



СТАНДАРТ ТРЕТЬЕГО ПОКОЛЕНИЯ

Т. А. Павловская

C/C++

Программирование на языке высокого уровня

для магистров и бакалавров

Допущено Министерством образования Российской Федерации в качестве
учебника для студентов высших учебных заведений, обучающихся
по направлению подготовки дипломированных специалистов
«Информатика и вычислительная техника»



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2013

ББК 32.973-018я7
УДК 681.3.06(075)
П12

Рецензенты:

Ревунков Г. И., кандидат технических наук, доцент, декан факультета информатики
и систем управления МГТУ им. Н. Э. Баумана

Варлинский Н. Н., кандидат технических наук, доцент кафедры МО ЭВМ
Санкт-Петербургского электротехнического университета

Фомичев В.С., доктор технических наук, профессор кафедры вычислительной техники
Санкт-Петербургского электротехнического университета

П12 **С/С++. Программирование на языке высокого уровня** / Т. А. Павловская. — СПб.:
Питер, 2013. — 461 с.: ил.

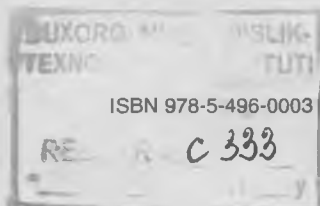
ISBN 978-5-496-00031-4

Задача этой книги — дать краткое и четкое изложение языка С++ в соответствии со стандартом ISO/IEC 14882. Она предназначена для студентов, изучающих язык «с нуля», но будет полезна и более искушенным в программировании. Цель, которая ставилась при написании книги — дать правильное представление о возможностях языка С++ и способах их применения, толчок к дальнейшему изучению этого и других языков программирования и ключ к пониманию современных объектно-ориентированных технологий.

Контрольные задания по ключевым темам представлены в 20 вариантах, и автор надеется, что преподаватели оценят проявленную о них заботу.

ББК 32.973-018я7
УДК 681.3.06(075)

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.
Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.



ISBN 978-5-496-00031-4

© ООО Издательство «Питер», 2013

Кратко

Предисловие . . .

Часть I. Структура

Глава 1. Базовые

Глава 2. Модуль

Глава 3. Технол

Упражнения к ча

Часть II. Объекты

Глава 4. Классы

Глава 5. Наслед

Глава 6. Шабло

Глава 7. Обраб

Глава 8. Преоб

Глава 9. Реком

Упражнения к ча

Часть III. Стандар

Глава 10. Поток

Глава 11. Строки

Глава 12. Конте

Глава 13. Итера

Глава 14. Алгори

Глава 15. Средс

Глава 16. Други

Упражнения к ча

Послесловие . . .

Литература . . .

Приложение 1. Оп

Приложение 2. Сп

Приложение 3. Ар

Приложение 4. За

Приложение 5. Ко

Приложение 6. Фу

Приложение 7. Та

Алфавитный указа

ББК 32.973-018я7

УДК 681.3.06(075)

П12

Рецензенты:

Ревунков Г. И., кандидат технических наук, доцент, декан факультета информатики и систем управления МГТУ им. Н. Э. Баумана

Варлинский Н. Н., кандидат технических наук, доцент кафедры МО ЭВМ Санкт-Петербургского электротехнического университета

Фомичев В. С., доктор технических наук, профессор кафедры вычислительной техники Санкт-Петербургского электротехнического университета

П12 **С/С++. Программирование на языке высокого уровня / Т. А. Павловская.** — СПб.: Питер, 2013. — 461 с.: ил.

ISBN 978-5-496-00031-4

Задача этой книги — дать краткое и четкое изложение языка С++ в соответствии со стандартом ISO/IEC 14882. Она предназначена для студентов, изучающих язык «с нуля», но будет полезна и более искушенным в программировании. Цель, которая ставилась при написании книги — дать правильное представление о возможностях языка С++ и способах их применения, толчок к дальнейшему изучению этого и других языков программирования и ключ к пониманию современных объектно-ориентированных технологий.

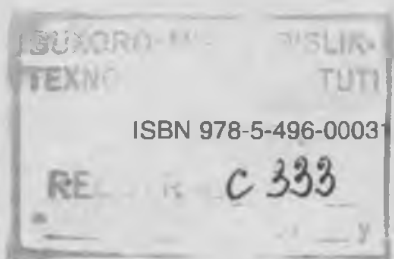
Контрольные задания по ключевым темам представлены в 20 вариантах, и автор надеется, что преподаватели оценят проявленную о них заботу.

ББК 32.973-018я7

УДК 681.3.06(075)

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.



ISBN 978-5-496-00031-4

© ООО Издательство «Питер», 2013

Краткое содержание

Предисловие	11
Часть I. Структурное программирование	13
Глава 1. Базовые средства языка C++.	15
Глава 2. Модульное программирование	72
Глава 3. Технология создания программ	102
Упражнения к части I	128
Часть II. Объектно-ориентированное программирование	173
Глава 4. Классы	178
Глава 5. Наследование	200
Глава 6. Шаблоны классов	211
Глава 7. Обработка исключительных ситуаций	222
Глава 8. Преобразования типов	231
Глава 9. Рекомендации по программированию	241
Упражнения к части II	248
Часть III. Стандартная библиотека	263
Глава 10. Потокковые классы	265
Глава 11. Строки	286
Глава 12. Контейнерные классы	295
Глава 13. Итераторы и функциональные объекты	328
Глава 14. Алгоритмы	343
Глава 15. Средства для численных расчетов	369
Глава 16. Другие средства стандартной библиотеки	378
Упражнения к части III	381
Послесловие	382
Литература	383
Приложение 1. Операции языка C++	384
Приложение 2. Спецификации формата для функций семейства printf	387
Приложение 3. Арифметические преобразования типов	390
Приложение 4. Заголовочные файлы стандартной библиотеки	391
Приложение 5. Константы, макросы и типы данных стандартной библиотеки	394
Приложение 6. Функции стандартной библиотеки	409
Приложение 7. Таблицы кодов ASCII	447
Алфавитный указатель	450

Содержание

Предисловие	11
Часть I. Структурное программирование	13
Глава 1. Базовые средства языка C++	15
Состав языка	15
Алфавит языка	17
Идентификаторы	18
Ключевые слова.	19
Знаки операций.	19
Константы	20
Комментарии	22
Типы данных C++	22
Концепция типа данных	22
Основные типы данных	23
Структура программы	26
Переменные и выражения	28
Переменные.	28
Операции	31
Выражения.	37
Базовые конструкции структурного программирования	38
Оператор «выражение»	40
Операторы ветвления	40
Операторы цикла	44
Операторы передачи управления	49
Указатели и массивы.	51
Указатели	51
Ссылки	58
Массивы	58
Типы данных, определяемые пользователем	65
Переименование типов (typedef)	65
Перечисления (enum)	66
Структуры (struct).	67
Объединения (union).	69

Глава 2. Модульное программирование	72
Функции	73
Объявление и определение функций	73
Глобальные переменные	76
Возвращаемое значение	76
Параметры функции	77
Рекурсивные функции	82
Перегрузка функций	83
Шаблоны функций	85
Функция <code>main()</code>	87
Функции стандартной библиотеки	88
Директивы препроцессора	93
Директива <code>#include</code>	93
Директива <code>#define</code>	94
Директивы условной компиляции	95
Директива <code>#undef</code>	96
Области действия и пространства имен	97
Внешние объявления	98
Поименованные области	99
 Глава 3. Технология создания программ	 102
Кодирование и документирование программы	102
Проектирование и тестирование программы	109
Динамические структуры данных	114
Линейные списки	115
Стеки	119
Очереди	121
Бинарные деревья	122
Реализация динамических структур с помощью массивов	126
 Упражнения к части I	 128
Циклические вычислительные процессы	128
Одномерные массивы	136
Двумерные массивы	139
Одномерные и двумерные массивы	142
Структуры	144
Указатели	151
Простейшие функции	151
Функции и файлы	151
Функции библиотеки для работы со строками и символами	159
Шаблоны функций	165
Модульное программирование	165
Динамические структуры данных	165

Часть II. Объектно-ориентированное программирование.	173
Глава 4. Классы	178
Описание класса	178
Описание объектов	180
Указатель this	181
Конструкторы	182
Конструктор копирования	184
Статические элементы класса	185
Статические поля	186
Статические методы	186
Дружественные функции и классы	187
Дружественная функция	187
Дружественный класс	188
Деструкторы	188
Перегрузка операций.	189
Перегрузка унарных операций	190
Перегрузка бинарных операций	191
Перегрузка операции присваивания	191
Перегрузка операций new и delete	192
Перегрузка операции приведения типа	195
Перегрузка операции вызова функции	195
Перегрузка операции индексирования	196
Указатели на элементы классов	197
Рекомендации по составу класса	198
Глава 5. Наследование.	200
Ключи доступа	200
Простое наследование.	201
Виртуальные методы	205
Механизм позднего связывания	207
Абстрактные классы	208
Множественное наследование	208
Отличия структур и объединений от классов.	210
Глава 6. Шаблоны классов	211
Создание шаблонов классов	211
Использование шаблонов классов.	219
Специализация шаблонов классов	220
Достоинства и недостатки шаблонов	221
Глава 7. Обработка исключительных ситуаций	222
Общий механизм обработки исключений.	223
Синтаксис исключений.	223

Перехват исключений	225
Список исключений функции	227
Исключения в конструкторах и деструкторах	228
Иерархии исключений	229
 Глава 8. Преобразования типов	 231
Операция приведения типов в стиле C	231
Операция <code>const_cast</code>	232
Операция <code>dynamic_cast</code>	232
Повышающее преобразование	233
Понижающее преобразование	233
Преобразование ссылок	235
Перекрестное преобразование	236
Операция <code>static_cast</code>	237
Операция <code>reinterpret_cast</code>	238
Динамическое определение типа	238
 Глава 9. Рекомендации по программированию	 241
Упражнения к части II	248
Классы	248
Наследование	252
Шаблоны классов	260
 Часть III. Стандартная библиотека	 263
 Глава 10. Потокные классы	 265
Стандартные потоки	267
Форматирование данных	269
Флаги и форматирующие методы	269
Манипуляторы	271
Методы обмена с потоками	273
Ошибки потоков	278
Файловые потоки	280
Строковые потоки	282
Потоки и типы, определенные пользователем	284
 Глава 11. Строки	 286
Конструкторы и присваивание строк	287
Операции	288

Функции.	288
Присваивание и добавление частей строк	288
Преобразования строк.	289
Поиск подстрок	291
Сравнение частей строк	293
Получение характеристик строк	294
 Глава 12. Контейнерные классы	295
Последовательные контейнеры	297
Векторы (vector)	299
Двусторонние очереди (deque).	304
Списки (list).	306
Стеки (stack)	310
Очереди (queue).	311
Очереди с приоритетами (priority_queue).	313
Ассоциативные контейнеры.	315
Словари (map)	316
Словари с дубликатами (multimap).	321
Множества (set)	321
Множества с дубликатами (multiset).	323
Битовые множества (bitset)	323
Пример использования контейнеров	325
 Глава 13. Итераторы и функциональные объекты	328
Итераторы	328
Обратные итераторы	331
Итераторы вставки	332
Потоковые итераторы	333
Функциональные объекты	334
Арифметические функциональные объекты	335
Предикаты	336
Отрицатели.	337
Связыватели	337
Адаптеры указателей на функции	338
Адаптеры методов	341
 Глава 14. Алгоритмы	343
Немодифицирующие операции с последовательностями.	344
adjacent_find	345
count, count_if	345
equal.	346
find, find_if	346
find_first_of	347
find_end	347
for_each	347

mismatch	348
search, search_n	348
Модифицирующие операции с последовательностями	348
copy, copy_backward	350
fill, fill_n	351
generate, generate_n	351
iter_swap, swap, swap_ranges	352
random_shuffle	352
remove, remove_if, remove_copy, remove_copy_if	353
replace, replace_if, replace_copy, replace_copy_if	354
reverse, reverse_copy	355
rotate, rotate_copy	355
transform	356
unique, unique_copy	357
Алгоритмы, связанные с сортировкой	357
binary_search	358
equal_range	359
inplace_merge	359
lexicographical_compare	359
lower_bound, upper_bound	360
max, min	360
max_element, min_element	360
merge	361
next_permutation, prev_permutation	361
nth_element	362
partial_sort, partial_sort_copy	363
partition, stable_partition	363
sort, stable_sort	364
Алгоритмы работы с множествами и пирамидами	364
includes	365
set_intersection	365
set_difference, set_symmetric_difference	366
set_union	366
Пример использования алгоритмов работы с множествами	366
make_heap	367
pop_heap	368
push_heap	368
sort_heap	368
Глава 15. Средства для численных расчетов	369
Обобщенные численные алгоритмы	369
accumulate	369
inner_product	370
partial_sum	370
adjacent_difference	371

Класс <code>valarray</code>	371
Вспомогательные классы	374
Глава 16. Другие средства стандартной библиотеки	378
Упражнения к части III	381
Послесловие	382
Литература	383
Приложение 1. Операции языка C++	384
Приложение 2. Спецификации формата для функций семейства <code>printf</code>	387
Приложение 3. Арифметические преобразования типов	390
Приложение 4. Заголовочные файлы стандартной библиотеки	391
Приложение 5. Константы, макросы и типы данных стандартной библиотеки	394
Приложение 6. Функции стандартной библиотеки	409
Состав заголовочных файлов	409
Алфавитный перечень функций.	416
Приложение 7. Таблицы кодов ASCII	447
Алфавитный указатель	450

За
со
се
то
ни
ле
ко
Кн
сп
ее
сп
гиз
Кр
ны
же
До
ка
пр
сы
ня
рас
ре
час
ни
пр
ка
ван
Бол
кот
ни
уни
В к
это

Класс <code>valarray</code>	371
Вспомогательные классы	374
Глава 16. Другие средства стандартной библиотеки	378
Упражнения к части III	381
Послесловие	382
Литература	383
Приложение 1. Операции языка C++	384
Приложение 2. Спецификации формата для функций семейства <code>printf</code>	387
Приложение 3. Арифметические преобразования типов	390
Приложение 4. Заголовочные файлы стандартной библиотеки	391
Приложение 5. Константы, макросы и типы данных стандартной библиотеки	394
Приложение 6. Функции стандартной библиотеки	409
Состав заголовочных файлов	409
Алфавитный перечень функций	416
Приложение 7. Таблицы кодов ASCII	447
Алфавитный указатель	450

ния». Синтаксис примеров соответствует стандарту C++, поэтому некоторые из них могут не компилироваться в оболочках, выпущенных до его утверждения.

Книга состоит из трех частей: «Структурное программирование», «Объемно-ориентированное программирование» и «Стандартная библиотека».

В первой части рассматриваются основные конструкции языка и базовая технология создания программ, отвечающих современным требованиям качества и надежности.

Вторая часть посвящена описанию средств языка, позволяющих успешно создавать программы большого объема — классов, наследования, шаблонов и итераторов. Материал этой части необходим также для освоения стандартной библиотеки.

В третьей части дано описание средств стандартной библиотеки C++. Рассматриваются потоки, контейнеры, итераторы, векторы, стандартные алгоритмы и многое другое.

Определения синтаксических элементов языка выделены в тексте книги жирным шрифтом.

В приложениях содержится большое количество справочной информации, полезной при написании любой программы.

Все ключевые слова, типы, константы, функции, макросы и классы, описанные в книге, можно найти по предметному указателю, что позволяет использовать книгу как справочник. Для удобства пользования после имени элемента в строке указателя через запятую приведено имя класса, к которому принадлежит данный элемент, или пояснение.

Пользуясь случаем, хочу выразить глубокую благодарность моим первым преподавателям программирования П. А. Шипилу и Г. Л. Голованевскому, сотрудникам издательства «Питер» А. В. Васильеву, коллегам А. В. Лаздину, Ю. А. Павлову и И. Яковлеву, всем корифеям, упомянутым в списке литературы, а моим родителям и сыну Евгению, нарисовавшему картинку.

Книга поддержана программой Министерства образования Российской Федерации «Государственная поддержка региональной научно-технической политики высшей школы и развития ее научного потенциала» на 2001 год.

Доброжелательную и конструктивную критику книги направляйте по адресу mx@tp2055.spb.edu или 2:5030/1016.20@fidonet.org.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на Web-сайте издательства <http://www.piter.com>.

ния». Синтаксис примеров соответствует стандарту C++, поэтому некоторые из них могут не компилироваться в оболочках, выпущенных до его утверждения.

Книга состоит из трех частей: «Структурное программирование», «Объектно-ориентированное программирование» и «Стандартная библиотека».

В первой части рассматриваются основные конструкции языка и базовая технология создания программ, отвечающих современным требованиям качества и надежности.

Вторая часть посвящена описанию средств языка, позволяющих успешно создавать программы большого объема — классов, наследования, шаблонов и исключений. Материал этой части необходим также для освоения стандартной библиотеки.

В третьей части дано описание средств стандартной библиотеки C++. Рассматриваются потоки, контейнеры, итераторы, векторы, стандартные алгоритмы и связи между ними.

Определения синтаксических элементов языка выделены в тексте книги полужирным шрифтом.

В приложениях содержится большое количество справочной информации, необходимой при написании любой программы.

Все ключевые слова, типы, константы, функции, макросы и классы, описанные в книге, можно найти по предметному указателю, что позволяет использовать ее и как справочник. Для удобства пользования после имени элемента в строке указателя через запятую приведено имя класса, к которому принадлежит данный элемент, или пояснение.

Пользуясь случаем, хочу выразить глубокую благодарность моим первым преподавателям программирования П. А. Шипилову и Г. Л. Голованевскому, сотруднику издательства «Питер» А. В. Васильеву, коллегам А. В. Лаздину, Ю. А. Щупаку и И. Яковлеву, всем корифеям, упомянутым в списке литературы, а также моим родителям и сыну Евгению, нарисовавшему картинку.

Книга поддержана программой Министерства образования Российской Федерации «Государственная поддержка региональной научно-технической политики высшей школы и развития ее научного потенциала» на 2001 год.

Доброжелательную и конструктивную критику книги направляйте по адресам mux@tp2055.spb.edu или 2:5030/1016.20@fidonet.org.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на Web-сайте издательства <http://www.piter.com>.

ЧАСТЬ I

Структурное программирование

*Написание хороших программ
требует ума, вкуса и терпения.*

Б. Страуструп

Традиционная технология программирования складывалась в условиях, когда основными потребителями программ были научные учреждения, вычислительные ресурсы были ограничены, а проблемы сопровождения по существу неизвестны. Основными критериями качества программы считались ее узко понимаемая эффективность и компактность. Со временем сложность программ возросла настолько, что на их разработку уходили годы труда большого коллектива, а в результате системы появлялись с опозданием и содержали тысячи ошибок.

Кризис программного обеспечения привел к необходимости создания нового способа создания программ, который снижал бы общие затраты на протяжении всего цикла программы, — от замысла до завершения эксплуатации. Такая технология появилась в начале 70-х годов и была названа структурным программированием. В его основе лежит сочетание теории программирования и личного опыта высококвалифицированных программистов, а также учет современных требований к программам и промышленного характера их производства.

Главное требование, которому должна удовлетворять программа, — работать в полном соответствии со спецификацией и адекватно реагировать на любые действия пользователя. Кроме этого, программа должна быть выпущена точно к заявленному сроку и допускать оперативное внесение необходимых изменений и дополнений. Объем занимаемой памяти и эффективность алгоритмов при этом, к сожалению, отходят на второй план. Иными словами, современные критерии качества программы — это, прежде всего, надежность, а также возможность точно планировать производство программы и ее сопровождение. Для достижения этих целей программа должна иметь простую структуру, быть хорошо читаемой и легко модифицируемой.

Структурное программирование — это технология создания программ, позволяющая путем соблюдения определенных правил уменьшить время разработки и количество ошибок, а также облегчить возможность модификации программы. Структурный подход охватывает все стадии разработки проекта: спецификацию, проектирование, собственно программирование и тестирование.

Особенно важно придерживаться определенной дисциплины при программировании на C++. Этот язык обладает настолько большой гибкостью и широкими возможностями, что, если не поставить себя в жесткие рамки с самого начала, программа быстро превратится в огромного неуправляемого монстра, не поддающегося отладке.

Структурный подход к программированию позволил успешно создавать достаточно крупные проекты, но сложность программного обеспечения продолжала возрастать, и требовались все более развитые средства ее преодоления. Идеи структурного программирования получили свое дальнейшее развитие в объектно-ориентированном программировании (ООП) — технологии, позволяющей достичь простоты структуры и управляемости очень крупных программных систем.

Не существует единственного самого лучшего способа создания программ. Для решения задач разного рода и уровня сложности требуется применять разные технологии программирования. В простейших случаях достаточно освоить азы структурного написания программ. Для создания же сложных проектов требуется не только свободно владеть языком в полном объеме, но и иметь представление о принципах проектирования и отладки программ, возможностях стандартной и других библиотек и т. д. Как правило, чем сложнее задача, тем больше времени требуется на освоение инструментов, необходимых для ее решения.

Мы начнем с изучения состава и основных конструкций языка — предмета для написания программ необходимого, но далеко не достаточного.

ГЛАВА 1

Базовые средства языка C++

Состав языка

В тексте на любом естественном языке можно выделить четыре основных элемента: символы, слова, словосочетания и предложения. Подобные элементы содержит и алгоритмический язык, только слова называют лексемами (элементарными конструкциями), словосочетания — выражениями, а предложения — операторами. Лексемы образуются из символов, выражения — из лексем и символов, а операторы — из символов, выражений и лексем (рис. 1.1):

- Алфавит языка, или его символы — это основные неделимые знаки, с помощью которых пишутся все тексты на языке.
- Лексема, или элементарная конструкция, — минимальная единица языка, имеющая самостоятельный смысл.
- Выражение задает правило вычисления некоторого значения.
- Оператор задает законченное описание некоторого действия.



Рис. 1.1. Состав алгоритмического языка

Для описания сложного действия требуется последовательность операторов. Операторы могут быть объединены в составной оператор, или блок¹. В этом случае они рассматриваются как один оператор.

Операторы бывают исполняемые и неисполняемые. Исполняемые операторы задают действия над данными. Неисполняемые операторы служат для описания данных, поэтому их часто называют операторами описания или просто описаниями.

Каждый элемент языка определяется синтаксисом и семантикой. Синтаксические определения устанавливают правила построения элементов языка, а семантика определяет их смысл и правила использования.

Объединенная единым алгоритмом совокупность описаний и операторов образует программу на алгоритмическом языке. Для того чтобы выполнить программу, требуется перевести ее на язык, понятный процессору — в машинные коды. Этот процесс состоит из нескольких этапов. Рисунок 1.2 иллюстрирует эти этапы для языка C++.

Сначала программа передается *препроцессору*, который выполняет *директивы*, содержащиеся в ее тексте (например, включение в текст так называемых заголовочных файлов — текстовых файлов, в которых содержатся описания используемых в программе элементов).

Получившийся полный текст программы поступает на вход *компилятора*, который выделяет лексемы, а затем на основе грамматики языка распознает выражения и операторы, построенные из этих лексем. При этом компилятор выявляет синтаксические ошибки и в случае их отсутствия строит *объектный модуль*.

Компоновщик, или редактор связей, формирует *исполняемый модуль* программы, подключая к объектному модулю другие объектные модули, в том числе содержащие функции библиотек, обращение к которым содержится в любой программе (например, для осуществления вывода на экран). Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки. Исполняемый модуль имеет расширение .exe и запускается на выполнение обычным образом.

Для **описания языка** в документации часто используется некоторый формальный метаязык, например, формулы Бэкуса—Наура или синтаксические диаграммы. Для наглядности и простоты изложения в этой книге используется широко распространенный неформальный способ описания, при котором необязательные части синтаксических конструкций заключаются в квадратные скобки, текст, который необходимо заменить конкретным значением, пишется по-русски, а выбор одного из нескольких элементов обозначается *вертикальной чертой*. Например, запись

```
[ void | int ] имя();
```

означает, что вместо конструкции имя необходимо указать конкретное имя в соответствии с правилами языка, а перед ним может находиться либо void, либо int, либо ничего. *Фигурные скобки* используются для группировки элементов,

¹ Блоком в языке C++ считается последовательность операторов, заключенная в фигурные скобки { }.

из которых требуется выбрать только один. В тех случаях, когда квадратные скобки являются элементом синтаксиса, это оговаривается особо.

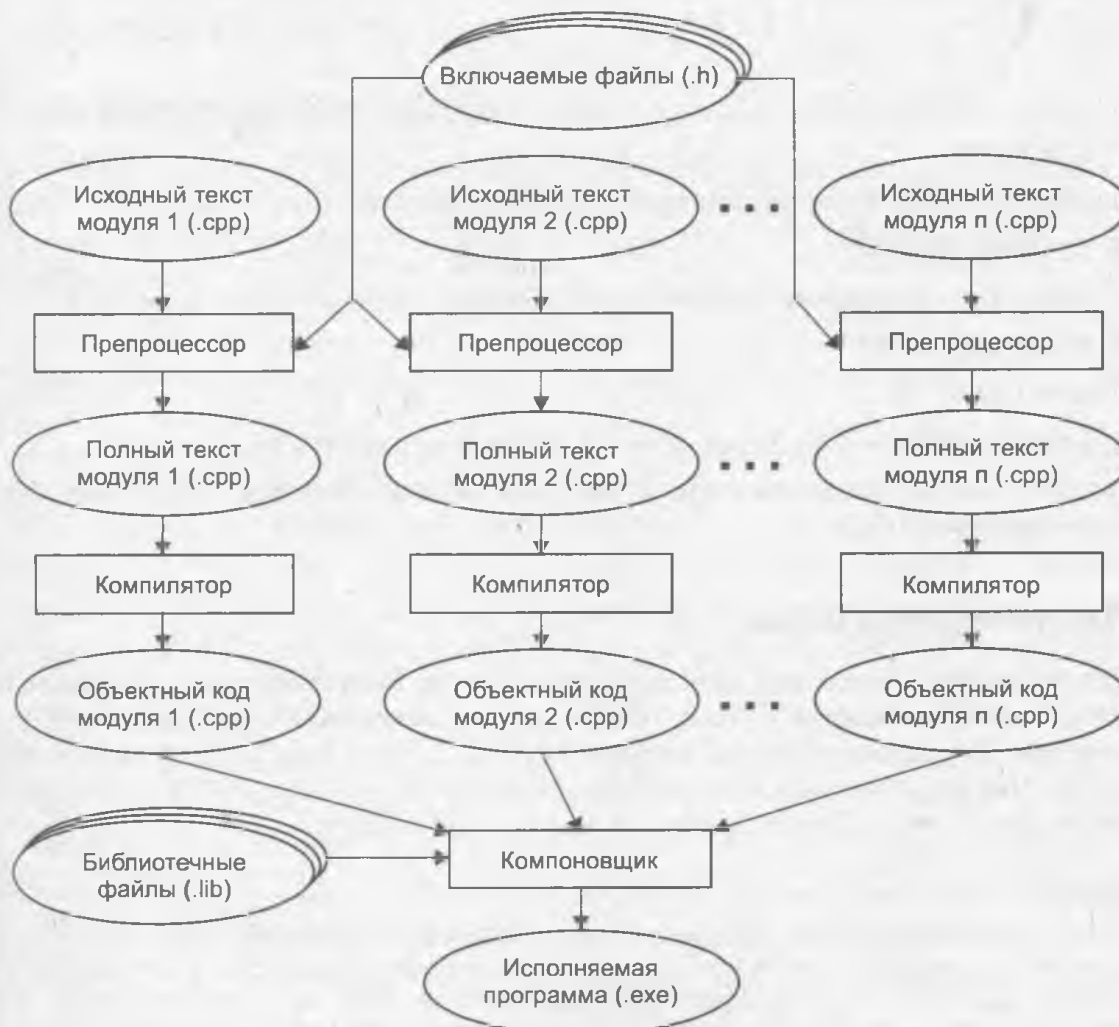


Рис. 1.2. Этапы создания исполняемой программы

Начнем изучение C++ с самого простого — с алфавита, а затем, осваивая все более сложные элементы, постепенно углубимся в дебри объектно-ориентированного программирования и постараемся в них не заблудиться. К концу изучения этой книги читателю будет легко и приятно порассуждать об «инкапсулированных абстракциях с четким протоколом доступа», о том, отчего нельзя «сгенерировать исключение, если конструктор копии объекта не является общедоступным», и о многих столь же интересных вещах.

Алфавит языка

Алфавит C++ включает:

- прописные и строчные латинские буквы и знак подчеркивания;



- ☐ арабские цифры от 0 до 9, шестнадцатичные цифры от A до F;
- ☐ специальные знаки:

" { } , ! [] () + - / % * . \

’ : ? < = > ! & # ~ ; ^

- ☐ пробельные символы: пробел, символы табуляции, символы перехода на новую строку.

Из символов алфавита формируются *лексемы* языка:

- ☐ идентификаторы;
- ☐ ключевые (зарезервированные) слова;
- ☐ знаки операций;
- ☐ константы;
- ☐ разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами, такими, как разделители или знаки операций.

Идентификаторы

Идентификатор — это имя программного объекта. В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, `sysop`, `SySoP` и `SYSOP` — три различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы внутри имен не допускаются.

СОВЕТ

Для улучшения читаемости программы следует давать объектам осмысленные имена. Существует соглашение о правилах создания имен, называемое венгерской нотацией (поскольку предложил ее сотрудник компании Microsoft венгр по национальности), по которому каждое слово, составляющее идентификатор, начинается с прописной буквы, а вначале ставится префикс, соответствующий типу величины, например, `iMaxLength`, `lpfnSetFirstDialog`. Другая традиция — разделять слова, составляющие имя, знаками подчеркивания: `max_length`, `number_of_galosh`.

Длина идентификатора по стандарту не ограничена, но некоторые компиляторы и компоновщики налагают на нее ограничения. Идентификатор создается на этапе объявления переменной, функции, типа и т. п., после этого его можно использовать в последующих операторах программы. При выборе идентификатора необходимо иметь в виду следующее:

- ☐ идентификатор не должен совпадать с ключевыми словами (см. следующий раздел) и именами используемых стандартных объектов языка;
- ☐ не рекомендуется начинать идентификаторы с символа подчеркивания, поскольку они могут совпасть с именами системных функций или переменных, и, кроме того, это снижает мобильность программы;

- на идентификаторы, используемые для определения внешних переменных, налагаются ограничения компоновщика (использование различных компоновщиков или версий компоновщика накладывает разные требования на имена внешних переменных).

Ключевые слова

Ключевые слова — это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Список ключевых слов C++ приведен в табл. 1.1.

Таблица 1.1. Список ключевых слов C++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Знаки операций

Знак операции — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Операции делятся на унарные, бинарные и тернарную по количеству участвующих в них операндов. Один и тот же знак может интерпретироваться по-разному в зависимости от контекста. Все знаки операций за исключением [], () и ? : представляют собой отдельные лексемы.

Знаки операций C++ описаны в разделе «Операции», с. 31, а также приведены в приложении 1. Большинство стандартных операций может быть переопределено (перегружено). Перегрузка операций рассмотрена на с. 189.

Константы

Константами называют неизменяемые величины. Различаются целые, вещественные, символьные и строковые константы. Компилятор, выделив константу в качестве лексемы, относит ее к одному из типов по ее внешнему виду¹.

Форматы констант, соответствующие каждому типу, приведены в табл. 1.2.

Таблица 1.2. Константы в языке C++

Константа	Формат	Примеры
Целая	Десятичный: последовательность десятичных цифр, начинающаяся не с нуля, если это не число нуль	8, 0, 199226
	Восьмеричный: нуль, за которым следуют восьмеричные цифры (0,1,2,3,4,5,6,7)	01, 020, 07155
	Шестнадцатеричный: 0x или 0X, за которым следуют шестнадцатеричные цифры (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)	0xA, 0x1B8, 0X00FF
Вещественная	Десятичный: [цифры].[цифры] ²	5.7, .001, 35.
	Экспоненциальный: [цифры][.][цифры]{E e}{+ -}[цифры] ³	0.2E6, .11e-3, 5E10
Символьная	Один или два символа, заключенных в апострофы	'A', 'ю', '*', 'db', '\0', '\n', '\012', '\x07\x07'
Строковая	Последовательность символов, заключенная в кавычки	"Здесь был Vasia", "\tЗначение r=\0xF5\n"

Допустимые диапазоны значений целых и вещественных констант приведены в табл. 1.4.

Если требуется сформировать *отрицательную* целую или вещественную константу, то перед константой ставится знак унарной операции изменения знака (-), например: -218, -022, -0x3C, -4.8, -0.1e4.

Вещественная константа в экспоненциальном формате представляется в виде *мантиссы* и *порядка*. Мантисса записывается слева от знака экспоненты (E или e), порядок — справа от знака. Значение константы определяется как произведение мантиссы и возведенного в указанную в порядке степень числа 10. Обратите внимание, что пробелы внутри числа не допускаются, а для отделения целой части от дробной используется не запятая, а точка.

¹ Программист может задать тип константы самостоятельно (см. стр. 24).

² Могут быть опущены либо целая часть, либо дробная, но не обе сразу.

³ Могут быть опущены либо целая часть, либо дробная, но не обе сразу. Если указаны обе части, символ точки обязателен.

Символьные константы, состоящие из одного символа, занимают в памяти один байт и имеют стандартный тип `char`. Двухсимвольные константы занимают два байта и имеют тип `int`, при этом первый символ размещается в байте с меньшим адресом (о типах данных рассказывается в следующем разделе).

Символ обратной косой черты используется для представления:

- ❑ кодов, не имеющих графического изображения (например, `\a` — звуковой сигнал, `\n` — перевод курсора в начало следующей строки);
- ❑ символов апострофа (`'`), обратной косой черты (`\`), знака вопроса (`?`) и кавычки (`"`);
- ❑ любого символа с помощью его шестнадцатеричного или восьмеричного кода, например, `\073`, `\0xF5`. Числовое значение должно находиться в диапазоне от 0 до 255.

Последовательности символов, начинающиеся с обратной косой черты, называются *управляющими*, или *escape-последовательностями*. В таблице 1.3 приведены их допустимые значения. Управляющая последовательность интерпретируется как одиночный символ. Если непосредственно за обратной косой чертой следует символ, не предусмотренный табл. 1.3, результат интерпретации не определен. Если в последовательности цифр встречается недопустимая, она считается концом цифрового кода.

Таблица 1.3. Управляющие последовательности в языке C++

Изображение	Шестнадцатеричный код	Наименование
<code>\a</code>	7	Звуковой сигнал
<code>\b</code>	8	Возврат на шаг
<code>\f</code>	C	Перевод страницы (формата)
<code>\n</code>	A	Перевод строки
<code>\r</code>	D	Возврат каретки
<code>\t</code>	9	Горизонтальная табуляция
<code>\v</code>	B	Вертикальная табуляция
<code>\\</code>	5C	Обратная косая черта
<code>\'</code>	27	Апостроф
<code>\"</code>	22	Кавычка
<code>\?</code>	3F	Вопросительный знак
<code>\0ddd</code>	—	Восьмеричный код символа
<code>\0xdd</code>	ddd	Шестнадцатеричный код символа

Управляющие последовательности могут использоваться и в строковых константах, называемых иначе *строковыми литералами*. Например, если внутри строки требуется записать кавычку, ее предваряют косой чертой, по которой компилятор отличает ее от кавычки, ограничивающей строку:

```
"Издательский дом \"Питер\""
```

Все строковые литералы рассматриваются компилятором как различные объекты. Строковые константы, отделенные в программе только пробельными символами, при компиляции объединяются в одну. Длинную строковую константу можно разместить на нескольких строках, используя в качестве знака переноса обратную косую черту, за которой следует перевод строки. Эти символы игнорируются компилятором, при этом следующая строка воспринимается как продолжение предыдущей. Например, строка

```
"Никто не доволен своей \  
внешностью, но все довольны \  
своим умом"
```

полностью эквивалентна строке

```
"Никто не доволен своей внешностью, но все довольны своим умом"
```

В конец каждого строкового литерала компилятором добавляется *нулевой символ*, представляемый управляющей последовательностью `\0`. Поэтому длина строки всегда на единицу больше количества символов в ее записи. Таким образом, пустая строка `" "` имеет длину 1 байт. Обратите внимание на разницу между строкой из одного символа, например, `"A"`, и символьной константой `'A'`.

Пустая символьная константа недопустима.

Комментарии

Комментарий либо начинается с двух символов «прямая косая черта» (`//`) и заканчивается символом перехода на новую строку, либо заключается между символами-скобками `/*` и `*/`. Внутри комментария можно использовать любые допустимые на данном компьютере символы, а не только символы из алфавита языка C++, поскольку компилятор комментарии игнорирует. Вложенные комментарии-скобки стандартом не допускаются, хотя в некоторых компиляторах разрешены.

СОВЕТ

Рекомендуется использовать для пояснений `//`-комментарии, а скобки `/* */` применять для временного исключения блоков кода при отладке. Советы по содержанию комментариев см. в разделе «Кодирование и документирование программы», с. 102.

Типы данных C++

Концепция типа данных

Основная цель любой программы состоит в обработке данных. Данные различного типа хранятся и обрабатываются по-разному. В любом алгоритмическом языке каждая константа, переменная, результат вычисления выражения или функции должны иметь определенный тип.

Тип данных определяет:

- *внутреннее представление* данных в памяти компьютера;

- *множество значений*, которые могут принимать величины этого типа;
- *операции и функции*, которые можно применять к величинам этого типа.

Исходя из этих характеристик, программист выбирает тип каждой величины, используемой в программе для представления реальных объектов. Обязательное описание типа позволяет компилятору производить проверку допустимости различных конструкций программы. От типа величины зависят машинные команды, которые будут использоваться для обработки данных.

Все типы языка C++ можно разделить на основные и *составные*. В языке C++ определено шесть основных типов данных для представления целых, вещественных, символьных и логических величин. На основе этих типов программист может вводить описание составных типов. К ним относятся массивы, перечисления, функции, структуры, ссылки, указатели, объединения и классы.

Основные типы данных

Основные (*стандартные*) типы данных часто называют *арифметическими*, поскольку их можно использовать в арифметических операциях. Для описания основных *типов* определены следующие ключевые слова:

- `int` (целый);
- `char` (символьный);
- `wchar_t` (расширенный символьный);
- `bool` (логический);
- `float` (вещественный);
- `double` (вещественный с двойной точностью).

Первые четыре типа называют целочисленными (*целыми*), последние два — *типами с плавающей точкой*. Код, который формирует компилятор для обработки целых величин, отличается от кода для величин с плавающей точкой.

Существует четыре *спецификатора типа*, уточняющих внутреннее представление и диапазон значений стандартных типов:

- `short` (короткий);
- `long` (длинный);
- `signed` (знаковый);
- `unsigned` (беззнаковый).

Целый тип (`int`)

Размер типа `int` не определяется стандартом, а зависит от компьютера и компилятора. Для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного — 4 байта.

Спецификатор `short` перед именем типа указывает компилятору, что под число требуется отвести 2 байта независимо от разрядности процессора. Спецификатор `long` означает, что целая величина будет занимать 4 байта. Таким образом, на 16-разрядном компьютере эквиваленты `int` и `short int`, а на 32-разрядном — `int` и `long int`.

Внутреннее представление величины целого типа — целое число в двоичном коде. При использовании спецификатора `signed` старший бит числа интерпретируется как знаковый (0 — положительное число, 1 — отрицательное). Спецификатор `unsigned` позволяет представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа. Таким образом, диапазон значений типа `int` зависит от спецификаторов. Диапазоны значений величин целого типа с различными спецификаторами для IBM PC-совместимых компьютеров приведены в табл. 1.4.

По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор `signed` можно опускать.

Константам, встречающимся в программе, приписывается тот или иной тип в соответствии с их видом. Если этот тип по каким-либо причинам не устраивает программиста, он может явно указать требуемый тип с помощью суффиксов `L`, `l` (`long`) и `U`, `u` (`unsigned`). Например, константа `32L` будет иметь тип `long` и занимать 4 байта. Можно использовать суффиксы `L` и `U` одновременно, например, `0x22UL` или `05Lu`.

ПРИМЕЧАНИЕ

Типы `short int`, `long int`, `signed int` и `unsigned int` можно сокращать до `short`, `long`, `signed` и `unsigned` соответственно.

Символьный тип (`char`)

Под величину символьного типа отводится количество байт, достаточное для размещения любого символа из набора символов для данного компьютера, что и обусловило название типа. Как правило, это 1 байт. Тип `char`, как и другие целые типы, может быть со знаком или без знака. В величинах со знаком можно хранить значения в диапазоне от -128 до 127. При использовании спецификатора `unsigned` значения могут находиться в пределах от 0 до 255. Этого достаточно для хранения любого символа из 256-символьного набора ASCII. Величины типа `char` применяются также для хранения целых чисел, не превышающих границы указанных диапазонов.

Расширенный символьный тип (`wchar_t`)

Тип `wchar_t` предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например, Unicode. Размер этого типа зависит от реализации; как правило, он соответствует типу `short`. Строковые константы типа `wchar_t` записываются с префиксом `L`, например, `L"Gates"`.

Логический тип (`bool`)

Величины логического типа могут принимать только значения `true` и `false`, являющиеся зарезервированными словами. Внутренняя форма представления значения `false` — 0 (нуль). Любое другое значение интерпретируется как `true`. При преобразовании к целому типу `true` имеет значение 1.

Типы с плавающей точкой (float, double и long double)

Стандарт C++ определяет три типа данных для хранения вещественных значений: float, double и long double.

Типы данных с плавающей точкой хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей — мантиссы и порядка. В IBM PC-совместимых компьютерах величины типа float занимают 4 байта, из которых один двоичный разряд отводится под знак мантиссы, 8 разрядов под порядок и 23 под мантиссу. Мантисса — это число, большее 1.0, но меньшее 2.0. Поскольку старшая цифра мантиссы всегда равна 1, она не хранится.

Для величин типа double, занимающих 8 байт, под порядок и мантиссу отводится 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка — его диапазон. Как можно видеть из табл. 1.4, при одинаковом количестве байтов, отводимом под величины типа float и long int, диапазоны их допустимых значений сильно различаются из-за внутренней формы представления.

Спецификатор long перед именем типа double указывает, что под величину отводится 10 байтов.

Константы с плавающей точкой имеют по умолчанию тип double. Можно явно указать тип константы с помощью суффиксов F, f (float) и L, l (long). Например, константа 2E+6L будет иметь тип long double, а константа 1.82f — тип float.

Таблица 1.4. Диапазоны значений простых типов данных для IBM PC

Тип	Диапазон значений	Размер (байт)
bool	true и false	1
signed char	-128 ... 127	1
unsigned char	0 ... 255	1
signed short int	-32 768 ... 32 767	2
unsigned short int	0 ... 65 535	2
signed long int	-2 147 483 648 ... 2 147 483 647	4
unsigned long int	0 ... 4 294 967 295	4
float	3.4e-38 ... 3.4e+38	4
double	1.7e-308 ... 1.7e+308	8
long double	3.4e-4932 ... 3.4e+4932	10

Для вещественных типов в таблице приведены абсолютные величины минимальных и максимальных значений.

Для написания переносимых на различные платформы программ нельзя делать предположений о размере типа int. Для его получения необходимо пользоваться операцией sizeof, результатом которой является размер типа в байтах. Например, для операционной системы MS-DOS sizeof (int) даст в результате 2, а для Windows 9X или OS/2 результатом будет 4.

В стандарте ANSI диапазоны значений для основных типов не задаются, определяются только соотношения между их размерами, например:

$$\text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}$$
$$\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$$

ПРИМЕЧАНИЕ

Минимальные и максимальные допустимые значения для целых типов зависят от реализации и приведены в заголовочном файле `<limits.h>` (`<climits>`), характеристики вещественных типов — в файле `<float.h>` (`<cfloat>`), а также в шаблоне класса `numeric_limits` (см. раздел «Другие средства стандартной библиотеки», с. 378, и приложение 5).

Различные виды целых и вещественных типов, различающиеся диапазоном и точностью представления данных, введены для того, чтобы дать программисту возможность наиболее эффективно использовать возможности конкретной аппаратуры, поскольку от выбора типа зависит скорость вычислений и объем памяти. Но оптимизированная для компьютеров какого-либо одного типа программа может стать не переносимой на другие платформы, поэтому в общем случае следует избегать зависимостей от конкретных характеристик типов данных.

Тип void

Кроме перечисленных, к основным типам языка относится тип `void`, но множество значений этого типа пусто. Он используется для определения функций, которые не возвращают значения, для указания пустого списка аргументов функции (о функциях рассказывается на с. 73), как базовый тип для указателей (с. 51) и в операции приведения типов (с. 56).

Структура программы

Программа на языке C++ состоит из *функций, описаний и директив препроцессора* (с. 16). Одна из функций должна иметь имя `main`. Выполнение программы начинается с первого оператора этой функции. Простейшее определение функции имеет следующий формат:

```
тип_возвращаемого_значения имя ([ параметры ]){  
    операторы, составляющие тело функции  
}
```

Как правило, функция используется для вычисления какого-либо значения, поэтому перед именем функции указывается его тип. О функциях рассказывается на с. 73, ниже приведены лишь самые необходимые сведения:

- ❑ если функция не должна возвращать значение, указывается тип `void`;
- ❑ тело функции является блоком и, следовательно, заключается в фигурные скобки;
- ❑ функции не могут быть вложенными;
- ❑ каждый оператор заканчивается точкой с запятой (кроме составного оператора).

Пример структуры программы, содержащей функции `main`, `f1` и `f2`:

```
директивы препроцессора
описания
int main(){
    операторы главной функции
}
int f1(){
    операторы функции f1
}
int f2(){
    операторы функции f2
}
```

Программа может состоять из нескольких *модулей* (исходных файлов).

Несколько предварительных замечаний о вводе/выводе. В языке C++ нет встроенных средств ввода/вывода — он осуществляется с помощью функций, типов и объектов, содержащихся в стандартных библиотеках. Используется два способа: функции, унаследованные из языка C, и объекты C++.

Основные *функции ввода/вывода* в стиле C:

```
int scanf (const char* format, ... ) // ввод
int printf(const char* format, ... ) // вывод
```

Они выполняют форматированный ввод и вывод произвольного количества величин в соответствии со строкой формата `format`. Строка формата содержит символы, которые при выводе копируются в поток (на экран) или запрашиваются из потока (с клавиатуры) при вводе, и *спецификации преобразования*, начинающиеся со знака %, которые при вводе и выводе заменяются конкретными величинами. Список наиболее употребительных спецификаций преобразования приведен в приложении 2.

Пример программы, использующей функции ввода/вывода в стиле C:

```
#include <stdio.h>
int main(){
    int i;
    printf("Введите целое число\n");
    scanf("%d", &i);
    printf("Вы ввели число %d, спасибо!", i);
    return 0;
}
```

Первая строка этой программы — директива препроцессора, по которой в текст программы вставляется заголовочный файл `<stdio.h>`, содержащий описание использованных в программе функций ввода/вывода (в данном случае угловые скобки являются элементом языка). Все директивы препроцессора начинаются со знака #. Директивы препроцессора описаны на с. 93.

Третья строка — описание переменной целого типа с именем `i`. Переменные рассматриваются на с. 28.

Функция `printf` в четвертой строке выводит приглашение «Введите целое число» и переходит на новую строку в соответствии с управляющей последователь-

ностью \n. Функция `scanf` заносит введенное с клавиатуры целое число в переменную `i` (знак `&` означает операцию получения адреса), а следующий оператор выводит на экран указанную в нем строку, заменив спецификацию преобразования на значение этого числа. Ввод/вывод в стиле C рассмотрен в разделе «Функции ввода/вывода» (с. 88).

А вот как выглядит та же программа с использованием *библиотеки классов C++*:

```
#include <iostream.h>
int main(){
    int i;
    cout << "Введите целое число\n";
    cin >> i;
    cout << "Вы ввели число " << i << ", спасибо!";
    return 0;
}
```

Заголовочный файл `<iostream.h>` содержит описание набора классов для управления вводом/выводом. В нем определены стандартные объекты-потoki `cin` для ввода с клавиатуры и `cout` для вывода на экран, а также операции помещения в поток `<<` и чтения из потока `>>`. Потoki рассмотрены в разделе «Потоковые классы» на с. 265.

В дальнейшем изложении будут использоваться оба способа, но в одной программе смешивать их не рекомендуется.

Переменные и выражения

В любой программе требуется производить вычисления. Для вычисления значений используются *выражения*, которые состоят из операндов, знаков операций и скобок. Операнды задают данные для вычислений. Операции задают действия, которые необходимо выполнить. Каждый операнд является, в свою очередь, выражением или одним из его частных случаев, например, константой или переменной. Операции выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки.

Рассмотрим составные части выражений и правила их вычисления.

Переменные

Переменная — это именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение. Во время выполнения программы значение переменной можно изменять. Перед использованием любая переменная должна быть описана.

Пример описания целой переменной с именем `a` и вещественной переменной `x`:

```
int a; float x;
```

Общий вид *оператора описания* переменных:

```
[класс памяти] [const] тип имя [инициализатор];
```

Рассмотрим правила задания составных частей этого оператора.

- Необязательный класс памяти может принимать одно из значений `auto`, `extern`, `static` и `register`. О них рассказывается чуть ниже.
- Модификатор `const` показывает, что значение переменной изменять нельзя. Такую переменную называют *именованной константой*, или просто *константой*.
- При описании можно присвоить переменной начальное значение, это называется *инициализацией*. Инициализатор можно записывать в двух формах — со знаком равенства:

`= значение`

или в круглых скобках:

`( значение )`

Константа должна быть инициализирована при объявлении. В одном операторе можно описать несколько переменных одного типа, разделяя их запятыми.

Примеры:

```
short int a = 1;           // целая переменная a
const char C = 'C';        // символьная константа C
char s, sf = 'f';          // инициализация относится только к sf
char t (54);
float c = 0.22, x(3), sum;
```

Если тип инициализирующего значения не совпадает с типом переменной, выполняются *преобразования типа* по определенным правилам (см. с. 38 и приложение 3).

Описание переменной, кроме типа и класса памяти, явно или по умолчанию задает ее *область действия*. Класс памяти и область действия зависят не только от собственно описания, но и от места его размещения в тексте программы.

Область действия идентификатора — это часть программы, в которой его можно использовать для доступа к связанной с ним области памяти. В зависимости от области действия переменная может быть локальной или глобальной.

Если переменная определена внутри блока (напомню, что блок ограничен фигурными скобками), она называется *локальной*, область ее действия — от точки описания до конца блока, включая все вложенные блоки. Если переменная определена вне любого блока, она называется *глобальной* и областью ее действия считается файл, в котором она определена, от точки описания до его конца.

Класс памяти определяет время жизни и область видимости программного объекта (в частности, переменной). Если класс памяти не указан явным образом, он определяется компилятором исходя из контекста объявления.

Время жизни может быть постоянным (в течение выполнения программы) и временным (в течение выполнения блока).

Областью видимости идентификатора называется часть текста программы, из которой допустим обычный доступ к связанной с идентификатором области памяти. Чаще всего область видимости совпадает с областью действия. Исключением является ситуация, когда во вложенном блоке описана переменная с таким

же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя он и входит в ее область действия. Тем не менее к этой переменной, если она глобальная, можно обратиться, используя операцию доступа к области видимости ::.

Для задания *класса памяти* используются следующие спецификаторы:

`auto` — *автоматическая* переменная. Память под нее выделяется в стеке и при необходимости инициализируется каждый раз при выполнении оператора, содержащего ее определение. Освобождение памяти происходит при выходе из блока, в котором описана переменная. Время ее жизни — с момента описания до конца блока. Для глобальных переменных этот спецификатор не используется, а для локальных он принимается по умолчанию, поэтому задавать его явным образом большого смысла не имеет.

`extern` — означает, что переменная определяется в другом месте программы (в другом файле или дальше по тексту). Используется для создания переменных, доступных во всех модулях программы, в которых они объявлены¹. Подробнее об употреблении внешних переменных рассказывается в разделе «Внешние объявления», с. 98.

`static` — *статическая* переменная. Время жизни — постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. В зависимости от расположения оператора описания статические переменные могут быть глобальными и локальными. Глобальные статические переменные видны только в том модуле, в котором они описаны.

`register` — аналогично `auto`, но память выделяется по возможности в регистрах процессора. Если такой возможности у компилятора нет, переменные обрабатываются как `auto`.

```
int a;                // 1 глобальная переменная a
int main(){
    int b;            // 2 локальная переменная b
    extern int x;      // 3 переменная x определена в другом месте
    static int c;      // 4 локальная статическая переменная c
    a = 1;             // 5 присваивание глобальной переменной
    int a;            // 6 локальная переменная a
    a = 2;             // 7 присваивание локальной переменной
    ::a = 3;           // 8 присваивание глобальной переменной
    return 0;
}
int x = 4;            // 9 определение и инициализация x
```

В этом примере глобальная переменная `a` определена вне всех блоков. Память под нее выделяется в сегменте данных в начале работы программы, областью действия является вся программа. Область видимости — вся программа, кроме строк 6–8, так как в первой из них определяется локальная переменная с тем же именем, область действия которой начинается с точки ее описания и заканчива-

¹ Если переменная в том же операторе инициализируется, спецификатор `extern` игнорируется.

ется при выходе из блока. Переменные `b` и `c` — локальные, область их видимости — блок, но время жизни различно: память под `b` выделяется в стеке при входе в блок и освобождается при выходе из него, а переменная `c` располагается в сегменте данных и существует все время, пока работает программа.

Если при определении начальное значение переменных явным образом не задается, компилятор присваивает глобальным и статическим переменным нулевое значение соответствующего типа. Автоматические переменные не инициализируются.

Имя переменной должно быть уникальным в своей области действия (например, в одном блоке не может быть двух переменных с одинаковыми именами). Более подробно области действия рассматриваются на с. 97.

СОВЕТ

Не жалейте времени на придумывание подходящих имен. Имя должно отражать смысл хранимой величины, быть легко распознаваемым и, желательно, не содержать символов, которые можно перепутать друг с другом, например, `l`, `1` (строчная `L`) или `I` (прописная `i`). Для разделения частей имени можно использовать знак подчеркивания. Не давайте переменным имена, демонстрирующие знание иностранного слэнга — ведь «как вы лодку назовете, так она и поплывет». Как правило, переменным с большой областью видимости даются более длинные имена (желательно с префиксом типа), а для переменных, вся жизнь которых проходит на протяжении нескольких строк исходного текста, хватит и одной буквы с комментарием при объявлении.

Описание переменной может выполняться в форме *объявления или определения*. *Объявление* информирует компилятор о типе переменной и классе памяти, а *определение содержит, кроме этого*, указание компилятору выделить память в соответствии с типом переменной. В C++ большинство объявлений являются одновременно и определениями. В приведенном выше примере только описание `z` является объявлением, но не определением.

Переменная может быть объявлена многократно, но определена только в одном месте программы, поскольку объявление просто описывает свойства переменной, а определение связывает ее с конкретной областью памяти.

Операции

В таблице 1.5 приведен список основных операций, определенных в языке C++, в соответствии с их приоритетами (по убыванию приоритетов, операции с разными приоритетами разделены чертой). Остальные операции будут вводиться по мере изложения. Полный список операций приведен в приложении 1.

В соответствии с количеством операндов, которые используются в операциях, они делятся на унарные (один операнд), бинарные (два операнда) и тернарную (три операнда).

Все приведенные в таблице операции, кроме условной и `sizeof`, могут быть перегружены. О перегрузке операций рассказывается на с. 189.

Таблица 1.5. Основные операции языка C++

Операция	Краткое описание
Унарные операции	
++	увеличение на 1
--	уменьшение на 1 ¹
sizeof	размер
~	поразрядное отрицание
!	логическое отрицание
-	арифметическое отрицание (унарный минус)
+	унарный плюс
&	взятие адреса
*	разадресация
new	выделение памяти
delete	освобождение памяти
(type)	преобразование типа
Бинарные и тернарная операции	
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
==	равно
!=	не равно
&	поразрядная конъюнкция (И)
^	поразрядное исключающее ИЛИ
	поразрядная дизъюнкция (ИЛИ)
&&	логическое И
	логическое ИЛИ

¹ Пробелы между символами внутри операции не допускаются.

Опер

?

=

*=

/=

%=

+=

-=

<<=

>>=

&=

|=

^=

Рассм

Опера

также

когда

форме

рующи

являет

#inc

int

i

p

p

p

p

p

p

}

Результ

Знач

Знач

Знач

Знач

Операн

чение (L

сток па

ции при

Операция	Краткое описание
? :	условная операция (тернарная)
=	присваивание
*=	умножение с присваиванием
/=	деление с присваиванием
%=	остаток от деления с присваиванием
+=	сложение с присваиванием
-=	вычитание с присваиванием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное И с присваиванием
=	поразрядное ИЛИ с присваиванием
^=	поразрядное исключающее ИЛИ с присваиванием
.	последовательное вычисление

Рассмотрим основные операции подробнее.

Операции увеличения и уменьшения на 1 (++ и --). Эти операции, называемые также инкрементом и декрементом, имеют две формы записи — *префиксную*, когда операция записывается перед операндом, и *постфиксную*. В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

```
#include <stdio.h>
int main(){
    int x = 3, y = 3;
    printf("Значение префиксного выражения: %d\n", ++x);
    printf("Значение постфиксного выражения: %d\n", y++);
    printf("Значение x после приращения: %d\n", x);
    printf("Значение y после приращения: %d\n", y);
    return 0;
}
```

Результат работы программы:

```
Значение префиксного выражения: 4
Значение постфиксного выражения: 3
Значение x после приращения: 4
Значение y после приращения: 4
```

Операндом операции инкремента в общем случае является так называемое *L-значение (L-value)*. Так обозначается любое выражение, адресуемое некоторый участок памяти, в который можно занести значение. Название произошло от операции присваивания, поскольку именно ее левая (Left) часть определяет, в какую

область памяти будет занесен результат операции. Переменная является частным случаем L-значения.

Операция определения размера sizeof предназначена для вычисления размера объекта или типа в байтах, и имеет две формы:

```
sizeof выражение
sizeof ( тип )
```

Пример:

```
#include <iostream.h>
int main(){
    float x = 1;
    cout << "sizeof (float) :" << sizeof (float);
    cout << "\nsizeof x :" << sizeof x;
    cout << "\nsizeof (x + 1.0) :" << sizeof (x + 1.0);
    return 0;
}
```

Результат работы программы:

```
sizeof (float) : 4
sizeof x : 4
sizeof (x + 1.0) : 8
```

Последний результат связан с тем, что вещественные константы по умолчанию имеют тип double, к которому, как к более длинному, приводится тип переменной x и всего выражения. Скобки необходимы для того чтобы выражение, стоящее в них, вычислялось раньше операции приведения типа, имеющей больший приоритет, чем сложение.

Операции отрицания (-, ! и ~). *Арифметическое отрицание* (унарный минус -) изменяет знак операнда целого или вещественного типа на противоположный. *Логическое отрицание* (!) дает в результате значение 0, если операнд есть истинный (не нуль), и значение 1, если операнд равен нулю. Операнд должен быть целого или вещественного типа, а может иметь также тип указатель. *Побитовое отрицание* (~), часто называемое побитовым, инвертирует каждый разряд в двоичном представлении целочисленного операнда.

Деление (/) и остаток от деления (%). *Операция деления* применима к операндам арифметического типа. Если оба операнда целочисленные, результат операции округляется до целого числа, в противном случае тип результата определяется правилами преобразования (см. раздел «Выражения», с. 7–38, и приложение 3). *Операция остатка от деления* применяется только к целочисленным операндам. Знак результата зависит от реализации.

```
#include <stdio.h>
int main(){
    int x = 11, y = 4;
    float z = 4;
    printf("Результаты деления: %d %f\n", x/y, x/z);
    printf("Остаток: %d\n", x%y);
    return 0;
}
```

Результат работы программы:

Результаты деления: 2 2.750000

Остаток: 3

Операции сдвига (<< и >>) применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом. При *сдвиге влево* (<<) освободившиеся разряды обнуляются. При *сдвиге вправо* (>) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа, и знаковым разрядом в противном случае. Операции сдвига не учитывают переполнение и потерю значимости.

Операции отношения (<, <=, >, >=, ==, !=) сравнивают первый операнд со вторым. Операнды могут быть арифметического типа или указателями. Результатом операции является значение true или false (любое значение, не равное нулю, интерпретируется как true). Операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции сравнения.

ВНИМАНИЕ

Обратите внимание на разницу между операцией проверки на равенство (==) и операцией присваивания (=), результатом которой является значение, присвоенное левому операнду.

Поразрядные операции (&, |, ^) применяются только к целочисленным операндам и работают с их двоичными представлениями. При выполнении операций операнды сопоставляются побитово (первый бит первого операнда с первым битом второго, второй бит первого операнда со вторым битом второго, и т. д.).

При *поразрядной конъюнкции*, или *поразрядном И* (операция обозначается &) бит результата равен 1 только тогда, когда соответствующие биты обоих операндов равны 1.

При *поразрядной дизъюнкции*, или *поразрядном ИЛИ* (операция обозначается |) бит результата равен 1 тогда, когда соответствующий бит хотя бы одного из операндов равен 1.

При *поразрядном исключаящем ИЛИ* (операция обозначается ^) бит результата равен 1 только тогда, когда соответствующий бит только одного из операндов равен 1.

```
#include <iostream.h>
int main(){
    cout << "\n 6 & 5 = " << (6 & 5);
    cout << "\n 6 | 5 = " << (6 | 5);
    cout << "\n 6 ^ 5 = " << (6 ^ 5);
    return 0;
}
```

Результат работы программы:

6 & 5 = 4

6 | 5 = 7

6 ^ 5 = 3

Логические операции (&& и ||). Операнды логических операций И (&&) и ИЛИ (||) могут иметь арифметический тип или быть указателями, при этом операнды в каждой операции могут быть различных типов. Преобразования типов не производятся, каждый операнд оценивается с точки зрения его эквивалентности нулю (операнд, равный нулю, рассматривается как false, не равный нулю — как true).

Результатом логической операции является true или false. Результат операции *логическое И* имеет значение true только если оба операнда имеют значение true. Результат операции *логическое ИЛИ* имеет значение true, если хотя бы один из операндов имеет значение true. Логические операции выполняются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

Операции присваивания (=, +=, -=, *= и т. д.). Операции присваивания могут использоваться в программе как законченные операторы.

Формат операции *простого присваивания* (=):

```
операнд_1 = операнд_2
```

Первый операнд должен быть L-значением, второй — выражением. Сначала вычисляется выражение, стоящее в правой части операции, а потом его результат записывается в область памяти, указанную в левой части (мнемоническое правило: «присваивание — это передача данных "налево"»). То, что ранее хранилось в этой области памяти, естественно, теряется.

```
#include <iostream.h>
int main(){
    int a = 3, b = 5, c = 7;
    a = b; b = a; c = c + 1;
    cout << "a = " << a;
    cout << "\t b = " << b;
    cout << "\t c = " << c;
    return 0;
}
```

Результат работы программы:

```
a = 5 b = 5 c = 8
```

ВНИМАНИЕ

При присваивании производится преобразование типа выражения к типу L-значения, что может привести к потере информации.

В сложных операциях присваивания (+, *=, /= и т. п.) при вычислении выражения, стоящего в правой части, используется и L-значение из левой части. Например, при сложении с присваиванием ко второму операнду прибавляется первый, и результат записывается в первый операнд, то есть выражение $a += b$ является более компактной записью выражения $a = a + b$.

Условная операция (? :). Эта операция тернарная, то есть имеет три операнда. Ее формат:

```
операнд_1 ? операнд_2 : операнд_3
```

Первый операнд может иметь арифметический тип или быть указателем. Он оценивается с точки зрения его эквивалентности нулю (операнд, равный нулю, рассматривается как `false`, не равный нулю — как `true`). Если результат вычисления операнда 1 равен `true`, то результатом условной операции будет значение второго операнда, иначе — третьего операнда. Вычисляется всегда либо второй операнд, либо третий. Их тип может различаться. Условная операция является сокращенной формой условного оператора `if` (он рассмотрен на с. 40).

```
#include <stdio.h>
int main(){
    int a = 11, b = 4, max;
    max = (b > a)? b : a;
    printf("Наибольшее число: %d", max);
    return 0;
}
```

Результат работы программы:

Наибольшее число: 11

Другой пример применения условной операции. Требуется, чтобы некоторая целая величина увеличивалась на 1, если ее значение не превышает `n`, а иначе принимала значение 1:

```
i = (i < n) ? i + 1: 1;
```

Не рассмотренные в этом разделе операции будут описаны позже.

Выражения

Как уже говорилось выше, выражения состоят из операндов, операций и скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд является, в свою очередь, выражением или одним из его частных случаев — константой или переменной.

Примеры выражений:

```
(a + 0.12)/6
x && y || !z
(t * sin(x)-1.05e4)/((2 * k + 2) * (2 * k + 3))
```

Операции выполняются в соответствии с *приоритетами*. Для изменения порядка выполнения операций используются круглые скобки. Если в одном выражении записано несколько операций одинакового приоритета, унарные операции, условная операция и операции присваивания выполняются *справа налево*, остальные — *слева направо*. Например, `a = b = c` означает `a = (b = c)`, `a a + b + c` означает `(a + b) + c`. Порядок вычисления подвыражений внутри выражений не определен: например, нельзя считать, что в выражении `(sin(x + 2) + cos(y + 1))` обращение к синусу будет выполнено раньше, чем к косинусу, и что `x + 2` будет вычислено раньше, чем `y + 1`.

Результат вычисления выражения характеризуется значением и типом. Например, если `a` и `b` — переменные целого типа и описаны так:

```
int a = 2, b = 5;
```


то выражение $a + b$ имеет значение 7 и тип `int`, а выражение $a = b$ имеет значение, равное помещенному в переменную a (в данном случае 5) и тип, совпадающий с типом этой переменной. Таким образом, в C++ допустимы выражения вида $a = b = c$: сначала вычисляется выражение $b = c$, а затем его результат становится правым операндом для операции присваивания переменной a .

В выражение могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат операции будет иметь тот же тип. Если операнды разного типа, перед вычислениями выполняются преобразования типов по определенным правилам, обеспечивающим преобразование более коротких типов в более длинные для сохранения значимости и точности.

Преобразования бывают двух типов:

- ☐ изменяющие внутреннее представление величин (с потерей точности или без потери точности);
- ☐ изменяющие только интерпретацию внутреннего представления.

К первому типу относится, например, преобразование целого числа в вещественное (без потери точности) и наоборот (возможно, с потерей точности), ко второму — преобразование знакового целого в беззнаковое.

В любом случае величины типов `char`, `signed char`, `unsigned char`, `short int` и `unsigned short int` преобразуются в тип `int`, если он может представить все значения, или в `unsigned int` в противном случае.

После этого операнды преобразуются к типу наиболее длинного из них, и он используется как тип результата. Правила преобразований приведены в приложении 3.

Программист может задать преобразования типа явным образом (об этом рассказывается в разделе «Преобразования типов» на с. 231).

Итак, мы потратили достаточно много времени, рассматривая самый нижний уровень конструкций, составляющих язык программирования, и теперь пора начинать строить из них нечто более осмысленное — сначала отдельные операторы, а затем программы. Даже самая простая программа должна создаваться по определенным правилам, тогда она будет надежна, эффективна и красива. Изучением этих правил мы и займемся в следующем разделе.

Создание простейших программ подробно рассматривается на первом семинаре практикума [11, с. 13].

Базовые конструкции структурного программирования

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур, называемых следованием, ветвлением и циклом. Этот результат установлен Боймом и Якопини еще в 1966 году путем доказательства того, что любую программу можно преобразовать в эквивалентную, состоящую только из этих структур и их комбинаций.

Следование, ветвление и цикл называют *базовыми конструкциями* структурного программирования. *Следованием* называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или со

ставных). *Ветвление* задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия. *Цикл* задает многократное выполнение оператора (рис. 1.3). Особенностью базовых конструкций является то, что любая из них имеет только один вход и один выход, поэтому конструкции могут вкладываться друг в друга произвольным образом, например, цикл может содержать следование из двух ветвлений, каждое из которых включает вложенные циклы (рис. 1.4).

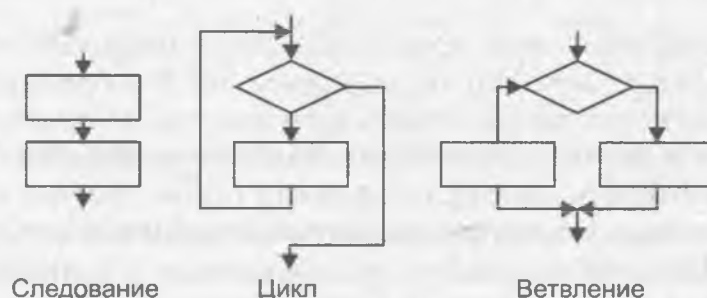


Рис. 1.3. Базовые конструкции структурного программирования

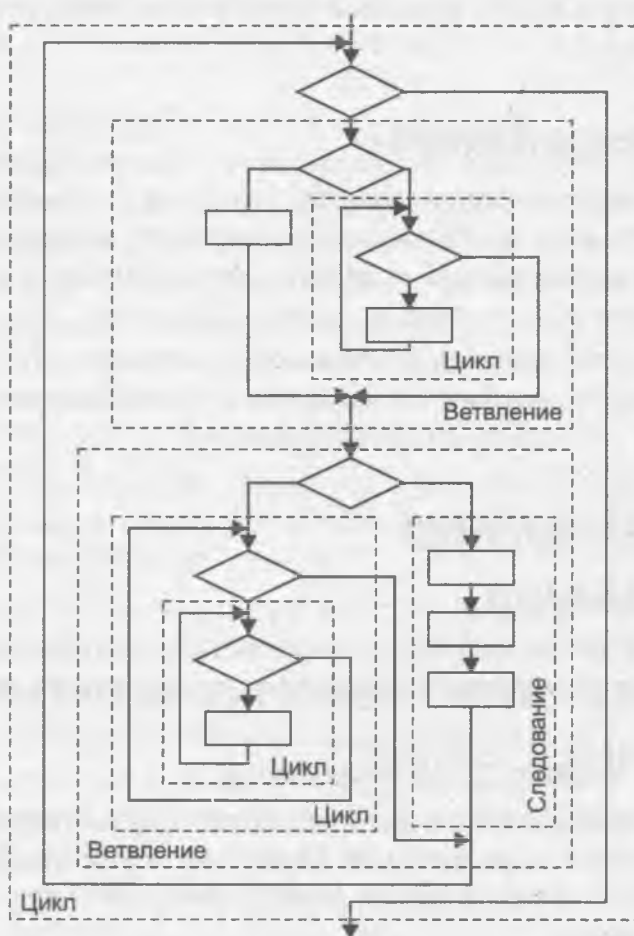


Рис. 1.4. Вложение базовых конструкций

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать (а программы чаще прихо-

дится читать, чем писать), отлаживать и при необходимости вносить в нее изменения. Структурное программирование часто называли «программированием без goto», и в этом есть большая доля правды: частое использование операторов передачи управления в произвольные точки программы затрудняет прослеживание логики ее работы. С другой стороны, никакие принципы нельзя возводить в абсолют, и есть ситуации, в которых использование goto оправдано и приводит, напротив, к упрощению структуры программы. О них говорится в разделе «Оператор goto» (с. 49).

В большинстве языков высокого уровня существует несколько реализаций базовых конструкций; в C++ есть три вида циклов и два вида ветвлений (на два и на произвольное количество направлений). Они введены для удобства программирования, и в каждом случае надо выбирать наиболее подходящие средства. Главное, о чем нужно помнить даже при написании самых простых программ, — что они должны состоять из четкой последовательности блоков строго определенной конфигурации. «Кто ясно мыслит, тот ясно излагает» — практика давно показала, что программы в стиле «поток сознания» нежизнеспособны, не говоря о том, что они просто некрасивы.

Рассмотрим операторы языка, реализующие базовые конструкции структурного программирования.

Оператор «выражение»

Любое выражение, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении выражения. Частным случаем выражения является *пустой оператор*; (он используется, когда по синтаксису оператор требуется, а по смыслу — нет). Примеры:

```
i++;           // выполняется операция инкремента
a *= b + c;    // выполняется умножение с присваиванием
fun(i, k);     // выполняется вызов функции
```

Операторы ветвления

Условный оператор if

Условный оператор if используется для разветвления процесса вычислений на два направления. Структурная схема оператора приведена на рис. 1.5. Формат оператора:

```
if ( выражение ) оператор_1; [else оператор_2;]
```

Сначала вычисляется выражение, которое может иметь арифметический тип или тип указателя. Если оно не равно нулю (имеет значение true), выполняется первый оператор, иначе — второй. После этого управление передается на оператор, следующий за условным.

Одна из ветвей может отсутствовать, логичнее опускать вторую ветвь вместе с ключевым словом else. Если в какой-либо ветви требуется выполнить несколько операторов, их необходимо заключить в блок, иначе компилятор не сможет понять, где заканчивается ветвление. Блок может содержать любые операторы,

в том числе описания и другие условные операторы (но не может состоять из одних описаний). Необходимо учитывать, что переменная, описанная в блоке, вне блока не существует.

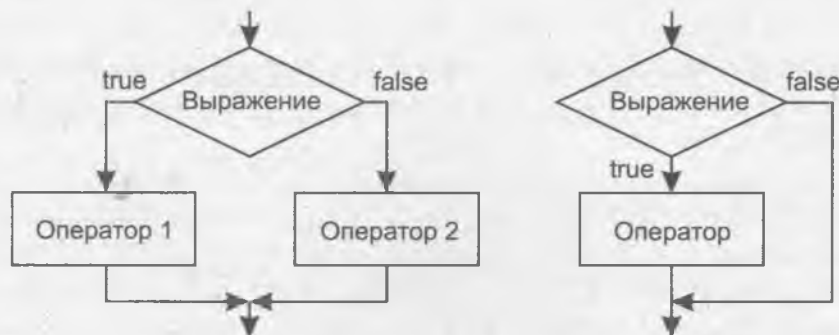


Рис. 1.5. Структурная схема условного оператора

Примеры:

```

if (a<0) b = 1; // 1
if (a<b && (a>d || a==0)) b++; else {b* = a; a = 0;} // 2
if (a<b) {if (a<c) m = a; else m = c;} else {if (b<c) m = b; else m = c;} // 3
if (a++) b++; // 4
if (b>a) max = b; else max = a; // 5
  
```

В *примере 1* отсутствует ветвь `else`. Подобная конструкция называется «пропуск оператора», поскольку присваивание либо выполняется, либо пропускается в зависимости от выполнения условия.

Если требуется проверить несколько условий, их объединяют знаками логических операций. Например, выражение в *примере 2* будет истинно в том случае, если выполнится одновременно условие `a<b` и одно из условий в скобках. Если опустить внутренние скобки, будет выполнено сначала логическое И, а потом — ИЛИ.

Оператор в *примере 3* вычисляет наибольшее значение из трех переменных. Фигурные скобки в данном случае не обязательны, так как компилятор относит часть `else` к ближайшему `if`.

Пример 4 напоминает о том, что хотя в качестве выражений в операторе `if` чаще всего используются операции отношения, это не обязательно.

Конструкции, подобные оператору в *примере 5*, проще и нагляднее записывать в виде условной операции (в данном случае: `max = (b > a) ? b : a;`).

Пример. Производится выстрел по мишени, изображенной на рис. 1.6. Определить количество очков.

```

#include <iostream.h>
int main(){
    float x, y; int kol;
    cout << "Введите координаты выстрела\n";
    cin >> x >> y;
    if ( x*x + y*y < 1 ) kol = 2;
    else if( x*x + y*y < 4 ) kol = 1;
  
```

```

else kol = 0;
cout << "\n Очков: " << kol;
return 0;
}

```

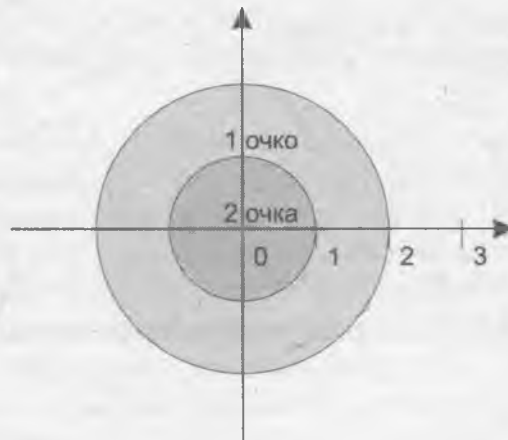


Рис. 1.6. Мишень

ВНИМАНИЕ

Распространенная ошибка при записи условных операторов — *использование в выражениях вместо проверки на равенство (==) простого присваивания (=)*, например, `if(a=1)b=0;`. Синтаксической ошибки нет, так как операция присваивания формирует результат, который и оценивается на равенство/неравенство нулю. В данном примере присваивание переменной `b` будет выполнено независимо от значения переменной `a`. Поэтому в выражениях проверки переменной на равенство константе константу рекомендуется записывать слева от операции сравнения: `if (1==a)b=0;`.

Вторая ошибка — *неверная запись проверки на принадлежность диапазону*. Например, чтобы проверить условие $0 < x < 1$, нельзя записать его в условном операторе непосредственно, так как будет выполнено сначала сравнение $0 < x$, а его результат (`true` или `false`, преобразованное в `int`) будет сравниваться с 1. Правильный способ записи: `if(0 < x && x < 1)...`

Тип переменных выбирается исходя из их назначения. Координаты выстрела нельзя представить целыми величинами, так как это приведет к потере точности результата, а счетчик очков не имеет смысла описывать как вещественный. Даже такую простую программу можно еще упростить с помощью промежуточной переменной и записи условия в виде двух последовательных, а не вложенных, операторов `if` (обратите внимание, что в первом варианте значение переменной `kol` присваивается ровно один раз, а во втором — от одного до трех раз в зависимости от выполнения условий):

```

#include <iostream.h>
int main(){
    float x, y, temp; int kol;
    cout << "Введите координаты выстрела\n";
    cin >> x >> y;
    temp = x * x + y * y;
    kol = 0;
}

```

```
if (temp < 4 ) kol = 1;  
if (temp < 1 ) kol = 2;  
cout << "\n Очков: " << kol;  
return 0;  
}
```

Если какая-либо переменная используется только внутри условного оператора, рекомендуется объявить ее внутри скобок, например:

```
if (int i = fun(t)) a -= i; else a += i;
```

Объявление переменной в тот момент, когда она требуется, то есть когда ей необходимо присвоить значение, является признаком хорошего стиля и позволяет избежать случайного использования переменной до ее инициализации. Объявлять внутри оператора `if` можно только одну переменную. Область ее видимости начинается в точке объявления и включает обе ветви оператора.

Оператор switch

Оператор `switch` (переключатель) предназначен для разветвления процесса вычислений на несколько направлений. Структурная схема оператора приведена на рис. 1.7. Формат оператора:

```
switch ( выражение ){  
    case константное_выражение_1: [список_операторов_1]  
    case константное_выражение_2: [список_операторов_2]  
    ...  
    case константное_выражение_n: [список_операторов_n]  
    [default: операторы ]  
}
```



Рис. 1.7. Структурная схема оператора switch

Выполнение оператора начинается с вычисления выражения (оно должно быть целочисленным), а затем управление передается первому оператору из списка, помеченного константным выражением, значение которого совпало с вычисленным. После этого, если выход из переключателя явно не указан, последовательно выполняются все остальные ветви.

Выход из переключателя обычно выполняется с помощью операторов `break` или `return`. Оператор `break` выполняет выход из самого внутреннего из объемлющих

его операторов switch, for, while и do. Оператор return выполняет выход из функции, в теле которой он записан.

Все константные выражения должны иметь разные значения, но быть одного и того же целочисленного типа. Несколько меток могут следовать подряд. Если совпадения не произошло, выполняются операторы, расположенные после слова default (а при его отсутствии управление передается следующему за switch оператору).

Пример (программа реализует простейший калькулятор на 4 действия):

```
#include <iostream.h>
int main(){
    int a, b, res;
    char op;
    cout << "\nВведите 1й операнд : ";    cin >> a;
    cout << "\nВведите знак операции : "; cin >> op;
    cout << "\nВведите 2й операнд : ";    cin >> b;
    bool f = true;
    switch (op){
        case '+': res = a + b; break;
        case '-': res = a - b; break;
        case '*': res = a * b; break;
        case '/': res = a / b; break;
        default : cout << "\nНеизвестная операция"; f = false;
    }
    if (f) cout << "\nРезультат : " << res;
    return 0;
}
```

ПРИМЕЧАНИЕ

В случае синтаксической ошибки в слове default сообщение об ошибке не выдается, поскольку компилятор воспримет это слово как допустимую метку оператора.

Операторы цикла

Операторы цикла используются для организации многократно повторяющихся вычислений. Любой цикл состоит из тела цикла, то есть тех операторов, которые выполняются несколько раз, начальных установок, модификации параметров цикла и проверки условия продолжения выполнения цикла (рис. 1.8).

Один проход цикла называется итерацией. Проверка условия выполняется на каждой итерации либо до тела цикла (тогда говорят о цикле с предусловием) либо после тела цикла (цикл с постусловием). Разница между ними состоит в том, что тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Проверка необходимости выполнения цикла с предусловием делается до тела цикла, поэтому возможно, что он не выполнится ни разу.

Пере
прод
ла, и
чика
Нача
стоит
испо
Цикл
прин
этого
редач
менд
Для
while
Цикл
Цикл
и име
wh
Выра
стым
ния в
ла. Е

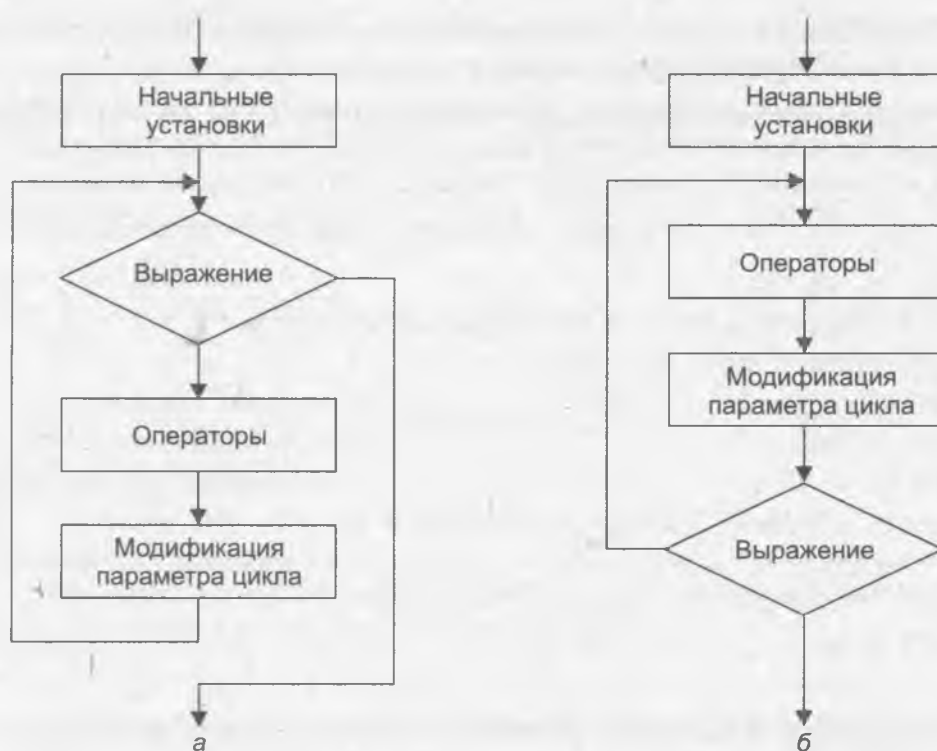


Рис. 1.8. Структурные схемы операторов цикла: а — цикл с предусловием; б — цикл с постусловием

Переменные, изменяющиеся в теле цикла и используемые при проверке условия продолжения, называются параметрами цикла. Целочисленные параметры цикла, изменяющиеся с постоянным шагом на каждой итерации, называются *счетчиками цикла*.

Начальные установки могут явно не присутствовать в программе, их смысл состоит в том, чтобы до входа в цикл задать значения переменным, которые в нем используются.

Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом. Для этого служат операторы `break`, `continue`, `return` и `goto` (см. раздел «Операторы передачи управления», с. 49). Передавать управление извне внутрь цикла не рекомендуется.

Для удобства, а не по необходимости, в C++ есть три разных оператора цикла — `while`, `do while` и `for`.

Цикл с предусловием (`while`)

Цикл с предусловием реализует структурную схему, приведенную на рис. 1.8, а, и имеет вид:

```
while ( выражение ) оператор
```

Выражение определяет условие повторения тела цикла, представленного простым или составным оператором. Выполнение оператора начинается с вычисления выражения. Если оно истинно (не равно `false`), выполняется оператор цикла. Если при первой проверке выражение равно `false`, цикл не выполнится ни

разу. Тип выражения должен быть арифметическим или приводимым к нему. Выражение вычисляется перед каждой итерацией цикла.

Пример (программа печатает таблицу значений функции $y=x^2+1$ во введенном диапазоне):

```
#include <stdio.h>
int main(){
    float Xn, Xk, Dx;
    printf("Введите диапазон и шаг изменения аргумента: ");
    scanf("%f%f%f", &Xn, &Xk, &Dx);
    printf("| X | Y |\n");           // шапка таблицы
    float X = Xn;                  // установка параметра цикла
    while (X <= Xk){               // проверка условия продолжения
        printf("| %5.2f | %5.2f |\n", X, X*X + 1); // тело цикла
        X += Dx;                  // модификация параметра
    }
    return 0;
}
```

Пример (программа находит все делители целого положительного числа):

```
#include <iostream.h>
int main(){
    int num;
    cout << "\nВведите число : "; cin >> num;
    int half = num / 2;           // половина числа
    int div = 2;                  // кандидат на делитель
    while (div <= half){
        if (!(num % div))cout << div << "\n";
        div++;
    }
    return 0;
}
```

Распространенный прием программирования — организация бесконечного цикла с заголовком `while (true)` либо `while (1)` и принудительным выходом из тела цикла по выполнению какого-либо условия.

В круглых скобках после ключевого слова `while` можно вводить описание переменной¹. Областью ее действия является цикл:

```
while (int x = 0){ ... /* область действия x */ }
```

Цикл с постусловием (do while)

Цикл с постусловием реализует структурную схему, приведенную на рис. 1.8, и имеет вид:

```
do оператор while выражение;
```

¹ В старых версиях компиляторов это требование стандарта может не поддерживаться.

Сначала выполняется простой или составной оператор, составляющий тело цикла, а затем вычисляется выражение. Если оно истинно (не равно false), тело цикла выполняется еще раз. Цикл завершается, когда выражение станет равным false или в теле цикла будет выполнен какой-либо оператор передачи управления. Тип выражения должен быть арифметическим или приводимым к нему.

Пример (программа осуществляет проверку ввода):

```
#include <iostream.h>
int main(){
    char answer;
    do{
        cout << "\nКупи слоника! "; cin >> answer;
    }while (answer != 'y');
    return 0;
}
```

Пример. Программа вычисляет квадратный корень вещественного аргумента X с заданной точностью Eps по итерационной формуле:

$$y_n = \frac{1}{2} (y_{n-1} + X/y_{n-1}),$$

где y_{n-1} — предыдущее приближение к корню (в начале вычислений выбирается произвольно), y_n — последующее приближение. Процесс вычислений прекращается, когда приближения станут отличаться друг от друга по абсолютной величине менее, чем на величину заданной точности. Для вычисления абсолютной величины используется стандартная функция `fabs()`, объявление которой находится в заголовочном файле `<math.h>`.

```
#include <stdio.h>
#include <math.h>
int main(){
    double X, Eps;    // аргумент и точность
    double Yp, Y = 1; // предыдущее и последующее приближение
    printf("Введите аргумент и точность: ");
    scanf("%lf%lf", &X, &Eps);
    do{
        Yp = Y;
        Y = (Yp + X/Yp)/2;
    }while (fabs(Y - Yp) >= Eps);
    printf("\nКорень из %lf равен %lf", X, Y);
    return 0;
}
```

Цикл с параметром (for)

Цикл с параметром имеет следующий формат:

`for ( инициализация; выражение; модификации) оператор;`

Инициализация используется для объявления и присвоения начальных значений величинам, используемым в цикле. В этой части можно записать несколько операторов, разделенных запятой (операцией «последовательное выполнение»), например, так.

```
for (int i = 0, j = 2; ...
int k, m;
for (k = 1, m = 0; ...
```

Областью действия переменных, объявленных в части инициализации цикла, является цикл¹. Инициализация выполняется один раз в начале исполнения цикла.

Выражение определяет условие выполнения цикла: если его результат, приведенный к типу `bool`, равен `true`, цикл выполняется. Цикл с параметром реализован как цикл с предусловием.

Модификации выполняются после каждой итерации цикла и служат обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую. Простой или составной *оператор* представляет собой тело цикла. Любая из частей оператора `for` может быть опущена (но точки с запятой надо оставить на своих местах!).

Пример (оператор, вычисляющий сумму чисел от 1 до 100):

```
for (int i = 1, s = 0; i<=100; i++) s += i;
```

Пример (программа печатает таблицу значений функции $y=x^2+1$ во введенном диапазоне):

```
#include <stdio.h>
int main(){
    float Xn, Xk, Dx, X;
    printf("Введите диапазон и шаг изменения аргумента: ");
    scanf("%f%f%f", &Xn, &Xk, &Dx);
    printf("| X | Y |\n");
    for (X = Xn; X<=Xk; X += Dx)
        printf("| %5.2f | %5.2f |\n", X, X*X + 1);
    return 0;
}
```

Пример (программа находит все делители целого положительного числа):

```
#include <iostream.h>
int main(){
    int num, half, div;
    cout << "\nВведите число : "; cin >> num;
    for (half = num / 2, div = 2; div <= half; div++)
        if (!(num % div))cout << div << "\n";
    return 0;
}
```

Два последних примера выполняют те же действия, что и примеры для цикла с предусловием, но записаны более компактно и наглядно: все действия, связанные с управлением циклом, локализованы в его заголовке.

Любой цикл `while` может быть приведен к эквивалентному ему циклу `for` и наоборот по следующей схеме:

¹ В старых версиях компиляторов это требование стандарта может интерпретироваться по-другому.

```
for (b1; b2; b3) оператор b1;
    while (b2){
        оператор; b3;}
```

Часто встречающиеся *ошибки при программировании циклов* — использование в теле цикла неинициализированных переменных и неверная запись условия выхода из цикла.

Чтобы избежать ошибок, рекомендуется:

- ❑ проверить, всем ли переменным, встречающимся в правой части операторов присваивания в теле цикла, присвоены до этого начальные значения (а также возможно ли выполнение других операторов);
- ❑ проверить, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла;
- ❑ предусмотреть аварийный выход из цикла по достижению некоторого количества итераций (см. пример в следующем разделе);
- ❑ и, конечно, не забывать о том, что если в теле цикла требуется выполнить более одного оператора, нужно заключать их в фигурные скобки.

Операторы цикла взаимозаменяемы, но можно привести некоторые *рекомендации* по выбору наилучшего в каждом конкретном случае.

Оператор `do while` обычно используют, когда цикл требуется обязательно выполнить хотя бы раз (например, если в цикле производится ввод данных).

Оператор `for` предпочтительнее в большинстве остальных случаев (однозначно — для организации циклов со счетчиками).

Оператором `while` удобнее пользоваться в случаях, когда число итераций заранее не известно, очевидных параметров цикла нет или модификацию параметров удобнее записывать не в конце тела цикла.

Операторы передачи управления

В C++ есть четыре оператора, изменяющих естественный порядок выполнения вычислений:

- ❑ оператор безусловного перехода `goto`;
- ❑ оператор выхода из цикла `break`;
- ❑ оператор перехода к следующей итерации цикла `continue`;
- ❑ оператор возврата из функции `return`.

Оператор `goto`

Оператор безусловного перехода `goto` имеет формат:

```
goto метка;
```

В теле той же функции должна присутствовать ровно одна конструкция вида:

```
метка: оператор;
```

Оператор `goto` передает управление на помеченный оператор. *Метка* — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан.

Использование оператора безусловного перехода оправдано в двух случаях:

- ❑ принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей;
- ❑ переход из нескольких мест функции в одно (например, если перед выходом из функции необходимо всегда выполнять какие-либо действия).

В остальных случаях для записи любого алгоритма существуют более подходящие средства, а использование `goto` приводит только к усложнению структуры программы и затруднению отладки¹. Применение `goto` нарушает принципы структурного и модульного программирования, по которым все блоки, из которых состоит программа, должны иметь только один вход и один выход.

В любом случае не следует передавать управление внутрь операторов `if`, `switch` и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию переменных, на операторы, расположенные после нее, поскольку в этом случае инициализация не будет выполнена:

```
int k; ...
goto metka; ...
{int a = 3, b = 4;
    k = a + b;
    metka: int m = k + 1; ...
}
```

После выполнения этого фрагмента программы значение переменной `m` не определено.

Оператор `break`

Оператор `break` используется внутри операторов цикла, `if` или `switch` для обеспечения перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится `break`.

Пример. Программа вычисляет значение гиперболического синуса вещественного аргумента `x` с заданной точностью `eps` с помощью разложения в бесконечный ряд.

$$\operatorname{sh} x = x + x^3/3! + x^5/5! + x^7/7! + \dots$$

Вычисление заканчивается, когда абсолютная величина очередного члена ряда прибавляемого к сумме, станет меньше заданной точности.

```
#include <iostream.h>
#include <math.h>
int main(){
```

¹ Даже в приведенных случаях допустимо применять `goto` только в случае, если в этих фрагментах кода не создаются локальные объекты. В противном случае возможно приращение деструктора при пропущенном конструкторе, что приводит к серьезным ошибкам в программе.

Опе

Опер
торы
шей

Опе

Опер
ет уп

ге

Выра
значе

Ука

Ука

Когда
int i=
указа
ни (i)
значе
для x
лями.Итак,
различ
личаю
ся сам
типом

```
const int MaxIter = 500;           // ограничитель количества итераций
double x, eps;
cout << "\nВведите аргумент и точность: ";
cin >> x >> eps;
bool ok = true;                    // признак успешного вычисления
double y = x, ch = x;              // сумма и первый член ряда
for (int n = 0; fabs(ch) > eps; n++) {
    ch *= x * x / (2 * n + 2) / (2 * n + 3); // очередной член ряда
    y += ch;
    if (n > MaxIter){
        ok = false; break;}
}
if (flag) cout << "\nЗначение функции: " << y;
else cout << "\nРяд расходится!";
return 0;
}
```

Оператор continue

Оператор перехода к следующей итерации цикла `continue` пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации.

Оператор return

Оператор возврата из функции `return` завершает выполнение функции и передает управление в точку ее вызова. Вид оператора:

```
return [ выражение ];
```

Выражение должно иметь скалярный тип. Если тип возвращаемого функцией значения описан как `void`, выражение должно отсутствовать.

Указатели и массивы

Указатели

Когда компилятор обрабатывает оператор определения переменной, например, `int i=10;`, он выделяет память в соответствии с типом (`int`) и инициализирует ее указанным значением (10). Все обращения в программе к переменной по ее имени (`i`) заменяются компилятором на адрес области памяти, в которой хранится значение переменной. Программист может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями.

Итак, *указатели предназначены для хранения адресов областей памяти*. В C++ различают три вида указателей — указатели на объект, на функцию и на `void`, отличающиеся свойствами и набором допустимых операций. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим конкретным типом.

Указатель на функцию содержит адрес в сегменте кода, по которому располагается исполняемый код функции, то есть адрес, по которому передается управление при вызове функции. Указатели на функции используются для косвенного вызова функции (не через ее имя, а через обращение к переменной, хранящей ее адрес), а также для передачи имени функции в другую функцию в качестве параметра. *Указатель функции* имеет тип «указатель функции, возвращающей значение заданного типа и имеющей аргументы заданного типа»:

тип (*имя) (список_типов_аргументов);

Например, объявление:

```
int (*fun) (double, double);
```

задает указатель с именем fun на функцию, возвращающую значение типа int и имеющую два аргумента типа double.

Указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа (основного или составного). Простейшее объявление *указателя на объект* (в дальнейшем называемого просто указателем) имеет вид:

тип *имя;

где тип может быть любым, кроме ссылки (о ссылках рассказывается на с. 58) и битового поля (см. с. 69), причем тип может быть к этому моменту только объявлен, но еще не определен (следовательно, в структуре, например, может присутствовать указатель на структуру того же типа).

Звездочка относится непосредственно к имени, поэтому для того, чтобы объявить несколько указателей, требуется ставить ее перед именем каждого из них. Например, в операторе

```
int *a, b, *c;
```

описываются два указателя на целое с именами a и c, а также целая переменная b.

Размер указателя зависит от модели памяти. Можно определить указатель на указатель и т. д.

Указатель на void применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Указателю на void можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом (см. с. 231).

Указатель может быть константой или переменной, а также указывать на константу или переменную. Рассмотрим примеры:

```
int i;                // целая переменная
const int ci = 1;     // целая константа
int * pi;             // указатель на целую переменную
const int * pci;      // указатель на целую константу
int * const cp = &i;  // указатель-константа на целую переменную
const int * const cpc = &ci; // указатель-константа на целую константу
```

Как в
теля и
а const
Для и
Велич
действ

Иници

Указат
мой не
свобод
место
ческой
через у
до кон
два сп
функци
delete.

При оп
есть пр
циализ
мах. И
ках, ли
Сущест
1. При

■ с

i

i

i

■ с

i

■ с

«

i

i

vo

vo

pt

2. Прис

char*

Здесь

дени.

Как видно из примеров, модификатор `const`, находящийся между именем указателя и звездочкой, относится к самому указателю и запрещает его изменение, а `const` слева от звездочки задает постоянство значения, на которое он указывает. Для инициализации указателей использована операция получения адреса `&`.

Величины типа указатель подчиняются общим правилам определения области действия, видимости и времени жизни.

Инициализация указателей

Указатели чаще всего используют при работе с динамической памятью, называемой некоторыми эстетами кучей (перевод с английского языка слова *heap*). Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти, называемым *динамическими переменными*, производится только через указатели. Время жизни динамических переменных — от точки создания до конца программы или до явного освобождения памяти. В C++ используется два способа работы с динамической памятью. Первый использует семейство функций `malloc` и достался в наследство от C, второй использует операции `new` и `delete`.

При определении указателя надо стремиться выполнить его инициализацию, то есть присвоение начального значения. Непреднамеренное использование неинициализированных указателей — распространенный источник ошибок в программах. Инициализатор записывается после имени указателя либо в круглых скобках, либо после знака равенства.

Существуют следующие способы инициализации указателя:

1. Присваивание указателю адреса существующего объекта:

- с помощью операции получения адреса:

```
int a = 5;           // целая переменная
int* p = &a;         // в указатель записывается адрес a
int* p (&a);          // то же самое другим способом
```

- с помощью значения другого инициализированного указателя:

```
int* r = p;
```

- с помощью имени массива или функции, которые трактуются как адрес (см. «Массивы», с. 58, и «Передача имен функций в качестве параметров», с. 80):

```
int b[10];            // массив
int* t = b;           // присваивание адреса начала массива

...

void f(int a){ /* ... */ } // определение функции
void (*pf)(int);          // указатель на функцию
pf = f;                   // присваивание адреса функции
```

2. Присваивание указателю адреса области памяти в явном виде:

```
char* vp = (char *)0xB8000000;
```

Здесь `0xB8000000` — шестнадцатеричная константа, `(char *)` — операция приведения типа: константа преобразуется к типу «указатель на `char`».

3. Присваивание пустого значения:

```
int* suxx = NULL;
int* rulez = 0;
```

В первой строке используется константа `NULL`, определенная в некоторых заголовочных файлах C как указатель, равный нулю. Рекомендуется использовать просто `0`, так как это значение типа `int` будет правильно преобразовано стандартными способами в соответствии с контекстом. Поскольку гарантируется, что объектов с нулевым адресом нет, пустой указатель можно использовать для проверки, ссылается указатель на конкретный объект или нет.

4. Выделение участка динамической памяти и присваивание ее адреса указателю:

- с помощью операции `new`:

```
int* n = new int;           // 1
int* m = new int (10);      // 2
int* q = new int [10];      // 3
```

- с помощью функции `malloc`¹:

```
int* u = (int *)malloc(sizeof(int)); // 4
```

В операторе 1 операция `new` выполняет выделение достаточного для размещения величины типа `int` участка динамической памяти и записывает адрес начала этого участка в переменную `n`. Память под саму переменную `n` (размера, достаточного для размещения указателя) выделяется на этапе компиляции.

В операторе 2, кроме описанных выше действий, производится инициализация выделенной динамической памяти значением `10`.

В операторе 3 операция `new` выполняет выделение памяти под 10 величин типа `int` (массива из 10 элементов) и записывает адрес начала этого участка в переменную `q`, которая может трактоваться как имя массива. Через имя можно обращаться к любому элементу массива. О массивах рассказывается на с. 58.

Если память выделить не удалось, по стандарту должно порождаться исключение `bad_alloc` (об исключениях рассказывается в разделе «Обработка исключительных ситуаций», с. 222, а об обработке ошибок выделения памяти — на с. 378). Старые версии компиляторов могут возвращать `0`.

В операторе 4 делается то же самое, что и в операторе 1, но с помощью функции выделения памяти `malloc`, унаследованной из библиотеки C. В функцию передается один параметр — количество выделяемой памяти в байтах. Конструкция `(int*)` используется для приведения типа указателя, возвращаемого функцией к требуемому типу (о явном преобразовании типов см. с. 231). Если память выделить не удалось, функция возвращает `0`.

Операцию `new` использовать предпочтительнее, чем функцию `malloc`, особенно при работе с объектами.

Освобождение памяти, выделенной с помощью операции `new`, должно выполняться с помощью `delete`, а памяти, выделенной функцией `malloc` — посредством

¹ Для того чтобы использовать `malloc`, требуется подключить к программе заголовочный файл `<malloc.h>`.

функции `free`. При этом переменная-указатель сохраняется и может инициализироваться повторно. Приведенные выше динамические переменные уничтожаются следующим образом:

```
delete n; delete m; delete [] q; free (u);
```

Если память выделялась с помощью `new[]`, для освобождения памяти необходимо применять `delete[]`. Размерность массива при этом не указывается. Если квадратных скобок нет, то никакого сообщения об ошибке не выдается, но помечен как свободный будет только первый элемент массива, а остальные окажутся недоступны для дальнейших операций. Такие ячейки памяти называются мусором.

ВНИМАНИЕ

Если переменная-указатель выходит из области своего действия, отведенная под нее память освобождается. Следовательно, динамическая переменная, на которую ссылался указатель, становится недоступной. При этом память из-под самой динамической переменной не освобождается. Другой случай появления «мусора» — когда инициализированному указателю присваивается значение другого указателя. При этом старое значение бесследно теряется.

С помощью комбинаций звездочек, круглых и квадратных скобок можно описывать составные типы и указатели на составные типы, например, в операторе

```
int *(*p[10])();
```

объявляется массив из 10 указателей на функции без параметров, возвращающих указатели на `int`.

По умолчанию квадратные и круглые скобки имеют одинаковый приоритет, больший, чем звездочка, и рассматриваются слева направо. Для изменения порядка рассмотрения используются круглые скобки.

При интерпретации сложных описаний необходимо придерживаться правила «изнутри наружу»:

- 1) если справа от имени имеются квадратные скобки, это массив, если скобки круглые — это функция;
- 2) если слева есть звездочка, это указатель на проинтерпретированную ранее конструкцию;
- 3) если справа встречается закрывающая круглая скобка, необходимо применить приведенные выше правила внутри скобок, а затем переходить наружу;
- 4) в последнюю очередь интерпретируется спецификатор типа.

Для приведенного выше описания порядок интерпретации указан цифрами:

```
int *(*p[10])();  
5  4 2 1  3      // порядок интерпретации описания
```

Операции с указателями

С указателями можно выполнять следующие операции: разадресация, или косвенное обращение к объекту (*), присваивание, сложение с константой, вычитание, инкремент (++), декремент (--), сравнение, приведение типов. При работе с указателями часто используется операция получения адреса (&).

Операция разадресации, или разыменования, предназначена для доступа к величине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины (если она не объявлена как константа):

```
char a;                // переменная типа char
char * p = new char;   /* выделение памяти под указатель и под динамическую
переменную типа char */
*p = 'Ю'; a = *p;      // присваивание значения обоим переменным
```

Как видно из примера, конструкцию `*имя_указателя` можно использовать в левой части оператора присваивания, так как она является L-значением (см. с. 33), то есть определяет адрес области памяти. Для простоты эту конструкцию можно считать именем переменной, на которую ссылается указатель. С ней допустимы все действия, определенные для величин соответствующего типа (если указатель инициализирован). На одну и ту же область памяти может ссылаться несколько указателей различного типа. Примененная к ним операция разадресации даст разные результаты. Например, программа

```
#include <stdio.h>
int main(){
    unsigned long int A = 0xcc77ffaa;
    unsigned short int* pint = (unsigned short int*) &A;
    unsigned char* pchar = (unsigned char *) &A;
    printf(" | %x | %x | %x |", A, *pint, *pchar);
    return 0;
}
```

на IBM PC выведет на экран строку¹:

```
| cc77ffaa | ffaa | aa |
```

Значения указателей `pint` и `pchar` одинаковы, но разадресация `pchar` дает в результате один младший байт по этому адресу, а `pint` — два младших байта.

В приведенном выше примере при инициализации указателей были использованы *операции приведения типов*. Синтаксис операции явного приведения типа прост: перед именем переменной в скобках указывается тип, к которому ее требуется преобразовать. При этом не гарантируется сохранение информации, поэтому в общем случае явных преобразований типа следует избегать. Подробнее операция приведения типов рассмотрена на с. 231.

При смешивании в выражении указателей разных типов явное преобразование типов требуется для всех указателей, кроме `void*`. Указатель может неявно преобразовываться в значение типа `bool` (например, в выражении условного оператора), при этом ненулевой указатель преобразуется в `true`, а нулевой в `false`.

Присваивание без явного приведения типов допускается в двух случаях:

- ❑ указателям типа `void*`;
- ❑ если тип указателей справа и слева от операции присваивания один и тот же.

¹ Старые версии компилятора могут выдать в результате строку `| ffaa | cc77 | ffaa |`

Таким образом, неявное преобразование выполняется только к типу `void*`. Значение 0 неявно преобразуется к указателю на любой тип. Присваивание указателей на объекты указателям на функции (и наоборот) недопустимо. Запрещено и присваивать значения указателям-константам, впрочем, как и константам любого типа (присваивать значения указателям на константу и переменным, на которые ссылается указатель-константа, допускается).

Арифметические операции с указателями (сложение с константой, вычитание, инкремент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, последовательно размещенными в памяти, например, с массивами.

Инкремент перемещает указатель к следующему элементу массива, *декремент* — к предыдущему. Фактически значение указателя изменяется на величину `sizeof(тип)`. Если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа, например:

```
short * p = new short [5];  
p++;           // значение p увеличивается на 2  
long * q = new long [5];  
q++;           // значение q увеличивается на 4
```

Разность двух указателей — это разность их значений, деленная на размер типа в байтах (в применении к массивам разность указателей, например, на третий и шестой элементы равна 3). Суммирование двух указателей не допускается.

При записи выражений с указателями следует обращать внимание на приоритеты операций. В качестве примера рассмотрим последовательность действий, заданную в операторе

```
*p++ = 10;
```

Операции разадресации и инкремента имеют одинаковый приоритет и выполняются справа налево, но, поскольку инкремент постфиксный, он выполняется после выполнения операции присваивания. Таким образом, сначала по адресу, записанному в указателе `p`, будет записано значение 10, а затем указатель будет увеличен на количество байт, соответствующее его типу. То же самое можно записать подробнее:

```
*p = 10; p++;
```

Выражение `(*p)++`, напротив, инкрементирует значение, на которое ссылается указатель.

Унарная операция получения адреса & применима к величинам, имеющим имя и размещенным в оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной. Примеры операции приводились выше.

Ссылки

Ссылка представляет собой *синоним имени*, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

тип & имя;

где тип — это тип величины, на которую указывает ссылка, & — оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа, например:

```
int kol;  
int& pal = kol;           // ссылка pal – альтернативное имя для kol  
const char& CR = '\n';    // ссылка на константу
```

Запомните следующие правила.

- ❑ Переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции (см. с. 77), описана как `extern` или ссылается на поле данных класса (см. с. 178).
- ❑ После инициализации ссылке не может быть присвоена другая переменная.
- ❑ Тип ссылки должен совпадать с типом величины, на которую она ссылается.
- ❑ Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений. Ссылки позволяют использовать в функциях переменные, передаваемые по адресу, без операции разадресации, что улучшает читаемость программы (подробнее об этом рассказывается в разделе «Параметры функции», с. 77).

Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

Массивы

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой величин одинакового типа требуется выполнять однообразные действия, им дают одно имя, а различают по порядковому номеру. Это позволяет компактно записывать множество операций с помощью циклов. Конечная именованная последовательность однотипных величин называется *массивом*. Описание массива в программе отличается от описания простой переменной наличием после имени квадратных скобок, в которых задается количество элементов массива (*размерность*):

```
float a [10]; // описание массива из 10 вещественных чисел
```

ВНИМАНИЕ

При описании массивов квадратные скобки являются элементом синтаксиса, а не указанием на необязательность конструкции.

Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, `const` и инициализатор), что и для простых переменных. Инициализирующие значения для массивов записываются в фигурных скобках. Значения элементам присваиваются по порядку. Если элементов в массиве больше, чем инициализаторов, элементы, для которых значения не указаны, обнуляются:

```
int b[5] = {3, 2, 1}; // b[0]=3, b[1]=2, b[2]=1, b[3]=0, b[4]=0
```

Размерность массива вместе с типом его элементов определяет объем памяти, необходимый для размещения массива, которое выполняется на этапе компиляции, поэтому размерность может быть задана только целой положительной *константой или константным выражением*. Если при описании массива не указана размерность, должен присутствовать инициализатор, в этом случае компилятор выделяет память по количеству инициализирующих значений. В дальнейшем мы увидим, что размерность может быть опущена также в списке формальных параметров (см. раздел «Параметры функции», с. 77).

Для доступа к элементу массива после его имени указывается номер элемента (*индекс*) в квадратных скобках. В следующем примере подсчитывается сумма элементов массива.

```
#include <iostream.h>
int main(){
    const int n = 10;
    int i, sum;
    int marks[n] = {3, 4, 5, 4, 4};
    for (i = 0, sum = 0; i < n; i++) sum += marks[i];
    cout << "Сумма элементов: " << sum;
    return 0;
}
```

Размерность массивов предпочтительнее задавать с помощью *именованных констант*, как это сделано в примере, поскольку при таком подходе для ее изменения достаточно скорректировать значение константы всего лишь в одном месте программы. Обратите внимание, что последний элемент массива имеет номер, на единицу меньший заданной при его описании размерности.

ВНИМАНИЕ

При обращении к элементам массива автоматический контроль выхода индекса за границу массива не производится, что может привести к ошибкам.

Пример. Сортировка целочисленного массива *методом выбора*. Алгоритм состоит в том, что выбирается наименьший элемент и меняется местами с первым элементом массива, затем рассматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым элементом, и так далее $n-1$ раз (при последнем проходе цикла при необходимости меняются местами предпоследний и последний элементы массива).

```
#include <iostream.h>
int main(){
```

```

const int n = 20;           // количество элементов массива
int b[n];                   // описание массива
int i;
for (i = 0; i < n; i++) cin >> b[i]; // ввод массива
for (i = 0; i < n-1; i++){    // n-1 раз ищем наименьший элемент
    // принимаем за наименьший первый из рассматриваемых элементов:
    int imin = i;
    // поиск номера минимального элемента из неупорядоченных:
    for (int j = i + 1; j < n; j++)
        // если нашли меньший элемент, запоминаем его номер:
        if (b[j] < b[imin]) imin = j;
    int a = b[i];           // обмен элементов
    b[i] = b[imin];         // с номерами
    b[imin] = a;           // i и imin
}
// вывод упорядоченного массива:
for (i = 0; i < n; i++) cout << b[i] << ' ';
return 0;
}

```

Процесс обмена элементов массива с номерами i и $imin$ через буферную переменную a на i -м проходе цикла проиллюстрирован на рис. 1.9. Цифры около стрелок обозначают порядок действий.

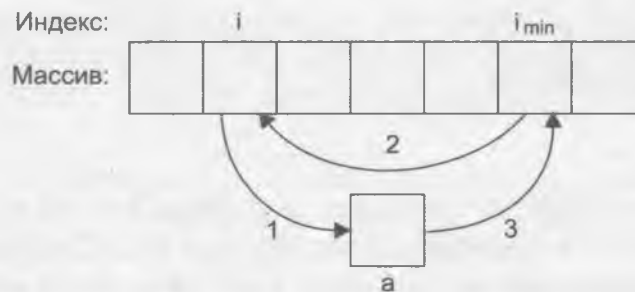


Рис. 1.9. Обмен значений двух переменных

Идентификатор массива является константным указателем на его нулевой элемент. Например, для массива из предыдущего листинга имя b — это то же самое, что $\&b[0]$, а к i -му элементу массива можно обратиться, используя выражение $*(b+i)$. Можно описать указатель, присвоить ему адрес начала массива и работать с массивом через указатель. Следующий фрагмент программы копирует все элементы массива a в массив b :

```

int a[100], b[100];
int *pa = a;           // или int *p = &a[0];
int *pb = b;
for(int i = 0; i < 100; i++)
    *pb++ = *pa++; // или pb[i] = pa[i];

```

Динамические массивы создают с помощью операции `new`, при этом необходимо указать тип и размерность, например:

```
int n = 100;
float *p = new float [n];
```

В этой строке создается переменная-указатель на `float`, в динамической памяти отводится непрерывная область, достаточная для размещения 100 элементов вещественного типа, и адрес ее начала записывается в указатель `p`. Динамические массивы нельзя при создании инициализировать, и они не обнуляются.

Преимущество динамических массивов состоит в том, что размерность может быть переменной, то есть объем памяти, выделяемой под массив, определяется на этапе выполнения программы. Доступ к элементам динамического массива осуществляется точно так же, как к статическим, например, к элементу номер 5 приведенного выше массива можно обратиться как `p[5]` или `*(p+5)`.

Альтернативный способ создания динамического массива — использование функции `malloc` библиотеки C:

```
int n = 100;
float *q = (float *) malloc(n * sizeof(float));
```

Операция преобразования типа, записанная перед обращением к функции `malloc`, требуется потому, что функция возвращает значение указателя типа `void*`, а инициализируется указатель на `float`.

Память, зарезервированная под динамический массив с помощью `new []`, должна освободиться операцией `delete []`, а память, выделенная функцией `malloc` — посредством функции `free`, например:

```
delete [] p; free (q);
```

При несоответствии способов выделения и освобождения памяти результат не определен. Размерность массива в операции `delete` не указывается, но квадратные скобки обязательны.

Многомерные массивы задаются указанием каждого измерения в квадратных скобках, например, оператор

```
int matr [6][8];
```

задает описание двумерного массива из 6 строк и 8 столбцов. В памяти такой массив располагается в последовательных ячейках *построчно*. Многомерные массивы размещаются так, что при переходе к следующему элементу быстрее всего изменяется последний индекс. Для *доступа* к элементу многомерного массива указываются все его индексы, например, `matr[i][j]`, или более экзотическим способом: `*(matr[i]+j)` или `*(*(matr+i)+j)`. Это возможно, поскольку `matr[i]` является адресом начала *i*-й строки массива.

При *инициализации* многомерного массива он представляется либо как массив из массивов, при этом каждый массив заключается в свои фигурные скобки (в этом случае левую размерность при описании можно не указывать), либо задается общий список элементов в том порядке, в котором элементы располагаются в памяти:

```
int mass2 [][]={ {1, 1}, {0, 2}, {1, 0} };
int mass2 [3][2]={1, 1, 0, 2, 1, 0};
```

Пример. Программа определяет в целочисленной матрице номер строки, которая содержит наибольшее количество элементов, равных нулю.

```

#include <stdio.h>
int main(){
    const int nstr = 4, nstb = 5;           // размерности массива
    int b[nstr][nstb];                     // описание массива
    int i, j;
    for (i = 0; i < nstr; i++)              // ввод массива
        for (j = 0; j < nstb; j++) scanf("%d", &b[i][j]);
    int istr = -1, MaxKol = 0;
    for (i = 0; i < nstr; i++){             // просмотр массива по строкам
        int Kol = 0;
        for (j = 0; j < nstb; j++) if (b[i][j] == 0) Kol++;
        if (Kol > MaxKol){istr = i; MaxKol = Kol;}
    }
    printf(" Исходный массив:\n");
    for (i = 0; i < nstr; i++){
        for (j = 0; j < nstb; j++) printf("%d ", b[i][j]);
        printf("\n");
    }
    if (istr == -1) printf("Нулевых элементов нет");
    else printf("Номер строки: %d", istr);
    return 0;}

```

Номер искомой строки хранится в переменной `istr`, количество нулевых элементов в текущей (*i*-й) строке — в переменной `Kol`, максимальное количество нулевых элементов — в переменной `MaxKol`. Массив просматривается по строкам, в каждой из них подсчитывается количество нулевых элементов (обратите внимание, что переменная `Kol` обнуляется перед просмотром каждой строки). Наибольшее количество и номер соответствующей строки запоминаются.

Для создания динамического многомерного массива необходимо указать в операции `new` все его размерности (самая левая размерность может быть переменной), например:

```

int nstr = 5;
int ** m = (int **) new int [nstr][10];

```

Более универсальный и безопасный способ выделения памяти под двумерный массив, когда обе его размерности задаются на этапе выполнения программы, приведен ниже:

```

int nstr, nstb;
cout << " Введите количество строк и столбцов :";
cin >> nstr >> nstb;
int **a = new int *[nstr];           // 1
for(int i = 0; i < nstr; i++)         // 2
    a[i] = new int [nstb];           // 3

```

В операторе 1 объявляется переменная типа «указатель на указатель на `int`» и выделяется память под массив указателей на строки массива (количество строк — `nstr`). В операторе 2 организуется цикл для выделения памяти под каждую строку массива. В операторе 3 каждому элементу массива указателей на строки присваи-

вается адрес начала участка памяти, выделенного под строку двумерного массива. Каждая строка состоит из *nstb* элементов типа *int* (рис. 1.10).

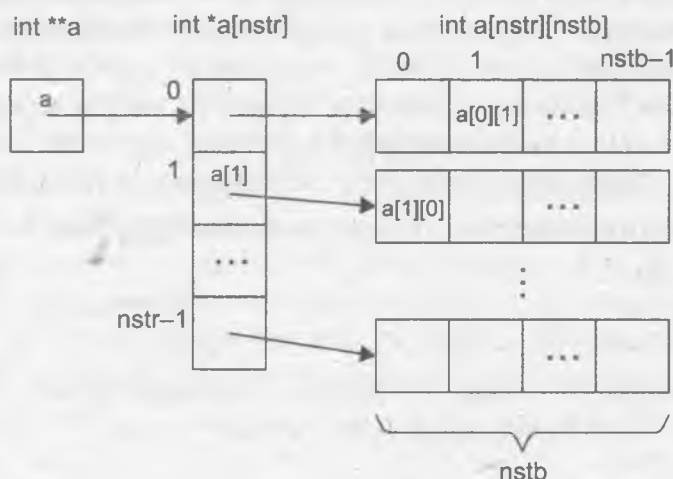


Рис. 1.10. Выделение памяти под двумерный массив

Освобождение памяти из-под массива с любым количеством измерений выполняется с помощью операции `delete []`. Указатель на константу удалить нельзя.

ПРИМЕЧАНИЕ

Для правильной интерпретации объявлений полезно запомнить мнемоническое правило: «суффикс прежде префикса». Если при описании переменной используются одновременно префикс `*` (указатель) и суффикс `[]` (массив), то переменная интерпретируется как массив указателей, а не указатель на массив: `int *p[10]`; — массив из 10 указателей на `int`.

Строки

Строка представляет собой массив символов, заканчивающийся нуль-символом. Нуль-символ — это символ с кодом, равным 0, что записывается в виде управляющей последовательности `'\0'`. По положению нуль-символа определяется фактическая длина строки. Строку можно инициализировать строковым литералом (см. с. 20):

```
char str[10] = "Vasia";
// выделено 10 элементов с номерами от 0 до 9
// первые элементы — 'V', 'a', 's', 'i', 'a', '\0'
```

В этом примере под строку выделяется 10 байт, 5 из которых занято под символы строки, а шестой — под нуль-символ. Если строка при определении инициализируется, ее размерность можно опускать (компилятор сам выделит соответствующее количество байт):

```
char str[] = "Vasia"; // выделено и заполнено 6 байт
```

Оператор

```
char *str = "Vasia"
```

создает не строковую переменную, а указатель на строковую константу, изменить которую невозможно (к примеру, оператор `str[1]='o'` не допускается). Знак

равенства перед строковым литералом означает инициализацию, а не присваивание. Операция присваивания одной строки другой не определена (поскольку строка является массивом) и может выполняться с помощью цикла или функций стандартной библиотеки. Библиотека предоставляет возможности копирования, сравнения, объединения строк, поиска подстроки, определения длины строки и т. д. (возможности библиотеки описаны в разделе «Функции работы со строками и символами», с. 91, и в приложении 6), а также содержит специальные функции ввода строк и отдельных символов с клавиатуры и из файла.

Пример. Программа запрашивает пароль не более трех раз.

```
#include <stdio.h>
#include <string.h>
int main(){
    char s[5], passw[] = "kuku"; // passw - эталонный пароль.
    // Можно описать как *passw = "kuku";
    int i, k = 0;
    for (i = 0; !k && i<3; i++){
        printf("\nвведите пароль:\n");
        gets(s); // функция ввода строки
        if (strstr(s,passw))k = 1; // функция сравнения строк
    }
    if (k) printf("\nпароль принят");
    else printf("\nпароль не принят");
    return 0;
}
```

При работе со строками часто используются указатели.

ВНИМАНИЕ

Распространенные ошибки при работе со строками — отсутствие нуль-символа и выход указателя при просмотре строки за ее пределы.

Рассмотрим процесс копирования строки *src* в строку *dest*. Очевидный алгоритм имеет вид:

```
char src[10], dest[10];
...
for (int i = 0; i<=strlen(src); i++) dest[i] = src[i];
```

Длина строки определяется с помощью функции *strlen*, которая вычисляет длину, выполняя поиск нуль-символа. Таким образом, строка фактически просматривается дважды. Более эффективным будет использовать проверку на нуль-символ непосредственно в программе. Увеличение индекса можно заменить инкрементом указателей (для этого память под строку *src* должна выделяться динамически, а также требуется определить дополнительный указатель и инициализировать его адресом начала строки *dest*):

```
#include <iostream.h>
int main(){
    char *src = new char [10];
```

```
char *dest = new char [10]. *d = dest;  
cin >> src;  
while ( *src != 0) *d++ = *src++;  
*d = 0;      // завершающий ноль  
cout << dest;  
return 0;  
}
```

В цикле производится посимвольное присваивание элементов строк с одновременной инкрементацией указателей. Результат операции присваивания — передаваемое значение, которое, собственно, и проверяется в условии цикла, поэтому можно поставить присваивание на место условия, а проверку на неравенство нулю опустить (при этом завершающий ноль копируется в цикле, и отдельного оператора для его присваивания не требуется). В результате цикл копирования строки принимает вид:

```
while ( *d++ = *src++);
```

Оба способа работы со строками (через массивы или указатели) приемлемы и имеют свои плюсы и минусы, но в общем случае лучше не изобретать велосипед, а пользоваться функциями библиотеки или определенным в стандартной библиотеке C++ классом `string`, который обеспечивает индексацию, присваивание, сравнение, добавление, объединение строк и поиск подстрок, а также преобразование из C-строк, то есть массивов типа `char`, в `string`, и наоборот (строки C++ описаны в разделе «Строки» на с. 286).

ПРИМЕЧАНИЕ

Строки C++ не обязательно должны состоять из символов типа `char`. Это позволяет использовать любой набор символов (не только ASCII), но для произвольного набора следует определить собственную операцию копирования, что может снизить эффективность работы.

Типы данных, определяемые пользователем

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе простых типов данных, массивов и указателей. Язык C++ позволяет программисту определять свои типы данных и правила работы с ними. Исторически для таких типов сложилось наименование, вынесенное в название главы, хотя правильное было бы назвать их типами, определяемыми программистом.

Переименование типов (`typedef`)

Для того чтобы сделать программу более ясной, можно задать типу новое имя с помощью ключевого слова `typedef`:

```
typedef тип новое_имя [ размерность ];
```

```
char *dest = new char [10]. *d = dest;  
cin >> src;  
while ( *src != 0) *d++ = *src++;  
*d = 0;      // завершающий ноль  
cout << dest;  
return 0;  
}
```

В цикле производится посимвольное присваивание элементов строк с одновременной инкрементацией указателей. Результат операции присваивания — передаваемое значение, которое, собственно, и проверяется в условии цикла, поэтому можно поставить присваивание на место условия, а проверку на неравенство нулю опустить (при этом завершающий ноль копируется в цикле, и отдельного оператора для его присваивания не требуется). В результате цикл копирования строки принимает вид:

```
while ( *d++ = *src++);
```

Оба способа работы со строками (через массивы или указатели) приемлемы и имеют свои плюсы и минусы, но в общем случае лучше не изобретать велосипед, а пользоваться функциями библиотеки или определенным в стандартной библиотеке C++ классом `string`, который обеспечивает индексацию, присваивание, сравнение, добавление, объединение строк и поиск подстрок, а также преобразование из C-строк, то есть массивов типа `char`, в `string`, и наоборот (строки C++ описаны в разделе «Строки» на с. 286).

ПРИМЕЧАНИЕ

Строки C++ не обязательно должны состоять из символов типа `char`. Это позволяет использовать любой набор символов (не только ASCII), но для произвольного набора следует определить собственную операцию копирования, что может снизить эффективность работы.

Типы данных, определяемые пользователем

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе простых типов данных, массивов и указателей. Язык C++ позволяет программисту определять свои типы данных и правила работы с ними. Исторически для таких типов сложилось наименование, вынесенное в название главы, хотя правильнее было бы назвать их типами, определяемыми программистом.

Переименование типов (`typedef`)

Для того чтобы сделать программу более ясной, можно задать типу новое имя с помощью ключевого слова `typedef`:

```
typedef тип новое_имя [ размерность ];
```

В данном случае квадратные скобки являются элементом синтаксиса. Размерность может отсутствовать. Примеры:

```
typedef unsigned int UINT;
typedef char Msg[100];
typedef struct{
    char fio[30];
    int date, code;
    double salary;} Worker;
```

Введенное таким образом имя можно использовать таким же образом, как и имена стандартных типов:

```
UINT i, j;           // две переменных типа unsigned int
Msg str[10];         // массив из 10 строк по 100 символов
Worker stuff[100];   // массив из 100 структур
```

Кроме задания типам с длинными описаниями более коротких псевдонимов `typedef` используется для облегчения переносимости программ: если машинно-зависимые типы объявить с помощью операторов `typedef`, при переносе программы потребуется внести изменения только в эти операторы.

Перечисления (enum)

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть не важны). Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант. Формат:

```
enum [ имя_типа ] { список_констант };
```

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

```
enum Err {ERR_READ, ERR_WRITE, ERR_CONVERT};
Err error;

switch (error){
    case ERR_READ:      /* операторы */ break;
    case ERR_WRITE:     /* операторы */ break;
    case ERR_CONVERT:   /* операторы */ break;
}
```

Константам `ERR_READ`, `ERR_WRITE`, `ERR_CONVERT` присваиваются значения 0, 1 и 2 соответственно.

Другой пример:

```
enum {two = 2, three, four, ten = 10, eleven, fifty = ten + 40};
```

В данном случае квадратные скобки являются элементом синтаксиса. Размерность может отсутствовать. Примеры:

```
typedef unsigned int UINT;
typedef char Msg[100];
typedef struct{
    char fio[30];
    int date, code;
    double salary;} Worker;
```

Введенное таким образом имя можно использовать таким же образом, как и имена стандартных типов:

```
UINT i, j;           // две переменных типа unsigned int
Msg str[10];         // массив из 10 строк по 100 символов
Worker stuff[100];   // массив из 100 структур
```

Кроме задания типам с длинными описаниями более коротких псевдонимов `typedef` используется для облегчения переносимости программ: если машинно-зависимые типы объявить с помощью операторов `typedef`, при переносе программы потребуется внести изменения только в эти операторы.

Перечисления (enum)

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть не важны). Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант. Формат:

```
enum [ имя_типа ] { список_констант };
```

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

```
enum Err {ERR_READ, ERR_WRITE, ERR_CONVERT};
Err error;

switch (error){
    case ERR_READ:      /* операторы */ break;
    case ERR_WRITE:     /* операторы */ break;
    case ERR_CONVERT:   /* операторы */ break;
}
```

Константам `ERR_READ`, `ERR_WRITE`, `ERR_CONVERT` присваиваются значения 0, 1 и 2 соответственно.

Другой пример:

```
enum {two = 2, three, four, ten = 10, eleven, fifty = ten + 40};
```

В данном случае квадратные скобки являются элементом синтаксиса. Размерность может отсутствовать. Примеры:

```
typedef unsigned int UINT;
typedef char Msg[100];
typedef struct{
    char fio[30];
    int date, code;
    double salary;} Worker;
```

Введенное таким образом имя можно использовать таким же образом, как и имена стандартных типов:

```
UINT i, j;           // две переменных типа unsigned int
Msg str[10];         // массив из 10 строк по 100 символов
Worker stuff[100];   // массив из 100 структур
```

Кроме задания типам с длинными описаниями более коротких псевдонимов, `typedef` используется для облегчения переносимости программ: если машинно-зависимые типы объявить с помощью операторов `typedef`, при переносе программы потребуется внести изменения только в эти операторы.

Перечисления (enum)

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть не важны). Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант. Формат:

```
enum [ имя_типа ] { список_констант };
```

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

```
enum Err {ERR_READ, ERR_WRITE, ERR_CONVERT};
Err error;

switch (error){
    case ERR_READ:      /* операторы */ break;
    case ERR_WRITE:     /* операторы */ break;
    case ERR_CONVERT:   /* операторы */ break;
}
```

Константам `ERR_READ`, `ERR_WRITE`, `ERR_CONVERT` присваиваются значения 0, 1 и 2 соответственно.

Другой пример:

```
enum {two = 2, three, four, ten = 10, eleven, fifty = ten + 40};
```


Константам `three` и `four` присваиваются значения 3 и 4, константе `eleven` — 11.

Имена перечисляемых констант должны быть уникальными, а значения могут совпадать. Преимущество применения перечисления перед описанием именованных констант и директивой `#define` (см. раздел «Директива `#define`», с. 94) состоит в том, что связанные константы нагляднее; кроме того, компилятор при инициализации констант может выполнять проверку типов.

При выполнении арифметических операций перечисления преобразуются в целые. Поскольку перечисления являются типами, определяемыми пользователем, для них можно вводить собственные операции (см. «Перегрузка операций», с. 189).

ПРИМЕЧАНИЕ

Диапазон значений перечисления определяется количеством бит, необходимым для представления всех его значений. Любое значение целочисленного типа можно явно привести к типу перечисления, но при выходе за пределы его диапазона результат не определен.

Структуры (`struct`)

В отличие от массива, все элементы которого однотипны, структура может содержать элементы разных типов. В языке C++ структура является видом класса и обладает всеми его свойствами, но во многих случаях достаточно использовать структуры так, как они определены в языке C:

```
struct [ имя_типа ] {  
    тип_1 элемент_1;  
    тип_2 элемент_2;  
    ...  
    тип_n элемент_n;  
} [ список_описателей ];
```

Элементы структуры называются *полями структуры* и могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него. Если отсутствует имя типа, должен быть указан список описателей переменных, указателей или массивов. В этом случае описание структуры служит определением элементов этого списка:

```
// Определение массива структур и указателя на структуру:  
struct {  
    char fio[30];  
    int date, code;  
    double salary;  
}stuff[100]. *ps;
```

Если список отсутствует, описание структуры определяет новый тип, имя которого можно использовать в дальнейшем наряду со стандартными типами, например:

```
struct Worker{           // описание нового типа Worker  
    char fio[30];  
    int date, code;
```

```

    double salary;
}; // описание заканчивается точкой с запятой
// определение массива типа Worker и указателя на тип Worker:
Worker stuff[100], *ps;

```

Имя структуры можно использовать сразу после его объявления (определение можно дать позднее) в тех случаях, когда компилятору не требуется знать размер структуры, например:

```

struct List:           // объявление структуры List
struct Link{
    List *p;           // указатель на структуру List
    Link *prev, *succ: // указатели на структуру Link
};
struct List { /* определение структуры List */};

```

Это позволяет создавать связанные списки структур.

Для инициализации структуры значения ее элементов перечисляют в фигурных скобках в порядке их описания:

```

struct{
    char fio[30];
    int date, code;
    double salary;
}worker = {"Страусенко", 31, 215, 3400.55};

```

При инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива (учитывая, что многомерный массив — это массив массивов):

```

struct complex{
    float real, im;
} compl [2][3] = {
    {{1. 1}, {1. 1}, {1. 1}}. // строка 1. то есть массив compl[0]
    {{2. 2}, {2. 2}, {2. 2}} // строка 2. то есть массив compl[1]
};

```

Для переменных одного и того же структурного типа определена операция присваивания, при этом происходит поэлементное копирование. Структуру можно передавать в функцию и возвращать в качестве значения функции. Другие операции со структурами могут быть определены пользователем (см. «Перегрузка операций», с. 189). Размер структуры не обязательно равен сумме размеров ее элементов, поскольку они могут быть выровнены по границам слова.

Доступ к полям структуры выполняется с помощью операций выбора . (точка) при обращении к полю через имя структуры и -> при обращении через указатель, например:

```

Worker worker, stuff[100], *ps;
j
worker.fio = "Страусенко";
stuff[8].code = 215;
ps->salary = 0.12;

```

Если элементом структуры является другая структура, то доступ к ее элементам выполняется через две операции выбора:

```
struct A {int a; double x;};  
struct B {A a; double x;} x[2];  
x[0].a.a = 1;  
x[1].x = 0.1;
```

Как видно из примера, поля разных структур могут иметь одинаковые имена, поскольку у них разная область видимости. Более того, можно объявлять в одной области видимости структуру и другой объект (например, переменную или массив) с одинаковыми именами, если при определении структурной переменной использовать слово `struct`, но не советую это делать — запутать компилятор труднее, чем себя.

Битовые поля

Битовые поля — это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «да/нет». Минимальная адресуемая ячейка памяти — 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа):

```
struct Options{  
    bool centerX:1;  
    bool centerY:1;  
    unsigned int shadow:2;  
    unsigned int palette:4;  
};
```

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом — по имени. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры. Следует учитывать, что операции с отдельными битами реализуются гораздо менее эффективно, чем с байтами и словами, так как компилятор должен генерировать специальные коды, и экономия памяти под переменные оборачивается увеличением объема кода программы. Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

Объединения (union)

Объединение (`union`) представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания такой же, как у структуры, только вместо ключевого слова `struct` используется слово `union`. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в переменной типа объединение хранится только одно значение, и ответственность за его правильное использование лежит на программисте.

Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно не требуется:

```

#include <iostream.h>
int main(){
    enum paytype {CARD, CHECK};
    paytype ptype;
    union payment{
        char card[25];
        long check;
    } info;
    /* присваивание значений info и ptype */
    switch (ptype){
        case CARD: cout << "Оплата по карте: " << info.card; break;
        case CHECK: cout << "Оплата чеком: " << info.check; break;
    }
    return 0;
}

```

Объединение часто используют в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Имя объединения можно не указывать, что позволяет обращаться к его полям непосредственно:

```

#include <iostream.h>
int main(){
    enum paytype {CARD, CHECK};
    struct{
        paytype ptype;
        union{
            char card[25];
            long check;
        };
    } info;
    ... /* присваивание значения info */
    switch (info.ptype){
        case CARD: cout << "Оплата по карте: " << info.card; break;
        case CHECK: cout << "Оплата чеком: " << info.check; break;
    }
    return 0;
}

```

Объединения применяются также для разной интерпретации одного и того же битового представления (но, как правило, в этом случае лучше использовать явные операции преобразования типов). В качестве примера рассмотрим работу со структурой, содержащей битовые поля:

```

struct Options{
    bool centerX:1;
    bool centerY:1;
    unsigned int shadow:2;
    unsigned int palette:4;
};

```

```
union{
    unsigned char ch;
    Options bit;
}option = {0xC4};
cout << option.bit.palette;
option.ch &= 0xF0; // наложение маски
```

По сравнению со структурами на объединения налагаются некоторые ограничения. Смысл некоторых из них станет понятен позже:

- ☐ объединение может инициализироваться только значением его первого элемента;
- ☐ объединение не может содержать битовые поля;
- ☐ объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания;
- ☐ объединение не может входить в иерархию классов.

```
union{
    unsigned char ch;
    Options bit;
}option = {0xC4};
cout << option.bit.palette;
option.ch &= 0xF0; // наложение маски
```

По сравнению со структурами на объединения налагаются некоторые ограничения. Смысл некоторых из них станет понятен позже:

- ❑ объединение может инициализироваться только значением его первого элемента;
- ❑ объединение не может содержать битовые поля;
- ❑ объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания;
- ❑ объединение не может входить в иерархию классов.

ГЛАВА 2

Модульное программирование

Модульность — фундаментальный аспект всех успешно работающих крупных систем.

Б. Страуструп

С увеличением объема программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является ее разбиение на части. В C++ задача может быть разделена на более простые и обозримые с помощью *функций*, после чего программу можно рассматривать в более укрупненном виде — на уровне взаимодействия функций. Это важно, поскольку человек способен помнить ограниченное количество фактов. Использование функций является первым шагом к повышению степени абстракции программы и ведет к упрощению ее структуры.

Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Процесс отладки программы, содержащей функции, можно лучше структурировать. Часто используемые функции можно помещать в библиотеки. Таким образом создаются более простые в отладке и сопровождении программы.

Следующим шагом в повышении уровня абстракции программы является группировка функций и связанных с ними данных в отдельные файлы (модули), компилируемые раздельно. Получившиеся в результате компиляции объектные модули объединяются в исполняемую программу с помощью компоновщика. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля и позволяя отлаживать программу по частям (или разными программистами).

Модуль содержит данные и функции их обработки. Другим модулям нежелательно иметь собственные средства обработки этих данных, они должны пользоваться для этого функциями первого модуля. Для того чтобы использовать модуль, нужно знать только его интерфейс, а не все детали его реализации. Чем

более независимы модули, тем меньший объем информации требуется. Разделение программы на модули — это способ решения задачи, которая должна

Скрытие деталей реализации — ключевой идеей как структурирования. Пример инкапсуляции: передача всех необходимых параметров такой функции, требующей возврата (имя, тип возвращаемого модуля являются заголовки в переменных и констант. Функции модулей программы должны (Модульность в языке C++ по пространствам имен, классов и т.д. говоря, раздельная компиляция его реализации).

Функции

Объявление и определение

Функция — это именованная последовательность каких-либо действий, выполняемых и возвращающих значение.

Любая программа на C++ состоит из `main` (с нее начинается выполнение в момент вызова). Любая программа должна иметь хотя бы одну функцию. Как и для других величин, объявление функции для того, чтобы компилятор мог найти ее.

Объявление функции (прототип) — это объявление имени функции, списка параметров, возвращаемого значения и список параметров. Кроме объявления, объявление функции включает в себя объявление операторов и описаний.

```
[ класс ] тип имя ([ список_
{ тело функции }
```

Рассмотрим составные части объявления функции.

- С помощью необязательного ключа `extern` объявление функции, исполняемой в другом модуле.

ГЛАВА 2

Модульное программирование

*Модульность — фундаментальный аспект
всех успешно работающих крупных систем.*

Б. Страуструп

С увеличением объема программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является ее разбиение на части. В С++ задача может быть разделена на более простые и обозримые с помощью *функций*, после чего программу можно рассматривать в более укрупненном виде — на уровне взаимодействия функций. Это важно, поскольку человек способен помнить ограниченное количество фактов. Использование функций является первым шагом к повышению степени абстракции программы и ведет к упрощению ее структуры.

Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Процесс отладки программы, содержащей функции, можно лучше структурировать. Часто используемые функции можно помещать в библиотеки. Таким образом создаются более простые в отладке и сопровождении программы.

Следующим шагом в повышении уровня абстракции программы является группировка функций и связанных с ними данных в отдельные файлы (модули), компилируемые отдельно. Получившиеся в результате компиляции объектные модули объединяются в исполняемую программу с помощью компоновщика. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля и позволяя отлаживать программу по частям (или разными программистами).

Модуль содержит данные и функции их обработки. Другим модулям нежелательно иметь собственные средства обработки этих данных, они должны пользоваться для этого функциями первого модуля. Для того чтобы использовать модуль, нужно знать только его интерфейс, а не все детали его реализации. Чем

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и раздельной компиляции (строго говоря, раздельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора класс можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и раздельной компиляции (строгое говоря, раздельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена и определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]  
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора класс можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется инкапсуляцией. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). *Интерфейсом модуля* являются заголовки всех функций и описания доступных извне типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

Модульность в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и отдельной компиляции (строго говоря, отдельная компиляция не является элементом языка, а относится к его реализации).

Функции

Функции используются для наведения порядка в хаосе алгоритмов.

Б. Страуструп

Объявление и определение функций

Функция — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя `main` (с нее начинается выполнение программы). Функция начинает выполняться в момент *вызова*. Любая функция должна быть *объявлена* и *определена*. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. *Определение функции* содержит, кроме объявления, *тело* функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]  
{ тело функции }
```

Рассмотрим составные части определения.

- С помощью необязательного модификатора `класс` можно явно задать область видимости функции, используя ключевые слова `extern` и `static`:

- `extern` — глобальная видимость во всех модулях программы (по умолчанию);
 - `static` — видимость только в пределах модуля, в котором определена функция.
- Тип возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, указывается тип `void`.
- Список параметров определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра, передаваемого в функцию, указывается его тип и имя (в объявлении имени можно опускать).
- Об исключениях, обрабатываемых функцией, рассказывается в разделе «Список исключений функции», с. 227.

В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать. На имена параметров ограничений по соответствию не накладывается, поскольку функцию можно вызывать с различными аргументами, а в прототипах имена компилятором игнорируются (они служат только для улучшения читаемости программы).

Функцию можно определить как *встроенную* с помощью модификатора `inline`, который рекомендует компилятору вместо обращения к функции помещать ее код непосредственно в каждую точку вызова. Модификатор `inline` ставится перед типом функции. Он применяется для коротких функций, чтобы снизить накладные расходы на вызов (сохранение и восстановление регистров, передача управления). Директива `inline` носит рекомендательный характер и выполняется компилятором по мере возможности. Использование `inline`-функций может увеличить объем исполняемой программы. Определение функции должно предшествовать ее вызовам, иначе вместо `inline`-расширения компилятор сгенерирует обычный вызов.

Тип возвращаемого значения и типы параметров совместно определяют *тип функции*.

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого функцией значения не `void`, она может входить в состав выражений или, в частном случае, располагаться в правой части оператора присваивания.

Пример функции, возвращающей сумму двух целых величин:

```
#include <iostream.h>
int sum(int a, int b);      // объявление функции
int main(){
    int a = 2, b = 3, c, d;
    c = sum(a, b);          // вызов функции
    cin >> d;
    cout << sum(c, d);      // вызов функции
```

Пример

#inc

struc

cl

ti

di

);

void

int r

W

—

fr

r

}

void

c

}

Все вел

кальны

при вхо

ские пе

сора на

для то

вающей

При вы

значени

сохраня

менных

#incl

void

ir

cc

wf

}

}

int m

- `extern` — глобальная видимость во всех модулях программы (по умолчанию);
 - `static` — видимость только в пределах модуля, в котором определена функция.
- Тип возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, указывается тип `void`.
 - Список параметров определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра, передаваемого в функцию, указывается его тип и имя (в объявлении имени можно опускать).
 - Об исключениях, обрабатываемых функцией, рассказывается в разделе «Список исключений функции», с. 227.

В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать. На имена параметров ограничений по соответствию не накладывается, поскольку функцию можно вызывать с различными аргументами, а в прототипах имена компилятором игнорируются (они служат только для улучшения читаемости программы).

Функцию можно определить как *встроенную* с помощью модификатора `inline`, который рекомендует компилятору вместо обращения к функции помещать ее код непосредственно в каждую точку вызова. Модификатор `inline` ставится перед типом функции. Он применяется для коротких функций, чтобы снизить накладные расходы на вызов (сохранение и восстановление регистров, передача управления). Директива `inline` носит рекомендательный характер и выполняется компилятором по мере возможности. Использование `inline`-функций может увеличить объем исполняемой программы. Определение функции должно предшествовать ее вызовам, иначе вместо `inline`-расширения компилятор сгенерирует обычный вызов.

Тип возвращаемого значения и типы параметров совместно определяют *тип функции*.

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого функцией значения не `void`, она может входить в состав выражений или, в частном случае, располагаться в правой части оператора присваивания.

Пример функции, возвращающей сумму двух целых величин:

```
#include <iostream.h>
int sum(int a, int b):      // объявление функции
int main(){
    int a = 2, b = 3, c, d;
    c = sum(a, b);          // вызов функции
    cin >> d;
    cout << sum(c, d);      // вызов функции
```

```

    return 0;
}
int sum(int a, int b){           // определение функции
    return (a + b);
}

```

Пример функции, выводящей на экран поля переданной ей структуры:

```

#include <iostream.h>
struct Worker{
    char fio[30];
    int date, code;
    double salary;
};
void print_worker(Worker);      //объявление функции
int main(){
    Worker stuff[100];
    _ /* формирование массива stuff */

    for (int i = 0; i<100; i++)print_worker(stuff[i]);    // вызов функции
    return 0;
}
void print_worker(Worker w){    //определение функции
    cout << w.fio << ' ' << w.date << ' ' << w.code << ' ' << w.salary << endl;
}

```

Все величины, описанные внутри функции, а также ее параметры, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память под локальные автоматические переменные. Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции для того, чтобы при выходе из нее можно было продолжить выполнение вызывающей функции.

При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются. Если этого требуется избежать, при объявлении локальных переменных используется модификатор `static`:

```

#include <iostream.h>
void f(int a){
    int m = 0;
    cout << "n m p\n";
    while (a--){
        static int n = 0;
        int p = 0;
        cout << n++ << ' ' << m++ << ' ' << p++ << '\n';
    }
}
int main(){ f(3); f(2); return 0;}

```

Статическая переменная *n* размещается в сегменте данных и инициализируется один раз при первом выполнении оператора, содержащего ее определение. Автоматическая переменная *m* инициализируется при каждом входе в функцию. Автоматическая переменная *p* инициализируется при каждом входе в блок цикла. Программа выведет на экран:

```
n m p
0 0 0
1 1 0
2 2 0
n m p
3 0 0
4 1 0
```

При совместной работе функции должны обмениваться информацией. Это можно осуществить с помощью глобальных переменных, через параметры и через возвращаемое функцией значение.

Глобальные переменные

Глобальные переменные видны во всех функциях, где не описаны локальные переменные с теми же именами, поэтому использовать их для передачи данных между функциями очень легко. Тем не менее это не рекомендуется, поскольку затрудняет отладку программы и препятствует помещению функций в библиотеки общего пользования. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

Возвращаемое значение

Механизм возврата из функции в вызвавшую ее функцию реализуется оператором

```
return [ выражение ];
```

Функция может содержать несколько операторов `return` (это определяется потребностями алгоритма). Если функция описана как `void`, выражение не указывается. Оператор `return` можно опускать для функции типа `void`, если возврат из нее происходит перед закрывающей фигурной скобкой, и для функции `main`. В этой книге для экономии места оператор `return` в функции `main` не указан, поэтому при компиляции примеров выдается предупреждение. Выражение, указанное после `return`, неявно преобразуется к типу возвращаемого функцией значения и передается в точку вызова функции.

Примеры:

```
int f1(){return 1;} // правильно
void f2(){return 1;} // неправильно, f2 не должна возвращать значение
double f3{return 1;} // правильно, 1 преобразуется к типу double
```

Статическая переменная *n* размещается в сегменте данных и инициализируется один раз при первом выполнении оператора, содержащего ее определение. Автоматическая переменная *m* инициализируется при каждом входе в функцию. Автоматическая переменная *p* инициализируется при каждом входе в блок цикла. Программа выведет на экран:

```
n m p
0 0 0
1 1 0
2 2 0
n m p
3 0 0
4 1 0
```

При совместной работе функции должны обмениваться информацией. Это можно осуществить с помощью глобальных переменных, через параметры и через возвращаемое функцией значение.

Глобальные переменные

Глобальные переменные видны во всех функциях, где не описаны локальные переменные с теми же именами, поэтому использовать их для передачи данных между функциями очень легко. Тем не менее это не рекомендуется, поскольку затрудняет отладку программы и препятствует помещению функций в библиотеки общего пользования. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

Возвращаемое значение

Механизм возврата из функции в вызвавшую ее функцию реализуется оператором

```
return [ выражение ];
```

Функция может содержать несколько операторов `return` (это определяется потребностями алгоритма). Если функция описана как `void`, выражение не указывается. Оператор `return` можно опускать для функции типа `void`, если возврат из нее происходит перед закрывающей фигурной скобкой, и для функции `main`. В этой книге для экономии места оператор `return` в функции `main` не указан, поэтому при компиляции примеров выдается предупреждение. Выражение, указанное после `return`, неявно преобразуется к типу возвращаемого функцией значения и передается в точку вызова функции.

Примеры:

```
int f1(){return 1;} // правильно
void f2(){return 1;} // неправильно, f2 не должна возвращать значение
double f3{return 1;} // правильно, 1 преобразуется к типу double
```

ВНИМАНИЕ

Нельзя возвращать из функции указатель на локальную переменную, поскольку память, выделенная локальным переменным при входе в функцию, освобождается после возврата из нее.

Пример:

```
int* f(){
    int a = 5;
    return &a; // нельзя!
```

Параметры функции

Механизм параметров является основным способом обмена информацией между вызываемой и вызывающей функциями. Параметры, перечисленные в заголовке описания функции, называются *формальными*, а записанные в операторе вызова функции — *фактическими*.

При вызове функции в первую очередь вычисляются выражения, стоящие на месте фактических параметров; затем в стеке выделяется память под формальные параметры функции в соответствии с их типом, и каждому из них присваивается значение соответствующего фактического параметра. При этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение.

Существует два способа передачи параметров в функцию: по значению и по адресу.

При передаче по значению в стек заносятся копии значений фактических параметров, и операторы функции работают с этими копиями. Доступа к исходным значениям параметров у функции нет, а, следовательно, нет и возможности их изменить.

При передаче по адресу в стек заносятся копии адресов параметров, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения параметров:

```
#include <iostream.h>
void f(int i, int* j, int& k);
int main(){
    int i = 1, j = 2, k = 3;
    cout << "i j k\n";
    cout << i << ' ' << j << ' ' << k << '\n';
    f(i, &j, k);
    cout << i << ' ' << j << ' ' << k;
    return 0;
}
void f(int i, int* j, int& k){
    i++; (*j)++; k++;
}
```

Результат работы программы:

```
i j k
1 2 3
1 3 4
```

Первый параметр (*i*) передается по значению. Его изменение в функции не влияет на исходное значение. Вторым параметром (*j*) передается по адресу с помощью указателя, при этом для передачи в функцию адреса фактического параметра используется операция взятия адреса, а для получения его значения функции требуется операция разыменования. Третий параметр (*k*) передается по адресу с помощью ссылки.

При передаче по ссылке в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменуются. Поэтому использование ссылок вместо указателей улучшает читаемость программы, избавляя от необходимости применять операции получения адреса и разыменования. Использование ссылок вместо передачи по значению более эффективно, поскольку не требует копирования параметров, что имеет значение при передаче структур данных большого объема.

Если требуется запретить изменение параметра внутри функции, используется модификатор `const`:

```
int f(const char*);
char* t(char* a, const int* b);
```

СОВЕТ

Рекомендуется указывать `const` перед всеми параметрами, изменение которых в функции не предусмотрено. Это облегчает отладку больших программ, так как по заголовку функции можно сделать вывод о том, какие величины в ней изменяются, а какие нет. Кроме того, на место параметра типа `const&` может передаваться константа, а для переменной при необходимости выполняются преобразования типа.

Таким образом, исходные данные, которые не должны изменяться в функции, предпочтительнее передавать ей с помощью константных ссылок.

По умолчанию параметры любого типа, кроме массива и функции (например, востребованного, структурного, перечисления, объединения, указателя), передаются в функцию по значению.

Передача массивов в качестве параметров

При использовании в качестве параметра *массива* в функцию передается указатель на его первый элемент, иными словами, массив всегда передается по адресу. При этом информация о количестве элементов массива теряется, и следует передавать его размерность через отдельный параметр¹ (в случае массива символов, то есть строки, ее фактическую длину можно определить по положению нуль-символа):

¹ Если размерность массива является константой, проблем не возникает, поскольку можно указать ее и при описании формального параметра, и в качестве границы циклов при обработке массива внутри функции.


```

#include <iostream.h>
int sum(const int* mas, const int n);
int const n = 10;
int main(){
    int marks[n] = {3, 4, 5, 4, 4};
    cout << "Сумма элементов массива: " << sum(marks, n);
    return 0;
}
int sum(const int* mas, const int n){
    // варианты: int sum(int mas[], int n)
    // или      int sum(int mas[n], int n)
    // (величина n должна быть константой)
    int s = 0;
    for (int i = 0; i<n; i++) s += mas[i];
    return s;
}

```

При передаче *многомерных массивов* все размерности, если они не известны на этапе компиляции, должны передаваться в качестве параметров. Внутри функции массив интерпретируется как одномерный, а его индекс пересчитывается в программе. В приведенном ниже примере с помощью функции подсчитывается сумма элементов двух двумерных массивов. Размерность массива *b* известна на этапе компиляции, под массив *a* память выделяется динамически:

```

#include <stdio.h>
#include <stdlib.h>
int sum(const int *a, const int nstr, const int nstb);
int main(){
    int b[2][2] = {{2, 2}, {4, 3}};
    printf("Сумма элементов b: %d\n", sum(&b[0][0], 2, 2));
    // имя массива передавать в sum нельзя из-за несоответствия типов
    int i, j, nstr, nstb, *a;
    printf("Введите количество строк и столбцов: \n");
    scanf("%d%d", &nstr, &nstb);
    a = (int *)malloc(nstr * nstb * sizeof(int));
    for (i = 0; i<nstr; i++)
        for (j = 0; j<nstb; j++)scanf("%d", &a[i * nstb + j]);
    printf("Сумма элементов a: %d\n", sum(a, nstr, nstb));
    return 0;}
int sum(const int *a, const int nstr, const int nstb){
    int i, j, s = 0;
    for (i = 0; i<nstr; i++)
        for (j = 0; j<nstb; j++)s += a[i * nstb + j];
    return s;
}

```

Для того чтобы работать с двумерным массивом естественным образом, можно применить альтернативный способ выделения памяти:

```

#include <iostream.h>
int sum(int **a, const int nstr, const int nstb);
int main(){
    int nstr, nstb;
    cin >> nstr >> nstb;
    int **a, i, j;
    // Формирование матрицы a:
    a = new int* [nstr];
    for (i = 0; i<nstr; i++)
        a[i] = new int [nstb];
    for (i = 0; i<nstr; i++)
        for (j = 0; j<nstb; j++)cin >> a[i][j];
    cout << sum(a, nstr, nstb);
    return 0;
}
int sum(int **a, const int nstr, const int nstb){
    int i, j, s = 0;
    for (i = 0; i<nstr; i++)
        for (j = 0; j<nstb; j++)s += a[i][j];
    return s;
}

```

В этом случае память выделяется в два этапа: сначала под столбец указателей на строки матрицы, а затем в цикле под каждую строку, как показано на рис. 1.10. Освобождение памяти должно выполняться в обратном порядке.

Передача имен функций в качестве параметров

Функцию можно вызвать через указатель на нее. Для этого объявляется указатель соответствующего типа и ему с помощью операции взятия адреса присваивается адрес функции:

```

void f(int a){ /* j */ }    // определение функции
void (*pf)(int);           // указатель на функцию

...

pf = &f;                  // указателю присваивается адрес функции
                           // (можно написать pf = f;)
pf(10);                   // функция f вызывается через указатель pf
                           // (можно написать (*pf)(10) )

```

Для того чтобы сделать программу легко читаемой, при описании указателей на функции используют переименование типов (typedef). Можно объявлять массивы указателей на функции (это может быть полезно, например, при реализации меню):

```

// Описание типа PF как указателя
// на функцию с одним параметром типа int:
typedef void (*PF)(int);
// Описание и инициализация массива указателей:
PF menu[] = {&new, &open, &save};
menu[1](10); // Вызов функции open

```

Здесь `new`, `open` и `save` — имена функций, которые должны быть объявлены ранее. Указатели на функции передаются в подпрограмму таким же образом, как и параметры других типов:

```
#include <iostream.h>
typedef void (*PF)(int);
void f1(PF pf){ // функция f1 получает в качестве параметра указатель типа PF
    pf(5);      //вызов функции, переданной через указатель
}
void f(int i ){cout << i;}
int main(){
    f1(f);
    return 0;
}
```

Тип указателя и тип функции, которая вызывается посредством этого указателя, должны совпадать в точности.

Параметры со значениями по умолчанию

Чтобы упростить вызов функции, в ее заголовке можно указать значения параметров по умолчанию. Эти параметры должны быть последними в списке и могут опускаться при вызове функции. Если при вызове параметр опущен, должны быть опущены и все параметры, стоящие за ним. В качестве значений параметров по умолчанию могут использоваться константы, глобальные переменные и выражения:

```
int f(int a, int b = 0);
void f1(int, int = 100, char* = 0); /* обратите внимание на пробел между * и
= (без него получилась бы операция сложного присваивания *=) */
void err(int errValue = errno);      // errno – глобальная переменная

f(100); f(a, 1);                     // варианты вызова функции f
f1(a); f1(a, 10); f1(a, 10, "Vasia"); // варианты вызова функции f1
f1(a, "Vasia")                       // неверно!
```

Функции с переменным числом параметров

Если список формальных параметров функции заканчивается многоточием, это означает, что при ее вызове на этом месте можно указать еще несколько параметров. Проверка соответствия типов для этих параметров не выполняется, `char` и `short` передаются как `int`, а `float` — как `double`. В качестве примера можно привести функцию `printf`, прототип которой имеет вид:

```
int printf(const char*, ...);
```

Это означает, что вызов функции должен содержать по крайней мере один параметр типа `char*` и может либо содержать, либо не содержать другие параметры:

```
printf("Введите исходные данные"); // один параметр
printf("Сумма: %5.2f рублей", sum); // два параметра
printf("%d %d %d %d", a, b, c, d);  // пять параметров
```

Для доступа к необязательным параметрам внутри функции используются макросы библиотеки `va_start`, `va_arg` и `va_end`, находящиеся в заголовочном файле `<stdarg.h>`. Эти макросы описаны в приложении 5.

Поскольку компилятор не имеет информации для контроля типов, вместо функций с переменным числом параметров предпочтительнее пользоваться параметрами по умолчанию или перегруженными функциями (см. «Перегрузка функций», с. 83), хотя можно представить случаи, когда переменное число параметров является лучшим решением.

Рекурсивные функции

Рекурсивной называется функция, которая вызывает саму себя. Такая рекурсия называется *прямой*. Существует еще *косвенная* рекурсия, когда две или более функций вызывают друг друга. Если функция вызывает себя, в стеке создается копия значений ее параметров, как и при вызове обычной функции, после чего управление передается первому исполняемому оператору функции. При повторном вызове этот процесс повторяется. Ясно, что для завершения вычислений каждая рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

Классическим примером рекурсивной функции является вычисление факториала (это не означает, что факториал следует вычислять именно так). Для того чтобы получить значение факториала числа n , требуется умножить на n факториал числа $(n-1)$. Известно также, что $0!=1$ и $1!=1$.

```
long fact(long n){
    if (n==0 || n==1) return 1;
    return (n * fact(n - 1));
}
```

То же самое можно записать короче:

```
long fact(long n){
    return (n>1) ? n * fact(n - 1) : 1;
}
```

Рекурсивные функции чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно, например, с двоичными деревьями (с. 122). Любую рекурсивную функцию можно реализовать без применения рекурсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно. Достоинством рекурсии является компактная запись, а недостатками — расход времени и памяти на повторные вызовы функции и передачу ей копий параметров, и, главное, опасность переполнения стека.

Перегрузка функций

Часто бывает удобно, чтобы функции, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Если это имя mnemonic, то есть несет нужную информацию, это делает программу более понятной, поскольку для каждого действия требуется помнить только одно имя. Использование нескольких функций с одним и тем же именем, но с различными типами параметров, называется *перегрузкой функций*.

Компилятор определяет, какую именно функцию требуется вызвать, по типу фактических параметров. Этот процесс называется разрешением перегрузки (перевод английского слова *resolution* в смысле «уточнение»). Тип возвращаемого функцией значения в разрешении не участвует. Механизм разрешения основан на достаточно сложном наборе правил, смысл которых сводится к тому, чтобы использовать функцию с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта функции, определяющей наибольшее значение:

```
// Возвращает наибольшее из двух целых:
int max(int, int);
// Возвращает подстроку наибольшей длины:
char* max(char*, char*);
// Возвращает наибольшее из первого параметра и длины второго:
int max (int, char*);
// Возвращает наибольшее из второго параметра и длины первого:
int max (char*, int);
void f(int a, int b, char* c, char* d){
    cout << max (a, b) << max(c, d) << max(a, c) << max(c, b);
}
```

При вызове функции `max` компилятор выбирает соответствующий типу фактических параметров вариант функции (в приведенном примере будут последовательно вызваны все четыре варианта функции).

Если точного соответствия не найдено, выполняются продвижения типов в соответствии с общими правилами (см. с. 38 и приложение 3), например, `bool` и `char` в `int`, `float` в `double` и т. п. Далее выполняются стандартные преобразования типов, например, `int` в `double` или указателей в `void*`. Следующим шагом является выполнение преобразований типа, заданных пользователем (об этих преобразованиях рассказывается во второй части книги, с. 195), а также поиск соответствий за счет переменного числа аргументов функций. Если соответствие на одном и том же этапе может быть получено более чем одним способом, вызов считается неоднозначным и выдается сообщение об ошибке.

Неоднозначность может появиться при:

- преобразовании типа;
- использовании параметров-ссылок;
- использовании аргументов по умолчанию.

Пример неоднозначности при преобразовании типа:

```
#include <iostream.h>
float f(float i){
    cout << "function float f(float i)" << endl;
    return i;
}
double f(double i){
    cout << "function double f(double i)" << endl;
    return i*2;
}
int main(){
    float x = 10.09;
    double y = 10.09;
    cout << f(x) << endl; // Вызывается f(float)
    cout << f(y) << endl; // Вызывается f(double)
    /* cout << f(10) << endl; Неоднозначность – как преобразовать 10: во float
или double? */
    return 0;
}
```

Для устранения этой неоднозначности требуется явное приведение типа для константы 10.

Пример неоднозначности при использовании параметров-ссылок: если одна из перегружаемых функций объявлена как `int f(int a, int b)`, а другая — как `int f(int a, int &b)`, то компилятор не сможет узнать, какая из этих функций вызывается, так как нет синтаксических различий между вызовом функции, которая получает параметр по значению, и вызовом функции, которая получает параметр по ссылке.

Пример неоднозначности при использовании аргументов по умолчанию:

```
#include <iostream.h>
int f(int a){return a;}
int f(int a, int b = 1){return a * b;}
int main(){
    cout << f(10, 2); // Вызывается f(int, int)
    /* cout << f(10); Неоднозначность – что вызывается: f(int, int) или
f(int) ? */
    return 0;
}
```

Ниже приведены правила описания перегруженных функций.

- ❑ Перегруженные функции должны находиться в *одной области видимости*, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках.
- ❑ Перегруженные функции могут иметь *параметры по умолчанию*, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.

- Функции не могут быть перегружены, если описание их параметров отличается только *модификатором* `const` или *использованием ссылки* (например, `int` и `const int` или `int` и `int&`).

Шаблоны функций

Многие алгоритмы не зависят от типов данных, с которыми они работают (классический пример — сортировка). Естественно желание параметризовать алгоритм таким образом, чтобы его можно было использовать для различных типов данных. Первое, что может прийти в голову — передать информацию о типе в качестве параметра (например, одним параметром в функцию передается указатель на данные, а другим — длина элемента данных в байтах). Использование дополнительного параметра означает генерацию дополнительного кода, что снижает эффективность программы, особенно при рекурсивных вызовах и вызовах во внутренних циклах; кроме того, отсутствует возможность контроля типов. Другим решением будет написание для работы с различными типами данных нескольких перегруженных функций, но в таком случае в программе будет несколько одинаковых по логике функций, и для каждого нового типа придется вводить новую.

В C++ есть мощное средство параметризации — шаблоны. Существуют шаблоны функций и шаблоны классов (о шаблонах классов см. с. 211). С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции в виде параметра на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному типу. Таким образом, создается функция, которая автоматически перегружает сама себя и при этом не содержит накладных расходов, связанных с параметризацией.

Формат простейшей функции-шаблона:

```
template <class Type> заголовок{
    /* тело функции */
}
```

Вместо слова `Type` может использоваться произвольное имя.

В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только типом, но и просто переменной, например:

```
template<class A, class B, int i> void f(){ ... }
```

Например, функция, сортирующая методом выбора (он был рассмотрен на с. 59) массив из n элементов любого типа, в виде шаблона может выглядеть так:

```
template <class Type>
void sort_vybor(Type *b, int n){
    Type a;           //буферная переменная для обмена элементов
    for (int i = 0; i<n-1; i++){
        int imin = i;
        for (int j = i + 1; j<n; j++)
            if (b[j] < b[imin]) imin = j;
```

```

        a = b[i]; b[i] = b[imin]; b[imin] = a;
    }
}

```

Главная функция программы, вызывающей эту функцию-шаблон, может иметь вид:

```

#include <iostream.h>
template <class Type> void sort_vybor(Type *b, int n);
int main(){
    const int n = 20;
    int i, b[n];
    for (i = 0; i<n; i++) cin >> b[i];
    sort_vybor(b, n);          // Сортировка целочисленного массива
    for (i = 0; i<n; i++) cout << b[i] << ' ';
    cout << endl;
    double a[] = {0.22, 117, -0.08, 0.21, 42.5};
    sort_vybor(a, 5);          // Сортировка массива вещественных чисел
    for (i = 0; i<5; i++) cout << a[i] << ' ';
    return 0;
}

```

Первый же вызов функции, который использует конкретный тип данных, приводит к созданию компилятором кода для соответствующей версии функции. Это процесс называется инстанцированием шаблона (instantiation). Конкретный тип для инстанцирования либо определяется компилятором автоматически, исходя из типов параметров при вызове функции, либо задается явным образом. При повторном вызове с тем же типом данных код заново не генерируется. На месте параметра шаблона, являющегося не типом, а переменной, должно указываться константное выражение¹.

Пример явного задания аргументов шаблона при вызове:

```

template<class X, class Y, class Z> void f(Y, Z);
void g(){
    f<int, char*, double>("Vasia", 3.0);
    f<int, char*>("Vasia", 3.0); // Z определяется как double
    f<int>("Vasia", 3.0); // Y определяется как char*, а Z – как double
    // f("Vasia", 3.0);        ошибка: X определить невозможно
}

```

Чтобы применить функцию-шаблон к типу данных, определенному пользователем (структуре или классу), требуется перегрузить операции для этого типа данных, используемые в функции (о перегрузке операций см. с. 189).

Как и обычные функции, шаблоны функций могут быть *перегружены как с помощью шаблонов, так и обычными функциями*.

Можно предусмотреть специальную обработку отдельных параметров и типов с помощью *специализации шаблона функции*. Допустим, мы хотим более эффективно

¹ Подробнее о параметрах шаблонов рассказывается в разделе «Шаблоны классов», с. 2

```

        a = b[i]; b[i] = b[imin]; b[imin] = a;
    }
}

```

Главная функция программы, вызывающей эту функцию-шаблон, может иметь вид:

```

#include <iostream.h>
template <class Type> void sort_vybor(Type *b, int n);
int main(){
    const int n = 20;
    int i, b[n];
    for (i = 0; i<n; i++) cin >> b[i];
    sort_vybor(b, n);          // Сортировка целочисленного массива
    for (i = 0; i<n; i++) cout << b[i] << ' ';
    cout << endl;
    double a[] = {0.22, 117, -0.08, 0.21, 42.5};
    sort_vybor(a, 5);          // Сортировка массива вещественных чисел
    for (i = 0; i<5; i++) cout << a[i] << ' ';
    return 0;
}

```

Первый же вызов функции, который использует конкретный тип данных, приводит к созданию компилятором кода для соответствующей версии функции. Этот процесс называется инстанцированием шаблона (instantiation). Конкретный тип для инстанцирования либо определяется компилятором автоматически, исходя из типов параметров при вызове функции, либо задается явным образом. При повторном вызове с тем же типом данных код заново не генерируется. На месте параметра шаблона, являющегося не типом, а переменной, должно указываться константное выражение¹.

Пример явного задания аргументов шаблона при вызове:

```

template<class X, class Y, class Z> void f(Y, Z);
void g(){
    f<int, char*, double>("Vasia", 3.0);
    f<int, char*>("Vasia", 3.0); // Z определяется как double
    f<int>("Vasia", 3.0); // Y определяется как char*, а Z – как double
    // f("Vasia", 3.0);        ошибка: X определить невозможно
}

```

Чтобы применить функцию-шаблон к типу данных, определенному пользователем (структуре или классу), требуется перегрузить операции для этого типа данных, используемые в функции (о перегрузке операций см. с. 189).

Как и обычные функции, шаблоны функций могут быть *перегружены как с помощью шаблонов, так и обычными функциями*.

Можно предусмотреть специальную обработку отдельных параметров и типов с помощью *специализации шаблона функции*. Допустим, мы хотим более эффективно

¹ Подробнее о параметрах шаблонов рассказывается в разделе «Шаблоны классов», с. 211.

реализовать общий алгоритм сортировки для целых чисел. В этом случае можно «вручную» задать вариант шаблона функции для работы с целыми числами:

```
void sort_vibor<int>(int *b, int n){  
    ... // Тело специализированного варианта функции  
}
```

Сигнатура шаблона функции включает не только ее тип и типы параметров, но и фактический аргумент шаблона. Обычная функция никогда не считается специализацией шаблона, несмотря на то, что может иметь то же имя и тип возвращаемого значения.

Функция `main()`

Функция, которой передается управление после запуска программы, должна иметь имя `main`. Она может возвращать значение в вызвавшую систему и принимать параметры из внешнего окружения. Возвращаемое значение должно быть целого типа. Стандарт предусматривает два формата функции:

```
// без параметров:  
тип main(){ /* ... */}  
// с двумя параметрами:  
тип main(int argc, char* argv[]){ /* ... */}
```

При запуске программы параметры разделяются пробелами. Имена параметров в программе могут быть любыми, но принято использовать `argc` и `argv`. Первый параметр (`argc`) определяет количество параметров, передаваемых функции, включая имя самой программы, второй параметр (`argv`) является указателем на массив указателей типа `char*`. Каждый элемент массива содержит указатель на отдельный параметр командной строки, хранящийся в виде С-строки, оканчивающейся нуль-символом. Первый элемент массива (`argv[0]`) ссылается на полное имя запускаемого на выполнение файла, следующий (`argv[1]`) указывает на первый параметр, `argv[2]` — на второй параметр, и так далее. Параметр `argv[argc]` должен быть равен 0.

Если функция `main()` ничего не возвращает, вызвавшая система получит значение, означающее успешное завершение. Ненулевое значение означает аварийное завершение. Оператор возврата из `main()` можно опускать.

```
#include <iostream.h>  
void main(int argc, char* argv[]){  
    for (int i = 0; i<argc; i++) cout << argv[i] << '\n';  
}
```

Пусть исполняемый файл программы имеет имя `main.exe` и вызывается из командной строки:

```
d:\cpp\main.exe one two three
```

На экран будет выведено:

```
D:\CPP\MAIN.EXE  
one  
two  
three
```

Функции стандартной библиотеки

Любая программа на C++ содержит обращения к стандартной библиотеке, в которой находятся определения типов, констант, макросов, функций и классов. Чтобы использовать их в программе, требуется с помощью директивы `#include` включить в исходный текст программы заголовочные файлы, в которых находятся соответствующие объявления. Сами библиотечные функции хранятся в скомпилированном виде и подключаются к программе на этапе компоновки. В программах на C++ могут использоваться функции, унаследованные от библиотеки C. Использование классов стандартной библиотеки рассматривается в третьей части книги.

Функции библиотеки можно разбить на группы по их назначению: ввод/вывод, обработка строк, математические функции, работа с динамической памятью, поиск и сортировка и т. д. Список функций библиотеки приведен в приложении 6. Ниже приведен краткий обзор функций ввода/вывода в стиле C, функций работы со строками и символами и математических функций.

Функции ввода/вывода

Ввод/вывод в C++ реализуется либо с помощью функций, унаследованных от библиотеки C, либо с помощью потоков C++. Смешивать эти два способа в одной программе можно только синхронизировав ввод с помощью функции `sync_with_stdio()`. Каждый способ имеет свои преимущества. Преимущество использования потоков в том, что они легче в использовании в простых случаях ввода/вывода, не требующих форматирования, а, главное, потоковые операции можно переопределить для собственных классов. Ввод/вывод в стиле C удобнее использовать при форматированном выводе в программах, не использующих объектно-ориентированную технику. Кроме того, существуют миллионы строк, написанных на C и перенесенных на C++, с которыми программисту приходится сталкиваться. Потоки C++ рассматриваются в разделе «Потоковые классы», с. 265.

Для использования функций ввода/вывода в стиле C необходимо подключить к программе заголовочный файл `<stdio.h>` или `<cstdio>`. При вводе/выводе данные рассматриваются как поток байтов. Физически поток представляет собой файл или устройство, например, клавиатуру или дисплей, рассматривающиеся как частный случай файла.

Открытие потока

Работа с потоком начинается с его *открытия*. Поток можно открыть для чтения и/или записи в двоичном или текстовом режиме. Функция открытия потока имеет формат:

```
FILE* fopen(const char* filename, const char* mode);
```

При успешном открытии потока функция возвращает указатель на предопределенную структуру типа `FILE`, содержащую всю необходимую для работы с потоком информацию, или `NULL` в противном случае. Первый параметр — имя открываемого файла в виде C-строки, второй — режим открытия файла:

"r" — файл открывается для чтения;

"w" — открывается пустой файл для записи (если файл существует, он стирается);

- *a" — файл открывается для добавления информации в его конец;
- "r+" — файл открывается для чтения и записи (файл должен существовать);
- "w+" — открывается пустой файл для чтения и записи (если файл существует, он стирается);
- "a+" — файл открывается для чтения и добавления информации в его конец.

Режим открытия может также содержать символы t (текстовый режим) или b (двоичный режим), отличающиеся обработкой символов перехода на новую строку. По умолчанию файл открывается в текстовом режиме, при котором комбинация символов «возврат каретки» и «перевод строки» (0x13 0x10) при вводе преобразуются в одиночный символ перевода строки (при выводе выполняется обратное преобразование). В двоичном режиме эти преобразования не выполняются.

Пример:

```
FILE *f = fopen("d:\\cpp\\data", "rb+");
```

Указатель f используется в дальнейших операциях с потоком. Его передают функциям ввода/вывода в качестве параметра.

При открытии потока с ним связывается область памяти, называемая *буфером*. При выводе вся информация направляется в буфер и накапливается там до заполнения буфера или до закрытия потока. Чтение осуществляется блоками, равными размеру буфера, и данные читаются из буфера. Буферизация позволяет более быстро и эффективно обмениваться информацией с внешними устройствами. Следует иметь в виду, что при аварийном завершении программы выходной буфер может быть не выгружен, и возможна потеря данных. С помощью функций `setbuf` и `setvbuf` можно управлять размерами и наличием буферов.

Существует пять предопределенных потоков, которые открываются в начале работы программы: стандартный ввод `stdin`, стандартный вывод `stdout`, стандартный вывод сообщений об ошибках `stderr`, стандартный дополнительный поток `stdaux` и стандартная печать `stdprn`. Первые три потока по умолчанию относятся к консоли. Эти указатели можно использовать в любой функции ввода/вывода там, где требуется указатель потока.

Ввод/вывод в поток

Ввод/вывод в поток можно осуществлять различными способами: в виде последовательности байтов, в виде символов и строк или с использованием форматных преобразований. Для каждого вида операций определен свой набор функций.

Операции ввода/вывода выполняются начиная с текущей позиции потока, определяемой положением *указателя потока*. Указатель устанавливается при открытии на начало или конец файла (в соответствии с режимом открытия) и изменяется автоматически после каждой операции ввода/вывода. Текущее положение указателя можно получить с помощью функций `ftell` и `fgetpos` и задать явным образом с помощью функций `fseek` и `fsetpos`. Эти функции нельзя использовать для стандартных потоков. Ниже перечислены основные функции ввода/вывода потока.

- ❑ Чтение и запись потока байтов выполняют функции `fread` и `fwrite`.
- ❑ Чтение символа из потока — `getc`, `fgetc`, из стандартного потока `stdin` — `getchar`.
- ❑ Запись символа в поток — `putc`, `fputc`, в стандартный поток `stdout` — `putchar`.
- ❑ Чтение строки из потока — `fgets`, из стандартного потока `stdin` — `gets`.
- ❑ Запись строки в поток — `fputs`, в стандартный поток `stdout` — `puts`.
- ❑ Форматированный ввод из потока — `fscanf`, из стандартного потока `stdin` — `scanf`, из строки — `sscanf`.
- ❑ Форматированный вывод в поток — `fprintf`, в стандартный поток `stdout` — `printf`, в строку — `sprintf`.

Заккрытие потока

Поток закрывается либо при завершении программы, либо явным образом с помощью функции `fclose`:

```
int fclose(FILE*);
```

Перед закрытием потока информация из связанных с ним буферов выгружается на диск. Рекомендуется всегда явным образом закрывать потоки, открытые для записи, чтобы избежать потери данных.

Обработка ошибок

Функции работы с потоком возвращают значения, которые рекомендуется анализировать в программе и обрабатывать ошибочные ситуации, возникающие, например, при открытии существующих файлов или чтении из потока. При работе с файлами часто используются функции `feof` и `ferror`:

```
int feof(FILE*)    возвращает не равное нулю значение, если достигнут
                  конец файла, в противном случае 0;

int ferror(FILE*)  возвращает не равное нулю значение, если обнаружена
                  ошибка ввода/вывода, в противном случае 0.
```

Пример работы с потоками

Допустим, что в файле хранятся сведения о мониторах. В каждой строке указан тип, оптовая и розничная цены и примечание. Для простоты данные в каждой строке записаны единообразно: первые 20 символов занимает тип монитора, далее по 5 символов целые числа, представляющие оптовую и розничную цены, затем примечание длиной не более 40 символов.

Приведенная ниже программа построчно считывает данные из текстового файла в буферную переменную `s`, затем формирует из них структуру `mon` и записывает ее в двоичном режиме в выходной файл. Далее иллюстрируется считывание из этого файла произвольной записи.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(){
```

```

FILE *fi, *fo;
if((fi = fopen("d:\\c\\file.txt", "r")) == 0){
    cout << "Ошибка открытия входного файла"; return 1;}
if((fo = fopen("d:\\c\\binfile.out", "w+b")) == 0){
    cout << "Ошибка открытия выходного файла"; return 1;}
const int dl = 80;
char    s[dl];
struct{
    char  type[20];
    int   opt, rozn;
    char  comm[40];
}mon;
int kol = 0; // Количество записей в файле
while (fgets(s, dl, fi)){
    // Преобразование строки в структуру:
    strncpy(mon.type, s, 19);      // Описание strncpy см. в приложении 6
    mon.type[19]='\0';
    mon.opt = atoi(&s[20]);        // Описание atoi см. в след. разделе
    mon.rozn = atoi(&s[25]);
    strncpy(mon.comm, &s[30], 40);
    fwrite(&mon, sizeof mon, 1, fo);
    kol++;
}
fclose(fi);
int i; cin >> i;                  // Номер записи
if (i >= kol){cout << "Запись не существует"; return 1;}
// Установка указателя текущей позиции файла на запись i:
fseek(fo, (sizeof mon)*i, SEEK_SET);
fread(&mon, sizeof mon, 1, fo);
cout << "mon.type " << mon.type << " opt " << mon.opt
    << " rozn " << mon.rozn << endl;
fclose(fo);
return 0;
}

```

Функции работы со строками и символами

Строка представляет собой массив символов, заканчивающийся нуль-символом. В C++ есть две возможности работы со строками: функции, унаследованные из библиотеки C (заголовочный файл `<string.h>` или `<cstring>`), и библиотечный класс C++ `string`, предоставляющий более широкие возможности представления, обработки и контроля строк (он рассматривается в разделе «Строки» на с. 286).

Библиотека C содержит функции копирования строк (`strcpy`, `strncpy`), сравнения (`strcmp`, `strncmp`), объединения строк (`strcat`, `strncat`), поиска подстроки (`strstr`), поиска вхождения символа (`strchr`, `strchr`, `strpbrk`), определения длины строки (`strlen`) и другие. Полный список функций приведен в приложении 6.

В заголовочных файлах `<stdlib.h>` и `<cstdlib>` содержатся полезные функции преобразования строк в числа (обратные преобразования можно сделать с помощью функции `sprintf`):

`double atof(const char* p)` преобразует переданную строку в `double`;
`int atoi(const char* p)` преобразует переданную строку в `int`;
`long atol(const char* p)` преобразует переданную строку в `long`.

Пробелы и табуляции в начале строки пропускаются. Преобразование прекращается при встрече недопустимого символа или конца строки. Если строку нельзя преобразовать в число, возвращается 0. Если число выходит за пределы диапазона данного типа, переменной `errno` (заголовочный файл `<cerrno>`) присваивается значение `ERANGE` и возвращается допустимое число.

Пример (программа заполняет массив типа `double` из строки):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(){
    char s[] = "2, 38.5, 70, 0, 0, 1", *p = s;
    double m[10];
    int i = 0;
    do{
        m[i++] = atof(p);
        if (i>9)break;
    }while(p = strchr(p, ','), p++);
    for( int k = 0; k<i; k++) printf("%5.2f ", m[k]);
    return 0;
}
```

Для работы с символами в стандартной библиотеке (заголовочные файлы `<ctype.h>` и `<sctype>`) есть следующие функции:

Имя	Проверка на принадлежность символа множеству
<code>isalnum</code>	букв и цифр (A–Z, a–z, 0–9)
<code>isalpha</code>	букв (A–Z, a–z)
<code>iscntrl</code>	управляющих символов (с кодами 0..31 и 127)
<code>isdigit</code>	цифр (0–9)
<code>isgraph</code>	печатаемых символов, кроме пробела (<code>isalpha</code> <code>isdigit</code> <code>ispunct</code>)
<code>intbl islower</code>	букв нижнего регистра (a–z)
<code>isprint</code>	печатаемых символов
<code>ispunct</code>	знаков пунктуации
<code>isspace</code>	символов-разделителей
<code>isupper</code>	букв верхнего регистра (A–Z)
<code>isxdigit</code>	шестнадцатеричных цифр (A–F, a–f, 0–9)

Функции принимают величину типа `int` и возвращают значение `true`, если условие выполняется. Рекомендуется пользоваться стандартными функциями, а не

писать собственные циклы проверки, так как это снижает количество ошибок в программе.

Кроме описанных выше, в библиотеке есть функции `tolower` и `toupper`, переводящие символ соответственно в нижний и верхний регистр.

Для каждой из перечисленных функций есть ее аналог для многобайтных символов типа `wchar_t`, содержащий в названии букву `w`. Эти функции также описаны в приложении 6.

Математические функции

C++ унаследовал из C стандартные математические функции, описание которых находится в заголовочных файлах `<math.h>` (`<cmath>`). Они позволяют получить абсолютное значение (`abs`, `fabs`), округленное число (`ceil`, `floor`), квадратный корень (`sqrt`), степень (`pow`), значения тригонометрических функций (`sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `asin`, `acos`, `atan`, `atan2`), экспоненту (`exp`), логарифм (`log`, `log10`), дробную и целую части числа (`modf`), остаток от деления (`fmod`) и другие.

Ошибки индицируются установкой `errno` из `<errno.h>` (`<cerrno>`) в значение `EDOM` для ошибок, связанных с выходом из области определения, и `ERANGE` для ошибок выхода за пределы диапазона.

Директивы препроцессора

Препроцессором называется первая фаза компилятора. Инструкции препроцессора называются директивами. Они должны начинаться с символа `#`, перед которым в строке могут находиться только пробельные символы (с. 17).

Директива `#include`

Директива `#include <имя_файла>` вставляет содержимое указанного файла в ту точку исходного файла, где она записана. Включаемый файл также может содержать директивы `#include`. Поиск файла, если не указан полный путь, ведется в стандартных каталогах включаемых файлов. Вместо угловых скобок могут использоваться кавычки (`" "`) — в этом случае поиск файла ведется в каталоге, содержащем исходный файл, а затем уже в стандартных каталогах.

Директива `#include` является простейшим средством обеспечения согласованности объявлений в различных файлах, она включает в них информацию об интерфейсе из заголовочных файлов.

Заголовочные файлы обычно имеют расширение `.h` и могут содержать:

- определения типов, констант, встроенных функций, шаблонов, перечислений;
- объявления функций, данных, имен, шаблонов;
- пространства имен;
- директивы препроцессора;
- комментарии.

В заголовочном файле не должно быть определений функций и данных. Эти правила не являются требованием языка, а отражают разумный способ использования директивы.

При указании заголовочных файлов стандартной библиотеки расширение `.h` можно опускать¹. Это сделано для того, чтобы не ограничивать способы их хранения. Для каждого файла библиотеки C с именем `<name.h>` имеется соответствующий файл библиотеки C++ `<sname>`, в котором те же средства описываются в пространстве имен `std`. Например, директива `#include <cstdio>` обеспечивает те же возможности, что и `#include <stdio.h>`, но при обращении к стандартным функциям требуется указывать имя пространства имен `std` (см. раздел «Поименованные области», с. 99).

Директива `#define`

Директива `#define` определяет подстановку в тексте программы. Она используется для определения:

- *символических констант*:

```
#define имя текст_подстановки
```

(все вхождения имени заменяются на текст подстановки);

- *макросов*, которые выглядят как функции, но реализуются подстановкой их текста в текст программы:

```
#define имя( параметры ) текст_подстановки
```

- *символов, управляющих условной компиляцией*. Они используются вместе с директивами `#ifdef` и `#ifndef`. Формат:

```
#define имя
```

Примеры:

```
#define VERSION 1
#define VASIA "Василий Иванович"
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MUX
```

Имена рекомендуется записывать прописными буквами, чтобы зрительно отличать их от имен переменных и функций. Параметры макроса используются при макроподстановке, например, если в тексте программы используется вызов макроса `y = MAX(sum1, sum2);`, он будет заменен на

```
y = ((sum1)>(sum2)?(sum1):(sum2));
```

Отсутствие круглых скобок может привести к неправильному порядку вычисления, поскольку препроцессор не оценивает вставляемый текст с точки зрения синтаксиса. Например, если к макросу `#define sqr(x) (x*x)` обратиться как `sqr(y+1)`, в результате подстановки получится выражение `(y+1*y+1)`.

Макросы и символические константы унаследованы из языка C, при написании программ на C++ их следует избегать. Вместо символических констант предпоч-

¹ Старые версии компиляторов могут не поддерживать это свежее требование стандарта.

тительнее использовать `const` или `enum`, а вместо макросов — встроенные функции или шаблоны.

Директивы условной компиляции

Директивы условной компиляции `#if`, `#ifdef` и `#ifndef` применяются для того, чтобы исключить компиляцию отдельных частей программы. Это бывает полезно при отладке или, например, при поддержке нескольких версий программы для различных платформ.

Формат директивы `#if`:

```
#if константное_выражение
[ #elif константное_выражение
... ]
[ #elif константное_выражение
... ]
[ #else
... ]
#endif
```

Количество директив `#elif` — произвольное. Исключаемые блоки кода могут содержать как описания, так и исполняемые операторы. Пример условного включения различных версий заголовочного файла:

```
#if VERSION == 1
    #define INCFILE "vers1.h"
#elif VERSION == 2
    #define INCFILE "vers2.h" /* и так далее */
#else
    #define INCFILE "versN.h"
#endif
#include INCFILE
```

В константных выражениях может использоваться проверка, определена ли константа, с помощью `defined(имя_константы)`, например:

```
#if defined(__BORLANDC__) && __BORLANDC__ == 0x530 // BC5.3:
typedef istream_iterator<int, char, char_traits<char>, ptrdiff_t>
istream_iter;
#elif defined(__BORLAND__) // BC5.2:
typedef istream_iterator<int, ptrdiff_t> istream_iter;
#else // VC5.0:
typedef istream_iterator<int> istream_iter;
#endif
```

Другое назначение директивы — временно закомментировать фрагменты кода, например:

```
#if 0
    int i, j;
    double x, y;
#endif
```


Поскольку допускается вложенность директив, такой способ весьма удобен.

Наиболее часто в программах используются директивы `#ifdef` и `#ifndef`, позволяющие управлять компиляцией в зависимости от того, определен ли с помощью директивы `#define` указанный в них символ (хотя бы как пустая строка, например, `#define 32_BIT_SUPPORT`):

```
#ifdef символ
// Расположенный ниже код компилируется, если символ определен
#endif символ
// Расположенный ниже код компилируется, если символ не определен
```

Действие этих директив распространяется до первого `#elif`, `#else` или `#endif`.

Директива `#ifndef` часто применяется для того, чтобы обеспечить включение заголовочного файла только один раз:

```
#ifndef HEADER_INCLUDED
#include "myheader.h"
#define HEADER_INCLUDED
#endif
```

Директива `#undef`

Директива `#undef` имя удаляет определение символа. Используется редко, например, для отключения какой-либо опции компилятора.

Предопределенные макросы

В C++ определено несколько макросов, предназначенных в основном для того, чтобы выдавать информацию о версии программы или месте возникновения ошибки.

`__cplusplus` — определен, если программа компилируется как файл C++. Многие компиляторы при обработке файла с расширением `.c` считают, что программа написана на языке C. Использование этого макроса позволяет указать, что можно использовать возможности C++:

```
#ifdef __cplusplus
// Действия, специфические для C++
#endif
```

Применяется, если требуется переносить код из C в C++ и обратно.

`__DATE__` — содержит строку с текущей датой в формате *месяц день год*, например:

```
printf(" Дата компиляции - %s \n", __DATE__);
```

`__FILE__` — содержит строку с полным именем текущего файла.

`__LINE__` — текущая строка исходного текста.

`__TIME__` — текущее время, например:

```
printf(" Ошибка в файле %s \n Время компиляции: %s\n ",
__FILE__, __TIME__);
```

Области действия и пространства имен

Каждый программный объект имеет область действия, которая определяется видом и местом его объявления. Существуют следующие категории области действия: блок, файл, функция, прототип функции, класс и поименованная область.

Блок. Идентификаторы, описанные внутри блока, являются локальными. Область действия идентификатора начинается в точке определения и заканчивается в конце блока, видимость — в пределах блока и внутренних блоков, время жизни — до выхода из блока. После выхода из блока память освобождается.

Файл. Идентификаторы, описанные вне любого блока, функции, класса или пространства имен, имеют глобальную видимость и постоянное время жизни и могут использоваться с момента их определения.

Функция. Единственными идентификаторами, имеющими такую область действия, являются метки операторов. В одной функции все метки должны различаться, но могут совпадать с метками других функций.

Прототип функции. Идентификаторы, указанные в списке параметров прототипа (объявления) функции, имеют областью действия только прототип функции.

Класс. Элементы структур, объединений и классов (за исключением статических элементов) являются видимыми лишь в пределах класса. Они образуются при создании переменной указанного типа и разрушаются при ее уничтожении.

Поименованная область. C++ позволяет явным образом задать область определения имен как часть глобальной области с помощью оператора `namespace`:

Напомню, что область видимости совпадает с областью действия за исключением ситуации, когда во вложенном блоке описана переменная с таким же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя он и входит в ее область действия. Тем не менее к этой переменной, если она глобальная, можно обратиться, используя операцию доступа к области видимости `::`. Способ обратиться к скрытой локальной переменной отсутствует.

В каждой области действия различают так называемые пространства имен. Пространство имен — область, в пределах которой идентификатор должен быть уникальным. В разных пространствах имена могут совпадать, поскольку разрешение ссылок осуществляется по контексту идентификатора в программе, например:

```
struct Node{  
    int Node;  
    int i;  
}Node;
```

В данном случае противоречия нет, поскольку имена типа, переменной и элемента структуры относятся к разным пространствам. В C++ определено четыре отдельных класса идентификаторов, в пределах каждого из которых имена должны быть уникальными.

□ К одному пространству имен относятся имена переменных, функций, типов, определенных пользователем (`typedef`) и констант перечислений (с. 66) в пре-

делах одной области видимости. Все они, кроме имен функций, могут быть переопределены во вложенных блоках.

- ❑ Другой класс имен образуют имена типов перечислений, структур, классов и объединений. Каждое имя должно отличаться от имен других типов в той же области видимости.
- ❑ Отдельный класс составляют элементы каждой структуры, класса и объединения. Имя элемента должно быть уникально внутри структуры, но может совпадать с именами элементов других структур.
- ❑ Метки образуют отдельное пространство имен.

Внешние объявления

Любая функция автоматически видна во всех модулях программы. Если требуется ограничить область действия функции файлом, в котором она описана, используется модификатор `static`.

Для того чтобы сделать доступной в нескольких модулях переменную или константу, необходимо:

- ❑ определить ее ровно в одном модуле как глобальную;
- ❑ в других модулях объявить ее как внешнюю с помощью модификатора `extern`.

Другой способ — поместить это объявление в заголовочный файл и включить его в нужные модули¹.

Все описания одной и той же переменной должны быть согласованы.

Пример описания двух глобальных переменных в файлах `one.cpp` и `two.cpp` с помощью заголовочного файла `my_header.h`:

```
// my_header.h — внешние объявления
extern int a;
extern double b;

// -----
// one.cpp
#include "my_header.h"
int a;

// -----
// two.cpp
#include "my_header.h"
double b;
```

Обе переменные доступны в файлах `one.cpp` и `two.cpp`.

¹ Напомню, что объявление, в отличие от определения, не создает переменную. Объявление с `extern` не должно содержать инициализацию: если она присутствует, модификатор `extern` игнорируется.

Если переменная описана как `static`, область ее действия ограничивается файлом, в котором она описана.

При *описании типа* следует придерживаться *правила одного определения*, то есть тип, используемый в программе, должен быть определен ровно один раз. Как правило, это делается в заголовочном файле, который затем подключается к модулям, использующим этот тип. Нарушение этого правила приводит к ошибкам, которые трудно обнаружить, поскольку компиляторы, как правило, не обладают возможностью сличать определения одного и того же типа в различных файлах.

Поименованные области

Поименованные области служат для логического группирования объявлений и ограничения доступа к ним. Чем больше программа, тем более актуально использование поименованных областей. Простейшим примером применения является отделение кода, написанного одним человеком, от кода, написанного другим. При использовании единственной глобальной области видимости формировать программу из отдельных частей очень сложно из-за возможного совпадения и конфликта имен. Использование поименованных областей препятствует доступу к ненужным средствам.

Объявление поименованной области (ее также называют пространством имен) имеет формат:

```
namespace [ имя_области ] { /* Объявления */ }
```

Поименованная область может объявляться неоднократно, причем последующие объявления рассматриваются как расширения предыдущих. Таким образом, поименованная область может объявляться и изменяться за рамками одного файла.

Если имя области не задано, компилятор определяет его самостоятельно с помощью уникального идентификатора, различного для каждого модуля. Объявление объекта в непоименованной области равнозначно его описанию как глобального с модификатором `static`. Помещать объявления в такую область полезно для того, чтобы сохранить локальность кода. Нельзя получить доступ из одного файла к элементу неименованной области другого файла.

Пример.

```
namespace demo{
    int i = 1;
    int k = 0;
    void func1(int);
    void func2(int) { /* ... */ }
}

namespace demo{           // Расширение
    // int i = 2;           Неверно – двойное определение
    void func1(double);     // Перегрузка
    void func2(int);        // Верно (повторное объявление)
}
```

В объявлении поименованной области могут присутствовать как объявления, так и определения. Логично помещать в нее только объявления, а определять их

позднее с помощью имени области и оператора доступа к области видимости `::`, например:

```
void demo::func1(int) { /* ... */ }
```

Это применяется для разделения интерфейса и реализации. Таким способом нельзя объявить новый элемент пространства имен.

Объекты, объявленные внутри области, являются видимыми с момента объявления. К ним можно явно обращаться с помощью имени области и оператора доступа к области видимости `::`, например:

```
demo::i = 100; demo::func2(10);
```

Если имя часто используется вне своего пространства, можно объявить его доступным с помощью *оператора* `using`:

```
using demo::i;
```

После этого можно использовать имя без явного указания области.

Если требуется сделать доступными все имена из какой-либо области, используется оператор `using namespace`:

```
using namespace demo;
```

Операторы `using` и `using namespace` можно использовать и внутри объявления поименованной области, чтобы сделать в ней доступными объявления из другой области:

```
namespace Department_of_Applied_Mathematics{
    using demo::i;
    // ...
}
```

Имена, объявленные в поименованной области явно или с помощью оператора `using`, имеют приоритет по отношению к именам, объявленным с помощью оператора `using namespace` (это имеет значение при включении нескольких поименованных областей, содержащих совпадающие имена).

Короткие имена пространств имен могут войти в конфликт друг с другом, а длинные непрактичны при написании реального кода, поэтому допускается вводить синонимы имен:

```
namespace DAM = Department_of_Applied_Mathematics;
```

Пространства имен стандартной библиотеки. Объекты стандартной библиотеки определены в пространстве имен `std`. Например, объявления стандартных средств ввода/вывода C в заголовочном файле `<stdio.h>` помещены в пространство имен следующим образом:

```
// stdio.h
namespace std{
    int feof(FILE *f);
    ...
}
using namespace std;
```

Это обеспечивает совместимость сверху вниз. Для тех, кто не желает присутствия неявно доступных имен, определен новый заголовочный файл `<cstdio>`:

```
// cstdio
namespace std{
int feof(FILE *f);
...
}
```

Если в программу включен файл `<cstdio>`, нужно указывать имя пространства имен явным образом:

```
std::feof(j);
```

Механизм пространств имен вместе с директивой `#include` обеспечивают необходимую при написании больших программ гибкость путем сочетания логического группирования связанных величин и ограничения доступа.

Как правило, в любом функционально законченном фрагменте программы можно выделить интерфейсную часть (например, заголовки функций, описания типов), необходимую для использования этого фрагмента, и часть реализации, то есть вспомогательные переменные, функции и другие средства, доступ к которым извне не требуется. Пространства имен позволяют скрыть детали реализации и, следовательно, упростить структуру программы и уменьшить количество потенциальных ошибок. Продуманное разбиение программы на модули, четкая спецификация интерфейсов и ограничение доступа позволяют организовать эффективную работу над проектом группы программистов.

ГЛАВА 3

Технология создания программ

Мы изучили достаточное количество средств языка C++ (большей частью относящихся к подмножеству языка C), и пришло время обсудить, какими же принципами надо руководствоваться, чтобы получить в результате написания программы не картонный домик, готовый рухнуть от любого неосторожного движения, а красивую и надежную конструкцию.

В этой главе три раздела. В первом приведены рекомендации, которые будут полезны при создании даже самых простых программ. Они касаются в основном записи алгоритмов на языке C++ и документирования программ. Технология проектирования и тестирования более крупных программ рассмотрена во втором разделе (с. 109). Третий раздел посвящен рассмотрению динамических структур данных, которые широко используются для создания эффективных процедур обработки данных.

Кодирование и документирование программы

С приобретением опыта программист вырабатывает собственные правила и стиль. При этом полезно учиться не только на собственных неудачах, и разумное следование приведенным ниже рекомендациям поможет избежать многих распространенных ошибок. Конечно, на все случаи жизни советы дать невозможно, ведь не зря программирование, особенно на заре его развития, считалось искусством.

Главная цель, к которой нужно стремиться, — получить легко читаемую программу возможно более простой структуры. Широко известен верный в большинстве случаев девиз — «Better simpler than clever» (лучше по-простому, чем по-умному). В конечном счете все технологии программирования направлены на достижение именно этой цели, поскольку только так можно добиться надежно-

сти и простоты модификации программы¹. Смысл девиза заключается в том, что если какое-либо действие можно запрограммировать разными способами, то предпочтение должно отдаваться не наиболее компактному и даже не наиболее эффективному, а такому, который легче для понимания. Особенно это важно в том случае, когда пишут программу одни, а сопровождают другие, что является широко распространенной практикой.

Если вы еще не прониклись приведенным выше тезисом, попробуйте разобраться в приведенной ниже программе. Несмотря на то, что на первый взгляд она кажется полнейшей абракадаброй, это программа на C, и она работает:

```
#include <stdio.h>
main (int t, int _, char *a){return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)): 1,t<_ ? main(t+1,_,a):3,main(-94,-27+t,a)&&t==2\
?_<13?main(2,_,+1,"%s %d %d\n"):9:16:t<0?t<-72?
main(,t,,"@n'+,#'/*s{}w+/w#cdnr/+,.{r/*de}+,/*{*,./w{%+,./w#q#n+./#{l+./n\
{n+./+n+./# :#q#n+./+k#;*,./'r : 'd* '3,}{w+K w'K: '+'}e#':dq# 'l q# 'd'K#!\
/+k#;q# 'r}eKK#}w' r}eKK{n l} ' /#:#q#n' )}{n l} ' /+n' :d}rw' i: # }{n l}!\
/n{n#'; r{#w' r nc{n l} ' /#{l.+ 'K {rw' iK{:[{n l} ' /w#q#n' wk nw' iwk{KK{n l}!\
w{%'l##w# ' i: :{n l} ' /{*q# 'ld;r'}{n lwb!/*de}'c :;{n l' -{ }rw' ' /+ .}## ' *}\
#nc.' ,#nw}' /+kd'+e}+;# 'rdq#w! nr' / ' ) }+}{rl# ' {n' ' )# }'+}## (!! /")
:t<-50?_==*a?putchar(31[a]):
main(-65,_,a+1):
main(((*a=='/')+t,_,a+1):
0<t?main(2,2,"%s")
:*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{ }:\nuwloca-0;m .vpbks,fxntdCeghiry"
),a+1);
}
```

Для тех, кто поленился набрать приведенный выше текст на клавиатуре, скомпилировать его и запустить программу на выполнение, приведу фрагмент ее вывода:

On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

...

On the twelfth day of Christmas my true love gave to me

¹ Может показаться странным, зачем с самого начала думать о модификации программы. Но ведь для любого сколько-нибудь удачного коммерческого продукта выход новых версий — единственный способ не потерять актуальность среди пользователей.

twelve drummers drumming, eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings:
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

Первый шаг в написании программы — *записать на естественном языке* (возможно, с применением обобщенных блок-схем), что именно и как она должна делать. Если вы не можете сформулировать алгоритм по-русски, велика вероятность того, что он плохо продуман (естественно, я не имею в виду, что надо «проговаривать» все на уровне отдельных операторов, например, «изменяя индекс от 1 до 100 с шагом 1...»). Такое описание алгоритма полезно по нескольким причинам: оно помогает в деталях продумать алгоритм, найти на самой ранней стадии некоторые ошибки, разбить программу на логическую последовательность блоков, а также обеспечить комментарии к программе.

Если алгоритм можно разбить на последовательность законченных действий (а к этому надо стремиться!), каждое законченное действие *оформляется в виде функции*. Каждая функция должна решать только одну задачу (не надо объединять два коротких независимых фрагмента в одну функцию). Размер функции может варьироваться в широких пределах, все зависит от того, какой размер имеет законченный фрагмент кода, выделяемый в функцию. Желательно, чтобы тело функции помещалось на 1–2 экрана: одинаково сложно разбираться в программе, содержащей несколько необъятных функций, и в россыпи из сотен подпрограмм по несколько строк каждая.

Если некоторые действия встречаются в теле программы хотя бы дважды, их также нужно оформить в виде функции. Однотипные действия оформляются в виде перегруженных функций или функций с параметрами. Короткие функции имеют смысл объявить с директивой *inline*.

Необходимо *тщательно выбирать имена переменных*. Правильно выбранные имена могут сделать программу в некоторой степени самодокументированной. Неудачные имена, наоборот, служат источником проблем. Не увлекайтесь сокращениями, они ухудшают читаемость, и часто можно забыть, как именно было сокращено то или иное слово. Общая тенденция состоит в том, что чем больше область видимости переменной, тем более длинное у нее имя. Перед таким именем часто ставится префикс типа (одна или несколько букв, по которым можно определить тип переменной). Для счетчиков коротких циклов, напротив, лучше обойтись однобуквенными именами типа *i*, *j* или *k*. Имена макросов предпочтительнее записывать заглавными буквами, чтобы отличать их от других объектов программы. Не рекомендуется использовать имена, начинающиеся с символа подчеркивания, имена типов, оканчивающиеся на *_t*, а также идентификаторы, совпадающие с именами ресурсов стандартной библиотеки C++.

Переменные желательно инициализировать при их объявлении, а объявлять как можно ближе к месту их непосредственного использования. С другой стороны, удобно все объявления локальных переменных функции располагать в начале блока так, чтобы их было просто найти. При небольших размерах функций оба эти пожелания достаточно легко совместить.

Локальные переменные предпочтительнее глобальных. Если глобальная переменная все же необходима, лучше объявить ее статической, что ограничит область ее действия одним исходным файлом.

Всю необходимую функции информацию нужно стремиться передавать ей в качестве параметров, а не через глобальные переменные, изменение которых трудно отследить.

Входные параметры функции, которые не должны в ней изменяться, следует передавать как константные ссылки, а не по значению. Кроме улучшения читаемости программы и уменьшения возможности случайных ошибок, этот способ гораздо более эффективен, особенно в случае передачи сложных объектов. Исключения составляют параметры, размер которых меньше размера указателя — их эффективнее передавать по значению (не забывая указывать `const`).

Выходные параметры функции предпочтительнее передавать по адресу, а не по ссылке, чтобы из семантики вызова функции можно было понять, что внутри нее параметр изменяется.

Нельзя возвращать из функции ссылку на локальную переменную, потому что она автоматически уничтожается при выходе из функции, которая является ее областью действия. Не рекомендуется возвращать ссылку на переменную, созданную внутри функции в динамической области памяти с помощью функции `malloc` или операции `new`, так как это приводит к трудно контролируемым утечкам памяти.

Следует избегать использования в программе чисел в явном виде. Константы должны иметь осмысленные имена, заданные через `const` или `enum` (последнее предпочтительнее, так как память под перечисление не выделяется). Символическое имя делает программу более понятной, а кроме того, при необходимости изменить значение константы это можно сделать всего лишь в одном месте программы.

Для записи каждого фрагмента алгоритма *необходимо использовать наиболее подходящие средства языка.* Любой цикл можно, в принципе, реализовать с помощью операторов `goto` и `if`, но это было бы нелепо, поскольку с помощью операторов цикла те же действия легче читаются, а компилятор генерирует более эффективный код. Ветвление на несколько направлений предпочтительнее реализовывать с помощью оператора `switch`, а не нескольких `if`, красивый способ передачи управления одной функции из группы — массив указателей на функции.

Следует избегать лишних проверок условий. Например, вместо операторов

```
if      (strstr(a, b) > 0 ) { ... }  
else if (strstr(a, b) < 0 ) { ... }  
else if (strstr(a, b) == 0) { ... }
```

лучше написать

```
int is_equal = strstr(a, b);  
if      (is_equal > 0) { ... }  
else if (is_equal < 0) { ... }  
else    { ... } // здесь is_equal == 0
```

Если первая ветвь оператора `if` содержит передачу управления, использовать `else` нет необходимости:

```

if (is_equal > 0) { ... break;}
if (is_equal < 0) { ... return;}
{ ... } // здесь is_equal == 0

```

Бессмысленно использовать проверку на неравенство нулю (или, что еще хуже, на равенство true или false):

```

bool is_busy;
...
if (is_busy == true) { ... } // плохо! Лучше if (is_busy)
if (is_busy == false) { ... } // плохо! Лучше if (!is_busy)
char s[80];
while (fgets(s) != NULL) { ... } // плохо! Лучше while (fgets(s))
while(a == 0) { ... } // можно while(!a)

```

Если одна из ветвей условного оператора гораздо короче, чем другая, *более короткую ветвь if лучше поместить сверху*, иначе вся управляющая структура может не поместиться на экране, что затруднит отладку.

В некоторых случаях условная операция лучше условного оператора:

```

if (z) i = j; else i = k; // лучше так: i = z ? j : k;

```

При использовании циклов надо стремиться объединять инициализацию, проверку условия выхода и приращение в одном месте. Рекомендации по выбору наиболее подходящего оператора цикла были приведены на с. 49. При записи итеративных циклов (в которых для проверки условия выхода используются соотношения переменных, формирующихся в теле цикла), необходимо *предусматривать аварийный выход* по достижении заранее заданного максимального количества итераций.

Необходимо проверять коды возврата ошибок и предусматривать печать сообщений в тех точках программы, куда управление при нормальной работе программы передаваться не должно (именно это сообщение вы с большой вероятностью получите при первом же запуске программы). Например, оператор switch должен иметь слово default с обработкой ситуации по умолчанию, особенно если в нем перечислены все возможные значения переключателя.

Сообщение об ошибке должно быть информативным и подсказывать пользователю, как ее исправить. Например, при вводе неверного значения в сообщении должен быть указан допустимый диапазон.

Операции выделения и освобождения динамической памяти следует помещать в одну и ту же функцию. Утечки памяти, когда ее выделили, а освободить забыли, создают большие проблемы в программах, продолжительность работы которых не ограничена: на серверах баз данных, в операционных системах и т. д.

После написания программу следует тщательно отредактировать (для этого полезно представить себе, к примеру, что ее писал ваш злейший враг, которого вы хотите уличить в полной некомпетентности) — убрать ненужные фрагменты, сгруппировать описания, оптимизировать проверки условий и циклы, проверить, оптимально ли разбиение на функции и т. д. Подходить к написанию программы нужно таким образом, чтобы ее можно было в любой момент передать другому программисту. Полезно дать почитать свою программу кому-нибудь из друзей

или коллег (а еще лучше — врагов или завистников), и в тех местах, которые не будут им понятны без устных комментариев, внести их прямо в текст программы.

Комментарии имеют очень важное значение, поскольку программист, как ни странно, чаще читатель, чем писатель. Даже если сопровождающим программистом является автор программы, разбираться через год в плохо документированном тексте — сомнительное удовольствие.

Дальнейшие советы касаются комментариев и форматирования текста программы. Программы на C++ весьма плохо читаемы (лучше, чем программы на Perl, но хуже, чем на Pascal), и в них особенно важно придерживаться хорошего стиля при форматировании и документации.

«Программа без комментариев несопровождаема. Это... часовая бомба, а не компьютерная программа» [9]. Программа, если она используется, живет не один год, потребность в каких-то ее новых свойствах появляется сразу же после ввода в эксплуатацию, и сопровождение программы занимает гораздо больший промежуток времени, чем ее написание. Основная часть документации должна находиться в тексте программы. Хорошие комментарии написать почти так же сложно, как и хорошую программу.

Комментарии должны представлять собой правильные предложения без сокращений и со знаками препинания¹ и не должны подтверждать очевидное (комментарии в этой книге не могут служить образцом, поскольку они предназначены для обучения, а не для сопровождения). Например, бессмысленны фразы типа «вызов функции f» или «описание переменных». Если комментарий к фрагменту программы занимает несколько строк, лучше разместить его до фрагмента, чем справа от него, и выровнять по вертикали. Абзацный отступ комментария должен соответствовать отступу комментируемого блока.

```
// Комментарий, описывающий,  
// что происходит в следующем ниже  
// блоке программы.  
{ /* Непонятный  
блок  
программы */ }
```

Для разделения функций и других логически законченных фрагментов пользуйтесь пустыми строками или комментарием вида

```
//-----
```

Вложенные блоки должны иметь отступ в 3–4 символа, причем блоки одного уровня вложенности должны быть выровнены по вертикали. Желательно, чтобы закрывающая фигурная скобка находилась строго под открывающей. Форматируйте текст по столбцам везде, где это возможно:

```
#include <string.h>  
#include <stdlib.h>  
int main(){
```

¹ Совсем хорошо, если они при этом не будут содержать орфографических ошибок.


```

double      m[10];                      // комментарий
const char  s[] = "2. 38.5, 70. 0, 0, 1"; // комментарий
char        *p = s;                     // комментарий
int         i = 0;                       // комментарий
// -----
do{
    m[i++] = atof(p);
    if (i > 9) break;
}while(p = strchr(p, '.'), p++);
for( int k = 0; k<i; k++)
    printf("%5.2f ", m[k]);
return 0;
}

```

Для улучшения ясности можно отступить от правила отступов:

```

if      ( is_best      ) best();
else if ( is_bad       ) worse();
else if ( is_vovochka ) worst();

```

Помечайте конец длинного составного оператора:

```

while(1){
    while(gets(s)){
        for(i = 0; i<10; i++){
            for(j = 0; j<10; j++){
                // две страницы кода
            } // for(j = 0; j<10; j++)
        } // for(i = 0; i<10; i++)
    } // while(gets(s))
} // while(1)

```

Не следует размещать в одной строке много операторов. Как и в русском языке, *после знаков препинания должны использоваться пробелы:*

```
f=a+b; // плохо! Лучше f = a + b;
```

Очень много конкретных рекомендаций по программированию и проектированию программ на C++ содержится в книге Алена И. Голуба [9]. Массу интересного можно почерпнуть, осилив без малого 1000 страниц книги Б. Страуструпа [17]. В заключение хотелось бы привести цитату из этой книги: «Вопрос “Как писать хорошие программы на C++?” напоминает вопрос “Как писать хорошую английскую прозу?”. Есть два совета: “Знай, что хочешь сказать” и “Тренируйся. Подражай хорошему стилю”. Оба совета годятся как для C++, так и для английской прозы, и обоим одинаково сложно следовать.»

Проектирование и тестирование программы

Законы Мэрфи

Все сложнее, чем кажется.

Все тянется дольше, чем можно ожидать.

Если что-то может испортиться, оно обязательно портится.

Комментарий Каллагана к законам Мэрфи

Мэрфи был оптимистом.

Начинающие программисты, особенно студенты, часто пишут программы так: получив задание, тут же садятся за компьютер и начинают кодировать те фрагменты алгоритма, которые им удастся придумать сразу. Переменным дают первые попавшиеся имена типа *a*, *b*, *c* или другие, более отражающие словарный запас автора, чем содержание величин. Когда компьютер зависает, безжалостно убивая первый порыв энтузиазма, делается перерыв, после которого написанные фрагменты стираются, и все повторяется заново.

В процессе работы несколько раз изменяются структуры данных, функции nervно удаляются и кодируются заново, а разбиение на модули делается только тогда, когда листать программу становится утомительно. Нечего говорить, что комментарии к программе не пишутся, а ее текст никак не форматируется. Периодически высказываются сомнения в правильности работы компилятора, компьютера и операционной системы¹.

Когда программа впервые доходит до стадии выполнения, в нее вводятся произвольные значения, после чего экран на некоторое время становится объектом пристального удивленного изучения. «Работает» такая программа обычно только в бережных руках хозяина на одном наборе исходных данных, а внесение даже небольших изменений может привести автора к потере веры в себя и ненависти к процессу программирования.

Ваша задача состоит в том, чтобы научиться подходить к программированию профессионально. В конце концов, профессионал отличается тем, что может достаточно точно оценить, сколько времени у него займет написание программы, которая будет работать в полном соответствии с поставленной задачей. Кроме «ума, вкуса и терпения»², для этого требуется опыт, а также знание основных принципов, выработанных программистами в течение более чем полувека развития этой дисциплины. Даже к написанию самых простых программ нужно подходить последовательно, соблюдая определенную дисциплину.

¹ Обязательно упоминается Билл Гейтс.

² Мне очень нравится это выражение Б. Страуструпа, поэтому я повторяю его уже второй раз.

ПРИМЕЧАНИЕ

Прежде чем говорить об основных этапах создания программы, хотелось бы упомянуть о более абстрактных вещах — принципах, которыми должен руководствоваться каждый программист. Очень часто программисты упрямо считают, что их основной целью является изобретение новых изощренных алгоритмов, а не выполнение полезной работы, и напоминают этим печально известную точку зрения «магазин — для продавца, а не для покупателя». Надо помнить о том, что программист в конечном счете всегда работает для пользователя программы и является членом коллектива, который должен обеспечить создание надежной программы в установленный срок.

Структурный подход к программированию, как уже упоминалось, охватывает все стадии разработки проекта: спецификацию, проектирование, собственно программирование и тестирование. Задачи, которые при этом ставятся, — уменьшение числа возможных ошибок за счет применения только допустимых структур, возможно более раннее обнаружение ошибок и упрощение процесса их исправления. Ключевыми идеями структурного подхода являются нисходящая разработка, структурное программирование и нисходящее тестирование.

Приведенные ниже этапы создания программ рассчитаны на достаточно большие проекты, разрабатываемые коллективом программистов. Для программы небольшого объема каждый этап упрощается, но содержание и последовательность этапов не изменяются.

I этап. Создание любой программы начинается с *постановки задачи*. Изначально задача ставится в терминах предметной области, и необходимо перевести ее в термины, более близкие к программированию. Поскольку программист редко досконально разбирается в предметной области, а заказчик — в программировании (простой пример: требуется написать бухгалтерскую программу), постановка задачи может стать весьма непростым итерационным процессом. Кроме того, при постановке задачи заказчик зачастую не может четко и полно сформулировать свои требования и критерии. В качестве иллюстрации приведу карикатуру «Качели» (рис. 3.1), которая появилась в 1973 году в информационном бюллетене вычислительного центра Лондонского университета и сразу стала широко известной, поскольку очень точно отражала процесс создания программы.

Постановка задачи завершается созданием *технического задания*, а затем *внешней спецификации программы*, включающей в себя:

- ☐ описание исходных данных и результатов (типы, форматы, точность, способ передачи, ограничения)¹;
- ☐ описание задачи, реализуемой программой;
- ☐ способ обращения к программе;
- ☐ описание возможных аварийных ситуаций и ошибок пользователя.

Таким образом, программа рассматривается как черный ящик, для которого определена функция и входные и выходные данные.

II этап. *Разработка внутренних структур данных.* Большинство алгоритмов зависит от того, каким образом организованы данные, поэтому интуитивно ясно,

¹ Под типами и форматами не имеются в виду типы языка программирования.

что начинать проектирование программы надо не с алгоритмов, а с разработки структур, необходимых для представления входных, выходных и промежуточных данных. При этом принимаются во внимание многие факторы, например, ограничения на размер данных, необходимая точность, требования к быстродействию программы. Структуры данных могут быть статическими или динамическими (динамические структуры рассматриваются в следующем разделе).



Рис. 3.1. «Качели»

III этап. Проектирование (определение общей структуры и взаимодействия модулей). На этом этапе применяется *технология нисходящего проектирования* программы, основная идея которого теоретически проста: разбиение задачи на подзадачи меньшей сложности, которые можно рассматривать отдельно. При этом используется метод пошаговой детализации. Можно представить себе этот процесс так, что сначала программа пишется на языке некоторой гипотетической машины, которая способна понимать самые обобщенные действия, а затем каждое из них описывается на более низком уровне абстракции, и так далее. Очень важной на этом этапе является *спецификация интерфейсов*, то есть способов взаимодействия подзадач.

Для каждой подзадачи составляется внешняя спецификация, аналогичная приведенной выше. На этом же этапе решаются вопросы разбиения программы

на модули, главный критерий — минимизация их взаимодействия. Одна задача может реализовываться с помощью нескольких модулей и наоборот, в одном модуле может решаться несколько задач. На более низкий уровень проектирования переходят только после окончания проектирования верхнего уровня. Алгоритмы записывают в обобщенной форме — например, словесной, в виде обобщенных блок-схем или другими способами. Если возникают трудности с записью алгоритма, он, скорее всего, плохо продуман.

На этапе проектирования следует учитывать возможность будущих модификаций программы и стремиться проектировать программу таким образом, чтобы вносить изменения было возможно проще. Поскольку неизвестно, какие изменения придется выполнить, это пожелание напоминает создание «общей теории всего»; на практике надо ограничиться разумными компромиссами. Программист, исходя из своего опыта и здравого смысла, решает, какие именно свойства программы может потребоваться изменить или усовершенствовать в будущем.

Процесс проектирования является итерационным, поскольку в программах реального размера невозможно продумать все детали с первого раза.

IV этап. Структурное программирование. Процесс программирования также организуется по принципу «сверху вниз»: вначале кодируются модули самого верхнего уровня и составляются тестовые примеры для их отладки, при этом на месте еще не написанных модулей следующего уровня ставятся «заглушки» — временные программы. «Заглушки» в простейшем случае просто выдают сообщение о том, что им передано управление, а затем возвращают его в вызывающий модуль. В других случаях «заглушка» может выдавать значения, заданные заранее или вычисленные по упрощенному алгоритму.

Таким образом, сначала создается логический скелет программы, который затем обрastaет плотью кода.

Казалось бы, более логично применять к процессу программирования восходящую технологию — написать и отладить сначала модули нижнего уровня, а затем объединять их в более крупные фрагменты, но этот подход имеет ряд недостатков.

Во-первых, в процессе кодирования верхнего уровня могут быть вскрыты те или иные трудности проектирования более низких уровней программы (просто потому, что при написании программы ее логика продумывается более тщательно, чем при проектировании). Если подобная ошибка обнаруживается в последнюю очередь, требуются дополнительные затраты на переделку уже готовых модулей нижнего уровня.

Во-вторых, для отладки каждого модуля, а затем более крупных фрагментов программы требуется каждый раз составлять свои тестовые примеры, и программист часто вынужден имитировать то окружение, в котором должен работать модуль. Нисходящая же технология программирования обеспечивает естественный порядок создания тестов — возможность нисходящей отладки, которая рассмотрена далее.

Рекомендации по записи алгоритмов на C++ (большинство из этих рекомендаций справедливы и для других языков высокого уровня) приведены в предыдущем разделе. Напомню, что главные цели — читаемость и простота структуры

программы в целом и любой из составляющих ее функций. *При программировании следует отделять интерфейс (функции, модуля, класса) от его реализации и ограничивать доступ к ненужной информации.* Небрежное даже в мелочах программирование может привести к огромным затратам на поиск ошибок на этапе отладки.

Этапы проектирования и программирования совмещены во времени: в идеале сначала проектируется и кодируется верхний уровень, затем — следующий, и так далее. Такая стратегия применяется потому, что в процессе кодирования может возникнуть необходимость внести изменения, отражающиеся на модулях нижнего уровня.

V этап. Нисходящее тестирование. Этот этап записан последним, но это не значит, что тестирование не должно проводиться на предыдущих этапах. Проектирование и программирование обязательно должны сопровождаться написанием *набора тестов* — проверочных исходных данных и соответствующих им наборов эталонных реакций.

Необходимо различать процессы тестирования и отладки программы. *Тестирование* — процесс, посредством которого проверяется правильность программы. Тестирование носит позитивный характер, его цель — показать, что программа работает правильно и удовлетворяет всем проектным спецификациям. *Отладка* — процесс исправления ошибок в программе, при этом цель исправить все ошибки не ставится. Исправляют ошибки, обнаруженные при тестировании. При планировании следует учитывать, что процесс обнаружения ошибок подчиняется закону насыщения, то есть большинство ошибок обнаруживается на ранних стадиях тестирования, и чем меньше в программе осталось ошибок, тем дольше искать каждую из них.

Для исчерпывающего тестирования программы необходимо *проверить каждую из ветвей алгоритма.* Общее число ветвей определяется комбинацией всех альтернатив на каждом этапе. Это конечное число, но оно может быть очень большим, поэтому программа разбивается на фрагменты, после исчерпывающего тестирования которых они рассматриваются как элементарные узлы более длинных ветвей. Кроме данных, обеспечивающих выполнение операторов в требуемой последовательности, тесты должны содержать *проверку граничных условий* (например, переход по условию $x > 10$ должен проверяться для значений, больших, меньших и равных 10). Отдельно проверяется реакция программы на *ошибочные исходные данные.*

Идея нисходящего тестирования предполагает, что к тестированию программы приступают еще до того, как завершено ее проектирование. Это позволяет раньше опробовать основные межмодульные интерфейсы, а также убедиться в том, что программа в основном удовлетворяет требованиям пользователя. Только после того как логическое ядро испытано настолько, что появляется уверенность в правильности реализации основных интерфейсов, приступают к кодированию и тестированию следующего уровня программы.

Естественно, полное тестирование программы, пока она представлена в виде скелета, невозможно, однако добавление каждого следующего уровня позволяет постепенно расширять область тестирования.

Этап комплексной отладки на уровне системы при нисходящем проектировании занимает меньше времени, чем при восходящем, и приносит меньше сюрпризов, поскольку вероятность появления серьезных ошибок, затрагивающих большую часть системы, гораздо ниже. Кроме того, для каждого подключаемого к системе модуля уже создано его окружение, и выходные данные отлаженных модулей можно использовать как входные для тестирования других, что облегчает процесс тестирования. Это не значит, что модуль надо подключать к системе совсем «сырым» — бывает удобным провести часть тестирования автономно, поскольку сгенерировать на входе системы все варианты, необходимые для тестирования отдельного модуля, трудно.

Рассмотрение методов отладки программ выходит за рамки данной книги, поскольку при отладке активно используются средства конкретной оболочки программирования.

Динамические структуры данных

Любая программа предназначена для обработки данных, от способа организации которых зависят алгоритмы работы, поэтому выбор структур данных должен предшествовать созданию алгоритмов. Выше были рассмотрены стандартные способы организации данных, предоставляемые языком C++, — основные и составные типы. Наиболее часто в программах используются массивы, структуры и их сочетания, например, массивы структур, полями которых являются массивы и структуры.

Память под данные выделяется либо на этапе компиляции (в этом случае необходимый объем должен быть известен до начала выполнения программы, то есть задан в виде константы), либо во время выполнения программы с помощью операции `new` или функции `malloc` (необходимый объем должен быть известен до распределения памяти). В обоих случаях выделяется непрерывный участок памяти.

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память выделяется по мере необходимости отдельными блоками, связанными друг с другом с помощью указателей. Такой способ организации данных называется *динамическими структурами данных*, поскольку их размер изменяется во время выполнения программы. Из динамических структур в программах чаще всего используются *линейные списки*, *стеки*, *очереди* и *бинарные деревья*. Они различаются способами связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения задач сортировки, поскольку упорядочивание динамических структур не требует перестановки элементов, а сводится к изменению указателей на эти элементы. Например, если в процессе выполнения программы требуется многократно упорядочивать большой массив данных, имеет смысл организовать его в виде линейного списка. При

решении задач поиска элемента в тех случаях, когда важна скорость, данные лучше всего представить в виде бинарного дерева.

Элемент любой динамической структуры данных представляет собой структуру (в смысле `struct`), содержащую по крайней мере два поля: для хранения данных и для указателя. Полей данных и указателей может быть несколько. Поля данных могут быть любого типа: основного, составного или типа указатель. Описание простейшего элемента (компоненты, узла) выглядит следующим образом:

```
struct Node{  
    Data d; // тип данных Data должен быть определен ранее  
    Node *p;  
};
```

Рассмотрим реализацию основных операций с динамическими структурами данных (в дальнейшем будет приведен пример реализации списка в виде шаблона класса, см. с. 211).

Линейные списки

Самый простой способ связать множество элементов — сделать так, чтобы каждый элемент содержал ссылку на следующий. Такой список называется *однонаправленным (односвязным)*. Если добавить в каждый элемент вторую ссылку — на предыдущий элемент, получится *двунаправленный список (двусвязный)*, если последний элемент связать указателем с первым, получится *кольцевой список*.

Каждый элемент списка содержит *ключ*, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных. Например, если создается линейный список из записей, содержащих фамилию, год рождения, стаж работы и пол, любая часть записи может выступать в качестве ключа: при упорядочивании списка по алфавиту ключом будет фамилия, а при поиске, к примеру, ветеранов труда ключом будет стаж. Ключи разных элементов списка могут совпадать.

Над списками можно выполнять следующие *операции*:

- ☐ начальное формирование списка (создание первого элемента);
- ☐ добавление элемента в конец списка;
- ☐ чтение элемента с заданным ключом;
- ☐ вставка элемента в заданное место списка (до или после элемента с заданным ключом);
- ☐ удаление элемента с заданным ключом;
- ☐ упорядочивание списка по ключу.

Рассмотрим двунаправленный линейный список. Для формирования списка и работы с ним требуется иметь по крайней мере один указатель — на начало списка. Удобно завести еще один указатель — на конец списка. Для простоты допустим, что список состоит из целых чисел, то есть описание элемента списка выглядит следующим образом:

```

struct Node{
    int d;
    Node *next;
    Node *prev;
};

```

Ниже приведена программа, которая формирует список из 5 чисел, добавляет число в список, удаляет число из списка и выводит список на экран. Указатель на начало списка обозначен `pbeg`, на конец списка — `pend`, вспомогательные указатели — `pv` и `pkey`.

```

#include <iostream.h>
struct Node{
    int d;
    Node *next;
    Node *prev;
};
//-----
Node * first(int d);
void add(Node **pend, int d);
Node * find(Node * const pbeg, int i);
bool remove(Node **pbeg, Node **pend, int key);
Node * insert(Node * const pbeg, Node **pend, int key, int d);
//-----
int main(){
    Node *pbeg = first(1);    // Формирование первого элемента списка
    Node *pend = pbeg;       // Список заканчивается, едва начавшись
    // Добавление в конец списка четырех элементов 2, 3, 4, и 5:
    for (int i = 2; i<6; i++)add(&pend, i);
    // Вставка элемента 200 после элемента 2:
    insert(pbeg, &pend, 2, 200);
    // Удаление элемента 5:
    if(!remove (&pbeg, &pend, 5))cout << "не найден";
    Node *pv = pbeg;
    while (pv){               // вывод списка на экран
        cout << pv->d << ' ';
        pv = pv->next;
    }
    return 0;
}
//-----
// Формирование первого элемента
Node * first(int d){
    Node *pv = new Node;
    pv->d = d;  pv->next = 0;    pv->prev = 0;
    return pv;
}
//-----

```

```

// Добавление в конец списка
void add(Node **pend, int d){
    Node *pv = new Node;
    pv->d = d;  pv->next = 0;  pv->prev = *pend;
    (*pend)->next = pv;
    *pend = pv;
}

//-----
// Поиск элемента по ключу
Node * find(Node * const pbeg, int d){
    Node *pv = pbeg;
    while (pv){
        if(pv->d == d)break;
        pv = pv->next;
    }
    return pv;
}

//-----
// Удаление элемента
bool remove(Node **pbeg, Node **pend, int key){
    if(Node *pkey = find(*pbeg, key)){           // 1
        if (pkey == *pbeg){                       // 2
            *pbeg = (*pbeg)->next;
            (*pbeg)->prev = 0;}
        else if (pkey == *pend){                   // 3
            *pend = (*pend)->prev;
            (*pend)->next = 0;}
        else{                                       // 4
            (pkey->prev)->next = pkey->next;
            (pkey->next)->prev = pkey->prev;}
        delete pkey;
        return true;                               // 5
    }
    return false;                                  // 6
}

//-----
// Вставка элемента
Node * insert(Node * const pbeg, Node **pend, int key, int d){
    if(Node *pkey = find(pbeg, key)){
        Node *pv = new Node;
        pv->d = d;
        // 1 - установление связи нового узла с последующим:
        pv->next = pkey->next;
        // 2 - установление связи нового узла с предыдущим:
        pv->prev = pkey;
        // 3 - установление связи предыдущего узла с новым:
        pkey->next = pv;
    }
}

```

```

// 4 - установление связи последующего узла с новым:
if( pkey != *pend) (pv->next)->prev = pv;
// Обновление указателя на конец списка.
// если узел вставляется в конец:
else *pend = pv;
return pv;
}
return 0;
}

```

Результат работы программы:

```
1 2 200 3 4
```

Все параметры, не изменяемые внутри функций, должны передаваться с модификатором `const`. Указатели, которые могут измениться (например, при удалении из списка последнего элемента указатель на конец списка требуется скорректировать), передаются по адресу.

Рассмотрим подробнее *функцию удаления элемента из списка* `remove`. Ее параметрами являются указатели на начало и конец списка и ключ элемента, подлежащего удалению. В строке 1 выделяется память под локальный указатель `pkey`, которому присваивается результат выполнения функции нахождения элемента по ключу `find`. Эта функция возвращает указатель на элемент в случае успешного поиска и 0, если элемента с таким ключом в списке нет. Если `pkey` получает ненулевое значение, условие в операторе `if` становится истинным (элемент существует), и управление передается оператору 2, если нет — выполняется возврат из функции со значением `false` (оператор 6).

Удаление из списка происходит по-разному в зависимости от того, находится элемент в начале списка, в середине или в конце. В операторе 2 проверяется, находится ли удаляемый элемент в начале списка — в этом случае следует скорректировать указатель `pbeg` на начало списка так, чтобы он указывал на следующий элемент в списке, адрес которого находится в поле `next` первого элемента. Новый начальный элемент списка должен иметь в своем поле указателя на предыдущий элемент значение 0.

Если удаляемый элемент находится в конце списка (оператор 3), требуется сменить указатель `pend` конца списка на предыдущий элемент, адрес которого можно получить из поля `prev` последнего элемента. Кроме того, нужно обнулить для нового последнего элемента указатель на следующий элемент. Если удаление происходит из середины списка, то единственное, что надо сделать, — обеспечить двустороннюю связь предыдущего и последующего элементов. После корректировки указателей память из-под элемента освобождается, и функция возвращает значение `true`.

Работа *функции вставки элемента* в список проиллюстрирована на рис. 3.2. Номера около стрелок соответствуют номерам операторов в комментариях.

Сортировка связанного списка заключается в изменении связей между элементами. Алгоритм состоит в том, что исходный список просматривается, и каждый элемент вставляется в новый список на место, определяемое значением его ключа.

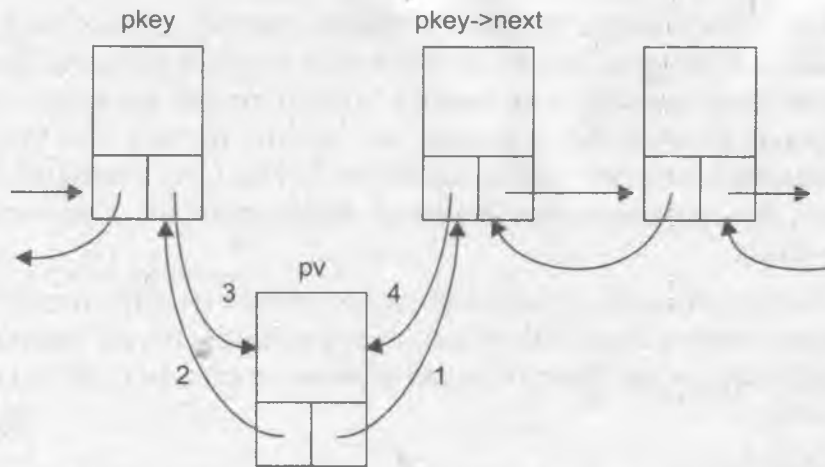


Рис. 3.2. Вставка элемента в список

Ниже приведена *функция формирования упорядоченного списка* (предполагается, что первый элемент существует):

```
void add_sort(Node **pbeg, Node **pend, int d){
    Node *pv = new Node; // добавляемый элемент
    pv->d = d;
    Node *pt = *pbeg;
    while (pt){ // просмотр списка
        if (d < pt->d){ // занести перед текущим элементом (pt)
            pv->next = pt;
            if (pt == *pbeg){ // в начало списка
                pv->prev = 0;
                *pbeg = pv;
            }
            else{ // в середину списка
                (pt->prev)->next = pv;
                pv->prev = pt->prev;
            }
            pt->prev = pv;
            return;
        }
        pt = pt->next;
    }
    pv->next = 0; // в конец списка
    pv->prev = *pend;
    (*pend)->next = pv;
    *pend = pv;
}
```

Стеки

Стек — это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO

(last in — first out, последним пришел — первым ушел). Стек проще всего представить себе как закрытую с одного конца узкую трубу, в которую бросают мячи. Достать первый брошенный мяч можно только после того, как вынуты все остальные. Кстати, сегмент стека назван так именно потому, что память под локальные переменные выделяется по принципу LIFO. Стеки широко применяются в системном программном обеспечении, компиляторах, в различных рекурсивных алгоритмах.

Ниже приведена программа, которая формирует стек из пяти целых чисел (1, 2, 3, 4, 5) и выводит его на экран. Функция помещения в стек по традиции называется `push`, а выборки — `pop`. Указатель для работы со стеком (`top`) всегда ссылается на его вершину.

```
#include <iostream.h>
struct Node{
    int d;
    Node *p;
};
Node * first(int d);
void push(Node **top, int d);
int pop(Node **top);
//-----
int main(){
    Node *top = first(1);
    for (int i = 2; i<6; i++)push(&top, i);
    while (top)
        cout << pop(&top) << ' ';
    return 0;
}
//-----
// Начальное формирование стека
Node * first(int d){
    Node *pv = new Node;
    pv->d = d;
    pv->p = 0;
    return pv;
}
//-----
// Занесение в стек
void push(Node **top, int d){
    Node *pv = new Node;
    pv->d = d;
    pv->p = *top;
    *top = pv;
}
//-----
// Выборка из стека
int pop(Node **top){
```

```

    int temp = (*top)->d;
    Node *pv = *top;
    *top = (*top)->p;
    delete pv;
    return temp;
}

```

Результат работы программы:

5 4 3 2 1

Очереди

Очередь — это частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (first in — first out, первым пришел — первым ушел). Очередь проще всего представить себе, постояв в ней час-другой. В программировании очереди применяются, например, в моделировании, диспетчеризации задач операционной системой, буферизованном вводе/выводе.

Ниже приведена программа, которая формирует очередь из пяти целых чисел и выводит ее на экран. Функция помещения в конец очереди называется add, а выборки — del. Указатель на начало очереди называется pbeg, указатель на конец — pend.

```

#include <iostream.h>
struct Node{
    int d;
    Node *p;
};
Node * first(int d);
void add(Node **pend, int d);
int del(Node **pbeg);
//-----
int main(){
    Node *pbeg = first(1);
    Node *pend = pbeg;
    for (int i = 2; i<6; i++)add(&pend, i);
    while (pbeg)
        cout << del(&pbeg) << ' ';
    return 0;
}
//-----
// Начальное формирование очереди
Node * first(int d){
    Node *pv = new Node;
    pv->d = d;
    pv->p = 0;
}

```

```

        return pv;
    }
    //-----
    // Добавление в конец
    void add(Node **pend, int d){
        Node *pv = new Node;
        pv->d = d;
        pv->p = 0;
        (*pend)->p = pv;
        *pend = pv;
    }
    //-----
    // Выборка
    int del(Node **pbeg){
        int temp = (*pbeg)->d;
        Node *pv = *pbeg;
        *pbeg = (*pbeg)->p;
        delete pv;
        return temp;
    }

```

Результат работы программы:

1 2 3 4 5

Бинарные деревья

Бинарное дерево — это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется *корнем* дерева.

На рисунке 3.3 приведен пример бинарного дерева (корень обычно изображается сверху, впрочем, книгу можно и перевернуть)¹. Узел, не имеющий поддеревьев, называется *листом*. Исходящие узлы называются предками, входящие — потомками. *Высота дерева* определяется количеством уровней, на которых располагаются его узлы.

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется *деревом поиска*. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу², двигаясь от корня и переходя

¹ Нужно не забывать о том, что в оперативной памяти ячейки расположены линейно по возрастанию адресов, а дерево — это только метод логической организации данных.

² Для так называемого *сбалансированного дерева*, в котором количество узлов справа и слева отличается не более чем на единицу, высота дерева равна двоичному логарифму количества узлов. Линейный список можно представить как вырожденное бинарное дерево, в котором каждый узел имеет не более одной ссылки. Для списка среднее время поиска равно половине длины списка.

на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму количества узлов.

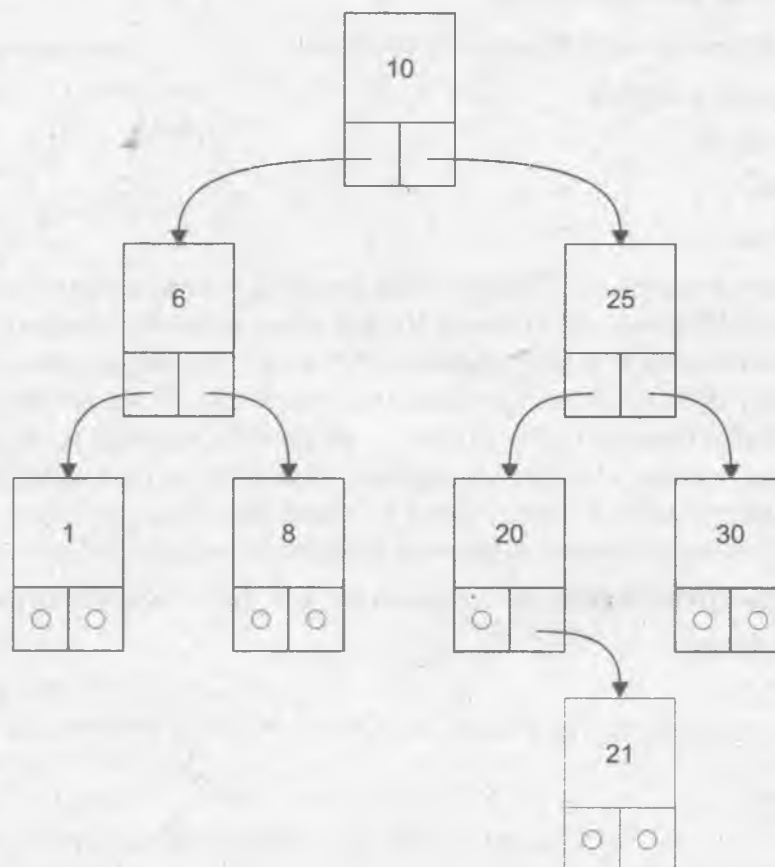


Рис. 3.3. Бинарное дерево

Дерево является рекурсивной структурой данных, поскольку каждое поддерево также является деревом. Действия с такими структурами изящнее всего описываются с помощью *рекурсивных алгоритмов*. Например, функцию обхода всех узлов дерева в общем виде можно описать так:

```

function way_around ( дерево ) {
    way_around ( левое поддерево )
    посещение корня
    way_around ( правое поддерево )
}
  
```

Можно обходить дерево и в другом порядке, например, сначала корень, потом поддерева, но приведенная функция позволяет получить на выходе отсортированную последовательность ключей, поскольку сначала посещаются вершины с меньшими ключами, расположенные в левом поддереве. Результат обхода дерева, изображенного на рис. 3.3:

1, 6, 8, 10, 20, 21, 25, 30

Если в функции обхода первое обращение идет к правому поддереву, результат обхода будет другим:

30, 25, 21, 20, 10, 8, 6, 1

Таким образом, деревья поиска можно применять для сортировки значений. При обходе дерева узлы не удаляются.

Для бинарных деревьев определены операции:

- ☐ включения узла в дерево;
- ☐ поиска по дереву;
- ☐ обхода дерева;
- ☐ удаления узла.

Для каждого рекурсивного алгоритма можно создать его нерекурсивный эквивалент. В приведенной ниже программе реализована нерекурсивная *функция поиска по дереву с включением* и рекурсивная функция обхода дерева. Первая функция осуществляет поиск элемента с заданным ключом. Если элемент найден, она возвращает указатель на него, а если нет — включает элемент в соответствующее место дерева и возвращает указатель на него. Для включения элемента необходимо помнить пройденный по дереву путь на один шаг назад и знать, выполняется ли включение нового элемента в левое или правое поддерево его предка¹.

Программа формирует дерево из массива целых чисел и выводит его на экран.

```
#include <iostream.h>
struct Node{
    int d;
    Node *left;
    Node *right;
};
Node * first(int d);
Node * search_insert(Node *root, int d);
void print_tree(Node *root, int l);
//-----
int main(){
    int b[] = {10, 25, 20, 6, 21, 8, 1, 30};
    Node *root = first(b[0]);
    for (int i = 1; i<8; i++)search_insert(root, b[i]);
    print_tree(root, 0);
    return 0;
}
//-----
// Формирование первого элемента дерева
Node * first(int d){
    Node *pv = new Node;
```

¹ «Довольно естественно испытывать некоторое недоверие к алгоритму поиска по дереву с включением» — Н. Вирт, автор классической книги «Алгоритмы + структуры данных = программы» [8].

```

    pv->d      = d;
    pv->left   = 0;
    pv->right  = 0;
    return pv;
}
//-----
// Поиск с включением
Node * search_insert(Node *root, int d){
    Node *pv = root, *prev;
    bool found = false;
    while (pv && !found){
        prev = pv;
        if      (d == pv->d) found = true;
        else if (d < pv->d) pv    = pv->left;
        else      pv    = pv->right;
    }
    if (found) return pv;
    // Создание нового узла:
    Node *pnew = new Node;
    pnew->d     = d;
    pnew->left  = 0;
    pnew->right = 0;
    if (d < prev->d)
        // Присоединение к левому поддереву предка:
        prev->left = pnew;
    else
        // Присоединение к правому поддереву предка:
        prev->right = pnew;
    return pnew;
}
//-----
// Обход дерева
void print_tree(Node *p, int level){
    if (p){
        print_tree(p->left, level + 1); // вывод левого поддерева
        for (int i = 0; i < level; i++) cout << "  ";
        cout << p->d << endl; // вывод корня поддерева
        print_tree(p->right, level + 1); // вывод правого поддерева
    }
}

```

Текущий указатель для поиска по дереву обозначен `pv`, указатель на предка `pv` обозначен `prev`, переменная `pnew` используется для выделения памяти под включаемый в дерево узел. Рекурсии удалось избежать, сохранив всего одну переменную (`prev`) и повторив при включении операторы, определяющие, к какому поддереву присоединяется новый узел.

Результат работы программы для дерева, изображенного на рис. 3.3:

```
      1
     6
    8
   10
  20
 21
25
30
```

Рассмотрим подробнее *функцию обхода дерева*. Вторым параметром в нее передается целая переменная, определяющая, на каком уровне находится узел. Корень находится на уровне 0. Дерево печатается по горизонтали так, что корень находится слева (посмотрите на результат работы программы, наклонив голову влево, и сравните с рис. 3.3). Перед значением узла для имитации структуры дерева выводится количество пробелов, пропорциональное уровню узла. Если закомментировать цикл печати пробелов, отсортированный по возрастанию массив будет выведен в столбик. Заметьте, что функция обхода дерева длиной всего в несколько строк может напечатать дерево любого размера — важно только следить, чтобы рекурсивные вызовы не переполнили стек.

Удаление узла из дерева представляет собой не такую простую задачу, поскольку удаляемый узел может быть корневым, содержать две, одну или ни одной ссылки на поддеревья. Для узлов, содержащих меньше двух ссылок, удаление тривиально. Чтобы сохранить упорядоченность дерева при удалении узла с двумя ссылками, его заменяют на узел с самым близким к нему ключом. Это может быть самый левый узел его правого поддерева или самый правый узел левого поддерева (например, чтобы удалить из дерева на рис. 3.3 узел с ключом 25, его нужно заменить на 21 или 30, узел 10 заменяется на 20 или 8, и т. д.). Реализация функции удаления из дерева оставлена читателю для самостоятельной работы.

Реализация динамических структур с помощью массивов

Операции выделения и освобождения памяти — дорогое удовольствие, поэтому если максимальный размер данных можно определить до начала использования и в процессе работы он не изменяется (например, при сортировке содержимого файла), более эффективным может оказаться однократное выделение непрерывной области памяти. Связи элементов при этом реализуются не через указатели, а через вспомогательные переменные или массивы, в которых хранятся номера элементов.

Проще всего реализовать таким образом стек. Кроме массива элементов, соответствующих типу данных стека, достаточно иметь одну переменную целого типа для хранения индекса элемента массива, являющегося вершиной стека. При помещении в стек индекс увеличивается на единицу, а при выборке — уменьшается.

Для реализации очереди требуются две переменных целого типа — для хранения индекса элементов массива, являющихся началом и концом очереди.

Для реализации линейного списка требуется вспомогательный массив целых чисел и еще одна переменная, например:

10 25 20 6 21 8 1 30 – массив данных
1 2 3 4 5 6 7 -1 – вспомогательный массив
0 – индекс первого элемента в списке

i -й элемент вспомогательного массива содержит для каждого i -го элемента массива данных индекс следующего за ним элемента. Отрицательное число используется как признак конца списка. Тот же массив после сортировки:

10 25 20 6 21 8 1 30 – массив данных
2 7 4 5 1 0 3 -1 – вспомогательный массив
6 – индекс первого элемента в списке

Для создания бинарного дерева можно использовать два вспомогательных массива (индексы вершин его правого и левого поддеревя). Отрицательное число используется как признак пустой ссылки. Например, дерево, приведенное на рис. 3.3, можно представить следующим образом:

10 25 20 6 21 8 1 30 – массив данных
3 2 -1 6 -1 -1 -1 -1 – левая ссылка
1 7 4 5 -1 -1 -1 -1 – правая ссылка

Память под такие структуры можно выделить либо на этапе компиляции, если размер можно задать константой, либо во время выполнения программы, например:

```
struct Node{  
    Data d; // тип данных Data должен быть определен ранее  
    int i;  
};  
Node spisok1[1000]; // на этапе компиляции  
Node *pspisok2 = new Node[m]; // на этапе выполнения
```

ВНИМАНИЕ

При работе с подобными структурами необходимо контролировать возможный выход индексов за границу массива.

Приведенный выше способ реализации позволяет использовать преимущества динамических структур (например, сортировать структуры из громоздких элементов данных без их физического перемещения в памяти), и при этом не расходовать время на выделение и освобождение памяти для каждого элемента данных.

Упражнения к части I

Циклические вычислительные процессы

Вариант 1

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} ax^2 + b & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x-a}{x-c} & \text{при } x > 0 \text{ и } b = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение $(A_{\text{ц}} \text{ ИЛИ } B_{\text{ц}}) \text{ И } (A_{\text{ц}} \text{ ИЛИ } C_{\text{ц}})$

не равно нулю, и целое значение в противном случае. Через $A_{\text{ц}}$, $B_{\text{ц}}$ и $C_{\text{ц}}$ обозначены целые части значений a, b, c , операции И и ИЛИ — поразрядные. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 2

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} \frac{1}{ax} - b & \text{при } x + 5 < 0 \text{ и } c = 0 \\ \frac{x-a}{x} & \text{при } x + 5 > 0 \text{ и } c \neq 0 \\ \frac{10x}{c-4} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение $(A_{\text{ц}} \text{ И } B_{\text{ц}}) \text{ ИЛИ } (B_{\text{ц}} \text{ И } C_{\text{ц}})$

не равно нулю, и целое значение в противном случае. Через $A_{\text{ц}}$, $B_{\text{ц}}$ и $C_{\text{ц}}$ обозначены целые части значений a, b, c , операции И и ИЛИ — поразрядные. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 3

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} ax^2 + bx + c & \text{при } a < 0 \text{ и } c \neq 0 \\ \frac{-a}{x - c} & \text{при } a > 0 \text{ и } c = 0 \\ a(x + c) & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение

Ац И (Вц ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И и ИЛИ — поразрядные. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 4

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} -ax - c & \text{при } c < 0 \text{ и } x \neq 0 \\ \frac{x - a}{-c} & \text{при } c > 0 \text{ и } x = 0 \\ \frac{bx}{c - a} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение

Ац ИЛИ Вц ИЛИ Сц

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операция ИЛИ — поразрядная. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 5

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} a - \frac{x}{10 + b} & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x - a}{x - c} & \text{при } x > 0 \text{ и } b = 0 \\ 3x + \frac{2}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

НЕ(Ац ИЛИ Вц) И (Ац МОД2 Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ, И, ИЛИ и МОД2 (сложение по модулю 2) — поразрядные. Значения a, b, c , Хнач., Хкон., dX ввести с клавиатуры.

Вариант 11

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX.

$$F = \begin{cases} ax^2 + \frac{b}{c} & \text{при } x < 1 \text{ и } c \neq 0 \\ \frac{x-a}{(x-c)^2} & \text{при } x > 15 \text{ и } c = 0 \\ \frac{x^2}{c^2} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац И Вц) МОД2 Сц

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И и МОД2 (сложение по модулю 2) — поразрядные. Значения a, b, c , Хнач., Хкон., dX ввести с клавиатуры.

Вариант 12

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX.

$$F = \begin{cases} ax^3 + b^2 + c & \text{при } x < 0.6 \text{ и } b + c \neq 0 \\ \frac{x-a}{x-c} & \text{при } x > 0.6 \text{ и } b + c = 0 \\ \frac{x}{c} + \frac{x}{a} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац ИЛИ Вц) И Сц

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И и ИЛИ — поразрядные. Значения a, b, c , Хнач., Хкон., dX ввести с клавиатуры.

Вариант 13

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX.

$$F = \begin{cases} ax^2 + b & \text{при } x - 1 < 0 \text{ и } b - x \neq 0 \\ \frac{x - a}{x} & \text{при } x - 1 > 0 \text{ и } b + x = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац ИЛИ Вц) МОД2 (Вц И Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И, ИЛИ и МОД2 (сложение по модулю 2) — поразрядные. Значения a, b, c , Хнач., Хкон., dX ввести с клавиатуры.

Вариант 14

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX.

$$F = \begin{cases} -ax^3 - b & \text{при } x + c < 0 \text{ и } a \neq 0 \\ \frac{x - a}{x - c} & \text{при } x + c > 0 \text{ и } a = 0 \\ \frac{x}{c} + \frac{c}{x} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац МОД2 Вц) ИЛИ (Ац МОД2 Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции ИЛИ и МОД2 (сложение по модулю 2) — поразрядные. Значения a, b, c , Хнач., Хкон., dX ввести с клавиатуры.

Вариант 15

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX.

$$F = \begin{cases} -ax^2 + b & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x}{x - c} + 5.5 & \text{при } x > 0 \text{ и } b = 0 \\ \frac{x}{-c} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение

НЕ(Ац ИЛИ Вц ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a , b , c , операции НЕ и ИЛИ — поразрядные. Значения a , b , c , Хнач., Хкон., dX ввести с клавиатуры.

Вариант 16

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX.

$$F = \begin{cases} a(x+c)^2 - b & \text{при } x = 0 \text{ и } b \neq 0 \\ \frac{x-a}{-c} & \text{при } x = 0 \text{ и } b = 0 \\ a + \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a , b , c — действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац МОД2 Вц) И НЕ(Ац ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a , b , c , операции НЕ, И, ИЛИ и МОД2 (сложение по модулю 2) — поразрядные. Значения a , b , c , Хнач., Хкон., dX ввести с клавиатуры.

Вариант 17

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX.

$$F = \begin{cases} ax^2 - cx + b & \text{при } x + 10 < 0 \text{ и } b \neq 0 \\ \frac{x-a}{x-c} & \text{при } x + 10 > 0 \text{ и } b = 0 \\ -\frac{x}{a-c} & \text{в остальных случаях} \end{cases}$$

где a , b , c — действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац ИЛИ Вц) И НЕ(Ац ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a , b , c , операции НЕ, И и ИЛИ — поразрядные. Значения a , b , c , Хнач., Хкон., dX ввести с клавиатуры.

Вариант 18

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX.

$$F = \begin{cases} ax^3 + bx^2 & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x-a}{x-c} & \text{при } x > 0 \text{ и } b = 0 \\ \frac{x+5}{c(x-10)} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение НЕ(Ац И Вц И Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ и И — поразрядные. Значения $a, b, c, \text{Хнач.}, \text{Хкон.}, dX$ ввести с клавиатуры.

Вариант 19

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX .

$$F = \begin{cases} a(x+7)^2 - b & \text{при } x < 5 \text{ и } b \neq 0 \\ \frac{x-cd}{ax} & \text{при } x > 5 \text{ и } b = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c, d — действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац МОД2 Вц) ИЛИ (Ац МОД2 Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции ИЛИ и МОД2 (сложение по модулю 2) — поразрядные. Значения $a, b, c, d, \text{Хнач.}, \text{Хкон.}, dX$ ввести с клавиатуры.

Вариант 20

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом dX .

$$F = \begin{cases} -\frac{2x-c}{cx-a} & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x-a}{x-c} & \text{при } x > 0 \text{ и } b = 0 \\ -\frac{x}{c} + \frac{-c}{2x} & \text{в остальных случаях} \end{cases}$$

где a, b, c — действительные числа.

Функция F должна принимать действительное значение, если выражение

НЕ(Ац ИЛИ Вц) И НЕ(Ац ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a , b , c , операции НЕ, И и ИЛИ — поразрядные. Значения a , b , c , Хнач., Хкон., dX ввести с клавиатуры.

Одномерные массивы

Примечание. Размерности массивов задаются именованными константами.

Вариант 1

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) сумму отрицательных элементов массива;
- 2) произведение элементов массива, расположенных между максимальным и минимальным элементами.

Упорядочить элементы массива по возрастанию.

Вариант 2

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) сумму положительных элементов массива;
- 2) произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.

Упорядочить элементы массива по убыванию.

Вариант 3

В одномерном массиве, состоящем из n целых элементов, вычислить:

- 1) произведение элементов массива с четными номерами;
- 2) сумму элементов массива, расположенных между первым и последним нулевыми элементами.

Преобразовать массив таким образом, чтобы сначала располагались все положительные элементы, а потом — все отрицательные (элементы, равные 0, считать положительными).

Вариант 4

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) сумму элементов массива с нечетными номерами;
- 2) сумму элементов массива, расположенных между первым и последним отрицательными элементами.

Сжать массив, удалив из него все элементы, модуль которых не превышает 1. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 5

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) максимальный элемент массива;

- 2) сумму элементов массива, расположенных до последнего положительного элемента.

Сжать массив, удалив из него все элементы, модуль которых находится в интервале $[a, b]$. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 6

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) минимальный элемент массива;
- 2) сумму элементов массива, расположенных между первым и последним положительными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, равные нулю, а потом — все остальные.

Вариант 7

В одномерном массиве, состоящем из n целых элементов, вычислить:

- 1) номер максимального элемента массива;
- 2) произведение элементов массива, расположенных между первым и вторым нулевыми элементами.

Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в нечетных позициях, а во второй половине — элементы, стоявшие в четных позициях.

Вариант 8

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) номер минимального элемента массива;
- 2) сумму элементов массива, расположенных между первым и вторым отрицательными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, модуль которых не превышает 1, а потом — все остальные.

Вариант 9

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) максимальный по модулю элемент массива;
- 2) сумму элементов массива, расположенных между первым и вторым положительными элементами.

Преобразовать массив таким образом, чтобы элементы, равные нулю, располагались после всех остальных.

Вариант 10

В одномерном массиве, состоящем из n целых элементов, вычислить:

- 1) минимальный по модулю элемент массива;
- 2) сумму модулей элементов массива, расположенных после первого элемента, равного нулю.

Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в четных позициях, а во второй половине — элементы, стоявшие в нечетных позициях.

Вариант 11

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) номер минимального по модулю элемента массива;
- 2) сумму модулей элементов массива, расположенных после первого отрицательного элемента.

Сжать массив, удалив из него все элементы, величина которых находится в интервале $[a, b]$. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 12

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) номер максимального по модулю элемента массива;
- 2) сумму элементов массива, расположенных после первого положительного элемента.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых лежит в интервале $[a, b]$, а потом — все остальные.

Вариант 13

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество элементов массива, лежащих в диапазоне от A до B ;
- 2) сумму элементов массива, расположенных после максимального элемента.

Упорядочить элементы массива по убыванию модулей элементов.

Вариант 14

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество элементов массива, равных 0;
- 2) сумму элементов массива, расположенных после минимального элемента.

Упорядочить элементы массива по возрастанию модулей элементов.

Вариант 15

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество элементов массива, больших C ;
- 2) произведение элементов массива, расположенных после максимального по модулю элемента.

Преобразовать массив таким образом, чтобы сначала располагались все отрицательные элементы, а потом — все положительные (элементы, равные 0, считать положительными).

Вариант 16

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество отрицательных элементов массива;

- 2) сумму модулей элементов массива, расположенных после минимального по модулю элемента.

Заменить все отрицательные элементы массива их квадратами и упорядочить элементы массива по возрастанию.

Вариант 17

В одномерном массиве, состоящем из n целых элементов, вычислить:

- 1) количество положительных элементов массива;
- 2) сумму элементов массива, расположенных после последнего элемента, равного нулю.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых не превышает 1, а потом — все остальные.

Вариант 18

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество элементов массива, меньших C ;
- 2) сумму целых частей элементов массива, расположенных после последнего отрицательного элемента.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, отличающиеся от максимального не более чем на 20%, а потом — все остальные.

Вариант 19

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) произведение отрицательных элементов массива;
- 2) сумму положительных элементов массива, расположенных до максимального элемента.

Изменить порядок следования элементов в массиве на обратный.

Вариант 20

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) произведение положительных элементов массива;
- 2) сумму элементов массива, расположенных до минимального элемента.

Упорядочить по возрастанию отдельно элементы, стоящие на четных местах, и элементы, стоящие на нечетных местах.

Двумерные массивы

Вариант 1

Дана целочисленная прямоугольная матрица. Определить:

- 1) количество строк, не содержащих ни одного нулевого элемента;
- 2) максимальное из чисел, встречающихся в заданной матрице более одного раза.

Вариант 2

Дана целочисленная прямоугольная матрица. Определить количество столбцов, не содержащих ни одного нулевого элемента.

Характеристикой строки целочисленной матрицы назовем сумму ее положительных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с ростом характеристик.

Вариант 3

Дана целочисленная прямоугольная матрица. Определить:

- 1) количество столбцов, содержащих хотя бы один нулевой элемент;
- 2) номер строки, в которой находится самая длинная серия одинаковых элементов.

Вариант 4

Дана целочисленная квадратная матрица. Определить:

- 1) произведение элементов в тех строках, которые не содержат отрицательных элементов;
- 2) максимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

Вариант 5

Дана целочисленная квадратная матрица. Определить:

- 1) сумму элементов в тех столбцах, которые не содержат отрицательных элементов;
- 2) минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали матрицы.

Вариант 6

Дана целочисленная прямоугольная матрица. Определить:

- 1) сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент;
- 2) номера строк и столбцов всех седловых точек матрицы.

ПРИМЕЧАНИЕ

Матрица A имеет седловую точку A_{ij} , если A_{ij} является минимальным элементом в i -й строке и максимальным в j -м столбце.

Вариант 7

Для заданной матрицы размером 8 на 8 найти такие k , что k -я строка матрицы совпадает с k -м столбцом.

Найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

Вариант 8

Характеристикой столбца целочисленной матрицы назовем сумму модулей отрицательных нечетных элементов. Переставляя столбцы заданной матрицы, расположить их в соответствии с ростом характеристик.

Найти сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.

Вариант 9

Соседями элемента A_{ij} в матрице назовем элементы A_{kl} с $i-1 \leq k \leq i+1$, $j-1 \leq l \leq j+1$, $(k, l) \neq (i, j)$. Операция сглаживания матрицы дает новую матрицу того же размера, каждый элемент которой получается как среднее арифметическое имеющихся соседей соответствующего элемента исходной матрицы. Построить результат сглаживания заданной вещественной матрицы размером 10 на 10.

В сглаженной матрице найти сумму модулей элементов, расположенных ниже главной диагонали.

Вариант 10

Элемент матрицы называется локальным минимумом, если он строго меньше всех имеющихся у него соседей. Подсчитать количество локальных минимумов заданной матрицы размером 10 на 10.

Найти сумму модулей элементов, расположенных выше главной диагонали.

Вариант 11

Коэффициенты системы линейных уравнений заданы в виде прямоугольной матрицы. С помощью допустимых преобразований привести систему к треугольному виду.

Найти количество строк, среднее арифметическое элементов которых меньше заданной величины.

Вариант 12

Уплотнить заданную матрицу, удаляя из нее строки и столбцы, заполненные нулями. Найти номер первой из строк, содержащих хотя бы один положительный элемент.

Вариант 13

Осуществить циклический сдвиг элементов прямоугольной матрицы на n элементов вправо или вниз (в зависимости от введенного режима). n может быть больше количества элементов в строке или столбце.

Вариант 14

Осуществить циклический сдвиг элементов квадратной матрицы размерности $M \times N$ вправо на k элементов таким образом: элементы 1-й строки сдвигаются в последний столбец сверху вниз, из него — в последнюю строку справа налево, из нее — в первый столбец снизу вверх, из него — в первую строку; для остальных элементов — аналогично.

Вариант 15

Дана целочисленная прямоугольная матрица. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.

Характеристикой строки целочисленной матрицы назовем сумму ее отрицательных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с убыванием характеристик.

Вариант 16

Упорядочить строки целочисленной прямоугольной матрицы по возрастанию количества одинаковых элементов в каждой строке.

Найти номер первого из столбцов, не содержащих ни одного отрицательного элемента.

Вариант 17

Путем перестановки элементов квадратной вещественной матрицы добиться того, чтобы ее максимальный элемент находился в левом верхнем углу, следующий по величине — в позиции (2,2), следующий по величине — в позиции (3,3) и т. д., заполнив таким образом всю главную диагональ.

Найти номер первой из строк, не содержащих ни одного положительного элемента.

Вариант 18

Дана целочисленная прямоугольная матрица. Определить:

- 1) количество строк, содержащих хотя бы один нулевой элемент;
- 2) номер столбца, в которой находится самая длинная серия одинаковых элементов.

Вариант 19

Дана целочисленная квадратная матрица. Определить:

- 1) сумму элементов в тех строках, которые не содержат отрицательных элементов;
- 2) минимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

Вариант 20

Дана целочисленная прямоугольная матрица. Определить:

- 1) количество отрицательных элементов в тех строках, которые содержат хотя бы один нулевой элемент;
- 2) номера строк и столбцов всех седловых точек матрицы.

ПРИМЕЧАНИЕ

Матрица A имеет седловую точку A_{ij} , если A_{ij} является минимальным элементом в i -й строке и максимальным в j -м столбце.

Одномерные и двумерные массивы**Вариант 1**

Два выпуклых многоугольника заданы на плоскости перечислением координат вершин в порядке обхода границы. Определить площади многоугольников и проверить, лежит ли один из них строго внутри другого.

Вариант 2

Из заданного на плоскости множества точек выбрать три различные точки так, чтобы разность между площадью круга, ограниченного окружностью, проходя-

щей через эти три точки, и площадью треугольника с вершинами в этих точках была минимальной.

Вариант 3

Даны два множества точек на плоскости. Выбрать три различные точки первого множества так, чтобы круг, ограниченный окружностью, проходящей через эти три точки, содержал все точки второго множества и имел минимальную площадь.

Вариант 4

Даны два множества точек на плоскости. Выбрать четыре различные точки первого множества так, чтобы квадрат с вершинами в этих точках накрывал все точки второго множества и имел минимальную площадь.

Вариант 5

Даны два множества точек на плоскости. Выбрать три различные точки первого множества так, чтобы треугольник с вершинами в этих точках накрывал все точки второго множества и имел минимальную площадь.

Вариант 6

Даны два множества точек на плоскости. Найти радиус и центр окружности, проходящей через n ($n \geq 3$) точек первого множества и содержащей строго внутри себя равное число точек первого и второго множеств.

Вариант 7

Даны два множества точек на плоскости. Из первого множества выбрать три различные точки так, чтобы треугольник с вершинами в этих точках содержал (строго внутри себя) равное количество точек первого и второго множеств.

Вариант 8

На плоскости заданы множество точек M и круг. Выбрать из M две различные точки так, чтобы наименьшим образом различались количества точек в круге, лежащие по разные стороны от прямой, проходящей через эти точки.

Вариант 9

Дано $3n$ точек на плоскости, причем никакие три из них не лежат на одной прямой. Построить множество n треугольников с вершинами в этих точках так, чтобы никакие два треугольника не пересекались и не содержали друг друга.

Вариант 10

Выбрать три различные точки из заданного множества точек на плоскости так, чтобы была минимальной разность между количествами точек, лежащих внутри и вне треугольника с вершинами в выбранных точках.

Вариант 11

Определить радиус и центр окружности, проходящей по крайней мере через три различные точки заданного множества точек на плоскости и содержащей внутри наибольшее количество точек этого множества.

Вариант 12

На плоскости заданы множество точек A и точка d вне его. Подсчитать количество различных неупорядоченных троек точек a, b, c из A таких, что четырехугольник $abcd$ является параллелограммом.

Вариант 13

На плоскости заданы множество точек A и множество окружностей B . Найти две такие различные точки из A , что проходящая через них прямая пересекается с максимальным количеством окружностей из B .

Вариант 14

Задано множество точек на плоскости. Найти все четверки точек, являющихся вершинами квадратов. Найти квадрат, внутри которого лежит наибольшее количество точек множества.

Вариант 15

Определить радиус и центр окружности минимального радиуса, проходящей хотя бы через три различные точки заданного множества точек на плоскости.

Вариант 16

Найти три треугольника с вершинами в заданном множестве точек на плоскости так, чтобы второй треугольник лежал строго внутри первого, а третий внутри второго.

Вариант 17

Дано множество точек на плоскости. Построить все возможные треугольники с вершинами в этом множестве точек и найти среди них такой, стороны которого пересекаются с максимальным количеством треугольников.

Вариант 18

На плоскости заданы множество точек и окружность радиусом R с центром в начале координат. Построить множество всех треугольников с вершинами в заданных точках, все три стороны которых пересекаются с окружностью, и найти среди них треугольник с минимальной площадью.

Вариант 19

Подсчитать количество равносторонних треугольников с различными длинами оснований и вершинами в заданном множестве точек на плоскости и определить, пересекаются ли они.

Вариант 20

Множество попарно различных плоскостей в трехмерном пространстве задано перечислением троек точек, через которые проходит каждая из плоскостей. Выбрать максимальное подмножество попарно непараллельных плоскостей.

Структуры**Вариант 1**

1. Описать структуру с именем STUDENT, содержащую следующие поля:

- NAME — фамилия и инициалы;

- GROUP — номер группы;
 - SES — успеваемость (массив из пяти элементов).
2. Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных в массив STUD1, состоящий из десяти структур типа STUDENT; записи должны быть упорядочены по возрастанию содержимого поля GROUP;
 - вывод на дисплей фамилий и номеров групп для всех студентов, включенных в массив, если средний балл студента больше 4,0;
 - если таких студентов нет, вывести соответствующее сообщение.

Вариант 2

1. Описать структуру с именем STUDENT, содержащую следующие поля:
- NAME — фамилия и инициалы;
 - GROUP — номер группы;
 - SES — успеваемость (массив из пяти элементов).
2. Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных в массив STUD1, состоящий из десяти структур типа STUDENT; записи должны быть упорядочены по возрастанию среднего балла;
 - вывод на дисплей фамилий и номеров групп для всех студентов, имеющих оценки 4 и 5;
 - если таких студентов нет, вывести соответствующее сообщение.

Вариант 3

1. Описать структуру с именем STUDENT, содержащую следующие поля:
- NAME — фамилия и инициалы;
 - GROUP — номер группы;
 - SES — успеваемость (массив из пяти элементов).
2. Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных в массив STUD1, состоящий из десяти структур типа STUDENT; записи должны быть упорядочены по алфавиту;
 - вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2;
 - если таких студентов нет, вывести соответствующее сообщение.

Вариант 4

1. Описать структуру с именем AEROFLOT, содержащую следующие поля:
- NAZN — название пункта назначения рейса;
 - NUMR — номер рейса;
 - TIP — тип самолета.
2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив AIRPORT, состоящий из семи элементов типа AEROFLOT; записи должны быть упорядочены по возрастанию номера рейса;
- вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры;
- если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 5

1. Описать структуру с именем AEROFLOT, содержащую следующие поля:
 - NAZN — название пункта назначения рейса;
 - NUMR — номер рейса;
 - TIP — тип самолета.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив AIRPORT, состоящий из семи элементов типа AEROFLOT; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;
 - вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры;
 - если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 6

1. Описать структуру с именем WORKER, содержащую следующие поля:
 - NAME — фамилия и инициалы работника;
 - POS — название занимаемой должности;
 - YEAR — год поступления на работу.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив TABL, состоящий из десяти структур типа WORKER; записи должны быть размещены по алфавиту.
 - вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры;
 - если таких работников нет, вывести на дисплей соответствующее сообщение.

Вариант 7

1. Описать структуру с именем TRAIN, содержащую следующие поля:
 - NAZN — название пункта назначения;
 - NUMR — номер поезда;
 - TIME — время отправления.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив RASP, состоящий из восьми элементов типа TRAIN; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;

- вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени;
- если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 8

1. Описать структуру с именем TRAIN, содержащую следующие поля:
 - NAZN — название пункта назначения;
 - NUMR — номер поезда;
 - TIME — время отправления.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив RASP, состоящий из шести элементов типа TRAIN; записи должны быть упорядочены по времени отправления поезда;
 - вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры;
 - если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 9

1. Описать структуру с именем TRAIN, содержащую следующие поля:
 - NAZN — название пункта назначения;
 - NUMR — номер поезда;
 - TIME — время отправления.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив RASP, состоящий из восьми элементов типа TRAIN; записи должны быть упорядочены по номерам поездов;
 - вывод на экран информации о поезде, номер которого введен с клавиатуры;
 - если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 10

1. Описать структуру с именем MARSH, содержащую следующие поля:
 - BEGST — название начального пункта маршрута;
 - TERM — название конечного пункта маршрута;
 - NUMER — номер маршрута.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив TRAFIC, состоящий из восьми элементов типа MARSH; записи должны быть упорядочены по номерам маршрутов;
 - вывод на экран информации о маршруте, номер которого введен с клавиатуры;
 - если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 11

1. Описать структуру с именем MARSH, содержащую следующие поля:
 - BEGST — название начального пункта маршрута;
 - TERM — название конечного пункта маршрута;
 - NUMER — номер маршрута.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив TRAFIC, состоящий из восьми элементов типа MARSH; записи должны быть упорядочены по номерам маршрутов;
 - вывод на экран информации о маршрутах, которые начинаются или кончаются в пункте, название которого введено с клавиатуры;
 - если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 12

1. Описать структуру с именем NOTE, содержащую следующие поля:
 - NAME — фамилия, имя;
 - TELE — номер телефона;
 - BDAY — день рождения (массив из трех чисел).
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив BLOCKNOTE, состоящий из восьми элементов типа NOTE; записи должны быть упорядочены по датам дней рождения;
 - вывод на экран информации о человеке, номер телефона которого введен с клавиатуры;
 - если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 13

1. Описать структуру с именем NOTE, содержащую следующие поля:
 - NAME — фамилия, имя;
 - TELE — номер телефона;
 - BDAY — день рождения (массив из трех чисел).
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив BLOCKNOTE, состоящий из восьми элементов типа NOTE; записи должны быть размещены по алфавиту;
 - вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры;
 - если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 14

1. Описать структуру с именем NOTE, содержащую следующие поля:
 - NAME — фамилия, имя;

- TELE — номер телефона;
- BDAY — день рождения (массив из трех чисел).

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив BLOCKNOTE, состоящий из восьми элементов типа NOTE; записи должны быть упорядочены по трем первым цифрам номера телефона;
- вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 15

1. Описать структуру с именем ZNAK, содержащую следующие поля:

- NAME — фамилия, имя;
- ZODIAC — знак Зодиака;
- BDAY — день рождения (массив из трех чисел).

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив BOOK, состоящий из восьми элементов типа ZNAK; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 16

1. Описать структуру с именем ZNAK, содержащую следующие поля:

- NAME — фамилия, имя;
- ZODIAC — знак Зодиака;
- BDAY — день рождения (массив из трех чисел).

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив BOOK, состоящий из восьми элементов типа ZNAK; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о людях, родившихся под знаком, наименование которого введено с клавиатуры;
- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 17

1. Описать структуру с именем ZNAK, содержащую следующие поля:

- NAME — фамилия, имя;
- ZODIAC — знак Зодиака;
- BDAY — день рождения (массив из трех чисел).

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив BOOK, состоящий из восьми элементов типа ZNAK; записи должны быть упорядочены по знакам Зодиака;
- вывод на экран информации о людях, родившихся в месяц, значение которого введено с клавиатуры;
- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 18

1. Описать структуру с именем PRICE, содержащую следующие поля:
 - TOVAR — название товара;
 - MAG — название магазина, в котором продается товар;
 - STOIM — стоимость товара в руб.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив SPISOK, состоящий из восьми элементов типа PRICE; записи должны быть размещены в алфавитном порядке по названиям товаров;
 - вывод на экран информации о товаре, название которого введено с клавиатуры;
 - если таких товаров нет, выдать на дисплей соответствующее сообщение.

Вариант 19

1. Описать структуру с именем PRICE, содержащую следующие поля:
 - TOVAR — название товара;
 - MAG — название магазина, в котором продается товар;
 - STOIM — стоимость товара в руб.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив SPISOK, состоящий из восьми элементов типа PRICE; записи должны быть размещены в алфавитном порядке по названиям магазинов;
 - вывод на экран информации о товарах, продающихся в магазине, название которого введено с клавиатуры;
 - если такого магазина нет, выдать на дисплей соответствующее сообщение.

Вариант 20

1. Описать структуру с именем ORDER, содержащую следующие поля:
 - PLAT — расчетный счет плательщика;
 - POL — расчетный счет получателя;
 - SUMMA — перечисляемая сумма в руб.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив SPISOK, состоящий из восьми элементов типа ORDER; записи должны быть размещены в алфавитном порядке по расчетным счетам плательщиков;

- вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры;
- если такого расчетного счета нет, выдать на дисплей соответствующее сообщение.

Указатели

Выполнить задания из разделов «Двумерные массивы» и «Структуры», используя динамическое выделение памяти.

Простейшие функции

Выполнить упражнения из раздела «Двумерные массивы», оформив каждый пункт задания в виде функции. Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Функции и файлы

Вариант 1

Для хранения данных о цветных планшетных сканерах описать структуру вида:

```
struct scan_info{
    char model[25];      // наименование модели
    int price;           // цена
    double x_size;       // горизонтальный размер области сканирования
    double y_size;       // вертикальный размер области сканирования
    int optr;            // оптическое разрешение
    int grey;            // число градаций серого
};
```

Написать функцию, которая записывает в бинарный файл данные о сканере из приведенной структуры. Структура файла: в первых двух байтах размещается значение типа `int`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах.

Написать функцию, которая извлекает из этого файла данные о сканере в структуру типа `scan_info`. Обязательный параметр — номер требуемой записи. Функция должна возвращать нулевое значение, если чтение прошло успешно, и `-1` в противном случае.

Привести пример программы, создающей файл с данными о сканерах (данные вводятся с клавиатуры) — 6–8 записей и выводящей на дисплей данные о запрошенной записи.

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 2

Для хранения данных о цветных планшетных сканерах описать структуру вида, описанного в варианте 1.

Написать функцию, которая записывает в бинарный файл данные о сканере из приведенной структуры. Структура файла: в первых двух байтах размещается значение типа `int`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах.

Написать функцию, которая сортирует записи в описанном выше бинарном файле по одной из следующих характеристик: цена либо число градаций серого. Обязательный параметр — признак, задающий критерий сортировки.

Привести пример программы, создающей файл с данными о сканерах (данные вводятся с клавиатуры) из не менее восьми записей и осуществляющий его сортировку.

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 3

Для хранения данных о цветных планшетных сканерах описать структуру вида, описанного в варианте 1.

Написать функцию, которая записывает в бинарный файл данные о сканере из приведенной структуры. Структура файла: в первых четырех байтах размещается значение типа `long`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах.

Написать функцию, которая сортирует записи в описанном выше бинарном файле по наименованию модели сканера.

Привести пример программы, создающей файл с данными о сканерах (данные вводятся с клавиатуры) из не менее восьми записей и осуществляющий его сортировку.

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 4

Для хранения данных о цветных планшетных сканерах описать структуру вида, описанного в варианте 1.

Написать функцию, которая динамически выделяет память под массив структур (не меньше шести элементов), заполняет его данными в режиме диалога и записывает массив в бинарный файл. Структура файла: в первых двух байтах размещается значение типа `int`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах.

Написать функцию, которая извлекает данные о сканере из описанного выше бинарного файла в структуру типа `scan_info`. Обязательный параметр — номер требуемой записи. Функция должна возвращать нулевое значение, если чтение прошло успешно, и `-1` в противном случае.

Привести пример программы, создающей файл с данными о сканерах (данные вводятся с клавиатуры) из не менее восьми записей и осуществляющий вывод на дисплей данных о требуемой записи.

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 5

Для хранения данных о цветных планшетных сканерах описать структуру вида, описанного в варианте 1.

Написать функцию, которая записывает данные о сканере из приведенной структуры в требуемую позицию в бинарном файле. Структура файла: в первых двух байтах размещается значение типа `int`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах. Запись может осуществляться в любую позицию, причем если между вводимой записью и последней (или началом файла) имеются пропуски, они заполняются нулями.

Написать функцию, которая «уплотняет» описанный выше бинарный файл путем удаления из него записей, содержащих все нули.

Привести пример программы, создающей файл с данными о сканерах (данные вводятся с клавиатуры) из не менее шести записей и осуществляющий его уплотнение.

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 6

Для хранения данных о цветных планшетных сканерах описать структуру вида, описанного в варианте 1.

Написать функцию, которая динамически выделяет память под массив структур (не меньше шести элементов), заполняет его данными в режиме диалога, и записывает массив в бинарный файл. Структура файла: в первых двух байтах размещается значение типа `int`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах.

Написать функцию, которая запрашивает данные о сканере в режиме диалога и замещает записи в бинарном файле по заданному номеру. Обязательный параметр — номер замещаемой записи. Функция должна возвращать нулевое значение, если запись прошла успешно, и `-1` в противном случае.

Привести пример программы, создающей файл с данными о сканерах (данные вводятся с клавиатуры) из не менее восьми записей и осуществляющий вставку новых данных о сканере.

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 7

Для хранения данных о цветных планшетных сканерах описать структуру вида, описанного в варианте 1.

Написать функцию, которая записывает в бинарный файл данные о сканере из приведенной структуры. Структура файла: в первых двух байтах размещается значение типа `int`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах.

Написать функцию, которая вводит данные о сканере с клавиатуры в структуру типа `scan_info`, и если данные об этом сканере отсутствуют в файле, помещает со-

держимое структуры в конец файла; в противном случае выдает соответствующее сообщение.

Привести пример программы, создающей файл с данными о сканерах (данные вводятся из текстового файла) — 6–8 записей и дополняющей файл записями о 2–3 сканерах, вводимых с клавиатуры.

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 8

Для хранения данных о цветных планшетных сканерах описать структуру вида, описанного в варианте 1.

Написать функцию, которая записывает в бинарный файл данные о сканере из приведенной структуры. Структура файла: в первых двух байтах размещается значение типа `int`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах.

Написать функцию, которая вводит данные о сканере с клавиатуры в структуру типа `scan_info` и помещает ее содержимое на место первой записи в файле. Файл должен существовать. При этом, запись ранее занимавшая первую позицию, помещается на вторую, вторая запись на третью, и т. д.

Привести пример программы, создающей файл с данными о сканерах (данные вводятся из текстового файла) — 6–8 записей и дополняющей этот файл 1–2 новыми записями, вводимыми с клавиатуры.

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 9

Для хранения данных о цветных планшетных сканерах описать структуру вида, описанного в варианте 1.

Написать функцию, которая запрашивает количество сканеров, информация о которых будет вводиться, динамически выделяет память под массив структур соответствующего размера и заполняет его данными в режиме диалога (с клавиатуры). При этом имя сканера может содержать пробелы.

Написать функцию, которая записывает данный массив в создаваемый бинарный файл. Если цена сканера меньше 200, то данные об этом сканере в файл не записываются. Информация об остальных сканерах помещается в бинарный файл, причем сначала пишутся данные о всех сканерах, имя которых начинается с заглавной буквы, а затем — с прописной.

Структура файла: в первых четырех байтах размещается значение типа `long`, определяющее количество сделанных в файл записей; далее без пропусков размещаются записи о сканерах.

Привести пример программы, создающей файл с данными о сканерах и осуществляющий вывод на дисплей данных о требуемой записи (либо всех, либо по номеру).

Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Вариант 10

Для хранения данных о ноутбуках описать структуру вида:

```
struct NOTEBOOK{
    char model[21]; // наименование
    struct size{    // габаритные размеры
        float x;
        float y;
        float z;
    };
    float w;        // вес
    int price;      // цена
}
```

Написать функцию, которая читает данные о ноутбуках из файла note.txt (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта (целое) — число записей в файле; далее записи в формате структуры NOTEBOOK.

Написать программу, в которой на основе разработанных функций осуществляется чтение данных только для тех ноутбуков, частота процессора которых больше 120 МГц, и запись в бинарный файл по убыванию цены.

Вариант 11

Для хранения данных о ноутбуках описать структуру вида, описанного в варианте 10.

Написать функцию, которая читает данные о ноутбуках из файла note.txt (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта (целое) — число записей в файле; далее записи в формате структуры NOTEBOOK.

Написать программу, в которой на основе разработанных функций осуществляется чтение данных только для тех ноутбуков, объем HDD которых меньше 1 Гбайт, и запись считанных данных в бинарный файл в алфавитном порядке по наименованию.

Вариант 12

Для хранения данных о ноутбуках описать структуру вида, описанного в варианте 10.

Написать функцию, которая читает данные о ноутбуках из файла note.txt (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта (целое) — число записей в файле; далее записи в формате структуры NOTEBOOK.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, целое количество которых в одном кубическом метре не превышает 285 штук.

Вариант 13

Для хранения данных о ноутбуках описать структуру вида, описанного в варианте 10.

Написать функцию, которая читает данные о ноутбуках из файла `note.txt` (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта (целое) — число записей в файле; далее записи в формате структуры `NOTEBOOK`.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, максимальный объем ОЗУ которых не менее 40 Мбайт, отсортированных по объему.

Вариант 14

Для хранения данных о ноутбуках описать структуру вида, описанного в варианте 10.

Написать функцию, которая читает данные о ноутбуках из файла `note.txt` (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта — целое число записей в файле; далее записи в формате структуры `NOTEBOOK`.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, диагональ дисплея которых больше одиннадцати дюймов.

Вариант 15

Для хранения данных о ноутбуках описать структуру вида:

```
struct NOTEBOOK{
    struct disp_res{    // разрешающая способность дисплея
        int x;          // по горизонтали
        int y;          // по вертикали
    };
    int f;              // частота регенерации
    float d;            // размер диагонали дисплея
    int price;          // цена
    char model[21];     // наименование
}
```

Написать функцию, которая читает данные о ноутбуках из файла `note.txt` (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта — целое число записей в файле; далее записи в формате структуры `NOTEBOOK`.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, вес которых менее 7 кг, отсортированных в порядке возрастания цены.

Вариант 16

Для хранения данных о ноутбуках описать структуру вида, описанного в варианте 15.

Написать функцию, которая читает данные о ноутбуках из файла `note.txt` (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта — целое число записей в файле; далее записи в формате структуры `NOTEBOOK`.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, объем видеопамати которых 2 Мбайт, отсортированных в порядке уменьшения тактовой частоты процессора.

Вариант 17

Для хранения данных о ноутбуках описать структуру вида, описанного в варианте 15.

Написать функцию, которая читает данные о ноутбуках из файла `note.txt` (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта — целое число записей в файле; далее записи в формате структуры `NOTEBOOK`.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, объем HDD которых больше 1 Гбайт, отсортированных в порядке возрастания размера диагонали дисплея.

Вариант 18

Для хранения данных о ноутбуках описать структуру вида, описанного в варианте 15.

Написать функцию, которая читает данные о ноутбуках из файла `note.txt` (см. с. 158) в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта — целое число записей в файле; далее записи в формате структуры `NOTEBOOK`.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, тактовая частота процессора которых больше 120 МГц, отсортированных в порядке уменьшения веса.

Вариант 19

Для хранения данных о ноутбуках описать структуру вида:

```
struct NOTEBOOK{
    struct disp_res{ // разрешающая способность дисплея
        int x;       // по горизонтали
        int y;       // по вертикали
```



```

};
int f;           // частота регенерации
float d;         // размер диагонали дисплея
float hdd;       // объем диска
char model[21]; // наименование
}

```

Написать функцию, которая читает данные о ноутбуках из файла `note.txt` в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта — целое число записей в файле; далее записи в формате структуры `NOTEBOOK`.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, тактовая частота процессора которых больше 120МГц, отсортированные в порядке возрастания цены.

Вариант 20

Для хранения данных о ноутбуках описать структуру вида, описанного в варианте 19.

Написать функцию, которая читает данные о ноутбуках из файла `note.txt` в структуру приведенного вида. Написать функцию, которая записывает содержимое структуры в конец бинарного файла. Структура бинарного файла: первые два байта — целое число записей в файле; далее записи в формате структуры `NOTEBOOK`.

Написать программу, в которой на основе разработанных функций осуществляется запись в двоичный файл данных только о тех ноутбуках, цена которых больше \$3500, отсортированные в порядке возрастания тактовой частоты процессора.

Пример файла `note.txt`:

Aser Note Light	2699	5.6	02.0x11.8x08.3	100	40	10.4	1	1024x0768	60	0.774
ASW ND5123T	3489	7.2	02.3x11.8x10.1	133	32	12.1	2	1024x0768	70	1.300
ARMNote TS80CD	3699	7.2	02.0x11.5x08.8	133	64	11.3	1	1024x0768	75	1.300
AST Ascentia P50	4499	7.5	02.3x11.3x09.0	133	40	11.3	1	0800x0600	70	0.774
BSI NP8657D	2605	8.0	02.3x11.8x09.3	133	40	11.3	1	1024x0768	60	0.810
BSI NP5265A	3765	8.2	02.5x12.0x09.0	150	32	12.1	2	1024x0768	70	1.300
Dell Xpi P100SD	3459	6.0	02.3x11.0x08.8	100	40	10.3	1	1024x0768	60	0.773
Digital HiNote	4799	4.0	01.3x11.0x08.8	120	40	10.4	1	0800x0600	56	1.000
Gateway Solo S5	4499	5.6	02.0x11.9x08.8	133	40	11.3	2	1024x0768	60	0.686
Hertz Z-Optima NB	3995	8.0	02.3x11.9x09.0	150	40	11.2	2	1024x0768	75	1.000
HP OmniBook 5500	6120	7.1	02.0x11.5x09.0	133	64	11.4	1	1024x0768	75	1.300
IBM ThinkPad 560	3749	4.1	01.3x11.8x08.8	120	40	12.1	2	1024x0768	85	0.774
NEC Versa 4080H	4780	6.6	02.3x11.8x09.5	120	48	10.4	1	0800x0600	70	0.776
Polywell Poly 500	3300	7.9	02.3x11.9x09.0	120	40	10.4	1	1024x0768	72	1.000
Samsung SENS 810	3667	8.7	02.3x11.5x09.5	100	32	11.4	2	1024x0768	75	0.773
Twinhead Slimnote	2965	7.4	02.0x11.5x08.0	075	64	10.4	1	1024x0768	70	0.772

В файле `note.txt` находится текстовая информация о ноутбуках. Каждая строка содержит данные об одной модели. Данные в строке размещаются в следующих полях:

- 1 : 20 — наименование модели;
- 21 : 24 — цена в долларах (целое число);
- 26 : 28 — масса ноутбука в кг (число с десятичной точкой из четырех символов);
- 30 : 43 — габаритные размеры ноутбука в дюймах (ВЫСОТАхДЛИНАхШИРИНА — три числа с десятичной точкой (4 символа, включая точку, разделенные 'x'));
- 44 : 47 — частота процессора в МГц (целое число из трех символов);
- 49 : 50 — максимальный объем ОЗУ в мегабайтах (целое число из двух символов);
- 52 : 55 — размер диагонали дисплея в дюймах (число с десятичной точкой из четырех символов, включая точку);
- 57 — размер видеопамати в мегабайтах — целое число из одного символа;
- 59 : 67 — разрешающая способность дисплея в пикселях (два целых числа, разделенные 'x');
- 69 : 70 — частота регенерации дисплея в Гц (целое число из двух символов);
- 72 : 76 — объем HDD в гигабайтах (число с десятичной точкой из пяти символов).

Все неописанные позиции заполнены пробелами.

Функции библиотеки для работы со строками и символами

Вариант 1

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение `DAT`.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое предложение текста;
- ☐ определяет количество предложений в тексте.

Вариант 2

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение `DAT`.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;

- ☐ по нажатию произвольной клавиши поочередно выделяет каждое слово текста;
- ☐ определяет количество слов в тексте.

Вариант 3

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое слово текста, оканчивающееся на гласную букву;
- ☐ определяет количество слов в тексте, оканчивающихся на гласную букву.

Вариант 4

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов. Текст должен состоять из трех предложений (длина строки текста не должна превышать 70 символов). Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое предложение текста в последовательности 2, 1, 3.

Вариант 5

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое из слов текста, у которых первый и последний символы совпадают;
- ☐ определяет количество слов в тексте, у которых первый и последний символы совпадают.

Вариант 6

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;

- ☐ по нажатию произвольной клавиши поочередно выделяет каждое слово текста, начинающееся на гласную букву;
- ☐ определяет количество слов в тексте, начинающихся на гласную букву.

Вариант 7

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ определяет количество символов в самом длинном слове;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое слово текста, содержащее максимальное количество символов.

Вариант 8

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ определяет количество символов в самом коротком слове;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое слово текста, содержащее минимальное количество символов.

Вариант 9

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ определяет в каждом предложении текста количество символов, отличных от букв и пробела;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое предложение текста, а в выделенном предложении — поочередно все символы, отличные от букв и пробела.

Вариант 10

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ определяет количество предложений текста и количество слов в каждом предложении;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое предложение текста, а в выделенном предложении — поочередно все слова.

Вариант 11

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ определяет количество букв 'а' в последнем слове текста;
- ☐ по нажатию произвольной клавиши выделяет последнее слово текста, а в выделенном слове поочередно все буквы 'а'.

Вариант 12

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ определяет самую длинную последовательность цифр в тексте (считать, что любое количество пробелов между двумя цифрами не прерывает последовательности цифр);
- ☐ по нажатию произвольной клавиши поочередно выделяет каждую последовательность цифр, содержащую максимальное количество символов.

Вариант 13

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ определяет порядковый номер заданного слова в каждом предложении текста (заданное слово вводится с клавиатуры);
- ☐ по нажатию произвольной клавиши поочередно выделяет каждое предложение текста, а в выделенном предложении — заданное слово.

Вариант 14

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 700 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет в тексте заданное слово (заданное слово вводить с клавиатуры);
- ☐ выводит текст на экран дисплея еще раз, выкидывая из него заданное слово и удаляя лишние пробелы.

Вариант 15

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 700 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет в тексте заданные слова, которые нужно поменять местами (заданные слова вводить с клавиатуры);
- ☐ выводит текст на экран дисплея еще раз, меняя в нем местами заданные слова и удаляя лишние пробелы.

Вариант 16

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 700 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит исходный текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет в тексте заданное слово (заданное слово вводить с клавиатуры);
- ☐ выводит текст на экран дисплея еще раз, заключая заданное слово в кавычки, и поочередно выделяет заданное слово вместе с кавычками.

Вариант 17

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 700 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит исходный текст на экран дисплея;
- ☐ выводит текст на экран дисплея еще раз, вставляя в каждое предложение в качестве последнего заданное слово, введенное с клавиатуры в качестве исходных данных.
- ☐ по нажатию произвольной клавиши поочередно выделяет в тексте вставленное слово.

Вариант 18

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 500 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет в тексте лишние пробелы между словами;
- ☐ выводит текст на экран дисплея еще раз, убирая лишние пробелы между словами и начиная каждое предложение с новой строки.

Вариант 19

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 700 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ по нажатию произвольной клавиши поочередно выделяет в тексте заданное слово (заданное слово вводить с клавиатуры);
- ☐ выводит текст на экран дисплея еще раз, заменяя в заданном слове строчные буквы прописными.

Вариант 20

С помощью текстового редактора создать файл, содержащий текст, длина которого не превышает 1000 символов (длина строки текста не должна превышать 70 символов).

Имя файла должно иметь расширение DAT.

Написать программу, которая:

- ☐ выводит текст на экран дисплея;
- ☐ определяет наибольшее количество подряд идущих пробелов в тексте;
- ☐ по нажатию произвольной клавиши поочередно выделяет каждую из последовательностей пробелов максимальной длины.

Шаблоны функций

Выполнить упражнения из раздела «Одномерные массивы», оформив каждый пункт задания в виде шаблона функции. Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается. Привести примеры программ, использующих эти шаблоны для типов `int`, `float` и `double`.

Модульное программирование

Выполнить упражнения из раздела «Функции и файлы», разместив описание структуры в заголовочном файле, а определения функций и главную функцию программы — в двух отдельных файлах.

Динамические структуры данных

Вариант 1

Составить программу, которая содержит динамическую информацию о наличии автобусов в автобусном парке.

Сведения о каждом автобусе содержат:

- ☐ номер автобуса;
- ☐ фамилию и инициалы водителя;
- ☐ номер маршрута.

Программа должна обеспечивать:

- ☐ начальное формирование данных о всех автобусах в парке в виде списка;
- ☐ при выезде каждого автобуса из парка вводится номер автобуса, и программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- ☐ при въезде каждого автобуса в парк вводится номер автобуса, и программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- ☐ по запросу выдаются сведения об автобусах, находящихся в парке, или об автобусах, находящихся на маршруте.

Вариант 2

Составить программу, которая содержит текущую информацию о книгах в библиотеке.

Сведения о книгах содержат:

- ☐ номер УДК;
- ☐ фамилию и инициалы автора;
- ☐ название;
- ☐ год издания;
- ☐ количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- ☐ начальное формирование данных о всех книгах в библиотеке в виде двоичного дерева;
- ☐ добавление данных о книгах, вновь поступающих в библиотеку;
- ☐ удаление данных о списываемых книгах;
- ☐ по запросу выдаются сведения о наличии книг в библиотеке, упорядоченные по годам издания.

Вариант 3

Составить программу, которая содержит текущую информацию о заявках на авиабилеты.

Каждая заявка содержит:

- ☐ пункт назначения;
- ☐ номер рейса;
- ☐ фамилию и инициалы пассажира;
- ☐ желаемую дату вылета.

Программа должна обеспечивать:

- ☐ хранение всех заявок в виде списка;
- ☐ добавление заявок в список;
- ☐ удаление заявок;
- ☐ вывод заявок по заданному номеру рейса и дате вылета;
- ☐ вывод всех заявок.

Вариант 4

Составить программу, которая содержит текущую информацию о заявках на авиабилеты.

Каждая заявка содержит:

- ☐ пункт назначения;
- ☐ номер рейса;
- ☐ фамилию и инициалы пассажира;
- ☐ желаемую дату вылета.

Программа должна обеспечивать:

- ☐ хранение всех заявок в виде двоичного дерева;
- ☐ добавление и удаление заявок;
- ☐ по заданному номеру рейса и дате вылета вывод заявок с их последующим удалением;
- ☐ вывод всех заявок.

Вариант 5

Составить программу, которая содержит текущую информацию о книгах в библиотеке.

Сведения о книгах содержат:

- ☐ номер УДК;
- ☐ фамилию и инициалы автора;
- ☐ название;
- ☐ год издания;
- ☐ количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- ☐ начальное формирование данных о всех книгах в библиотеке в виде списка;
- ☐ при взятии каждой книги вводится номер УДК, и программа уменьшает значение количества книг на единицу или выдает сообщение о том, что требуемой книги в библиотеке нет, или требуемая книга находится на руках;
- ☐ при возвращении каждой книги вводится номер УДК, и программа увеличивает значение количества книг на единицу;
- ☐ по запросу выдаются сведения о наличии книг в библиотеке.

Вариант 6

Составить программу, которая содержит динамическую информацию о наличии автобусов в автобусном парке.

Сведения о каждом автобусе содержат:

- ☐ номер автобуса;
- ☐ фамилию и инициалы водителя;
- ☐ номер маршрута;
- ☐ признак того, где находится автобус — на маршруте или в парке.

Программа должна обеспечивать:

- ☐ начальное формирование данных о всех автобусах в виде списка;
- ☐ при выезде каждого автобуса из парка вводится номер автобуса, и программа устанавливает значение признака «автобус на маршруте»;
- ☐ при въезде каждого автобуса в парк вводится номер автобуса, и программа устанавливает значение признака «автобус в парке»;
- ☐ по запросу выдаются сведения об автобусах, находящихся в парке, или об автобусах, находящихся на маршруте.

Вариант 7

Составить программу, отыскивающую проход по лабиринту.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице.

Программа находит проход через лабиринт, двигаясь от заданного входа. После отыскания прохода программа выводит найденный путь в виде координат квадратов. Для хранения пути использовать стек.

Вариант 8

Гаражная стоянка имеет одну стояночную полосу, причем въезд и выезд находятся в одном конце полосы. Если владелец автомашины приходит забрать свой автомобиль, который не является ближайшим к выходу, то все автомашины, загораживающие проезд, удаляются, машина данного владельца выводится со стоянки, а другие машины возвращаются на стоянку в исходном порядке.

Написать программу, которая моделирует процесс прибытия и отъезда машин. Прибытие или отъезд автомашины задается командной строкой, которая содержит признак прибытия или отъезда и номер машины. Программа должна выводить сообщение при прибытии или выезде любой машины. При выезде автомашины со стоянки сообщение должно содержать число раз, которое машина удалялась со стоянки для обеспечения выезда других автомобилей.

Вариант 9

Составить программу, моделирующую заполнение гибкого магнитного диска.

Общий объем памяти на диске 360 Кбайт. Файлы имеют произвольную длину от 18 байт до 32 Кбайт. В процессе работы файлы либо записываются на диск, либо удаляются с него.

В начале работы файлы записываются подряд друг за другом. После удаления файла на диске образуется свободный участок памяти, и вновь записываемый файл либо размещается на свободном участке, либо, если файл не вмещается в свободный участок, размещается после последнего записанного файла.

В случае, когда файл превосходит длину самого большого свободного участка, выдается аварийное сообщение. Требование на запись или удаление файла задается в командной строке, которая содержит имя файла, его длину в байтах, признак записи или удаления. Программа должна выдавать по запросу сведения о занятых и свободных участках памяти на диске.

Указание: следует создать список занятых участков и список свободных участков памяти на диске.

Вариант 10

В файловой системе каталог файлов организован как линейный список.

Для каждого файла в каталоге содержатся следующие сведения:

- ☐ имя файла;
- ☐ дата создания;
- ☐ количество обращений к файлу.

Составить программу, которая обеспечивает:

- ☐ начальное формирование каталога файлов;
- ☐ вывод каталога файлов;
- ☐ удаление файлов, дата создания которых меньше заданной;
- ☐ выборку файла с наибольшим количеством обращений.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 11

Предметный указатель организован как линейный список.

Каждая компонента указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти.

Составить программу, которая обеспечивает:

- ☐ начальное формирование предметного указателя;
- ☐ вывод предметного указателя;
- ☐ вывод номеров страниц для заданного слова.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 12

Текст помощи для некоторой программы организован как линейный список.

Каждая компонента текста помощи содержит термин (слово) и текст, содержащий пояснения к этому термину. Количество строк текста, относящихся к одному термину, от одной до пяти.

Составить программу, которая обеспечивает:

- ☐ начальное формирование текста помощи;
- ☐ вывод текста помощи;
- ☐ вывод поясняющего текста для заданного термина.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 13

Картотека в бюро обмена квартир организована как линейный список.

Сведения о каждой квартире содержат:

- ☐ количество комнат;
- ☐ этаж;
- ☐ площадь;
- ☐ адрес.

Составить программу, которая обеспечивает:

- ☐ начальное формирование картотеки;
- ☐ ввод заявки на обмен;
- ☐ поиск в картотеке подходящего варианта: при равенстве количества комнат и этажа и различии площадей в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в список;
- ☐ вывод всего списка.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 14

Англо-русский словарь построен как двоичное дерево.

Каждая компонента содержит английское слово, соответствующее ему русское слово и счетчик количества обращений к данной компоненте.

Первоначально дерево формируется согласно английскому алфавиту. В процессе эксплуатации словаря при каждом обращении к компоненте в счетчик обращений добавляется единица.

Составить программу, которая:

- ☐ обеспечивает начальный ввод словаря с конкретными значениями счетчиков обращений;
- ☐ формирует новое представление словаря в виде двоичного дерева по следующему алгоритму: а) в старом словаре ищется компонента с наибольшим значением счетчика обращений; б) найденная компонента заносится в новый словарь и удаляется из старого; в) переход к п. а) до исчерпания исходного словаря;
- ☐ производит вывод исходного и нового словарей.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 15

Анкета для опроса населения содержит две группы вопросов.

Первая группа содержит сведения о респонденте:

- ☐ возраст;
- ☐ пол;
- ☐ образование (начальное, среднее, высшее).

Вторая группа содержит собственно вопрос анкеты, ответ на который либо ДА, либо НЕТ.

Составить программу, которая:

- ☐ обеспечивает начальный ввод анкет и формирует из них линейный список;
- ☐ на основе анализа анкет выдает ответы на следующие вопросы: а) сколько мужчин старше 40 лет, имеющих высшее образование, ответили ДА на вопрос анкеты; а) сколько женщин моложе 30 лет, имеющих среднее образование, ответили НЕТ на вопрос анкеты; а) сколько мужчин моложе 25 лет, имеющих начальное образование, ответили ДА на вопрос анкеты;
- ☐ производит вывод всех анкет и ответов на вопросы.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 16

Составить программу, которая содержит текущую информацию о книгах в библиотеке.

Сведения о книгах содержат:

- ☐ номер УДК;
- ☐ фамилию и инициалы автора;
- ☐ название;
- ☐ год издания;
- ☐ количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- ☐ начальное формирование данных о всех книгах в библиотеке в виде списка;
- ☐ добавление данных о книгах, вновь поступающих в библиотеку;
- ☐ удаление данных о списываемых книгах;
- ☐ по запросу выдаются сведения о наличии книг в библиотеке, упорядоченные по годам издания.

Вариант 17

На междугородной телефонной станции картотека абонентов, содержащая сведения о телефонах и их владельцах, организована как линейный список.

Составить программу, которая:

- ☐ обеспечивает начальное формирование картотеки в виде линейного списка;
- ☐ производит вывод всей картотеки;
- ☐ вводит номер телефона и время разговора;
- ☐ выводит извещение на оплату телефонного разговора.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 18

На междугородной телефонной станции картотека абонентов, содержащая сведения о телефонах и их владельцах, организована как двоичное дерево.

Составить программу, которая:

- ☐ обеспечивает начальное формирование картотеки в виде двоичного дерева;
- ☐ производит вывод всей картотеки;
- ☐ вводит номер телефона и время разговора;
- ☐ выводит извещение на оплату телефонного разговора.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 19

Автоматизированная информационная система на железнодорожном вокзале содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается:

- ☐ номер поезда;
- ☐ станция назначения;
- ☐ время отправления.

Данные в информационной системе организованы в виде линейного списка.

Составить программу, которая:

- ☐ обеспечивает первоначальный ввод данных в информационную систему и формирование линейного списка;
- ☐ производит вывод всего списка;
- ☐ вводит номер поезда и выводит все данные об этом поезде;
- ☐ вводит название станции назначения и выводит данные обо всех поездах, следующих до этой станции.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 20

Автоматизированная информационная система на железнодорожном вокзале содержит сведения об отправлении поездов дальнего следования.

Для каждого поезда указывается:

- ☐ номер поезда;
- ☐ станция назначения;
- ☐ время отправления.

Данные в информационной системе организованы в виде двоичного дерева.

Составить программу, которая:

- ☐ обеспечивает первоначальный ввод данных в информационную систему и формирование двоичного дерева;
- ☐ производит вывод всего дерева;
- ☐ вводит номер поезда и выводит все данные об этом поезде;
- ☐ вводит название станции назначения и выводит данные о всех поездах, следующих до этой станции.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

ЧАСТЬ II

Объектно-ориентированное программирование

В окончательном виде любая программа представляет собой набор инструкций процессора. Все, что написано на любом языке программирования, — более удобная, упрощенная запись этого набора инструкций, облегчающая написание, отладку и последующую модификацию программы. Чем выше уровень языка, тем в более простой форме записываются одни и те же действия. Например, для реализации цикла на языке ассемблера требуется записать последовательность инструкций, позаботившись о размещении переменных в регистрах, а в С или Паскале для этого достаточно одного оператора.

С ростом объема программы становится невозможным удерживать в памяти все детали, и становится необходимым структурировать информацию, выделять главное и отбрасывать несущественное. Этот процесс называется повышением степени абстракции программы.

Первым шагом к повышению абстракции является *использование функций*, позволяющее после написания и отладки функции отвлечься от деталей ее реализации, поскольку для вызова функции требуется знать только ее интерфейс. Если глобальные переменные не используются, интерфейс полностью определяется заголовком функции.

Следующий шаг — *описание собственных типов данных*, позволяющих структурировать и группировать информацию, представляя ее в более естественном виде. Например, можно представить с помощью одной структуры все разнородные сведения, относящиеся к одному виду товара на складе.

Для работы с собственными типами данных требуются специальные функции. Естественно сгруппировать их с описанием этих типов данных в одном месте программы, а также по возможности отделить от ее остальных частей. При этом для использования этих типов и функций не требуется полного знания того, как именно они написаны — необходимы только описания интерфейсов. *Объединение в модули* описаний типов данных и функций, предназначенных для работы

с ними, со скрыванием от пользователя модуля несущественных деталей, является дальнейшим развитием структуризации программы.

Все три описанных выше метода повышения абстракции преследуют цель упростить структуру программы, то есть *представить ее в виде меньшего количества более крупных блоков и минимизировать связи между ними*. Это позволяет управлять большим объемом информации и, следовательно, успешно отлаживать более сложные программы.

Введение понятия *класса* является естественным развитием идей модульности. В классе структуры данных и функции их обработки объединяются. Класс используется только через его интерфейс — детали реализации для пользователя класса несущественны. Идея классов отражает строение объектов реального мира — ведь каждый предмет или процесс обладает набором характеристик или отличительных черт, иными словами, свойствами и поведением. Программы часто предназначены для моделирования предметов, процессов и явлений реального мира¹, поэтому в языке программирования удобно иметь адекватный инструмент для представления моделей.

Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы. Напомню, что тип задает внутреннее представление данных в памяти компьютера, множество значений, которое могут принимать величины этого типа, а также операции и функции, применяемые к этим величинам. Все это можно задать и в классе.

Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса за интерфейсом (ведь и в реальном мире можно, например, управлять автомобилем, не имея представления о принципе внутреннего сгорания и устройстве двигателя, а пользоваться телефоном — не зная, «как идет сигнал, принципов связи и кто клал кабель»²). Интерфейсом класса являются заголовки его методов. Таким образом, класс как модель объекта реального мира является черным ящиком, замкнутым по отношению к внешнему миру.

Идея классов является основой объектно-ориентированного программирования (ООП). Основные принципы ООП были разработаны еще в языках Simula-67³ и Smalltalk, но в то время не получили широкого применения из-за трудностей освоения и низкой эффективности реализации. В C++ эти концепции реализованы эффективно, красиво и непротиворечиво, что и явилось основой успешного распространения этого языка и внедрения подобных средств в другие языки программирования.

ООП — это не просто набор новых средств, добавленных в язык (на C++ можно успешно писать и без использования ООП, и наоборот, возможно написать объ-

¹ Например, Б. Страуструп разработал C++ для моделирования телефонных коммутационных систем.

² © БГ

³ Обратите внимание на год: не так часто появляются новые идеи в программировании.

ектную по сути программу на языке, не содержащим специальных средств поддержки объектов). ООП часто называют новой парадигмой программирования.

Красивый термин «парадигма» означает набор теорий, стандартов и методов, которые совместно представляют собой способ организации знаний — иными словами, способ видения мира. В программировании этот термин используется для определения модели вычислений, то есть способа структурирования информации, организации вычислений и данных. Объектно-ориентированная программа строится в терминах объектов и их взаимосвязей.

Выбор степени абстракции определяется типом задачи, которую требуется решить. Не имеет смысла использовать сложные технологии для решения простых задач, а попытка «врукопашную» справиться со сложными проблемами обречена на провал. С другой стороны, сложные технологии требуют больших затрат времени на их освоение.

Например, если требуется напечатать письмо, для этой цели подойдет простейший текстовый редактор, имеющий минимум возможностей, которым можно за 10 минут обучить даже собственную бабушку; подготовка статьи с формулами потребует освоения более сложного текстового процессора типа Microsoft Word, а для создания рекламной брошюры с иллюстрациями лучше всего подойдет один из издательских пакетов, для овладения которым потребуется не одна неделя. Так и в программировании: идеи ООП не очень просты для понимания и, в особенности, для практического использования (их неграмотное применение приносит гораздо больше вреда, чем пользы), а освоение существующих стандартных библиотек требует времени и достаточно высокого уровня первоначальной подготовки.

Конкретные величины типа данных «класс» называются *экземплярами класса*, или *объектами*. Объекты взаимодействуют между собой, посылая и получая сообщения. *Сообщение* — это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью вызова соответствующих функций. Таким образом, с помощью ООП легко реализуется так называемая «событийно-управляемая модель», когда данные активны и управляют вызовом того или иного фрагмента программного кода¹.

ПРИМЕЧАНИЕ

Примером реализации событийно-управляемой модели может служить любая программа, управляемая с помощью меню. После запуска такая программа пассивно ожидает действий пользователя и должна уметь правильно отреагировать на любое из них. Событийная модель является противоположностью традиционной (директивной), когда код управляет данными: программа после старта предлагает пользователю выполнить некоторые действия (ввести данные, выбрать режим) в соответствии с жестко заданным алгоритмом.

¹ Событийно-управляемая модель не является частью ООП и может быть реализована и без использования объектов (пример — программирование на языке C под Windows с использованием функций API).

Основными свойствами ООП являются инкапсуляция, наследование и полиморфизм. Ниже кратко поясняется их смысл, а полное представление о них можно получить¹ после изучения этой части книги.

Объединение данных с функциями их обработки в сочетании со скрытием ненужной для использования этих данных информации называется *инкапсуляцией* (encapsulation). Эта идея не нова и применялась в структурном и модульном программировании, а в ООП получила свое логическое завершение. Инкапсуляция повышает степень абстракции программы: данные класса и реализация его функций находятся ниже уровня абстракции, и для написания программы информация о них не требуется. Кроме того, инкапсуляция позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним (например, при необходимости сменить способ хранения данных с массива на стек). Простота модификации, как уже неоднократно отмечалось, является очень важным критерием качества программы.

Инкапсуляция позволяет использовать класс в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти, а также создавать библиотеки классов для применения во многих программах.

Наследование — это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы. Выделение общих черт различных классов в один класс-предок является мощным механизмом абстракции — ведь и любая наука начинается с абстрагирования и классификации, которые помогают справиться со сложностью рассматриваемой предметной области.

Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные — на ветвях и листьях. В С++ каждый класс может иметь сколько угодно потомков и предков. Иногда предки называются надклассами или суперклассами, а потомки — подклассами или субклассами.

Третьим китом, на котором стоит ООП, является *полиморфизм* — возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы.

Понятие полиморфизма используется в С++ весьма широко. Простым примером полиморфизма может служить рассмотренная в первой части книги перегрузка функций, когда из нескольких вариантов выбирается наиболее подходящая функция по соответствию ее прототипа передаваемым параметрам. Другой пример — использование шаблонов функций (в дальнейшем мы рассмотрим и шаблоны классов), когда один и тот же код видоизменяется в соответствии с типом, переданным в качестве параметра. Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов (он будет рассмотрен на с. 205).

¹ А можно и не получить.

Благодаря тому, что программа представляется в терминах поведения объектов, при программировании используются понятия, более близкие к предметной области, следовательно, программа легче читается и понимается. Это является большим преимуществом ООП. Однако проектирование объектно-ориентированной программы представляет собой весьма сложную задачу, поскольку в процесс добавляется еще один важный этап — *разработка иерархии классов*.

Плохо спроектированная иерархия приводит к созданию сложных и запутанных программ. Другим препятствием к применению ООП является большой объем информации, которую требуется освоить, и ее значительная сложность.

Важно до начала проектирования правильно определить, требуется ли вообще применять объектно-ориентированный подход. Если в иерархии классов нет необходимости, то, как правило, достаточно ограничиться модульной технологией, при этом можно успешно использовать классы как стандартной библиотеки, так и собственной разработки. Естественно, для применения стандартных классов требуется сначала изучить необходимый синтаксис и механизмы, а затем — конкретные свойства этих классов¹.

¹ В программах, приведенных в этой книге, уже незаметным образом применялись классы стандартной библиотеки `cin` и `cout`. Использование других классов может на первый взгляд показаться не столь простым, но эффект оправдает затраченные на изучение усилия.

ГЛАВА 4

Классы

В этой главе рассматривается фундамент, без которого невозможно написать ни одну объектно-ориентированную программу — синтаксические правила описания объектов. В конце этой части мы вернемся к обсуждению принципов ООП и методов проектирования объектных программ, поскольку «только хорошее понимание идей, стоящих за свойствами языка, ведет к мастерству» (Б. Страуструп).

Описание класса

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются *полями* (по аналогии с полями структуры), а функции класса — *методами*¹. Поля и методы называются *элементами класса*. Описание класса в первом приближении выглядит так:

```
class <имя>{  
    [ private: ]  
        <описание скрытых элементов>  
    public:  
        <описание доступных элементов>  
}; // Описание заканчивается точкой с запятой
```

Спецификаторы доступа `private` и `public` управляют видимостью элементов класса. Элементы, описанные после служебного слова `private`, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию. Интерфейс класса опи-

¹ В литературе чаще встречаются термины «данные-члены» и «функции-члены», а также «компонентные данные» и «компонентные функций», но по необъяснимым причинам я предпочитаю «поля» и «методы». В тех случаях, когда из контекста понятно, что речь идет о функциях-членах класса, а не об обычных функциях, методы называются просто «функциями».

сывается после спецификатора `public`. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций `private` и `public`, порядок их следования значения не имеет.

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
- могут быть описаны с модификатором `const`, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
- могут быть описаны с модификатором `static` (об этом рассказывается в разделе «Статические поля», с. 186), но не как `auto`, `extern` и `register`.

Инициализация полей при описании не допускается.

Классы могут быть *глобальными* (объявленными вне любого блока) и *локальными* (объявленными внутри блока, например, функции или другого класса).

Ниже перечислены некоторые особенности локального класса:

- внутри локального класса можно использовать типы, статические (`static`) и внешние (`extern`) переменные, внешние функции и элементы перечислений из области, в которой он описан; запрещается использовать автоматические переменные из этой области;
- локальный класс не может иметь статических элементов;
- методы этого класса могут быть описаны только внутри класса;
- если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

В качестве примера создадим класс, моделирующий персонаж компьютерной игры. Для этого требуется задать его свойства (например, количество шупалец, силу или наличие гранатомета) и поведение. Естественно, пример будет схематичен, поскольку приводится лишь для демонстрации синтаксиса.

```
class monstr{
    int health, ammo;
public:
    monstr(int he = 100, int am = 10){ health = he; ammo = am;}
    void draw(int x, int y, int scale, int position);
    int get_health(){return health;}
    int get_ammo(){return ammo;}
};
```

В этом классе два скрытых поля — `health` и `ammo`, получить значения которых извне можно с помощью методов `get_health()` и `get_ammo()`. Доступ к полям с помощью методов в данном случае кажется искусственным усложнением, но надо учитывать, что полями реальных классов могут быть сложные динамические структуры, и получение значений их элементов не так тривиально. Кроме того, очень важной является возможность вносить в эти структуры изменения, не затрагивая интерфейс класса.

Все методы класса имеют непосредственный доступ к его скрытым полям, иными словами, тела функций класса входят в область видимости `private` элементов класса.

В приведенном классе содержится три определения методов и одно объявление (метод `draw`). Если тело метода определено внутри класса, он является *встроенным* (`inline`). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости (`::`):

```
void monstr::draw(int x, int y, int scale, int position){
    /* тело метода */
}
```

Метод можно определить как встроенный и вне класса с помощью директивы `inline` (как и для обычных функций, она носит рекомендательный характер):

```
inline int monstr::get_ammo(){
    return ammo;
}
```

В каждом классе есть хотя бы один метод, имя которого совпадает с именем класса. Он называется *конструктором* и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных. Мы подробно рассмотрим конструкторы в разделе «Конструкторы», с. 182.

Типы данных `struct` и `union` являются видами класса, разница между ними будет объяснена позже, на с. 210. Другой пример описания класса, а также пример локального класса приведен в разделе «Создание шаблонов классов» на с. 211.

Описание объектов

Конкретные переменные типа «класс» называются *экземплярами класса, или объектами*. Время жизни и видимость объектов зависит от вида и места их описания и подчиняется общим правилам C++:

```
monstr Vasia;           // Объект класса monstr с параметрами по умолчанию
monstr Super(200, 300); // Объект с явной инициализацией
monstr stado[100];      // Массив объектов с параметрами по умолчанию
monstr *beavis = new monstr(10); // Динамический объект
                               //(второй параметр задается по умолчанию)
monstr &butthead = Vasia; // Ссылка на объект
```

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор (деструкторы описаны далее на с. 188).

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операция `.` (точка) при обращении к элементу через имя объекта и операция `->` при обращении через указатель, например:

```
int n = Vasia.get_ammo();
stado[5].draw;
cout << beavis->get_health();
```

Обратиться таким образом можно только к элементам со спецификатором `public`. Получить или изменить значения элементов со спецификатором `private` можно только через обращение к соответствующим методам.

Можно создать *константный объект*, значения полей которого изменять запрещается. К нему должны применяться только константные методы:

```
class monstr{
-
int get_health() const {return health;}
};
const monstr Dead(0,0);    // Константный объект
cout << Dead.get_health();
```

Константный метод:

- объявляется с ключевым словом `const` после списка параметров;
- не может изменять значения полей класса;
- может вызывать только константные методы;
- может вызываться для любых (не только константных) объектов.

Рекомендуется описывать как константные те методы, которые предназначены для получения значений полей.

Указатель `this`

Каждый объект содержит свой экземпляр полей класса. Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов с полями именно того объекта, для которого они были вызваны. Это обеспечивается передачей в функцию скрытого параметра `this`, в котором хранится константный указатель на вызвавший функцию объект. Указатель `this` неявно используется внутри метода для ссылок на элементы объекта. В явном виде этот указатель применяется в основном для возвращения из метода указателя (`return this;`) или ссылки (`return *this;`) на вызвавший объект.

Для иллюстрации использования указателя `this` добавим в приведенный выше класс `monstr` новый метод, возвращающий ссылку на наиболее здорового (поле `health`) из двух монстров, один из которых вызывает метод, а другой передается ему в качестве параметра (метод нужно поместить в секцию `public` описания класса):

```
monstr & the_best(monstr &M){
    if( health > M.get_health()) return *this;
    return M;
```



```

}
... monstr Vasia(50), Super(200);
// Новый объект Best инициализируется значениями полей Super:
monstr Best = Vasia.the_best(Super);

```

Указатель `this` можно также применять для идентификации поля класса в том случае, когда его имя совпадает с именем формального параметра метода. Другой способ идентификации поля использует операцию доступа к области видимости:

```

void cure(int health, int ammo){
    this -> health += health;    // Использование this
    monstr:: ammo += ammo;      // Использование операции ::
}

```

Конструкторы

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов.

- ❑ Конструктор *не возвращает значение*, даже типа `void`. Нельзя получить указатель на конструктор.
- ❑ Класс может иметь *несколько конструкторов* с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
- ❑ Конструктор, вызываемый без параметров, называется *конструктором по умолчанию*.
- ❑ *Параметры конструктора* могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.
- ❑ Если программист не указал ни одного конструктора, компилятор создает его *автоматически*. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов (см. раздел «Простое наследование», с. 201). В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.
- ❑ *Конструкторы не наследуются*.
- ❑ Конструкторы нельзя описывать с модификаторами `const`, `virtual` и `static`.
- ❑ Конструкторы глобальных объектов вызываются до вызова функции `main`. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).
- ❑ Конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

```

ия_класса ия_объекта [(список параметров)];
// Список параметров не должен быть пустым
ия_класса (список параметров);

```

```
// Создается объект без имени (список может быть пустым)
иня_класса иня_объекта = выражение;
// Создается объект без имени и копируется
```

Примеры:

```
monstr Super(200, 300), Vasia(50), Z;
monstr X = monstr(1000);
monstr Y = 500;
```

В первом операторе создаются три объекта. Значения не указанных параметров устанавливаются по умолчанию.

Во втором операторе создается безымянный объект со значением параметра `health = 1000` (значение второго параметра устанавливается по умолчанию). Выделяется память под объект `X`, в которую копируется безымянный объект.

В последнем операторе создается безымянный объект со значением параметра `health = 500` (значение второго параметра устанавливается по умолчанию). Выделяется память под объект `Y`, в которую копируется безымянный объект. Такая форма создания объекта возможна в том случае, если для инициализации объекта допускается задать один параметр.

В качестве примера класса с несколькими конструкторами усовершенствуем описанный ранее класс `monstr`, добавив в него поля, задающие цвет (`skin`) и имя (`name`):

```
enum color {red, green, blue}; // Возможные значения цвета
class monstr{
    int health, ammo;
    color skin;
    char *name;
public:
    monstr(int he = 100, int am = 10);
    monstr(color sk);
    monstr(char * nam);
    int get_health(){return health;}
    int get_ammo(){return ammo;}
};
//-----
monstr::monstr(int he, int am){
    health = he; ammo = am; skin = red; name = 0;
}
//-----
monstr::monstr(color sk){
    switch (sk){
        case red : health = 100; ammo = 10; skin = red; name = 0; break;
        case green: health = 100; ammo = 20; skin = green; name = 0; break;
        case blue : health = 100; ammo = 40; skin = blue; name = 0; break;
    }
}
```

```
//-----
monstr::monstr(char * nam){
    name = new char [strlen(nam) + 1];
    // К длине строки добавляется 1 для хранения нуль-символа
    strcpy(name, nam);
    health = 100; ammo = 10; skin = red;
}

//-----
monstr * m = new monstr ("Ork");
monstr Green (green);
```

Первый из приведенных выше конструкторов является конструктором по умолчанию, поскольку его можно вызвать без параметров. Объекты класса `monstr` теперь можно инициализировать различными способами, требуемый конструктор будет вызван в зависимости от списка значений в скобках. При задании нескольких конструкторов следует соблюдать те же правила, что и при написании перегруженных функций — у компилятора должна быть возможность распознать нужный вариант.

ПРИМЕЧАНИЕ

Перегружать можно не только конструкторы, но и другие методы класса.

Существует еще один способ инициализации полей в конструкторе (кроме использованного в приведенной выше программе присваивания полям значений формальных параметров) — с помощью списка инициализаторов, расположенных после двоеточия между заголовком и телом конструктора:

```
monstr::monstr(int he, int am):
    health (he), ammo (am), skin (red), name (0){}
```

Поля перечисляются через запятую. Для каждого поля в скобках указывается инициализирующее значение, которое может быть выражением. Без этого способа не обойтись при инициализации *полей-констант*, *полей-ссылок* и *полей-объектов*. В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам.

ПРИМЕЧАНИЕ

Конструктор не может вернуть значение, чтобы сообщить об ошибке во время инициализации. Для этого можно использовать механизм обработки исключительных ситуаций (см. раздел «Исключения в конструкторах и деструкторах», с. 228).

Конструктор копирования

Конструктор копирования — это специальный вид конструктора, получающий в качестве единственного параметра ссылку на объект этого же класса:

```
T::T(const T&) { ... /* Тело конструктора */ }
```

где `T` — имя класса.

Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего:

- при описании нового объекта с инициализацией другим объектом;
- при передаче объекта в функцию по значению;
- при возврате объекта из функции¹.

Если программист не указал ни одного конструктора копирования, компилятор создает его автоматически. Такой конструктор выполняет поэлементное копирование полей. Если класс содержит указатели или ссылки, это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

Запишем конструктор копирования для класса `monstr`. Поскольку в нем есть поле `name`, содержащее указатель на строку символов, конструктор копирования должен выделять память под новую строку и копировать в нее исходную:

```
monstr::monstr(const monstr &M){
    if (M.name){
        name = new char [strlen(M.name) + 1];
        strcpy(name, M.name);}
    else name = 0;
    health = M.health; ammo = M.ammo; skin = M.skin;
}

monstr Vasia (blue);
monstr Super = Vasia;           // Работает конструктор копирования
monstr *m = new monstr ("Ork");
monstr Green = *m;              // Работает конструктор копирования
```

ПРИМЕЧАНИЕ

Любой конструктор класса, принимающий один параметр какого-либо другого типа, называется *конструктором преобразования*, поскольку он осуществляет преобразование из типа параметра в тип этого класса.

Правила написания конструкторов классов, входящих в иерархию, описаны в разделе «Простое наследование», с. 201.

Статические элементы класса

С помощью модификатора `static` можно описать статические поля и методы класса. Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

А также при обработке исключений.

Статические поля

Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

Ниже перечислены особенности статических полей.

- ❑ Память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора (определение должно быть записано вне функций):

```
class A{
    public:
        static int count; // Объявление в классе
};

int A::count;           // Определение в глобальной области
                        // По умолчанию инициализируется нулем
// int A::count = 10;   Пример инициализации произвольным значением
```

- ❑ Статические поля доступны как через имя класса, так и через имя объекта:

```
A *a, b;

cout << A::count << a->count << b.count;
// Будет выведено одно и то же
```

- ❑ *На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как private, нельзя изменить с помощью операции доступа к области действия, как описано выше. Это можно сделать только с помощью статических методов (см. далее).*
- ❑ *Память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции sizeof.*

Статические методы

Статические методы предназначены для обращения к статическим полям класса. Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель `this`. Обращение к статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

```
class A{
    static int count; // Поле count – скрытое
    public:
        static void inc_count(){ count++; }
};
```

```

-
A::int count;           // Определение в глобальной области
void f(){
    A a;
    // a.count++ – нельзя, поле count скрытое
    // Изменение поля с помощью статического метода:
    a.inc_count();      // или A::inc_count();
}

```

Статические методы не могут быть константными (const) и виртуальными (virtual).

Дружественные функции и классы

Иногда желательно иметь непосредственный доступ извне к скрытым полям класса, то есть расширить интерфейс класса. Для этого служат дружественные функции и дружественные классы.

Дружественная функция

Дружественные функции применяются для доступа к скрытым полям класса и представляют собой альтернативу методам. Метод, как правило, описывает свойство объекта, а в виде дружественных функций оформляются действия, не являющиеся свойствами класса, но концептуально входящие в его интерфейс и нуждающиеся в доступе к его скрытым полям, например, переопределенные операции вывода объектов (см. с. 284).

Ниже перечислены правила описания и особенности дружественных функций.

- Дружественная функция объявляется *внутри класса*, к элементам которого ей нужен доступ, с ключевым словом `friend`. В качестве параметра ей должен передаваться объект или ссылка на объект класса, поскольку указатель `this` ей не передается.
- Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие спецификаторов доступа, место размещения ее объявления в классе безразлично.
- Одна функция может быть дружественной сразу нескольким классами.

В качестве примера ниже приведено описание двух функций, дружественных классу `monstr`. Функция `kill` является методом класса `hero`, а функция `steal_ammo` не принадлежит ни одному классу. Обеим функциям в качестве параметра передается ссылка на объект класса `monstr`.

```

class monstr; // Предварительное объявление класса
class hero{
public:
    void kill(monstr &);
};

```



```
class monstr{  
    friend int steal_ammo(monstr &);  
    friend void hero::kill(monstr &);  
    // Класс hero должен быть определен ранее  
};  
int steal_ammo(monstr &M){return --M.ammo;}  
void hero::kill(monstr &M){M.health = 0; M.ammo = 0;}
```

Использования дружественных функций нужно по возможности избегать, поскольку они нарушают принцип инкапсуляции и, таким образом, затрудняют отладку и модификацию программы.

Дружественный класс

Если все методы какого-либо класса должны иметь доступ к скрытым полям другого, весь класс объявляется дружественным с помощью ключевого слова `friend`. В приведенном ниже примере класс `mistress` объявляется дружественным классу `hero`:

```
class hero{  
    friend class mistress;  
}  
class mistress{  
    void f1();  
    void f2();  
}
```

Функции `f1` и `f2` являются дружественными по отношению к классу `hero` (хотя и описаны без ключевого слова `friend`) и имеют доступ ко всем его полям.

Объявление `friend` не является спецификатором доступа и не наследуется.

ПРИМЕЧАНИЕ

Обратите внимание на то, что класс сам определяет, какие функции и классы являются дружественными, а какие нет.

Деструкторы

Деструктор — это особый вид метода, применяющийся для освобождения памяти, занимаемой объектом. Деструктор вызывается автоматически, когда объект выходит из области видимости:

- ☐ для локальных объектов — при выходе из блока, в котором они объявлены;
- ☐ для глобальных — как часть процедуры выхода из `main`;
- ☐ для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции `delete`.

ВНИМАНИЕ

Автоматический вызов деструктора объекта при выходе из области действия указателя на него не производится.

Имя деструктора начинается с тильды (~), непосредственно за которой следует имя класса. Деструктор:

- не имеет аргументов и возвращаемого значения;
- не может быть объявлен как `const` или `static`;
- не наследуется;
- может быть виртуальным (см. раздел «Виртуальные методы», с. 205).

Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.

Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически — иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструктор для рассматриваемого примера (с. 183) должен выглядеть так:

```
monstr::~monstr() {delete [] name;}
```

Деструктор можно вызвать явным образом путем указания полностью уточненного имени, например:

```
monstr *m; ...
m -> ~monstr();
```

Это может понадобиться для объектов, которым с помощью перегруженной операции `new` выделялся конкретный адрес памяти. Без необходимости явно вызывать деструктор объекта не рекомендуется.

Перегрузка операций

C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Эта дает возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в C++, за исключением:

```
*      ?:      ::      #      ##      sizeof
```

Перегрузка операций осуществляется с помощью методов специального вида (*функций-операций*) и подчиняется следующим правилам:

- при перегрузке операций сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
- для стандартных типов данных переопределять операции нельзя;

- ❑ функции-операции не могут иметь аргументов по умолчанию;
- ❑ функции-операции наследуются (за исключением `-`);
- ❑ функции-операции не могут определяться как `static`.

Функцию-операцию можно определить тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс¹.

Функция-операция содержит ключевое слово `operator`, за которым следует знак переопределяемой операции:

тип `operator` операция (список параметров) { тело функции }

Перегрузка унарных операций

Унарная функция-операция, определяемая *внутри класса*, должна быть представлена с помощью нестатического метода без параметров, при этом операндом является вызвавший ее объект, например:

```
class monstr{
    monstr & operator ++() {++health; return *this;}
}
monstr Vasia;
cout << (++Vasia).get_health();
```

Если функция определяется *вне класса*, она должна иметь один параметр типа класса:

```
class monstr{
    friend monstr & operator ++( monstr &M);
};
monstr& operator ++(monstr &M) {++M.health; return M;}
```

Если не описывать функцию внутри класса как дружественную, нужно учитывать доступность изменяемых полей. В данном случае поле `health` недоступно извне, так как описано со спецификатором `private`, поэтому для его изменения требуется использование соответствующего метода. Введем в описание класса `monstr` метод `change_health`, позволяющий изменить значение поля `health`

```
void change_health(int he){ health = he;}
```

Тогда можно перегрузить операцию инкремента с помощью обычной функции, описанной вне класса:

```
monstr& operator ++(monstr &M){
    int h = M.get_health(); h++;
    M.change_health(h);
}
```

¹ Особый случай: функция-операция, первый параметр которой имеет стандартный тип, не может определяться как метод класса.

```

    return M;
}

```

Операции постфиксного инкремента и декремента должны иметь первый параметр типа `int`. Он используется только для того, чтобы отличить их от префиксной формы:

```

class monstr{
    monstr operator ++(int){
        monstr M(*this); health++;
        return M;
    }
};
monstr Vasia;
cout << (Vasia++).get_health();

```

Перегрузка бинарных операций

Бинарная функция-операция, определяемая *внутри класса*, должна быть представлена с помощью нестатического метода с параметрами, при этом вызвавший ее объект считается первым операндом:

```

class monstr{
    bool operator >(const monstr &M){
        if( health > M.get_health()) return true;
        return false;
    }
};

```

Если функция определяется *вне класса*, она должна иметь два параметра типа класса:

```

bool operator >(const monstr &M1, const monstr &M2){
    if( M1.get_health() > M2.get_health()) return true;
    return false;
}

```

Перегрузка операции присваивания

Операция присваивания определена в любом классе по умолчанию как поэлементное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. Если класс содержит поля, память под которые выделяется динамически, необходимо определить собственную операцию присваивания. Чтобы сохранить семантику присваивания, операция-функция должна возвращать ссылку на объект, для которого она вызвана, и принимать в качестве параметра единственный аргумент — ссылку на присваиваемый объект.

```

const monstr& operator = (const monstr &M){
    // Проверка на самоприсваивание:

```

```

    if (&M == this) return *this;
    if (name) delete [] name;
    if (M.name){
        name = new char [strlen(M.name) + 1];
        strcpy(name, M.name);}
    else name = 0;
    health = M.health; ammo = M.ammo; skin = M.skin;
    return *this;
}

```

Возврат из функции указателя на объект делает возможной цепочку операций присваивания:

```

monstr A(10), B, C;
C = B = A;

```

Операцию присваивания можно определять только как метод класса. Она не наследуется.

Перегрузка операций new и delete

Чтобы обеспечить альтернативные варианты управления памятью, можно определять собственные варианты операций new и new[] для выделения динамической памяти под объект и массив объектов соответственно, а также операции delete и delete [] для ее освобождения.

Эти функции-операции должны соответствовать следующим правилам:

- ☐ им не требуется передавать параметр типа класса;
- ☐ первым параметром функциям new и new[] должен передаваться размер объекта типа size_t (это тип, возвращаемый операцией sizeof, он определяется в заголовочном файле <stddef.h>); при вызове он передается в функции неявным образом;
- ☐ они должны определяться с типом возвращаемого значения void*, даже если return возвращает указатель на другие типы (чаще всего на класс);
- ☐ операция delete должна иметь тип возврата void и первый аргумент типа void*;
- ☐ операции выделения и освобождения памяти являются статическими элементами класса.

Поведение перегруженных операций должно соответствовать действиям, выполняемым ими по умолчанию. Для операции new это означает, что она должна возвращать правильное значение, корректно обрабатывать запрос на выделение памяти нулевого размера и порождать исключение при невозможности выполнения запроса (об исключениях рассказывается в разделе «Обработка исключительных ситуаций» на с. 222). Для операции delete следует соблюдать условие, что удаление нулевого указателя должно быть безопасным, поэтому внутри операции необходима проверка указателя на нуль и отсутствие каких-либо действий в случае равенства.

Стандартные операции выделения и освобождения памяти могут использоваться в области действия класса наряду с перегруженными (с помощью операции доступа к области видимости :: для объектов этого класса и непосредственно — для любых других).

Перегрузка операции выделения памяти применяется для экономии памяти, повышения быстродействия программы или для размещения данных в некоторой конкретной области. Например, пусть описывается класс, содержащий указатель на некоторый объект:

```
class Obj { _ };
class pObj{
    private:
        Obj *p;
};
```

При выделении памяти под объект типа pObj с помощью стандартной операции `new`

```
pObj *p = new pObj;
```

фактическое количество байтов будет превышать `sizeof(pObj)`, поскольку `new` обычно записывает в начало выделяемой области ее размер (для того чтобы правильно обрабатывала операция `delete`):

Для небольших объектов эти накладные расходы могут оказаться весьма значительными. Для экономии памяти можно написать собственную операцию `new` класса pObj, которая будет выделять большой блок памяти, а затем размещать в нем указатели на Obj. Для этого в объект pObj вводится статическое поле `headOfFree`, в котором хранится указатель на первую свободную ячейку блока для размещения очередного объекта.

Неиспользуемые ячейки связываются в список. Чтобы не занимать память под поле связи, используется объединение (`union`), с помощью которого одна и та же ячейка используется либо для размещения указателя на объект, либо для связи со следующей свободной ячейкой:

```
class pObj{
    public:
        static void * operator new(size_t size);
    private:
        union{
            Obj *p;          // Указатель на объект
            pObj *next;      // Указатель на следующую свободную ячейку
        };
        static const int BLOCK_SIZE; // Размер блока
        // Заголовок списка свободных ячеек:
        static pObj *headOfFree;
};

void * pObj::operator new(size_t size){
    // Перенаправить запросы неверного количества памяти
```



```

// стандартной операции new:
if (size != sizeof(pObj)) return ::operator new(size);
pObj *p = headOfFree; // Указатель на первую свободную ячейку
// Переместить указатель списка свободных ячеек:
if (p) headOfFree = p -> next;
// Если свободной памяти нет, выделяем очередной блок:
else {
    pObj *newblock = static_cast<pObj*>1
        (::operator new(BLOCK_SIZE * sizeof(pObj)));
    // Все ячейки свободны, кроме первой (она будет
    // занята), связываем их:
    for (int i = 1; i < BLOCK_SIZE - 1; ++i)
        newblock[i].next = &newblock[i + 1];
    newblock[BLOCK_SIZE - 1].next = 0;
    // Устанавливаем начало списка свободных ячеек:
    headOfFree = &newblock[1];
    p = newblock;
}
return p; // Возвращаем указатель на выделенную память
}

```

Перегруженная операция `new` наследуется, поэтому она вызывается для производных объектов. Если их размер не соответствует размеру базового (а так, скорее всего, и есть), это может вызвать проблемы. Чтобы их избежать, в начале операции проверяется соответствие размеров. Если размер объекта не равен тому, для которого перегружена операция `new`, запрос на выделение памяти передается стандартной операции `new`.

В программе, использующей класс `pObj`, должна присутствовать инициализация его статических полей (статические поля рассматривались на с. 186):

```

pObj *pObj::headOfFree; // Устанавливается в 0 по умолчанию
const int pObj::BLOCK_SIZE = 1024;

```

Как видно из этого примера, помимо экономии памяти достигается еще и высокое быстродействие, ведь в большинстве случаев выделение памяти сводится к нескольким простым операторам.

Естественно, что если операция `new` перегружена, то же самое должно быть выполнено и для операции `delete` (например, в нашем случае стандартная операция `delete` не найдет в начале объекта верной информации о его размерах, что приведет к неопределенному поведению программы).

В рассмотренном примере операция `delete` должна добавлять освобожденную ячейку памяти к списку свободных ячеек:

```

void pObj::operator delete(void * ObjToDie, size_t size){
    if (ObjToDie == 0) return;
    if (size != sizeof(pObj)){

```

¹ Здесь использовано явное преобразование типа с помощью операции `static_cast`. О нем рассказывается в разделе «Операция `static_cast`» на с. 237.

```

        ::operator delete(ObjToDie); return;
    }
    pObj *p = static_cast<pObj*>(ObjToDie);
    p->next = headOfFree;
    headOfFree = p;
}

```

В операции delete выполнена проверка соответствия размеров объектов, аналогичная приведенной в операции new.

Перегрузка операции приведения типа

Можно определить функции-операции, которые будут осуществлять преобразование объекта класса к другому типу. Формат:

```
operator имя_нового_типа ();
```

Тип возвращаемого значения и параметры указывать не требуется. Можно определять виртуальные функции преобразования типа.

Пример:

```

monstr::operator int(){return health;}

monstr Vasia; cout << int(Vasia);

```

Перегрузка операции вызова функции

Класс, в котором определена операция вызова функции, называется *функциональным*. От такого класса не требуется наличия других полей и методов:

```

class if_greater{
public:
    int operator () (int a, int b) const {
        return a > b;
    }
};

```

Использование такого класса имеет весьма специфический синтаксис. Рассмотрим пример:

```

if_greater x;
cout << x(1, 5) << endl;           // Результат - 0
cout << if_greater()(5, 1) << endl; // Результат - 1

```

Поскольку в классе if_greater определена операция вызова функции с двумя параметрами, выражение x(1, 5) является допустимым (то же самое можно записать в виде x.operator () (1, 5)). Как видно из примера, объект функционального класса используется так, как если бы он был функцией.

Во втором операторе вывода выражение if_greater() используется для вызова конструктора по умолчанию класса if_greater. Результатом выполнения этого выражения является объект класса if_greater. Далее, как и в предыдущем случае, для этого объекта вызывается функция с двумя аргументами, записанными в круглых скобках.

Операцию () (а также операцию []) можно определять только как метод класса. Можно определить перегруженные операции вызова функции с различным количеством аргументов. Функциональные объекты широко применяются в стандартной библиотеке C++.

Перегрузка операции индексирования

Операция индексирования [] обычно перегружается, когда тип класса представляет множество значений, для которого индексирование имеет смысл. Операция индексирования должна возвращать ссылку на элемент, содержащийся в множестве. Покажем это на примере класса Vect, предназначенного для хранения и работы с безопасным массивом целых чисел:

```
#include <iostream.h>
#include <stdlib.h>
class Vect{
public:
    explicit Vect(int n = 10);
    Vect(const int a[], int n); //инициализация массивом
    ~Vect() { delete [] p; }
    int& operator [] (int i);
    void Print();

private:
    int* p;
    int size;
};

Vect::Vect(int n) : size(n){
    p = new int[size];
}

Vect::Vect(const int a[], int n) : size(n){
    p = new int[size];
    for (int i = 0; i < size; i++) p[i] = a[i];
}

// Перегрузка операции индексирования:
int& Vect::operator [] (int i){
    if(i < 0 || i >= size){
        cout << "Неверный индекс (i = " << i << ")" << endl;
        cout << "Завершение программы" << endl;
        exit(0);
    }
    return p[i];
}

void Vect::Print(){
    for (int i = 0; i < size; i++)
        cout << p[i] << " ";
    cout << endl;
}
```

```
int main(){
    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    Vect a(arr, 10);
    a.Print();
    cout << a[5] << endl;
    cout << a[12] << endl;
    return 0;
}
```

Результат работы программы:

```
1 2 3 4 5 6 7 8 9 10
6
Неверный индекс (i = 12)
Завершение программы
```

Перегруженная операция индексирования получает целый аргумент и проверяет, лежит ли его значение в пределах диапазона массива. Если да, то возвращает адрес элемента, что соответствует семантике стандартной операции индексирования.

В данном примере конструктор с параметром по умолчанию объявлен как `explicit` для того, чтобы он не являлся конструктором преобразования типа (см. с. 185), вызываемым неявно. Ключевое слово `explicit` указывает на то, что этот конструктор будет вызываться только явным образом.

Указатели на элементы классов

К элементам классов можно обращаться с помощью указателей. Для этого определены операции `*` и `->`. Указатели на поля и методы класса определяются по-разному.

Формат указателя на метод класса:

```
возвр_тип (имя_класса::*имя_указателя)(параметры);
```

Например, описание указателя на методы класса `monstr`

```
int get_health() {return health;}
int get_ammo() {return ammo;}
```

(а также на другие методы этого класса с такой же сигнатурой) будет иметь вид:

```
int (monstr::*pget)();
```

Такой указатель можно задавать в качестве параметра функции. Это дает возможность передавать в функцию имя метода:

```
void fun(int (monstr::*pget)()){
    (*this.*pget)();    // Вызов функции через операцию .*
    (this->*pget)();    // Вызов функции через операцию ->*
}
```

Можно настроить указатель на конкретный метод с помощью операции взятия адреса:

```
// Присваивание значения указателю:
pget = & monstr::get_health;
monstr Vasia, *p;
p = new monstr;
// Вызов функции через операцию .* :
int Vasin_health = (Vasia.*pget)();
// Вызов функции через операцию ->* :
int p_health = (p->*pget)();
```

Ниже перечислены правила использования указателей на методы классов.

- ☐ Указателю на метод можно присваивать только адреса методов, имеющих соответствующий заголовок.
- ☐ Нельзя определить указатель на статический метод класса.
- ☐ Нельзя преобразовать указатель на метод в указатель на обычную функцию, не являющуюся элементом класса.

Как и указатели на обычные функции, указатели на методы используются в том случае, когда возникает необходимость вызвать метод, имя которого неизвестно. Однако в отличие указателя на переменную или обычную функцию, указатель на метод не ссылается на определенный адрес памяти. Он больше похож на индекс в массиве, поскольку задает смещение. Конкретный адрес в памяти получается путем сочетания указателя на метод с указателем на определенный объект.

ПРИМЕЧАНИЕ

Методы, вызываемые через указатели, могут быть виртуальными. При этом вызывается метод, соответствующий типу объекта, к которому применялся указатель.

Формат указателя на поле класса:

```
тип_данных(имя_класса::*имя_указателя);
```

В определение указателя можно включить его инициализацию в форме:

```
&имя_класса::*имя_поля; // Поле должно быть public
```

Если бы поле `health` было объявлено как `public`, определение указателя на него имело бы вид:

```
int (monstr::*phealth) = &monstr::health;
cout << Vasia.*phealth; // Обращение через операцию .*
cout << p->*phealth;    // Обращение через операцию ->*
```

Обратите внимание на то, что указатели на поля классов не являются обычными указателями — ведь при присваивании им значений они не ссылаются на конкретный адрес памяти, поскольку память выделяется не под классы, а под объекты классов.

Рекомендации по составу класса

Как правило, класс как тип, определенный пользователем, должен содержать скрытые (`private`) поля и следующие функции:

- ❑ *конструкторы*, определяющие, как инициализируются объекты класса;
- ❑ набор *методов, реализующих свойства класса* (при этом методы, возвращающие значения скрытых полей класса, описываются с модификатором `const`, указывающим, что они не должны изменять значения полей);
- ❑ набор *операций*, позволяющих копировать, присваивать, сравнивать объекты и производить с ними другие действия, требующиеся по сути класса;
- ❑ класс *исключений*, используемый для сообщений об ошибках с помощью генерации исключительных ситуаций (об этом будет рассказано на с. 228).

Если существуют функции, которые работают с классом или несколькими классами через интерфейс (то есть доступ к скрытым полям им не требуется), можно описать их вне классов, чтобы не перегружать интерфейсы, а для обеспечения логической связи поместить их в общее с этими классами пространство имен, например:

```
namespace Staff{  
    class monstr { /* _ */};  
    class hero { /* _ */};  
    void interact(hero, monstr);  
}
```


ГЛАВА 5

Наследование

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства. При большом количестве никак не связанных классов управлять ими становится невозможным. Наследование позволяет справиться с этой проблемой путем упорядочивания и ранжирования классов, то есть объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

Ключи доступа

При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью *ключей доступа* `private`, `protected` и `public`:

```
class имя : [private | protected | public] базовый_класс  
{тело класса};
```

Если базовых классов несколько, они перечисляются через запятую. Ключ доступа может стоять перед каждым классом, например:

```
class A { _ };  
class B { _ };  
class C { _ };  
class D: A, protected B, public C { _ };
```

По умолчанию для классов используется ключ доступа `private`, а для структур — `public`.

До сих пор мы рассматривали только применяемые к элементам класса *спецификаторы доступа* `private` и `public`. Для любого элемента класса может также использоваться спецификатор `protected`, который для одиночных классов, не входящих в иерархию, равносителен `private`. Разница между ними проявляется при наследовании, что можно видеть из приведенной таблицы:

Ключ доступа	Спецификатор базового класса	Доступ в производном классе	Комментарий
private	private	private	все private
	protected	private	
	public	private	
protected	private	private	все protected, кроме private
	protected	protected	
	public	protected	
public	private	private	не изменяется
	protected	protected	
	public	public	

Как видно из таблицы, `private` элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним может осуществляться только через методы базового класса.

Элементы `protected` при наследовании с ключом `private` становятся в производном классе `private`, в остальных случаях права доступа к ним не изменяются.

Доступ к элементам `public` при наследовании становится соответствующим ключу доступа.

Если базовый класс наследуется с ключом `private`, можно выборочно сделать некоторые его элементы доступными в производном классе, объявив их в секции `public` производного класса с помощью операции доступа к области видимости:

```
class Base{
    ...
    public: void f();
};
class Derived : private Base{
    ...
    public: Base::void f();
};
```

Простое наследование

Простым называется наследование, при котором производный класс имеет одного родителя. Для различных методов класса существуют разные правила насле-

дования — например, конструкторы и операция присваивания в производном классе не наследуются, а деструкторы наследуются. Рассмотрим наследование классов и проблемы, возникающие при этом, на примере.

Создадим производный от класса `monstr` класс `daemon`, добавив полезную в некоторых случаях способность думать. Базовый класс приведен с некоторыми методами, добавленными в предыдущих разделах:

```
enum color {red, green, blue};
// ----- Класс monstr -----
class monstr{
    // ----- Скрытые поля класса:
    int health, ammo;
    color skin;
    char *name;
public:
    // ----- Конструкторы:
    monstr(int he = 100, int am = 10);
    monstr(color sk);
    monstr(char * nam);
    monstr(monstr &M);
    // ----- Деструктор:
    ~monstr() {delete [] name;}
    // ----- Операции:
    monstr& operator ++(){
        ++health; return *this;
    }
    monstr operator ++(int){
        monstr M(*this); health++; return M;
    }
    operator int(){
        return health;
    }
    bool operator >(monstr &M){
        if( health > M.get_health()) return true;
        return false;
    }
    const monstr& operator = (monstr &M){
        if (&M == this) return *this;
        if (name) delete [] name;
        if (M.name){
            name = new char [strlen(M.name) + 1];
            strcpy(name, M.name);}
        else name = 0;
        health = M.health; ammo = M.ammo; skin = M.skin;
        return *this;
    }
    // ----- Методы доступа к полям:
```

```

    int get_health() const {return health;}
    int get_ammo() const {return ammo;}
    // ----- Методы, изменяющие значения полей:
    void change_health(int he){ health = he;}
    // ----- Прочие методы:
    void draw(int x, int y, int scale, int position):
};
// ----- Реализация класса monstr -----
monstr::monstr(int he, int am):
    health(he), ammo(am), skin(red), name(0){}
monstr::monstr(monstr &M){
    if (M.name){
        name = new char [strlen(M.name) + 1];
        strcpy(name, M.name);}
    else name = 0;
    health = M.health; ammo = M.ammo; skin = M.skin;
}
monstr::monstr(color sk){
    switch (sk){
        case red : health = 100; ammo = 10; skin = red; name = 0; break;
        case green: health = 100; ammo = 20; skin = green; name = 0; break;
        case blue: health = 100; ammo = 40; skin = blue; name = 0; break;
    }
}
monstr::monstr(char * nam){
    name = new char [strlen(nam)+1];
    strcpy(name, nam);
    health = 100; ammo = 10; skin = red;
}
void monstr::draw(int x, int y, int scale, int position)
{ /* _ Отрисовка monstr */ }

// ----- Класс daemon -----
class daemon : public monstr{
    int brain;
public:
    // ----- Конструкторы:
    daemon(int br = 10){brain = br;};
    daemon(color sk) : monstr(sk) {brain = 10;}
    daemon(char * nam) :onstr(nam) {brain = 10;}
    daemon(daemon &M) :onstr(M) {brain = M.brain;}
    // ----- Операции:
    const daemon& operator = (daemon &M){
        if (&M == this) return *this;
        brain = M.brain;
        monstr::operator = (M);
        return *this;
    }
};

```

```

    }
    // ----- Методы, изменяющие значения полей:
    void think();
    // ----- Прочие методы:
    void draw(int x, int y, int scale, int position):
};
// ----- Реализация класса daemon -----
void daemon::think(){ /* ... */ }
void daemon::draw(int x, int y, int scale, int position)
{ /* ... Отрисовка daemon */ }

```

В классе `daemon` введено поле `brain` и метод `think`, определены собственные конструкторы и операция присваивания, а также переопределен метод отрисовки `draw`. Все поля класса `monstr`, операции (кроме присваивания) и методы `get_health`, `get_ammo` и `change_health` наследуются в классе `daemon`, а деструктор формируется по умолчанию.

Рассмотрим правила наследования различных методов.

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными ниже правилами.

- ❑ Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть тот, который можно вызвать без параметров). Это использовано в первом из конструкторов класса `daemon`.
- ❑ Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами; в порядке их объявления в классе, а затем исполняется конструктор класса.
- ❑ В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.

ВНИМАНИЕ

Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации (это продемонстрировано в трех последних конструкторах).

Не наследуется и **операция присваивания**, поэтому ее также требуется явно определить в классе `daemon`. Обратите внимание на запись функции-операции: в ее теле применен явный вызов функции-операции присваивания из базового класса. Чтобы лучше представить себе синтаксис вызова, ключевое слово `operator` вместе со знаком операции можно интерпретировать как имя функции-операции.

Вызов функций базового класса предпочтительнее копирования фрагментов кода из функций базового класса в функции производного. Кроме сокращения объема кода, этим достигается упрощение модификации программы: изменения требуется вносить только в одну точку программы, что сокращает количество возможных ошибок.

Ниже перечислены правила наследования *деструкторов*.

- ❑ *Деструкторы не наследуются*, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.
- ❑ В отличие от конструкторов, при написании деструктора производного класса в нем *не требуется явно вызывать деструкторы базовых классов*, поскольку это будет сделано автоматически.
- ❑ Для иерархии классов, состоящей из нескольких уровней, деструкторы *вызываются в порядке, строго обратном вызову конструкторов*: сначала вызывается деструктор класса, затем — деструкторы элементов класса, а потом деструктор базового класса.

Поля, унаследованные из класса `monstr`, недоступны функциям производного класса, поскольку они определены в базовом классе как `private`. Если функциям, определенным в `daemon`, требуется работать с этими полями, можно либо описать их в базовом классе как `protected`, либо обращаться к ним с помощью функций из `monstr`, либо явно переопределить их в `daemon` так, как было показано в предыдущем разделе (с. 201).

Рассматривая наследование *методов*, обратите внимание на то, что в классе `daemon` описан метод `draw`, переопределяющий метод с тем же именем в классе `monstr` (поскольку отрисовка различных персонажей, естественно, выполняется по-разному). Таким образом, производный класс может не только дополнять, но и корректировать поведение базового класса¹. Доступ к переопределенному методу базового класса для производного класса выполняется через имя, уточненное с помощью операции доступа к области видимости (`::`).

Виртуальные методы

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса (при открытом наследовании):

```
// Описывается указатель на базовый класс:
monstr *p;
// Указатель ссылается на объект производного класса:
p = new daemon;
```

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается, поэтому при выполнении оператора, например,

```
p->draw(1, 1, 1, 1);
```

будет вызван метод класса `monstr`, а не класса `daemon`, поскольку ссылки на методы разрешаются во время компоновки программы. Этот процесс называется *ранним*

¹ Переопределять в производном классе рекомендуется только виртуальные методы (см. след. раздел).

связыванием. Чтобы вызвать метод класса `daemon`, можно использовать явное преобразование типа указателя:

```
((daemon * p))->draw(1. 1. 1. 1);
```

Это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс. Другой пример — связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

В C++ реализован механизм *позднего связывания*, когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов и рассмотрен в следующем разделе.

Для определения *виртуального метода* используется спецификатор `virtual`, например:

```
virtual void draw(int x, int y, int scale, int position);
```

Рассмотрим правила описания и использования виртуальных методов.

- ❑ Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе *с тем же именем и набором параметров*, автоматически становится виртуальным, а *с отличающимся набором параметров* — обычным.
- ❑ Виртуальные методы *наследуются*, то есть переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя.
- ❑ Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости.
- ❑ Виртуальный метод не может объявляться с модификатором `static`, но может быть объявлен как дружественный.
- ❑ Если в классе вводится описание виртуального метода, он должен быть определен хотя бы как чисто виртуальный.

Чисто виртуальный метод содержит признак `= 0` вместо тела, например:

```
virtual void f(int) = 0;
```

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Если определить метод `draw` в классе `monstr` как виртуальный, решение о том, метод какого класса вызвать, будет приниматься в зависимости от типа объекта, на который ссылается указатель:

```
monstr *r, *p;
r = new monstr;           // Создается объект класса monstr
p = new daemon;           // Создается объект класса daemon
r->draw(1. 1. 1. 1);       // Вызывается метод monstr::draw
p->draw(1. 1. 1. 1);       // Вызывается метод daemon::draw
```

```
p-> monstr::draw(1, 1, 1, 1): // Обход механизма виртуальных методов
```

Если объект класса `daemon` будет вызывать метод `draw` не непосредственно, а косвенно (то есть из другого метода, определенного в классе `monstr`), будет вызван метод `draw` класса `daemon`.

Итак, *виртуальным называется метод, ссылка на который разрешается на этапе выполнения программы* (перевод красивого английского слова *virtual* — в данном значении всего-навсего «фактический», то есть ссылка разрешается по факту вызова).

Механизм позднего связывания

Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает *таблицу виртуальных методов* (`vtbl`), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в `vtbl` одно и то же смещение для каждого класса в пределах иерархии.

Каждый объект содержит скрытое дополнительное *поле ссылки* на `vtbl`, называемое `vptr`. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).

На этапе компиляции ссылки на виртуальные методы заменяются на обращения к `vtbl` через `vptr` объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы.

Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта. Деструктор передает операции `delete` размер объекта, имеющий тип `size_t`. Если удаляемый объект является производным и в нем не определен виртуальный деструктор, передаваемый размер объекта может оказаться неправильным.

Четкого правила, по которому метод следует делать виртуальным, не существует. Можно только дать рекомендацию объявлять виртуальными методы, для которых есть вероятность, что они будут переопределены в производных классах. Методы, которые во всей иерархии останутся неизменными или те, которыми производные классы пользоваться не будут, делать виртуальными нет смысла. С другой стороны, при проектировании иерархии не всегда можно предсказать, каким образом будут расширяться базовые классы (особенно при проектировании библиотек классов), а объявление метода виртуальным обеспечивает гибкость и возможность расширения.

Для пояснения последнего тезиса представим себе, что вызов метода `draw` осуществляется из метода перемещения объекта. Если текст метода перемещения не зависит от типа перемещаемого объекта (поскольку принцип перемещения всех

объектов одинаков, а для отрисовки вызывается конкретный метод), переопределять этот метод в производных классах нет необходимости, и он может быть описан как неvirtуальный. Если метод `draw` виртуальный, метод перемещения сможет без перекомпиляции работать с объектами любых производных классов — даже тех, о которых при его написании ничего известно не было.

Виртуальный механизм работает только при использовании указателей или ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется *полиморфным*. В данном случае полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется *абстрактным*. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться *только в качестве базового* для других классов — объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении.

При определении абстрактного класса необходимо иметь в виду следующее:

- ❑ абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;
- ❑ допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;
- ❑ если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать *полиморфные функции*, работающие с объектом любого типа в пределах одной иерархии.

Множественное наследование

Множественное наследование означает, что класс имеет несколько базовых классов. Если в базовых классах есть одноименные элементы, при этом может произойти конфликт идентификаторов, который устраняется с помощью операции доступа к области видимости:

```
class monstr{
    public: int get_health();
};
```

```
class hero{
    public: int get_health();
};
class ostrich: public monstr, public hero{
};
int main(){
    ostrich A;
    cout << A.monstr::get_health();
    cout << A.hero::get_health();
}
```

Как видно из примера, для вызова метода `get_health` требуется явно указывать класс, в котором он описан. Использование обычной для вызова метода класса конструкции `A.get_health()` приведет к ошибке, поскольку компилятор не в состоянии разобраться, к методу какого из базовых классов требуется обратиться.

Если у базовых классов есть общий предок, это приведет к тому, что производный от этих базовых класс унаследует два экземпляра полей предка, что чаще всего является нежелательным. Чтобы избежать такой ситуации, требуется при наследовании общего предка определить его как виртуальный класс:

```
class monstr{
};
class daemon: virtual public monstr{
};
class lady: virtual public monstr{
};
class baby: public daemon, public lady{
};
```

Класс `baby` содержит только один экземпляр полей класса `monstr`.

Множественное наследование применяется для того, чтобы обеспечить производный класс свойствами двух или более базовых. Чаще всего один из этих классов является основным, а другие обеспечивают некоторые дополнительные свойства, поэтому они называются классами подмешивания. По возможности классы подмешивания должны быть виртуальными и создаваться с помощью конструкторов без параметров, что позволяет избежать многих проблем, возникающих при ромбовидном наследовании (когда у базовых классов есть общий предок).

Отличия структур и объединений от классов

Структуры (struct) и объединения (union) представляют собой частные случаи классов.

Структуры отличаются от классов тем, что доступ к элементам, а также базовый класс при наследовании по умолчанию считаются public. Структуры предпочтительнее использовать для классов, все элементы которых доступны.

Отличия *объединений* от классов перечислены ниже:

- ☐ доступ в объединениях по умолчанию public, кроме того, в них вообще нельзя явным образом использовать спецификаторы доступа;
- ☐ объединение не может участвовать в иерархии классов;
- ☐ элементами объединения не могут быть объекты, содержащие конструкторы и деструкторы;
- ☐ объединение может иметь конструктор и другие методы, только не статические;
- ☐ в анонимном объединении нельзя описывать методы.

ГЛАВА 6

Шаблоны классов

В первой части этой книги были рассмотрены шаблоны функций (с. 85), с помощью которых можно отделить алгоритм от конкретных типов данных, с которыми он работает, передавая тип в качестве параметра. Шаблоны классов предоставляют аналогичную возможность, позволяя создавать параметризованные классы.

Параметризованный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемому в качестве параметра. Наиболее широкое применение шаблоны находят при создании *контейнерных классов*. Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними. Стандартная библиотека C++ содержит множество контейнерных классов для организации структур данных различного вида (они описаны в разделе «Контейнерные классы» на с. 295).

Преимущество использования шаблонов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

Создание шаблонов классов

Рассмотрим процесс создания шаблона класса на примере. В разделе «Линейные списки» (с. 115) был описан двусвязный список и приведены алгоритмы работы с ним. Поскольку списки часто применяются для организации данных, удобно описать список в виде класса, а так как может потребоваться хранить данные различных типов, этот класс должен быть параметризованным.

Сначала рассмотрим непараметризованную версию класса «список».

Список состоит из узлов, связанных между собой с помощью указателей. Каждый узел хранит целое число, являющееся ключом списка. Опишем вспомогательный класс для представления одного узла списка:

```
class Node{  
    public:
```



```

int d;                // Данные
Node *next;           // Указатель на последующий узел
Node *prev;           // Указатель на предыдущий узел
Node(int dat = 0){    // Конструктор
    d = dat; next = 0; prev = 0;
}
};

```

Поскольку этот класс будет описан внутри класса, представляющего список, поля для простоты доступа из внешнего класса сделаны доступными (public). Это позволяет обойтись без функций доступа и изменения полей. Назовем класс списка List:

```

class List{
    class Node{
    };
    Node *pbeg, *pend; // Указатели на начало и конец списка
public:
    List(){pbeg = 0; pend = 0;} // Конструктор
    ~List();                  // Деструктор
    void add(int d);           // Добавление узла в конец списка
    Node * find(int i);        // Поиск узла по ключу
    // Вставка узла d после узла с ключом key:
    Node * insert(int key, int d);
    bool remove(int key);      // Удаление узла
    void print();              // Печать списка в прямом направлении
    void print_back();         // Печать списка в обратном направлении
};

```

Рассмотрим реализацию методов класса. Метод add выделяет память под новый объект типа Node и присоединяет его к списку, обновляя указатели на его начало и конец:

```

void List::add(int d){
    Node *pv = new Node(d);    // Выделение памяти под новый узел
    if (pbeg == 0)pbeg = pend = pv; // Первый узел списка
    else{
        // Связывание нового узла с предыдущим:
        pv->prev = pend;
        pend->next = pv;
        pend = pv;}           // Обновление указателя на конец списка
}

```

При желании получить отсортированный список этот метод можно заменить на метод, аналогичный функции формирования отсортированного списка add_sort, приведенной в разделе «Линейные списки» на с. 119.

Метод find выполняет поиск узла с заданным ключом и возвращает указатель на него в случае успешного поиска и 0 в случае отсутствия такого узла в списке:

```

Node * List::find( int d ){
    Node *pv = pbeg;

```

```

while (pv){
    if(pv->d == d)break;
    pv = pv->next;
}
return pv;
}

```

Метод insert вставляет в список узел после узла с ключом key и возвращает указатель на вставленный узел. Если такого узла в списке нет, вставка не выполняется и возвращается значение 0:

```

Node * List::insert(int key, int d){
    if(Node *pkey = find(key)){ // Поиск узла с ключом key
        // Выделение памяти под новый узел и его инициализация:
        Node *pv = new Node(d);
        // Установление связи нового узла с последующим:
        pv->next = pkey->next;
        // Установление связи нового узла с предыдущим:
        pv->prev = pkey;
        // Установление связи предыдущего узла с новым:
        pkey->next = pv;
        // Установление связи последующего узла с новым:
        if( pkey != pend) (pv->next)->prev = pv;
        // Обновление указателя на конец списка,
        // если узел вставляется в конец:
        else pend = pv;
        return pv;
    }
    return 0;
}

```

Метод remove удаляет узел с заданным ключом из списка и возвращает значение true в случае успешного удаления и false, если узел с таким ключом в списке не найден:

```

bool List::remove(int key){
    if(Node *pkey = find(key)){
        if (pkey == pbeg){ // Удаление из начала списка
            pbeg = pbeg->next;
            pbeg->prev = 0;}
        else if (pkey == pend){ // Удаление из конца списка
            pend = pend->prev;
            pend->next = 0;}
        else { // Удаление из середины списка
            (pkey->prev)->next = pkey->next;
            (pkey->next)->prev = pkey->prev;}
        delete pkey;
        return true;}
    return false;
}

```

Методы печати списка в прямом и обратном направлении поэлементно просматривают список, переходя по соответствующим ссылкам:

```
void List::print(){
    Node *pv = pbeg;
    cout << endl << "list: ";
    while (pv){
        cout << pv->d << ' ';
        pv = pv->next;}
    cout << endl;
}

void List::print_back(){
    Node *pv = pend;
    cout << endl << "list back: ";
    while (pv){
        cout << pv->d << ' ';
        pv = pv->prev;}
    cout << endl;
}
```

Деструктор списка освобождает память из-под всех его элементов:

```
List::~~List(){
    if (pbeg != 0){
        Node *pv = pbeg;
        while (pv){
            pv = pv->next;
            delete pbeg;
            pbeg = pv;}
    }
}
```

Ниже приведен пример программы, использующей класс List. Программа аналогична приведенной на с. 116: она формирует список из 5 чисел, выводит его на экран, добавляет число в список, удаляет число из списка и снова выводит его на экран:

```
int main(){
    List L;
    for (int i = 1; i<6; i++) L.add(i);
    L.print();
    L.print_back();
    L.insert(2, 200);
    if (!L.remove(5))cout << "not found";
    L.print();
    L.print_back();
}
```

Класс List предназначен для хранения целых чисел. Чтобы хранить в нем данные любого типа, требуется описать этот класс как шаблон и передать тип в качестве параметра.

Синтаксис описания шаблона:

```
template <описание_параметров_шаблона> определение_класса;
```

Параметры шаблона перечисляются через запятую. В качестве параметров могут использоваться типы, шаблоны и переменные.

Типы могут быть как стандартными, так и определенными пользователем. Для их описания используется ключевое слово `class`. Внутри шаблона параметр типа может применяться в любом месте, где допустимо использовать спецификацию типа, например:

```
template <class Data> class List{
    class Node{
    public:
        Data d;
        Node *next;
        Node *prev;
        Node(Data dat = 0){d = dat; next = 0; prev = 0;}
    };
};
```

Класс `Data` можно рассматривать как формальный параметр, на место которого при компиляции будет подставлен конкретный тип данных.

Для любых параметров шаблона могут быть заданы значения по умолчанию, например:

```
template<class T> class myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map{
    C<K> key;
    C<V> value;
};
```

Область действия параметра шаблона — от точки описания до конца шаблона, поэтому параметр можно использовать при описании следующих за ним, например:

```
template<class T, T* p, class U = T> class X { /* ... */ };
```

Методы шаблона класса автоматически становятся шаблонами функций. Если метод описывается вне шаблона, его заголовок должен иметь следующие элементы:

```
template <описание_параметров_шаблона>
    возвращ_тип_иня_класса <параметры_шаблона >::
    имя_функции (список_параметров_функции)
```

Описание параметров шаблона в заголовке функции должно соответствовать шаблону класса, при этом имена параметров могут не совпадать. Проще рассмотреть синтаксис описания методов шаблона на примере:

```
template <class Data> void List<Data>::print()
{ /* тело функции */ }
```

Здесь `<class Data>` — описание параметра шаблона, `void` — тип возвращаемого функцией значения, `List` — имя класса, `<Data>` — параметр шаблона, `print` — имя функции без параметров.

В случае нескольких параметров порядок их следования в описании параметров и параметрах шаблона должен быть один и тот же, например:

```
template<class T1, class T2> struct A{
    void f1();
};
template<class T2, class T1> void A<T2, T1>::f1(){ _ }
```

Ниже перечислены правила описания шаблонов.

- ☐ Локальные классы не могут содержать шаблоны в качестве своих элементов.
- ☐ Шаблоны методов не могут быть виртуальными.
- ☐ Шаблоны классов могут содержать статические элементы, дружественные функции и классы.
- ☐ Шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.
- ☐ Внутри шаблона нельзя определять friend-шаблоны.

В качестве примера шаблона рассмотрим полное описание параметризованного класса двусвязного списка `List`.

```
template <class Data> class List{
    class Node{
    public:
        Data d;
        Node *next, *prev;
        Node(Data dat = 0){d = dat; next = 0; prev = 0;}
    };
    Node *pbeg, *pend;
    public:
        List(){pbeg = 0; pend = 0;}
        ~List():
        void add(Data d);
        Node * find(Data i);
        Node * insert(Data key, Data d);
        bool remove(Data key);
        void print();
        void print_back();
    };
    //-----
    template <class Data>
    List <Data>::~~List(){
        if (pbeg != 0){
            Node *pv = pbeg;
            while (pv){
                pv = pv->next; delete pbeg;
                pbeg = pv;}
        }
```

```
}  
//-----  
template <class Data>  
void List <Data>::print(){  
    Node *pv = pbeg;  
    cout << endl << "list: ";  
    while (pv){  
        cout << pv->d << ' ';  
        pv = pv->next;}  
    cout << endl;  
}  
//-----  
template <class Data>  
void List <Data>::print_back(){  
    Node *pv = pend;  
    cout << endl << " list back: ";  
    while (pv){  
        cout << pv->d << ' ';  
        pv = pv->prev;}  
    cout << endl;  
}  
//-----  
template <class Data>  
void List <Data>::add(Data d){  
    Node *pv = new Node(d);  
    if (pbeg == 0)pbeg = pend = pv;  
    else{  
        pv->prev = pend;  
        pend->next = pv;  
        pend = pv;}  
}  
//-----  
template <class Data>  
Node * List <Data>::find(Data d){  
    Node *pv = pbeg;  
    while (pv){  
        if(pv->d == d)break;  
        pv = pv->next;  
    }  
    return pv;  
}  
//-----  
template <class Data>  
Node * List <Data>::insert(Data key, Data d){  
    if(Node *pkey = find(key)){  
        Node *pv = new Node(d);  
        pv->next = pkey->next;
```



```

        pv->prev = pkey;
        pkey->next = pv;
        if( pkey != pend)(pv->next)->prev = pv;
        else pend = pv;
        return pv;}
    return 0;
}
//-----
template <class Data>
bool List <Data>::remove(Data key){
    if(Node *pkey = find(key)){
        if (pkey == pbeg){
            pbeg = pbeg->next; pbeg->prev = 0;}
        else if (pkey == pend){
            pend = pend->prev; pend->next = 0;}
        else {
            (pkey->prev)->next = pkey->next;
            (pkey->next)->prev = pkey->prev;}
        delete pkey; return true;}
    return false;
}

```

Если требуется использовать шаблон List для хранения данных не встроенного, а определенного пользователем типа, в описание этого типа необходимо добавить перегрузку операции вывода в поток (об этом рассказывается в разделе «Потоки и типы, определенными пользователем» на с. 284) и сравнения на равенство, а если для его полей используется динамическое выделение памяти, то и операцию присваивания.

При определении синтаксиса шаблона было сказано, что в него, кроме типов и шаблонов, могут передаваться *переменные*. Они могут быть целого или перечисляемого типа, а также указателями или ссылками на объект или функцию. В теле шаблона они могут применяться в любом месте, где допустимо использовать константное выражение. В качестве примера создадим шаблон класса, содержащего блок памяти определенной длины и типа:

```

template <class Type, int kol>
class Block{
public:
    Block(){p = new Type [kol];}
    ~Block(){delete [] p;}
    operator Type *():
protected:
    Type * p;
};
template <class Type, int kol>
Block <Type, kol>::operator Type *(){
    return p;
}

```

После создания и отладки шаблоны классов удобно помещать в заголовочные файлы.

Использование шаблонов классов

Чтобы создать при помощи шаблона конкретный объект конкретного класса (этот процесс называется инстанцированием), при описании объекта после имени шаблона в угловых скобках перечисляются его аргументы:

```
имя_шаблона <аргументы>
    имя_объекта [(параметры_конструктора)];
```

Аргументы должны соответствовать параметрам шаблона. Имя шаблона вместе с аргументами можно воспринимать как уточненное имя класса. Примеры создания объектов по шаблонам, описанным в предыдущем разделе:

```
List <int> List_int;
List <double> List_double;
List <monstr> List_monstr;
Block <char, 128> buf;
Block <monstr, 100> stado;
```

При использовании параметров шаблона по умолчанию список аргументов может оказаться пустым, при этом угловые скобки опускать нельзя:

```
template<class T = char> class String;
String<>* p;
```

Если параметром шаблона является шаблон, имеющий специализацию, она учитывается при инстанцировании:

```
template<class T> class A{           // Исходный шаблон
    int x;
};
template<class T> class A<T*> {     // Специализация шаблона
    long x;
};
template<template<class U> class V> class C{
    V<int> y;
    V<int*> z;
};

C<A> c;
```

В этом примере `V<int>` внутри `C<A>` использует исходный шаблон, поэтому `c.y.x` имеет тип `int`, а `V<int*>` использует специализацию шаблона, поэтому `c.z.x` имеет тип `long`.

На месте формальных параметров, являющихся переменными целого типа, должны стоять константные выражения.

После создания объектов с помощью шаблона с ними можно работать так же, как с объектами обычных классов, например:

```
for (int i = 1; i<10; i++)List_double.add(i * 0.08);
List_double.print();
//-----
for (int i = 1; i<10; i++)List_monstr.add(i);
```

```
List_monstr.print();
//-----
strcpy(buf, "Очень важное сообщение");
cout << buf << endl;
```

Для упрощения использования шаблонов классов можно применить переименование типов с помощью typedef:

```
typedef List <double> Ldbl;
Ldbl List_double;
```

Специализация шаблонов классов

Каждая версия класса или функции, создаваемая по шаблону, содержит одинаковый базовый код; изменяется только то, что связано с параметрами шаблона. При этом эффективность работы версий, создаваемых для различных типов данных, может сильно различаться.

Если для какого-либо типа данных существует более эффективный код, можно либо предусмотреть для этого типа специальную реализацию отдельных методов, либо полностью переопределить (специализировать) шаблон класса.

Для *специализации метода* требуется определить вариант его кода, указав в заголовке конкретный тип данных. Например, если заголовок обобщенного метода print шаблона List имеет вид

```
template <class Data> void List <Data>::print();
```

специализированный метод для вывода списка символов будет выглядеть следующим образом:

```
void List <char>::print(){
    ... // Тело специализированного варианта метода print
}
```

Если в программе создать экземпляр шаблона List типа char, соответствующий вариант метода будет вызван автоматически.

При *специализации целого класса* после описания обобщенного варианта класса помещается полное описание специализированного класса, при этом требуется заново определить все его методы. Допустим, требуется специализировать шаблон Block, описанный в предыдущем разделе, для хранения 100 целых величин¹:

```
class Block<int, 100>{
public:
    Block(){p = new int [100];}
    ~Block(){delete [] p;}
    operator int *():
protected:
    int * p;
};
```

¹ Не стоит искать в этом глубокий смысл — пример приведен для демонстрации синтаксиса.

```
Block<int, 100>::operator int *(){  
    return p;  
}
```

При определении экземпляров шаблона `Block` с параметрами `int` и `100` будет задействован специализированный вариант.

Достоинства и недостатки шаблонов

Шаблоны представляют собой мощное и эффективное средство обращения с различными типами данных, которое можно назвать параметрическим полиморфизмом, а также обеспечивают безопасное использование типов, в отличие от макросов препроцессора. Однако следует иметь в виду, что программа, использующая шаблоны, содержит полный код для каждого порожденного типа, что может увеличить размер исполняемого файла. Кроме того, с некоторыми типами данных шаблоны могут работать не так эффективно, как с другими. В этом случае имеет смысл использовать специализацию шаблона.

Стандартная библиотека C++ предоставляет большой набор шаблонов для различных способов организации хранения и обработки данных (см. раздел «Контейнерные классы», с. 295).

ГЛАВА 7

Обработка исключительных ситуаций

Исключительная ситуация, или исключение — это возникновение непредвиденного или аварийного события, которое может порождаться некорректным использованием аппаратуры. Например, это деление на ноль или обращение по несуществующему адресу памяти. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. С++ дает программисту возможность восстанавливать программу и продолжать ее выполнение.

Исключения С++ не поддерживают обработку асинхронных событий, таких, как ошибки оборудования или обработку прерываний, например, нажатие клавиш Ctrl+C. Механизм исключений предназначен только для событий, которые происходят в результате работы самой программы и указываются явным образом. Исключения возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. При этом другая часть программы может попытаться сделать что-нибудь иное.

Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработка. Это важно не только для лучшей структуризации программы. Главной причиной является то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место возникновения. Это особенно актуально при использовании библиотечных функций и программ, состоящих из многих модулей.

Другое достоинство исключений состоит в том, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение, параметры или глобальные переменные, поэтому интерфейс функций не раздувается. Это особенно важно, например, для конструкторов, которые по синтаксису не могут возвращать значение.

ПРИМЕЧАНИЕ

В принципе, ничто не мешает рассматривать в качестве исключений не только ошибки, но и нормальные ситуации, возникающие при обработке данных, но это не имеет преимуществ перед другими решениями и не улучшает структуру и читаемость программы.

Общий механизм обработки исключений

Место, в котором может произойти ошибка, должно входить в контролируемый блок — составной оператор, перед которым записано ключевое слово `try`.

Рассмотрим, каким образом реализуется обработка исключительных ситуаций.

- Обработка исключения начинается с появления ошибки. Функция, в которой она возникла, генерирует исключение. Для этого используется ключевое слово `throw` с параметром, определяющим вид исключения. Параметр может быть константой, переменной или объектом и используется для передачи информации об исключении его обработчику.
- Отыскивается соответствующий обработчик исключения и ему передается управление.
- Если обработчик исключения не найден, вызывается стандартная функция `terminate`, которая вызывает функцию `abort`, аварийно завершающую текущий процесс. Можно установить собственную функцию завершения процесса.

В первой части книги (см. раздел «Функции», с. 73) говорилось о том, что при вызове каждой функции в стеке создается область памяти для хранения локальных переменных и адреса возврата в вызывающую функцию. Термин *стек вызовов* обозначает последовательность вызванных, но еще не завершившихся функций. *Раскручиванием стека* называется процесс освобождения памяти из-под локальных переменных и возврата управления вызывающей функции. Когда функция завершается, происходит естественное раскручивание стека. Тот же самый механизм используется и при обработке исключений. Поэтому после того, как исключение было зафиксировано, исполнение не может быть продолжено с точки генерации исключения. Подробнее этот механизм рассматривается в следующем разделе.

Синтаксис исключений

Ключевое слово `try` служит для обозначения *контролируемого блока* — кода, в котором может генерироваться исключение. Блок заключается в фигурные скобки:

```
try{  
    ...  
}
```

Все функции, прямо или косвенно вызываемые из `try`-блока, также считаются ему принадлежащими.

Генерация (порождение) исключения происходит по ключевому слову `throw`, которое употребляется либо с параметром, либо без него:

```
throw [ выражение ];
```

Тип выражения, стоящего после `throw`, определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, и происходит поиск соответствующего обработчика и передача ему управления. Как правило, исключение генерируется не непосредственно в `try`-блоке, а в функциях, прямо или косвенно в него вложенных.

Не всегда исключение, возникшее во внутреннем блоке, может быть сразу правильно обработано. В этом случае используются вложенные контролируемые блоки, и исключение передается на более высокий уровень с помощью ключевого слова `throw` без параметров.

Обработчики исключений начинаются с ключевого слова `catch`, за которым в скобках следует тип обрабатываемого исключения. Они должны располагаться непосредственно за `try`-блоком. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Синтаксис обработчиков напоминает определение функции с одним параметром — типом исключения. Существует три формы записи:

```
catch(тип имя){ ... /* тело обработчика */ }
catch(тип){ ... /* тело обработчика */ }
catch(_){ ... /* тело обработчика */ }
```

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий — например, вывода информации об исключении. Вторая форма не предполагает использования информации об исключении, играет роль только его тип. Многоточие вместо параметра обозначает, что обработчик перехватывает все исключения. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа следует помещать после всех остальных. Пример:

```
catch(int i){
    ... // Обработка исключений типа int
}
catch(const char *){
    ... // Обработка исключений типа const char*
}
catch(Overflow){
    ... // Обработка исключений класса Overflow
}
catch(j){
    ... // Обработка всех необслуженных исключений
}
```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений. Туда же, минуя код всех обработчиков, передается управление, если исключение в `try`-блоке не было сгенерировано.

Перехват исключений

Когда с помощью `throw` генерируется исключение, функции исполнительной библиотеки C++ выполняют следующие действия:

- 1) создают копию параметра `throw` в виде статического объекта, который существует до тех пор, пока исключение не будет обработано;
- 2) в поисках подходящего обработчика раскручивают стек, вызывая деструкторы локальных объектов, выходящих из области действия;
- 3) передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

При раскручивании стека все обработчики на каждом уровне просматриваются последовательно, от внутреннего блока к внешнему, пока не будет найден подходящий обработчик.

Обработчик считается найденным, если тип объекта, указанного после `throw`:

- тот же, что и указанный в параметре `catch` (параметр может быть записан в форме `T`, `const T`, `T&` или `const T&`, где `T` — тип исключения);
- является производным от указанного в параметре `catch` (если наследование производилось с ключом доступа `public`);
- является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре `catch`.

Из вышесказанного следует, что обработчики производных классов следует размещать до обработчиков базовых, поскольку в противном случае им никогда не будет передано управление. Обработчик указателя типа `void` автоматически скрывает указатель любого другого типа, поэтому его также следует размещать после обработчиков указателей конкретного типа.

Рассмотрим пример.

```
#include <fstream.h>
class Hello{
// Класс, информирующий о своем создании и уничтожении
public:
    Hello(){cout << "Hello!" << endl;}
    ~Hello(){cout << "Bye!" << endl;}
};
void f1(){
    ifstream ifs("\\INVALID\\FILE\\NAME"); // Открываем файл
    if (!ifs){
        cout << "Генерируем исключение" << endl;
        throw "Ошибка при открытии файла";}
}
void f2(){
    Hello H; // Создаем локальный объект
    f1();    // Вызываем функцию, генерирующую исключение
}
int main(){
```

```

try{
    cout << "Входим в try-блок" << endl;
    f2();
    cout << "Выходим из try-блока" << endl;
}
catch(int i){
    cout << "Вызван обработчик int, исключение - " << i << endl;
    return -1;
}
catch(const char * p){
    cout << "Вызван обработчик const char*, исключение - "
        << p << endl;
    return -1;
}
catch(j){
    cout << "Вызван обработчик всех исключений" << endl;
    return -1;
}
return 0; // Все обошлось благополучно
}

```

Результаты выполнения программы:

```

Входим в try-блок
Hello!
Генерируем исключение
Bye!
Вызван обработчик const char *, исключение - Ошибка при открытии файла

```

Обратите внимание, что после порождения исключения был вызван деструктор локального объекта, хотя управление из функции `f1` было передано обработчику, находящемуся в функции `main`. Сообщение «Выходим из try-блока» не было выведено. Для работы с файлом в программе использовались потоки (о них рассказывается в главе 10 «Потоковые классы» на с. 265).

Таким образом, механизм исключений позволяет корректно уничтожать объекты при возникновении ошибочных ситуаций. Поэтому *выделение и освобождение ресурсов полезно оформлять в виде классов*, конструктор которых выделяет ресурс, а деструктор освобождает. В качестве примера можно привести класс для работы с файлом. Конструктор класса открывает файл, а деструктор — закрывает. В этом случае есть гарантия, что при возникновении ошибки файл будет корректно закрыт, и информация не будет утеряна.

Как уже упоминалось, исключение может быть как стандартного, так и определенного пользователем типа. При этом нет необходимости определять этот тип глобально — достаточно, чтобы он был известен в точке порождения исключения и в точке его обработки. Класс для представления исключения можно описать внутри класса, при работе с которым оно может возникать. Конструктор копирования этого класса должен быть объявлен как `public`, поскольку иначе будет невозможно создать копию объекта при генерации исключения (конструктор копирования, создаваемый по умолчанию, имеет спецификатор `public`).

Список исключений функции

В заголовке функции можно задать список исключений, которые она может прямо или косвенно породить. Поскольку заголовок является интерфейсом функции, указание в нем списка исключений дает пользователям функции необходимую информацию для ее использования, а также гарантию, что при возникновении непредвиденного исключения эта ситуация будет обнаружена.

Алгоритм обработки исключения представлен на рис. 7.1.

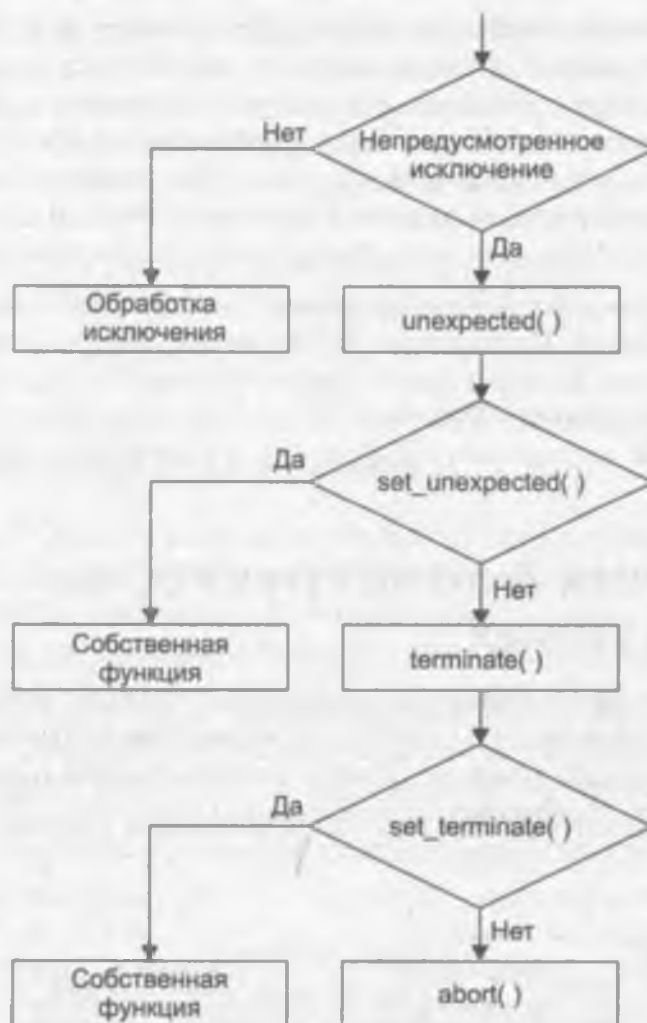


Рис. 7.1. Алгоритм обработки исключения

Типы исключений перечисляются в скобках через запятую после ключевого слова `throw`, расположенного за списком параметров функции, например:

```
void f1() throw (int, const char*) { /* Тело функции */ }
void f2() throw (Oops*) { /* Тело функции */ }
```

Функция `f1` должна генерировать исключения только типов `int` и `const char*`. Функция `f2` должна генерировать только исключения типа указателя на класс `Oops` или производных от него классов.

Если ключевое слово `throw` не указано, функция может генерировать любое исключение. Пустой список означает, что функция не должна порождать исключений:

```
void f() throw (){
    // Тело функции. не порождающей исключений
}
```

Исключения не входят в прототип функции. При переопределении в производном классе виртуальной функции можно задавать список исключений, такой же или более ограниченный, чем в соответствующей функции базового класса.

Указание списка исключений ни к чему не обязывает — функция может прямо или косвенно породить исключение, которое она обещала не использовать. Эта ситуация обнаруживается во время исполнения программы и приводит к вызову стандартной функции `unexpected`, которая по умолчанию просто вызывает функцию `terminate`. С помощью функции `set_unexpected` можно установить собственную функцию, которая будет вызываться вместо `terminate` и определять действие программы при возникновении непредвиденной исключительной ситуации.

Функция `terminate` по умолчанию вызывает функцию `abort`, которая завершает выполнение программы. С помощью функции `set_terminate` можно установить собственную функцию, которая будет вызываться вместо `abort` и определять способ завершения программы. Функции `set_unexpected` и `set_terminate` описаны в заголовочном файле `<exception>`.

Исключения в конструкторах и деструкторах

Язык C++ не позволяет возвращать значение из конструктора и деструктора. Механизм исключений дает возможность сообщить об ошибке, возникшей в конструкторе или деструкторе объекта. Для иллюстрации создадим класс `Vector`, в котором ограничивается количество запрашиваемой памяти:

```
class Vector{
public:
    class Size{}:           // Класс исключения
    enum {max = 32000}:     // Максимальная длина вектора
    Vector(int n)           // Конструктор
        { if (n<0 || n>max) throw Size(); ... }
};
```

При использовании класса `Vector` можно предусмотреть перехват исключений типа `Size`:

```
try{
    Vector *p = new Vector(i);
    ...
}
```

```
catch(Vector::Size){  
    // Обработка ошибки размера вектора  
}
```

В обработчике может использоваться стандартный набор основных способов выдачи сообщений об ошибке и восстановления. Внутри класса, определяющего исключение, может храниться информация об исключении, которая передается обработчику. Смысл этой техники заключается в том, чтобы обеспечить передачу информации об ошибке из точки ее обнаружения в место, где для обработки ошибки имеется достаточно возможностей.

Если в конструкторе объекта генерируется исключение, автоматически вызываются деструкторы для полностью созданных в этом блоке к текущему моменту объектов, а также для полей данных текущего объекта, являющихся объектами, и для его базовых классов. Например, если исключение возникло при создании массива объектов, деструкторы будут вызваны только для успешно созданных элементов.

Если объект создается в динамической памяти с помощью операции `new` и в конструкторе возникнет исключение, память из-под объекта корректно освобождается.

Иерархии исключений

Использование собственных классов исключений предпочтительнее применения стандартных типов данных. С помощью классов можно более гибко организовать передачу информации об исключении, легче дифференцировать обработку исключений, а кроме того, появляется возможность использовать иерархии классов.

Поскольку механизм управления исключениями позволяет создать обработчик для базового класса, родственные исключения часто можно представить в виде иерархии. Производя исключения от общего базового класса, можно в обработчике перехватывать ссылку или указатель на базовый класс, используя полиморфизм. Например, в математической библиотеке можно организовать классы следующим образом:

```
class Matherr{};  
class Overflow: public Matherr{};    // Переполнение  
class Underflow: public Matherr{};   // Исчезновение порядка  
class ZeroDivide: public Matherr{};  // Деление на ноль
```

Для представления ошибок ввода/вывода могут использоваться следующие классы:

```
class IOerr{};  
class Readerr: public IOerr{}; // Ошибка чтения  
class Writerr: public IOerr{}; // Ошибка записи  
class Seekerr: public IOerr{}; // Ошибка поиска
```

В зависимости от обстоятельств можно использовать либо обработчик исключений базового класса, который будет перехватывать и производные исключения, либо собственные обработчики производных классов.

Существует ряд *стандартных исключений*, которые генерируются операциями или функциями C++ (см. главу 16 «Другие средства стандартной библиотеки», с. 378). Все они являются производными от библиотечного класса `exception`, описанного в заголовочном файле `<stdexcept>`. Например, операция `new` при неудачном выделении памяти генерирует исключение типа `bad_alloc`.

Программист может определить собственные исключения, производные от стандартных.

ГЛАВА 8

Преобразования типов

При выполнении программы производятся явные и неявные преобразования величин из одного типа в другой. Неявные преобразования выполняются в соответствии с правилами, приведенными на с. 38 и в приложении 3. Для выполнения явных преобразований типа в C++ существует целая группа операций — `const_cast`, `dynamic_cast`, `reinterpret_cast` и `static_cast`, а также операция приведения типа, унаследованная из языка C, которая уже использовалась в этой книге (с. 56). Для начала рассмотрим ее подробнее.

Операция приведения типов в стиле C

Операция может записываться в двух формах:

тип (выражение)
(тип) выражение

Результатом операции является значение заданного типа, например:

```
int a = 2;  
float b = 6.8;  
printf("%lf %d", double(a), (int) b);
```

Величина `a` преобразуется к типу `double`, а переменная `b` — к типу `int` с отсечением дробной части, в обоих случаях внутренняя форма представления результата операции преобразования иная, чем форма исходного значения.

Необходимость в преобразовании типа возникает, например, в случае, когда функция возвращает указатель на тип `void`, который требуется присвоить переменной конкретного типа для последующего выполнения с ней каких-либо действий:

```
float *q = (float *) malloc(100 * sizeof(float));
```

Явное преобразование типа является источником возможных ошибок, поскольку вся ответственность за его результат возлагается на программиста. Поэтому в C++ введены операции, позволяющие выполнять частичный контроль выпол-

няемых преобразований или сделать намерения программиста более явными для понимания. Рассмотренное выше преобразование в стиле C оставлено в C++ только для нисходящей совместимости, и *использовать его не рекомендуется*. В зависимости от вида требуемого преобразования необходимо использовать соответствующую ему операцию приведения типа.

ПРИМЕЧАНИЕ

В ранних версиях компиляторов C++ операции приведения типа `const_cast`, `dynamic_cast`, `reinterpret_cast` и `static_cast` не поддерживаются.

Операция `const_cast`

Операция служит для удаления модификатора `const`. Как правило, она используется при передаче в функцию константного указателя на место формального параметра, не имеющего модификатора `const`. Формат операции:

```
const_cast <тип> (выражение)
```

Обозначенный тип должен быть таким же, как и тип выражения, за исключением модификатора `const`. Обычно это указатель. Операция формирует результат указанного типа.

Необходимость введения этой операции обусловлена тем, что программист, реализующий функцию, не обязан описывать не изменяемые в ней формальные параметры как `const`, хотя это и рекомендуется. Правила C++ запрещают передачу константного указателя на место обычного. Операция `const_cast` введена для того, чтобы обойти это ограничение. Естественно, функция не должна пытаться изменить значение, на которое ссылается передаваемый указатель, иначе результат выполнения программы не определен.

Пример:

```
void print(int *p){    // Функция не изменяет значение *p
    cout << *p;
}
const int *p;
```

```
/* print(&p): Ошибка, поскольку p объявлен как указатель на константу */
```

Операция `const_cast` используется в том случае, когда программист уверен, что в теле функции значение, на которое ссылается указатель, не изменяется. Естественно, если есть возможность добавить к описанию формального параметра модификатор `const`, это предпочтительнее использования преобразования типа при вызове функции.

Операция `dynamic_cast`

Операция применяется для преобразования указателей родственных классов иерархии, в основном — указателя базового класса в указатель на производный

класс, при этом во время выполнения программы производится проверка допустимости преобразования.

Формат операции:

`dynamic_cast <тип *> (выражение)`

Выражение должно быть указателем или ссылкой на класс, тип — базовым или производным для этого класса. После проверки допустимости преобразования в случае успешного выполнения операция формирует результат заданного типа, в противном случае для указателя результат равен нулю¹, а для ссылки порождается исключение `bad_cast`. Если заданный тип и тип выражения не относятся к одной иерархии, преобразование не допускается.

Преобразование из базового класса в производный называют *понижающим* (downcast), так как графически в иерархии наследования принято изображать производные классы ниже базовых. Приведение из производного класса в базовый называют *повышающим* (upcast), а приведение между производными классами одного базового или, наоборот, между базовыми классами одного производного — *перекрестным* (crosscast).

Повышающее преобразование

Выполнение с помощью операции `dynamic_cast` повышающего преобразования равносильно простому присваиванию (при открытом наследовании):

```
class B { /* ... */ };
class C: public B { /* ... */ };
C* c = new C;
B* b = dynamic_cast<B*>(c); // Эквивалентно B* b = c;
```

Понижающее преобразование

Чаще всего операция `dynamic_cast` применяется при понижающем преобразовании — когда компилятор не имеет возможности проверить правильность приведения.

Производные классы могут содержать функции, которых нет в базовых классах. Для их вызова через указатель базового класса нужно иметь уверенность, что этот указатель в действительности ссылается на объект производного класса. Такая проверка производится в момент выполнения приведения типа с использованием RTTI (run-time type information) — «информации о типе во время выполнения программы». Для того чтобы проверка допустимости могла быть выполнена, аргумент операции `dynamic_cast` должен быть полиморфного типа, то есть иметь хотя бы один виртуальный метод (см. с. 205).

¹ Если выражение равно нулю, результат также равен нулю.

ПРИМЕЧАНИЕ

Для использования RTTI необходимо подключить к программе заголовочный файл `<typeinfo>`. Кроме того, необходимо, чтобы был установлен соответствующий режим компилятора.

Для полиморфного объекта реализация операции `dynamic_cast` весьма эффективна, поскольку ссылка на информацию о типе объекта заносится в таблицу виртуальных методов, и доступ к ней осуществляется легко.

С точки зрения логики требование, чтобы объект был полиморфным, также оправдано: ведь если класс не имеет виртуальных методов, его нельзя безопасным образом использовать, не зная точный тип указателя. А если тип известен, использовать операцию `dynamic_cast` нет необходимости.

Результат применения операции `dynamic_cast` к указателю всегда требуется проверять явным образом. В приведенном ниже примере описан полиморфный базовый класс `B` и производный от него класс `C`, в котором определена функция `f2`. Для того чтобы вызывать ее из функции `demo` только в случае, когда последней передается указатель на объект производного класса, используется операция `dynamic_cast` с проверкой результата преобразования:

```
#include <iostream.h>
#include <typeinfo.h>
class B{
    public: virtual void f1(){};
};
class C: public B{
    public: void f2(){cout << "f2:"};
};
void demo(B* p){
    C* c = dynamic_cast<C*>(p);
    if (c) c->f2();
    else cout << "Передан не класс C";
}
int main(){
    B* b = new B;
    demo(b); // Выдается сообщение "Передан не класс C"
    C* c = new C;
    demo(c); // Выдается сообщение "f2" (правильно)
    return 0;
}
```

При использовании в этом примере вместо `dynamic_cast` приведения типов в стиле `C`, например:

```
C* c = (C*) p;
```

проконтролировать допустимость операции невозможно, и если указатель `p` на самом деле не ссылается на объект класса `C`, это приведет к ошибке.

Другим недостатком приведения в стиле `C` является невозможность преобразования в производный виртуального базового класса, это запрещено синтакси-

чески. С помощью операции `dynamic_cast` такое преобразование возможно при условии, что класс является полиморфным и преобразование недвусмысленно. Рассмотрим пример, в котором выполняется *понижающее преобразование виртуального базового класса*:

```
#include <iostream.h>
#include <typeinfo.h>
class A{
    public: virtual ~A(){};
class B: public virtual A{};
class C: public virtual A{};
class D: public B, public C{};
void demo(A *a){
    D* d = dynamic_cast<D*>(a);
    if (d) { _ }
}
int main(){
    D *d = new D; demo(d);
    return 0;
}
```

Преобразование ссылок

Для аргумента-ссылки смысл операции преобразования несколько иной, чем для указателя. Поскольку ссылка всегда указывает на конкретный объект, операция `dynamic_cast` должна выполнять преобразование именно к типу этого объекта. Корректность приведения проверяется автоматически, в случае несовпадения порождается исключение `bad_cast`:

```
#include <iostream.h>
#include <typeinfo.h>
class B{
    public: virtual void f1(){};
};
class C: public B{
    public: void f2(){ _ };
};
void demo(B& p){
    try{
        C& c = dynamic_cast<C&>(p);
        c.f2();
    }
    catch(Bad_cast){
        _
    }
}
int main(){
    B* b = new B; demo(*b);    // Порождается исключение
```



```

C* c = new C; demo(*c);    // Правильно
return 0;
}

```

Перекрестное преобразование

Операция `dynamic_cast` позволяет выполнять безопасное преобразование типа между производными классами одного базового класса, например:

```

#include <iostream.h>
#include <typeinfo.h>
class B{
    public: virtual void f1(){};
};
class C: public B{
    public: void f2(){ _ };
};
class D: public B{ _ };
void demo(D* p){
    C* c = dynamic_cast<C*>(p);
    if(c)c->f2();
    else cout <<" не выполнено ";
}
int main(){
    B* b = new C; demo((D*)b);
    return 0;
}

```

Классы C и D являются производными от класса B. Функции `demo` передается указатель на класс D, являющийся на самом деле указателем на «братский» для него класс C, поэтому динамическое преобразование типа из D в C в функции `demo` завершается успешно.

ПРИМЕЧАНИЕ

Указатель для передачи в функцию сформирован искусственным образом исключительно для демонстрации возможности динамического преобразования типа.

При необходимости можно осуществить преобразование между базовыми классами одного производного класса, например:

```

#include <iostream.h>
#include <typeinfo.h>
class B{
    public: virtual void f1(){ _ }; // B   C
};                                     // \ /
class C{                               //   D
    public: virtual void f2(){ _ };
};
class D: public B, public C{};

```

```

void demo(B* b){
    C* c = dynamic_cast<C*>(b);
    if(c)c->f2();
}
int main(){
    D* d = new D; demo(d);
    return 0;
}

```

Класс D является потомком B и C, поэтому содержит методы обоих классов. Если в функцию demo передается на самом деле указатель не на B, а на D, его можно преобразовать к его второму базовому классу C.

Операция static_cast

Операция static_cast используется для преобразования типа на этапе компиляции между:

- целыми типами;
- целыми и вещественными типами;
- целыми и перечисляемыми типами;
- указателями и ссылками на объекты одной иерархии, при условии, что оно однозначно и не связано с понижающим преобразованием виртуального базового класса.

Формат операции:

```
static_cast <тип> (выражение)
```

Результат операции имеет указанный тип, который может быть ссылкой, указателем, арифметическим или перечисляемым типом.

При выполнении операции внутреннее представление данных может быть модифицировано, хотя численное значение остается неизменным. Например:

```

float f = 100;
int i = static_cast <int> (f);
// Целые и вещественные имеют различное внутреннее представление

```

Такого рода преобразования применяются обычно для подавления сообщений компилятора о возможной потере данных в том случае, когда есть уверенность, что требуется выполнить именно это действие. Результат преобразования остается на совести программиста.

Операция static_cast позволяет выполнять преобразования из производного класса в базовый и наоборот без ограничений:

```

class B{};
class C: public B{};
C c;
B *bp = static_cast<B*>(c); // Производный -> базовый
B b;
C &cp = static_cast<C&>(b); // Базовый -> производный

```

Преобразование выполняется при компиляции, при этом объекты могут не быть полиморфными. Программист должен сам отслеживать допустимость дальнейших действий с преобразованными величинами.

В общем случае использование для преобразования указателей родственных классов иерархии предпочтительнее использовать операцию `dynamic_cast`. В этом случае если преобразование возможно на этапе компиляции, генерируется тот же код, что и для `static_cast`. Кроме того, `dynamic_cast` допускает перекрестное преобразование, нисходящее приведение виртуального базового класса и производит проверку допустимости приведения во время выполнения.

Операция `reinterpret_cast`

Операция `reinterpret_cast` применяется для преобразования не связанных между собой типов, например, указателей в целые или наоборот, а также указателей типа `void*` в конкретный тип. При этом внутреннее представление данных остается неизменным, а изменяется только точка зрения компилятора на данные.

Формат операции:

`reinterpret_cast <тип> (выражение)`

Результат операции имеет указанный тип, который может быть ссылкой, указателем, целым или вещественным типом.

Пример:

```
char *p = reinterpret_cast <char*>(malloc(100));  
long l = reinterpret_cast <long>(p);
```

Различие между `static_cast` и `reinterpret_cast` позволяет компилятору производить минимальную проверку при использовании `static_cast`, а программисту — обозначать опасные преобразования с помощью `reinterpret_cast`. Результат преобразования остается на совести программиста.

Динамическое определение типа

Механизм идентификации типа во время выполнения программы (RTTI) позволяет определять, на какой тип в текущий момент времени ссылается указатель, а также сравнивать типы объектов. Для доступа к RTTI в стандарт языка введена операция `typeid` и класс `type_info`¹.

Формат операции `typeid`:

`typeid (тип)`
`typeid (выражение)`

Операция принимает в качестве параметра имя типа или выражение и возвращает ссылку на объект класса `type_info`, содержащий информацию о типе. Если операция не может определить тип операнда, порождается исключение `bad_typeid`.

¹ В старых версиях компиляторов этот класс может называться `Type_info` или `typeinfo`.

Когда операнд представляет собой указатель или ссылку на полиморфный тип, результатом является динамическая информация о типе (то есть объект `type_info` содержит информацию о типе объекта, на который в данный момент ссылается указатель).

Если операндом является выражение, представляющее собой ссылку на непотоморфный тип, объект `type_info` содержит информацию о типе выражения, а не о типе объекта, на который оно ссылается.

Операцию `typeid` можно использовать как с основными, так и с производными типами данных.

Класс `type_info` описан в заголовочном файле `<typeinfo>` следующим образом:

```
class type_info{
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
};
```

Метод `name` возвращает указатель на строку, представляющую имя типа, описываемого объектом типа `type_info`.

Виртуальный деструктор делает класс `type_info` полиморфным.

Конструктор копирования и операция присваивания объявлены как `private`, чтобы исключить возможность случайного копирования и присваивания объектов класса.

Операции `==` и `!=` позволяют сравнивать два объекта на равенство и неравенство, а функция `before` выполняет побуквенное сравнение имен двух типов. Для сравнения используется конструкция вида:

```
typeid(T1).before(typeid(T2))
```

Если имя типа `T1` лексикографически предшествует имени `T2`, результат будет истинным.

Точная информация о типе объекта во время выполнения программы может потребоваться, например, когда программист расширяет функциональность некоторого библиотечного базового класса с помощью производного, и невозможно или бессмысленно добавлять к базовому классу виртуальные функции. Например:

```
#include <typeinfo.h>
class B{
public: virtual ~B(){};
};
class C: public B{
public: virtual void some_method(){ _ };
```

```
};  
void demo(B* p){  
    if (typeid(*p) == typeid(C))  
        dynamic_cast<C*>(p)->some_method();  
}  
int main(){  
    C* c = new C; demo(c);  
    return 0;  
}
```

Информацию о типе полезно использовать и в диагностических целях:

```
void print_type(some_obj *p){  
    cout << typeid(*p).name();  
}
```

Операция `typeid` не должна применяться вместо виртуальных функций и в случаях, когда тип объекта можно определить на этапе компиляции.

ГЛАВА 9

Рекомендации по программированию

В этой части были рассмотрены объектно-ориентированные возможности языка C++, и теперь пришло время остановиться на проблемах, которые возникают в процессе написания программ: какие средства языка предпочтительнее выбирать в разных случаях, на что обратить особое внимание, чего нужно избегать, чтобы программа получилась эффективной и легко сопровождаемой.

Однозначные рекомендации, как всегда, дать невозможно — только понимание механизма работы C++ позволит грамотно его использовать.

Создание любого программного продукта начинается с процесса проектирования, и одна из первых задач, возникающих при этом, — определить, должна программа быть объектно-ориентированной или нет. Объектно-ориентированное программирование, примененное к задаче, в которой в нем нет необходимости, только увеличит объем программы и сложность ее написания. Если в процессе анализа постановки задачи выясняется, что необходимости в иерархии классов нет, чаще всего можно обойтись структурным подходом. Б. Страуструп считает, что «везде, где не нужно более одного объекта определенного типа, достаточно стиля программирования с сокращением данных при помощи модулей».

Смешивать два подхода в одном проекте не рекомендуется, поскольку может получиться, что готовый продукт обладает недостатками и структурного, и объектно-ориентированного принципов построения.

Технологии проектирования программ посвящено много литературы, и освещение этого вопроса не входит в задачу данной книги. Тем, кто собирается приниматься за серьезные программные проекты, рекомендуется предварительно изучить ставшие классическими книги Ф. Брукса [6], Г. Буча [7] и Б. Страуструпа [17]. Далее рассматриваются вопросы, на которые следует обращать внимание на следующем этапе создания программы — в процессе написания программного кода.

При создании класса, то есть нового типа данных, следует хорошо продумать его интерфейс — средства работы с классом, доступные использующим его программам. Интерфейс хорошо спроектированного класса интуитивно ясен, непро-

тиворечив и обозрим. Как правило, он должен включать только методы, но не поля данных. Поля данных должны быть скрытыми (*private*). Это дает возможность впоследствии изменить реализацию класса без изменений в его интерфейсе, а также регулировать доступ к полям класса с помощью набора предоставляемых пользователю методов.

Не следует определять методы типа `get/set` для всех скрытых полей класса — это все равно, что открыть к ним доступ, только более сложным способом. Важно помнить, что поля класса вводятся только для того, чтобы реализовать свойства класса, представленные в его интерфейсе с помощью методов. Конечно, нет ничего плохого в том, чтобы установить или получить значение некоторого поля с помощью метода, если таким образом реализуется свойство класса (таким образом, код будет тот же, а его смысл — совсем другим).

Не нужно расширять интерфейс класса без необходимости, «на всякий случай», поскольку увеличение количества методов ведет к трудности понимания класса пользователем¹. В идеале интерфейс должен быть полным, то есть предоставлять возможность выполнить любые разумные действия с классом, и минимальным — без дублирования и пересечения возможностей методов.

В виде методов рекомендуется определять только действия, реализующие свойства класса. Если какое-либо действие можно реализовать, не обращаясь к скрытым полям класса, его нет необходимости описывать как метод; лучше описать его как обычную функцию, поместив ее в общее с классом пространство имен. Если функция выполняет действие, не являющееся свойством класса, но нуждается в доступе к его скрытым полям, ее следует объявить как дружественную. Но в общем случае *дружественных функций и классов надо избегать*, поскольку главной идеей ООП является минимизация связей между инкапсулированными классами.

Для увеличения производительности программы наиболее часто вызываемые методы можно объявить как встроенные (*inline*). В основном это касается коротких методов, тело которых оказывается меньше размера кода, генерируемого для их вызова. Кроме ускорения программы за счет исключения вызовов, это дает возможность компилятору производить более полную оптимизацию. Однако необходимо учитывать, что директива *inline* носит для компилятора рекомендательный характер, и он может ей не последовать — например, если метод содержит сложные циклы или объявлен как виртуальный. Кроме того, если программа или компилятор используют указатель на метод или функцию, дополнительно будет сгенерирована их невстраиваемая копия. Еще одним соображением, которое требуется принимать во внимание, является то, что многие отладчики имеют проблемы со встраиваемыми функциями.

Конструкторы и деструкторы делать встраиваемыми не рекомендуется, поскольку в них фактически присутствует код, помещаемый компилятором, и размер этого кода может быть весьма значительным (например, в конструкторе производного класса должны быть вызваны конструкторы всех базовых и вложенных классов).

¹ Под пользователем имеется в виду программист, использующий класс.

Перегруженные операции класса должны иметь интуитивно понятный общепринятый смысл (например, не следует заставлять операцию `+` выполнять что-либо, кроме сложения или добавления). В основном перегрузка операций используется для создаваемых программистом арифметических типов, а в остальных случаях для реализации каких-либо действий с объектами класса предпочтительнее использовать методы, поскольку им можно дать осмысленные имена.

Если какая-либо операция перегружена, следует, если возможно, перегрузить и аналогичные операции, например, `+`, `+=` и `++` (компилятор этого автоматически не сделает). При этом операции должны иметь ту же семантику, что и их стандартные аналоги.

Основой любого класса являются его *конструкторы и деструкторы*. Как известно, при отсутствии явно заданных конструкторов, конструктора копирования и операции присваивания компилятор создает их *автоматически*.

И конструктор копирования, и операция присваивания, создаваемые по умолчанию, выполняют поэлементное копирование из области-источника в область-приемник. Если объект содержит указатели, это приведет к тому, что после копирования два соответствующих указателя разных объектов будут ссылаться на одну и ту же область памяти. При уничтожении первого из объектов эта память будет освобождена, а повторная попытка освободить ее при уничтожении второго объекта приведет к неопределенному поведению программы. Поэтому для классов, содержащих поля-указатели, следует всегда явно определять конструктор копирования и операцию присваивания, выполняющие выделение памяти под динамические поля объекта.

Кроме того, *динамическая память, выделенная в конструкторе объекта, должна освобождаться в его деструкторе*. Невыполнение этого требования приводит к утечкам памяти. Удаление нулевого указателя безопасно (при этом ничего не происходит), поэтому если конструкторы, конструкторы копирования и операция присваивания написаны правильно, любой указатель либо ссылается на выделенную область памяти, либо равен нулю, и к нему можно применять `delete` без проверки.

Разница между конструктором копирования и операцией присваивания заключается в том, что последняя работает в том случае, когда объект-приемник уже существует, поэтому в ней перед выделением динамической памяти следует освободить занятую ранее. Из этого следует, что при реализации операции присваивания для классов, содержащих поля-указатели, необходимо проводить проверку на самоприсваивание и в этом случае оставить объект без изменений. Необходимо также помнить о том, что операция присваивания должна возвращать ссылку на константу. Таким образом, определение операции присваивания должно иметь вид:

```
class X{
    const X & operator=(const X & r);
    -
};
const X & X::operator=(const X & r){
    if(this != &r){
```

```

        // Копирование
    }
    return *this;
}

```

В конструкторах для задания начальных значений полям рекомендуется использовать инициализацию, а не присваивание:

```

class X{
    string s;
public:
    X(const char * str);
};
// Вариант конструктора с присваиванием:
X::X(const char * str){ s = str; }
// Вариант конструктора с инициализацией:
X::X(const char * str): s(str){}

```

Инициализация более универсальна, так как может применяться в тех случаях, когда присваиванием пользоваться нельзя (например, при задании значений константным полям или ссылкам). Кроме того, она выполняется более эффективно, потому что создание объекта в C++ начинается с инициализации его полей конструктором по умолчанию, после чего выполняется вызываемый конструктор. Таким образом, для первого варианта рассмотренного примера сначала будет вызван конструктор класса `string` по умолчанию, и только потом будет выполнено присваивание, в то время как второй вариант позволяет сразу же вызвать нужный конструктор. Для сложных классов, составляющих иерархию, разница между приведенными вариантами гораздо значительнее.

Необходимо учитывать и тот факт, что поля инициализируются в порядке их объявления, а не в порядке появления в списке инициализации. Поэтому для уменьшения числа возможных ошибок порядок указания полей в списке инициализации конструктора должен соответствовать порядку их объявления в классе.

Статические поля не должны инициализироваться в конструкторе, поскольку им нужно присваивать начальное значение только один раз для каждого класса, а конструктор выполняется для каждого объекта класса. Статические поля инициализируются в глобальной области определения (вне любой функции).

Конструкторы копирования также должны использовать списки инициализации полей, поскольку иначе для базовых классов и вложенных объектов будут вызваны конструкторы по умолчанию:

```

class X{
public:
    X(void);
    // Конструктор копирования:
    X(const X & r);
};
class Y: public X{
    string s;
public:
    // Вложенный объект

```

```
// Конструктор копирования:
Y(const Y & r): X(r), s(r.s){}
};
```

Операция присваивания не наследуется, поэтому она должна быть определена в производных классах. При этом из нее следует явным образом вызывать соответствующую операцию базового класса (см. с. 204).

Наследование классов предоставляет программисту богатейшие возможности — виртуальные и неvirtуальные базовые классы, открытое, защищенное и закрытое наследование, виртуальные методы и т. д. Выбор наиболее подходящих возможностей для целей конкретного проекта основывается на знании механизма их работы и взаимодействия.

Открытое наследование класса Y из класса X означает, что Y представляет собой разновидность класса X, то есть более конкретную, частную концепцию. Базовый класс X является более общим понятием, чем Y¹. Везде, где можно использовать X, можно использовать и Y, но не наоборот (вспомните, что на место ссылок на базовый класс можно передавать ссылку на любой из производных). Необходимо помнить, что во время выполнения программы не существует иерархии классов и передачи сообщений объектам базового класса из производных — есть только конкретные объекты классов, поля которых формируются на основе иерархии на этапе компиляции.

Методы, которые должны иметь все производные классы, но которые не могут быть реализованы на уровне базового класса, должны быть виртуальными. Например, все объекты иерархии должны уметь выводить информацию о себе. Поскольку она хранится в различных полях производных классов, эту функцию нельзя реализовать в базовом классе. Естественно назвать ее во всех классах одинаково и объявить как виртуальную с тем, чтобы другие методы базового класса могли вызывать ее в зависимости от фактического типа объекта, с которым они работают. По этой причине деструкторы объявляются как виртуальные.

При переопределении виртуальных методов нельзя изменять наследуемое значение аргумента по умолчанию, поскольку по правилам C++ оно определяется типом указателя, а не фактическим типом объекта, вызвавшего метод:

```
#include <iostream.h>
class X{
public:
    virtual void fun(int a = 0){cout << a;}
};
class Y: public X{
public:
    virtual void fun(int a = 1) {cout << a;}
};
int main(){
    X *px = new X;
    px->fun(); // Выводится 0
```

¹ Например, каждый программист — человек, но не каждый человек — программист.

```

X *py = new Y;
py->fun(); // Выводится 0
}

```

Невиртуальные методы переопределять в производных классах не рекомендуется, поскольку производные классы должны наследовать свойства базовых. Иными словами, неvirtуальный метод инвариантен относительно специализации, то есть сохраняет те свойства, которые должны наследоваться из базового класса независимо от того, как конкретизируется (специализируется) производный класс. Специализация производного класса достигается добавлением новых методов и переопределением существующих виртуальных методов.

Главное преимущество наследования состоит в том, что можно на уровне базового класса написать универсальный код, с помощью которого можно работать и с объектами производного класса, что реализуется с помощью виртуальных методов.

Альтернативным наследованию методом использования одним классом другого является *вложение*, когда один класс является полем другого:

```

class X{ j };
class Y{
    X x;
};

```

Вложение представляет отношения классов «Y содержит X» или «Y реализуется посредством X». Необходимость использовать вложение вместо наследования можно определить, задав себе вопрос, может ли у Y быть несколько объектов класса X. Если требуется, к примеру, описать класс для моделирования самолета, будет логично описать в нем поле типа «двигатель»: самолет содержит двигатель, но не является его разновидностью.

Отношение «реализуется посредством» используется вместо наследования тогда, когда про классы X и Y нельзя сказать, что Y является разновидностью X, но при этом Y использует часть функциональности X. Следует предпочитать вложение наследованию.

В случаях, когда между классами нет логической взаимосвязи, а требуется просто использовать часть кода одного класса в другом, может быть полезным использовать закрытое наследование (с ключом доступа `private`). Этот способ используется, когда в производном классе требуется доступ к защищенным элементам базового класса и замещение его виртуальных методов.

Шаблоны классов используются для создания семейств классов, поведение которых не зависит от типа объектов. Шаблоны следует использовать аккуратно, отдавая себе отчет в том, что для каждого типа порождается собственная копия шаблона, что может привести к разбуханию кода. Для уменьшения размера кода рекомендуется вынести в базовый класс все методы, не использующие информацию о типе объекта, и унаследовать от него шаблонный класс со всеми остальными методами. Это позволит избежать дублирования тех методов, тексты которых не зависят от параметра шаблона.

Исключения используются, как правило, в тех случаях, когда иного способа сообщить об ошибке не существует (например, если она произошла в конструкторе или перегруженной операции), а также когда ошибка неисправимая или очень редкая и неожиданная. Обработка исключений несколько уменьшает производительность программы и, с моей точки зрения, ухудшает ее читаемость, поэтому без необходимости пользоваться исключениями не нужно.

Для более подробного изучения рассмотренных в этом разделе вопросов рекомендуется обратиться к книгам А. Голуба [9] и С. Мейерса [13].

Упражнения к части II

Классы

Вариант 1

Описать класс, реализующий стек. Написать программу, использующую этот класс для моделирования Т-образного сортировочного узла на железной дороге. Программа должна разделять на два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность формирования состава из файла и с клавиатуры.

Вариант 2

Описать класс, реализующий бинарное дерево, обладающее возможностью добавления новых элементов, удаления существующих, поиска элемента по ключу, а также последовательного доступа ко всем элементам.

Написать программу, использующую этот класс для представления англо-русского словаря. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса. Предусмотреть возможность формирования словаря из файла и с клавиатуры.

Вариант 3

Построить систему классов для описания плоских геометрических фигур: круг, квадрат, прямоугольник. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и вращения на заданный угол.

Написать программу, демонстрирующую работу с этими классами. Программа должна содержать меню, позволяющее осуществить проверку всех методов классов.

Вариант 4

Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса, создания и уничтожения объектов этого класса.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 5

Составить описание класса для представления комплексных чисел. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 6

Составить описание класса для объектов-векторов, задаваемых координатами концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 7

Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменение размеров, построение наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 8

Составить описание класса для определения одномерных массивов целых чисел (векторов). Предусмотреть возможность обращения к отдельному элементу массива с контролем выхода за пределы массива, возможность задания произвольных границ индексов при создании объекта и выполнения операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов, умножения и деления всех элементов массива на скаляр, вывода на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 9

Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массива, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывод на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 10

Составить описание класса многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Предусмотреть методы для вычисления значения многочлена для заданного аргумента, операции сложения,

вычитания и умножения многочленов с получением нового объекта-многочлена, вывод на экран описания многочлена.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 11

Составить описание класса одномерных массивов строк, каждая строка задается длиной и указателем на выделенную для нее память. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массивов, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывод на экран элемента массива и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 12

Составить описание класса, обеспечивающего представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывода на экран подматрицы любого размера и всей матрицы.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 13

Написать класс для эффективной работы со строками, позволяющий форматировать и сравнивать строки, хранить в строках числовые значения и извлекать их. Для этого необходимо реализовать:

- ☐ перегруженные операторы присваивания и конкатенации;
- ☐ операции сравнения и приведения типов;
- ☐ преобразование в число любого типа;
- ☐ форматный вывод строки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 14

Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки книг по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 15

Описать класс «записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например,

по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 16

Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 17

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- ☐ сложение, вычитание, умножение, деление (+, -, *, /) (умножение и деление как на другую матрицу, так и на число);
- ☐ комбинированные операции присваивания (+-, --, *-, /-);
- ☐ операции сравнения на равенство/неравенство;
- ☐ операции вычисления обратной и транспонированной матрицы, операцию возведения в степень;
- ☐ методы вычисления детерминанта и нормы;
- ☐ методы, реализующие проверку типа матрицы (квадратная, диагональная, нулевая, единичная, симметрическая, верхняя треугольная, нижняя треугольная);
- ☐ операции ввода/вывода в стандартные потоки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 18

Описать класс «множество», позволяющий выполнять основные операции — добавление и удаление элемента, пересечение, объединение и разность множеств.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 19

Описать класс, реализующий стек. Написать программу, использующую этот класс для отыскания прохода по лабиринту.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице. После отыскания прохода программа печатает найденный путь в виде координат квадратов.

по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 16

Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 17

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- ☐ сложение, вычитание, умножение, деление (+, -, *, /) (умножение и деление как на другую матрицу, так и на число);
- ☐ комбинированные операции присваивания (+=, -=, *=, /=);
- ☐ операции сравнения на равенство/неравенство;
- ☐ операции вычисления обратной и транспонированной матрицы, операцию возведения в степень;
- ☐ методы вычисления детерминанта и нормы;
- ☐ методы, реализующие проверку типа матрицы (квадратная, диагональная, нулевая, единичная, симметрическая, верхняя треугольная, нижняя треугольная);
- ☐ операции ввода/вывода в стандартные потоки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 18

Описать класс «множество», позволяющий выполнять основные операции — добавление и удаление элемента, пересечение, объединение и разность множеств.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 19

Описать класс, реализующий стек. Написать программу, использующую этот класс для отыскания прохода по лабиринту.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице. После отыскания прохода программа печатает найденный путь в виде координат квадратов.

Вариант 20

Описать класс «предметный указатель». Каждая компонента указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Предусмотреть возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Наследование

Вариант 1

Создать класс CFile, инкапсулирующий в себе такие функции работы с файлами, как Open, Close, Seek, Read, Write, GetPosition и GetLength. На базе этого класса создать производный класс CMyDataFile — файл, содержащий в себе данные не-которого определенного типа MyData, а также заголовок, облегчающий доступ к этому файлу.

Написать программу, демонстрирующую работу с этими классами. Программа должна содержать меню, позволяющее осуществить проверку всех методов классов.

Вариант 2

Создать класс CPoint — точка. На его основе создать классы CcoloredPoint и CLine. На основе класса CLine создать класс CColoredLine и класс CPolyLine — многоугольник. Все классы должны иметь методы для установки и получения значений всех координат, а также изменения цвета и получения текущего цвета.

Написать демонстрационную программу, в которой будет использоваться список объектов этих классов в динамической памяти.

Вариант 3

Создать абстрактный класс CVehicle. На его основе реализовать классы CPlane, CCar и CShip. Классы должны иметь возможность задавать и получать координаты, параметры средств передвижения (цена, скорость, год выпуска). Для самолета должна быть определена высота, для самолета и корабля — количество пассажиров. Для корабля — порт приписки.

Написать программу, создающую список объектов этих классов в динамической памяти. Программа должна содержать меню, позволяющее осуществить проверку всех методов классов.

Вариант 4

1. Описать базовый класс «Элемент».

Поля:

- имя элемента (указатель на строку символов);
- количество входов элемента;
- количество выходов элемента.

Методы:

- конструктор класса;
- деструктор класса;
- метод, задающий имя элемента.

2. На основе класса «Элемент» описать производный класс «Комбинационный», представляющий собой комбинационный элемент (двоичный вентиль), который может иметь несколько входов и один выход.

Поля:

- указатель, используемый для динамического размещения полей, содержащих значения входов.

Методы:

- конструктор;
- конструктор копирования;
- деструктор;
- метод, задающий значение на входах экземпляра класса;
- метод, позволяющий опрашивать состояние отдельного входа экземпляра класса;
- метод, вычисляющий значение выхода (по варианту задания).

3. На основе класса «Элемент» описать производный класс «Память», представляющих собой триггер. Триггер имеет входы, соответствующие типу триггера (см. ниже вариант задания), и входы установки и сброса. Все триггеры считаются синхронными, сам синхровход в состав триггера не включается.

Поля:

- массив значений входов объекта класса (задается статически), в массиве учитываются все входы (управляющие и информационные);
- состояние на прямом выходе триггера;
- состояние на инверсном выходе триггера.

Методы:

- конструктор (по умолчанию сбрасывает экземпляр класса);
- конструктор копирования;
- деструктор;
- метод, задающий значение на входах экземпляра класса;
- методы, позволяющие опрашивать состояния отдельного входа экземпляра класса;
- метод, вычисляющий состояние экземпляра класса (по варианту задания) в зависимости от текущего состояния и значений на входах;
- метод, переопределяющий операцию «—» для экземпляров класса.

4. Создать класс «Регистр», используя класс «Память» как включаемый класс.

Поля:

- состояние входа «Сброс» — один для экземпляра класса;

- состояние входа «Установка» — один для экземпляра класса;
- статический массив типа «Память» заданной в варианте размерности;
- статический(е) массив(ы), содержащие значения на соответствующих входах элементов массива типа «Память».

Методы:

- метод, задающий значение на входах экземпляра класса (желательно в качестве параметров передавать методу указатели на массивы значений);
- метод, позволяющий опрашивать состояние отдельного выхода экземпляра класса;
- метод, вычисляющий значение нового состояния экземпляра класса.

Все поля классов «Элемент», «Комбинационный» и «Память» должны быть описаны с ключевым словом `private`.

В задании перечислены только обязательные члены и методы класса. Можно задавать дополнительные члены и методы, если они не отменяют обязательные и обеспечивают дополнительные удобства при работе с данными классами, например, описать функции вычисления выхода/состояния как виртуальные.

5. Для проверки функционирования созданных классов написать программу, использующую эти классы. В программе должны быть продемонстрированы все свойства созданных классов.

Конкретный тип комбинационного элемента, тип триггера и разрядность регистра выбираются в соответствии с вариантом задания.

Вариант	Комбинационный элемент	Число входов	Триггер	Разрядность регистра
1	И-НЕ	4	RS	8
2	ИЛИ	5	RST	10
3	МОД2-НЕ	6	D	12
4	И	8	T	8
5	ИЛИ-НЕ	8	V	9
6	И	4	RS	10
7	ИЛИ-НЕ	5	JK	11
8	МОД2	5	D	8
9	И	4	T	10
10	ИЛИ	3	JK	8
11	И-НЕ	3	RS	12
12	ИЛИ-НЕ	4	RST	4
13	МОД2	5	D	10
14	МОД2-НЕ	6	T	10
15	ИЛИ-НЕ	8	V	10

Вариант	Комбинационный элемент	Число входов	Триггер	Разрядность регистра
16	И	8	JK	6
17	И-НЕ	8	RS	10
18	ИЛИ	8	T	10
19	МОД2	6	JK	8
20	МОД2-НЕ	5	V	10

Вариант 5

Описать базовый класс СТРОКА.

Обязательные поля класса:

- ☐ указатель на char — хранит адрес динамически выделенной памяти для размещения символов строки;
- ☐ значение типа int — хранит длину строки в байтах.

Обязательные методы должны выполнять следующие действия:

- ☐ конструктор без параметров;
- ☐ конструктор, принимающий в качестве параметра Си-строку (заканчивается нулевым байтом);
- ☐ конструктор, принимающий в качестве параметра символ;
- ☐ конструктор копирования;
- ☐ получение длины строки;
- ☐ очистка строки (сделать строку пустой);
- ☐ деструктор.

Описать производный от СТРОКА класс СТРОКА_ИДЕНТИФИКАТОР.

Строки данного класса строятся по правилам записи идентификаторов в языке С и могут включать в себя только те символы, которые могут входить в состав С-идентификаторов. Если исходные данные противоречат правилам записи идентификатора, то создается пустая СТРОКА_ИДЕНТИФИКАТОР.

Обязательные методы:

- ☐ конструктор без параметров;
- ☐ конструктор, принимающий в качестве параметра Си-строку (заканчивается нулевым байтом);
- ☐ конструктор, принимающий в качестве параметра символ;
- ☐ конструктор копирования;
- ☐ перевод всех символов строки в верхний регистр;
- ☐ перевод всех символов строки в нижний регистр;
- ☐ поиск первого вхождения символа в строку;
- ☐ деструктор.

Переопределить следующие операции:

- ☐ присваивание (=);
- ☐ сложение (+) — операция конкатенации строк;
- ☐ вычитание (-) — из строки (первый операнд) удаляются все символы, входящие в строку, — второй операнд, при этом может получиться пустая строка;
- ☐ оператор > — проверка на больше. Строка считается больше другой, если код символа первой строки в i -й позиции (i изменяется от 0 до $n-1$, где n — длина более короткой строки) больше кода символа в той же позиции во второй строке, длины строк могут не совпадать.
- ☐ оператор < — проверка на меньше. Строка считается меньше другой, если код символа первой строки в i -й позиции (i изменяется от 0 до $n-1$, где n — длина более короткой строки) меньше кода символа в той же позиции кода символа в той же позиции во второй строке, длины строк могут не совпадать.

Разработчик вправе вводить любое (с обоснованием необходимости) число дополнительных полей и методов.

Написать тестовую программу, которая:

- ☐ динамически выделяет массив указателей на базовый класс (4–6);
- ☐ в режиме диалога заполняет этот массив указателями на производные классы, при этом экземпляры производных классов создаются динамически с заданием начальных значений;
- ☐ для созданных экземпляров производных классов выполняет проверку всех разработанных методов с выводом исходных данных и результатов на дисплей.

Для конструкторов копирования каждого класса предусмотреть диагностическую печать количества его вызовов в определенное место дисплея (рекомендуется использовать статические члены класса).

Режим диалога обеспечивается с помощью иерархического меню.

Вариант 6

Описать базовый класс СТРОКА.

Обязательные поля класса:

- ☐ указатель на char — хранит адрес динамически выделенной памяти для размещения символов строки;
- ☐ значение типа int — хранит длину строки в байтах.

Обязательные методы должны выполнять следующие действия:

- ☐ конструктор без параметров;
- ☐ конструктор, принимающий в качестве параметра Си-строку (заканчивается нулевым байтом);
- ☐ конструктор, принимающий в качестве параметра символ;
- ☐ конструктор копирования;
- ☐ получение длины строки;

- ☐ очистка строки (сделать строку пустой);
- ☐ деструктор.

Описать производный от СТРОКА класс БИТОВАЯ_СТРОКА.

Строки данного класса могут содержать только символы '0' или '1'. Если в составе инициализирующей строки будут встречены любые символы, отличные от допустимых, БИТОВАЯ_СТРОКА принимает нулевое значение. Содержимое данных строк рассматривается как двоичное число.

Отрицательные числа хранятся в дополнительном коде.

Обязательные методы:

- ☐ конструктор без параметров;
- ☐ конструктор, принимающий в качестве параметра Си-строку (заканчивается нулевым байтом);
- ☐ конструктор копирования;
- ☐ деструктор;
- ☐ изменение знака на противоположный (перевод числа в дополнительный код).

Переопределить следующие операции (длина строки результата равна длине большей из строк; в случае необходимости более короткая битовая строка расширяется влево знаковым разрядом):

- ☐ присваивание (-);
- ☐ сложение (+) — арифметическая сумма строк;
- ☐ операция (==) — проверка на равенство.

Разработчик вправе вводить любое (с обоснованием необходимости) число дополнительных полей и методов.

Написать тестовую программу, которая:

- ☐ динамически выделяет массив указателей на базовый класс (4–6);
- ☐ в режиме диалога заполняет этот массив указателями на производные классы, при этом экземпляры производных классов создаются динамически с заданием начальных значений;
- ☐ для созданных экземпляров производных классов выполняет проверку всех разработанных методов с выводом исходных данных и результатов на дисплей.

Для конструкторов копирования каждого класса предусмотреть диагностическую печать количества его вызовов в определенное место дисплея (рекомендуется использовать статические члены класса).

Режим диалога обеспечивается с помощью иерархического меню.

Вариант 7

Описать базовый класс СТРОКА.

Обязательные поля класса:

- ☐ указатель на char — хранит адрес динамически выделенной памяти для размещения символов строки;

☐ значение типа `int` — хранит длину строки в байтах.

Обязательные методы должны выполнять следующие действия:

- ☐ конструктор без параметров;
- ☐ конструктор, принимающий в качестве параметра Си-строку (заканчивается нулевым байтом);
- ☐ конструктор, принимающий в качестве параметра символ;
- ☐ конструктор копирования;
- ☐ получение длины строки;
- ☐ очистка строки (сделать строку пустой);
- ☐ деструктор.

Описать производный от `СТРОКА` класс `ДЕСЯТИЧНАЯ_СТРОКА`.

Строки данного класса могут содержать только символы десятичных цифр и символы `-` и `+`, задающие знак числа. Символы `-` или `+` могут находиться только в первой позиции числа, причем символ `+` может отсутствовать, в этом случае число считается положительным. Если в составе инициализирующей строки будут встречены любые символы, отличные от допустимых, `ДЕСЯТИЧНАЯ_СТРОКА` принимает нулевое значение. Содержимое данных строк рассматривается как десятичное число.

Обязательные методы:

- ☐ конструктор без параметров;
- ☐ конструктор, принимающий в качестве параметра Си-строку (заканчивается нулевым байтом);
- ☐ конструктор копирования;
- ☐ деструктор;
- ☐ метод, определяющий, можно ли представить данное число в формате `int`;

Переопределить следующие операции:

- ☐ присваивание (`=`);
- ☐ вычитание (`-`) — арифметическая разность строк;
- ☐ операция `>` — проверка на больше (по значению);
- ☐ операция `<` — проверка на меньше (по значению).

Разработчик вправе вводить любое (с обоснованием необходимости) число дополнительных полей и методов.

Написать тестовую программу, которая:

- ☐ динамически выделяет массив указателей на базовый класс (4–6);
- ☐ в режиме диалога заполняет этот массив указателями на производные классы, при этом экземпляры производных классов создаются динамически с заданием начальных значений;
- ☐ для созданных экземпляров производных классов выполняет проверку всех разработанных методов с выводом исходных данных и результатов на дисплей.

Для конструкторов копирования каждого класса предусмотреть диагностическую печать количества его вызовов в определенное место дисплея (рекомендуется использовать статические члены класса).

Режим диалога обеспечивается с помощью иерархического меню.

Вариант 8

Описать базовый класс СТРОКА.

Обязательные поля класса:

- ☐ указатель на `char` — хранит адрес динамически выделенной памяти для размещения символов строки;
- ☐ значение типа `int` — хранит длину строки в байтах.

Обязательные методы должны выполнять следующие действия:

- ☐ конструктор без параметров;
- ☐ конструктор, принимающий в качестве параметра Си-строку (заканчивается нулевым байтом);
- ☐ конструктор, принимающий в качестве параметра символ;
- ☐ конструктор копирования;
- ☐ получение длины строки;
- ☐ очистка строки (сделать строку пустой);
- ☐ деструктор.

Производный от СТРОКА класс КОМПЛЕКСНОЕ_ЧИСЛО.

Строки данного класса состоят из двух полей, разделенных символом `i`.

Первое поле задает значение реальной части числа, а второе — мнимой. Каждое из полей может содержать только символы десятичных цифр и символы `-` и `+`, задающие знак числа. Символы `-` или `+` могут находиться только в первой позиции числа, причем символ `+` может отсутствовать, в этом случае число считается положительным. Если в составе инициализирующей строки будут встречены любые символы, отличные от допустимых, КОМПЛЕКСНОЕ_ЧИСЛО принимает нулевое значение. Примеры строк: `33i12`, `-7i100`, `+5i-21`.

Обязательные методы:

- ☐ конструктор без параметров;
- ☐ конструктор, принимающий в качестве параметра Си-строку (заканчивается нулевым байтом);
- ☐ конструктор копирования;
- ☐ деструктор.

Переопределить следующие операции:

- ☐ присваивание (`=`);
- ☐ операция (`--`) — проверка на равенство;
- ☐ умножение (`*`) — умножение чисел.

Разработчик вправе вводить любое (с обоснованием необходимости) число дополнительных полей и методов.

Написать тестовую программу, которая:

- ☐ динамически выделяет массив указателей на базовый класс (4–6);
- ☐ в режиме диалога заполняет этот массив указателями на производные классы, при этом экземпляры производных классов создаются динамически с заданием начальных значений;
- ☐ для созданных экземпляров производных классов выполняет проверку всех разработанных методов с выводом исходных данных и результатов на дисплей.

Для конструкторов копирования каждого класса предусмотреть диагностическую печать количества его вызовов в определенное место дисплея (рекомендуется использовать статические члены класса).

Режим диалога обеспечивается с помощью иерархического меню.

Шаблоны классов

Вариант 1

Создать шаблон класса «стек». Использовать его при решении задачи 1 из раздела «Классы».

Вариант 2

Создать шаблон класса «стек». Использовать его при решении задачи 7 из раздела «Динамические структуры данных» (см. упражнения к первой части).

Вариант 3

Создать шаблон класса «стек». Использовать его при решении задачи 8 из раздела «Динамические структуры данных» (см. упражнения к первой части).

Вариант 4

Создать шаблон класса «однонаправленный линейный список». Использовать его при решении задачи 1 из раздела «Динамические структуры данных» (см. упражнения к первой части).

Вариант 5

Создать шаблон класса «однонаправленный линейный список». Использовать его при решении задачи 3 из раздела «Динамические структуры данных» (см. упражнения к первой части).

Вариант 6

Создать шаблон класса «бинарное дерево». Использовать его при решении задачи 2 из раздела «Динамические структуры данных» (см. упражнения к первой части).

Вариант 7

Создать шаблон класса «бинарное дерево». Использовать его при решении задачи 4 из раздела «Динамические структуры данных» (см. упражнения к первой части).

Вариант 17

Создать шаблон класса «однонаправленный линейный список указателей». Использовать его при решении задачи 11 из раздела «Динамические структуры данных» (см. упражнения к первой части).

Вариант 18

Создать шаблон класса «бинарное дерево». Использовать его при решении задачи 2 из раздела «Классы».

Вариант 19

Создать шаблон класса «бинарное дерево». Использовать его при решении задачи 18 из раздела «Динамические структуры данных» (см. упражнения к первой части).

Вариант 20

Создать шаблон класса «бинарное дерево», содержащее указатели на элементы данных. Использовать его при решении задачи 20 из раздела «Динамические структуры данных» (см. упражнения к первой части).

ЧАСТЬ III

Стандартная библиотека

В любой программе, кроме операторов языка, используются средства библиотек, включаемых в среду программирования. Различные среды предоставляют в распоряжение программиста разные наборы средств, облегчающих создание программ, — например, компиляторы Microsoft Visual C++ и Borland C++ содержат библиотеки классов для написания приложений Windows. Часть библиотек стандартизована, то есть должна поставляться с любым компилятором языка C++.

Стандартную библиотеку C++ можно условно разделить на две части. К первой относятся функции, макросы, типы и константы, унаследованные из библиотеки C, а ко второй — классы и другие средства C++. Первая часть библиотеки рассматривалась в разделе «Функции стандартной библиотеки» (с. 88). Список констант, макросов и типов приведен в приложении 5, а список функций — в приложении 6. Вторая часть содержит классы, шаблоны и другие средства для ввода, вывода, хранения и обработки данных как стандартных типов, так и типов, определенных пользователем.

Классы стандартной библиотеки можно разделить на группы в соответствии с их назначением.

- *Потоковые классы* предназначены для управления потоками данных между оперативной памятью и внешними устройствами (например, дисками и консолью), а также в пределах оперативной памяти.
- *Строковый класс* предназначен для удобной и защищенной от ошибок работы с символьными строками.
- *Контейнерные классы* реализуют наиболее распространенные структуры для хранения данных — например, списки, вектора и множества. В библиотеку входят также *алгоритмы*, использующие эти контейнеры.
- *Итераторы* предназначены для унифицированного доступа к элементам контейнерных и других классов.
- *Математические классы* поддерживают эффективную обработку массивов с плавающей точкой и работу с комплексными числами.

- *Диагностические классы* обеспечивают динамическую идентификацию типов и объектно-ориентированную обработку ошибок.
- *Остальные классы* обеспечивают динамическое распределение памяти, адаптацию к локальным особенностям, обработку функциональных объектов и т. д.

Часть библиотеки, в которую входят контейнерные классы, алгоритмы и итераторы, называют *стандартной библиотекой шаблонов* (STL — Standard Template Library).

Для использования средств стандартной библиотеки в программу требуется включить с помощью директивы `#include` соответствующие заголовочные файлы. Например, потоки описаны в `<iostream>`, а списки — в `<list>`. Элементы заголовочных файлов без расширения `.h` определены в пространстве имен `std`, а одноименные файлы с расширением `.h` — в глобальном пространстве имен. Список заголовочных файлов стандартной библиотеки приведен в приложении 4.

Имена заголовочных файлов C++ для функций библиотеки C, определенные в пространстве имен `std`, начинаются с буквы `c`, например, `<cstdio>`, `<cstring>`, `<ctime>`. Для каждого заголовочного файла вида `<сХ>` существует файл `<Х.h>`, определяющий те же имена в глобальном пространстве имен (см. с. 100)¹.

¹ Наибольшая путаница возникает с именами заголовков строк: `string` — заголовок для работы со строковыми классами C++, `cstring` — версия заголовка функций C в пространстве имен `std`, `string.h` — то же самое в глобальном пространстве имен, `cstring.h` — старый заголовок для работы со строковыми классами C++ в глобальном пространстве имен в одной из оболочек.

ГЛАВА 10

Потоковые классы

Поток — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Потоки C++, в отличие от функций ввода/вывода в стиле C, обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис.

Чтение данных из потока называется *извлечением*, вывод в поток — *помещением*, или *включением*. Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специальную область оперативной памяти — *буфер*. Фактическая передача данных выполняется при выводе после заполнения буфера, а при вводе — если буфер исчерпан.

По направлению обмена потоки можно разделить на *входные* (данные вводятся в память), *выходные* (данные выводятся из памяти) и *двунаправленные* (допускающие как извлечение, так и включение).

По виду устройств, с которыми работает поток, можно разделить потоки на стандартные, файловые и строковые.

Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея, *файловые потоки* — для обмена информацией с файлами на внешних носителях данных (например, на магнитном диске), а *строковые потоки* — для работы с массивами символов в оперативной памяти.

Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов — `ios` и `streambuf`. Класс `ios` содержит общие для ввода и вывода поля и методы, класс `streambuf` обеспечивает буферизацию потоков и их взаимодействие с физическими устройствами. От этих классов наследуется класс `istream` для входных потоков и `ostream` — для выходных. Два последних класса являются базовыми для класса `iostream`, реализующего двунаправленные потоки. Ниже в иерархии классов располагаются файловые и строковые потоки. Далее перечислены часто используемые классы потоков.

<code>ios</code>	— базовый класс потоков;
<code>istream</code>	— класс входных потоков;
<code>ostream</code>	— класс выходных потоков;
<code>iostream</code>	— класс двунаправленных потоков;
<code>istringstream</code>	— класс входных строковых потоков;
<code>ostringstream</code>	— класс выходных строковых потоков;
<code>stringstream</code>	— класс двунаправленных строковых потоков;
<code>ifstream</code>	— класс входных файловых потоков;
<code>ofstream</code>	— класс выходных файловых потоков;
<code>fstream</code>	— класс двунаправленных файловых потоков.

Описания классов находятся в заголовочных файлах:

<code><ios></code>	— базовый класс потоков ввода/вывода;
<code><iosfwd></code>	— предварительные объявления средств ввода/вывода;
<code><istream></code>	— шаблон потока ввода;
<code><ostream></code>	— шаблон потока вывода;
<code><iostream></code>	— стандартные объекты и операции с потоками ввода/вывода;
<code><fstream></code>	— потоки ввода/вывода в файлы;
<code><sstream></code>	— потоки ввода/вывода в строки;
<code><streambuf></code>	— буферизация потоков ввода/вывода;
<code><iomanip></code>	— манипуляторы (о них рассказывается далее, см. с. 271).

Подключение к программе файлов `<fstream>` и `<sstream>` автоматически подключает и файл `<iostream>`, так как он является для них базовым.

Основным преимуществом потоков по сравнению с функциями ввода/вывода, унаследованными из библиотеки C, является контроль типов, а также расширяемость, то есть возможность работать с типами, определенными пользователем. Для этого требуется переопределить операции потоков (об этом рассказывается в разделе «Потоки и типы, определенными пользователем», с. 284).

Кроме того, потоки могут работать с расширенным набором символов `wchar_t`. Для этого используются классы `wistream`, `wostream`, `wofstream` и т. д.¹. К недостаткам потоков можно отнести снижение быстродействия программы, которое в зависимости от реализации компилятора может быть весьма значительным.

¹ На самом деле потоки реализованы в библиотеке с помощью шаблонов, параметром которых является тип символа. Имена классов для параметров `char` и `wchar_t` определены с помощью `typedef`.

Стандартные потоки

Заголовочный файл `<iostream>` содержит, кроме описания классов для ввода/вывода, четыре предопределенных объекта.

Объект	Класс	Описание
<code>cin</code>	<code>istream</code>	Связывается с клавиатурой (стандартным буферизованным вводом)
<code>cout</code>	<code>ostream</code>	Связывается с экраном (стандартным буферизованным выводом)
<code>cerr</code>	<code>ostream</code>	Связывается с экраном (стандартным небуферизованным выводом), куда направляются сообщения об ошибках
<code>clog</code>	<code>ostream</code>	Связывается с экраном (стандартным буферизованным выводом), куда направляются сообщения об ошибках

Эти объекты создаются при включении с программу заголовочного файла `<iostream>`, при этом становятся доступными связанные с ними средства ввода/вывода. Имена этих объектов можно переназначить на другие файлы или символьные буферы.

В классах `istream` и `ostream` операции извлечения из потока `>>` и помещения в поток `<<` определены путем перегрузки операций сдвига. Пример:

```
#include <iostream.h>
int main(){
    int i;
    cin >> i;
    cout << "Вы ввели " << i;
    return 0;
}
```

Операции извлечения и чтения в качестве результата своего выполнения формируют ссылку на объект типа `istream` для извлечения и ссылку на `ostream` — для чтения. Это позволяет формировать цепочки операций, что проиллюстрировано последним оператором приведенного примера. Вывод при этом выполняется слева направо.

Как и для других перегруженных операций, для вставки и извлечения невозможно изменить приоритеты, поэтому в необходимых случаях используются скобки:

```
// Скобки не требуются — приоритет сложения больше, чем << :
cout << i + j;
// Скобки необходимы — приоритет операции отношения меньше, чем << :
cout << (i < j);
cout << (i << j);    // Правая операция << означает сдвиг
```

Величины при вводе должны разделяться пробельными символами (пробелами, знаками табуляции или перевода строки). Извлечение прекращается, если очередной символ оказался недопустимым.

Если в операции помещения в поток встречается выражение, изменяющее некоторую переменную, то она не должна присутствовать в цепочке операций более одного раза, поскольку в таком случае результат может зависеть от реализации компилятора.

Операции << и >> перегружены для всех встроенных типов данных, что позволяет автоматически выполнять ввод и вывод в соответствии с типом величин. Это означает, что при вводе последовательность символов преобразуется во внутреннее представление величины, стоящей справа от знака извлечения, а при выводе выполняется обратное преобразование, например:

```
#include <iostream.h>
int main(){
    int i = 0xD;
    double d;
    // Символы из потока ввода преобразуются в double:
    cin >> d;
    // int и double преобразуются в строку символов:
    cout << i << ' ' << d;
    return 0;
}
```

Рассмотрим, как обрабатываются с помощью этих операций данные различных типов.

Числовые значения можно вводить в десятичной или шестнадцатеричной системе счисления (с префиксом 0x) со знаком или без знака. Вещественные числа представляются в форме с фиксированной точкой или с порядком. Например, если для предыдущего примера с клавиатуры вводится последовательность символов 1.53e-2, она интерпретируется как вещественное число с порядком и преобразуется во внутреннее представление, соответствующее типу double. При выводе выполняется обратное преобразование, и на экран выводятся символы:

```
13 0.0153
```

Поскольку ввод буферизован, помещение в буфер ввода происходит после нажатия клавиши перевода строки, после чего из буфера выполняется операция извлечения из потока. Это дает возможность исправлять введенные символы до того, как нажата клавиша Enter.

При *вводе строк* извлечение происходит до ближайшего пробела (вместо него в строку заносится нуль-символ):

```
char str1[100], str2[100];
cin >> str1 >> str2;
```

Если с клавиатуры вводится строка "раз два три четыре пять", переменные str1 и str2 примут значения "раз" и "два" соответственно, а остаток строки воспринят не будет. При необходимости ввести из входного потока строку целиком (до символа '\n') пользуются методами get или getline (см. с. 273).

Значения указателей выводятся в шестнадцатеричной системе счисления. Под любую величину при выводе отводится столько позиций, сколько требуется для ее представления. Чтобы отделить одну величину от другой, используются пробелы:

```
cout << i << ' ' << d << "    " << j;
```

Если формат вывода он может скорректировать форматирования и та

Форматир

В потоковых классах цью флагов, маннну.

Флаги и форми

Флаги представляют long класса ios. Флаг

Таблица 10.1. Флаги

Флаг	Положен
skipws	0x0001
left	0x0002
right	0x0004
internal	0x0008
dec	0x0010
oct	0x0020
hex	0x0040
showbase	0x0080
showpoint	0x0100
uppercase	0x0200
showpos	0x0400
scientific	0x0800
fixed	0x1000
unitbuf	0x2000
stdio	0x4000

Если формат вывода, используемый по умолчанию, не устраивает программиста, он может скорректировать его с помощью методов классов ввода/вывода; флагов форматирования и так называемых манипуляторов. Об этом рассказывается далее.

Форматирование данных

В потоковых классах форматирование выполняется тремя способами — с помощью флагов, манипуляторов и форматирующих методов.

Флаги и форматирующие методы

Флаги представляют собой отдельные биты, объединенные в поле `x_flags` типа `long` класса `ios`. Флаги перечислены в табл. 10.1.

Таблица 10.1. Флаги форматирования

Флаг	Положение	Умолчание	Описание действия при установленном бите
<code>skipws</code>	<code>0x0001</code>	+	При извлечении пробельные символы игнорируются
<code>left</code>	<code>0x0002</code>		Выравнивание по левому краю поля
<code>right</code>	<code>0x0004</code>	+	Выравнивание по правому краю поля
<code>internal</code>	<code>0x0008</code>		Знак числа выводится по левому краю, число — по правому. Промежуток заполняется символами <code>x_fill</code> (см. ниже), по умолчанию пробелами
<code>dec</code>	<code>0x0010</code>	+	Десятичная система счисления
<code>oct</code>	<code>0x0020</code>		Восьмеричная система счисления
<code>hex</code>	<code>0x0040</code>		Шестнадцатеричная система счисления
<code>showbase</code>	<code>0x0080</code>		Выводится основание системы счисления (<code>0x</code> для шестнадцатеричных чисел и <code>0</code> для восьмеричных)
<code>showpoint</code>	<code>0x0100</code>		При выводе вещественных чисел печатать десятичную точку и дробную часть
<code>uppercase</code>	<code>0x0200</code>		При выводе использовать символы верхнего регистра
<code>showpos</code>	<code>0x0400</code>		Печатать знак при выводе положительных чисел
<code>scientific</code>	<code>0x0800</code>		Печатать вещественные числа в форме мантиисы с порядком
<code>fixed</code>	<code>0x1000</code>		Печатать вещественные числа в форме с фиксированной точкой (точность определяется полем <code>x_precision</code> , см. ниже)
<code>unitbuf</code>	<code>0x2000</code>		Выгружать буферы всех потоков после каждого вывода
<code>stdio</code>	<code>0x4000</code>		Выгружать буферы потоков <code>stdout</code> и <code>stderr</code> после каждого вывода

ПРИМЕЧАНИЕ

Флаги (`left`, `right` и `internal`), (`dec`, `oct` и `hex`), а также (`scientific` и `fixed`) взаимно исключают друг друга, то есть в каждый момент может быть установлен только один флаг из каждой группы.

Для управления флагами в классе `ios` есть *методы* `flags`, `setf` и `unsetf`:

- `long ios::flags(long):` — возвращает текущие флаги потока;
- `long ios::flags(long):` — присваивает флагам значение параметра;
- `long ios::setf(long, long):` — присваивает флагам, биты которых установлены в первом параметре, значение соответствующих битов второго параметра;
- `long ios::setf(long):` — устанавливает флаги, биты которых установлены в параметре;
- `long ios::unsetf(long):` — сбрасывает флаги, биты которых установлены в параметре.

Все функции возвращают прежние флаги потока.

Кроме флагов, для форматирования используются следующие *поля* класса `ios`:

- `int x_width` — минимальная ширина поля вывода;
- `int x_precision` — количество цифр в дробной части при выводе вещественных чисел;
- `int x_fill` — символ заполнения поля вывода.

Для управления этими полями используются *методы* `width`, `precision` и `fill`:

- `int ios::width()` — возвращает значение ширины поля вывода;
- `int ios::width(int)` — устанавливает ширину поля вывода в соответствии со значением параметра;
- `int ios::precision()` — возвращает значение точности представления при выводе вещественных чисел;
- `int ios::precision(int)` — устанавливает значение точности представления при выводе вещественных чисел, возвращает старое значение точности;
- `char fill()` — возвращает текущий символ заполнения;
- `char fill(char)` — устанавливает значение текущего символа заполнения, возвращает старое значение символа.

Перед установкой некоторых флагов требуется сбросить флаги, которые не могут быть установлены одновременно с ними. Для этого удобно использовать вторым параметром метода `setf` перечисленные ниже *статические константы* класса `ios`:

- `adjustfield (left | right | internal)`
- `basefield (dec | oct | hex)`
- `floatfield (scientific | fixed)`

Пример форматирования при выводе с помощью флагов и методов:

```
#include <iostream.h>
int main(){
    long a = 1000, b = 077;
    cout.width(7);
    cout.setf(ios::hex | ios::showbase | ios::uppercase);
    cout << a;
    cout.width(7);
    cout << b << endl;
    double d = 0.12, c = 1.3e-4;
    cout.setf(ios::left);
    cout << d << endl;
    cout << c;
    return 0;
}
```

В результате работы программы в первой строке будут прописными буквами выведены переменные *a* и *b* в шестнадцатеричном представлении, под каждую из них отводится по 7 позиций (функция *width* действует только на одно выводимое значение, поэтому ее вызов требуется повторить дважды). Значения переменных *c* и *d* прижаты к левому краю поля:

```
0X3E8  0X3F
0.12
0.00013
```

Манипуляторы

Манипуляторами называются функции, которые можно включать в цепочку операций помещения и извлечения для форматирования данных. Манипуляторы делятся на *простые*, не требующие указания аргументов, и *параметризованные*. Пользоваться манипуляторами более удобно, чем методами установки флагов форматирования.

Простые манипуляторы

Ниже перечислены манипуляторы, не требующие указания аргументов.

- dec* — устанавливает при вводе и выводе флаг десятичной системы счисления;
- oct* — устанавливает при вводе и выводе флаг восьмеричной системы счисления;
- hex* — устанавливает при вводе и выводе флаг шестнадцатеричной системы счисления;
- ws* — устанавливает при вводе извлечение пробельных символов;
- endl* — при выводе включает в поток символ новой строки и выгружает буфер;
- ends* — при выводе включает в поток нулевой символ;
- flush* — при выводе выгружает буфер.

Изменения системы счисления действуют до следующего явного изменения.

Пример:

```
cout << 13 << hex << ' ' << 13 << oct << ' ' << 13 << endl;
```

Если другие значения флагов установлены по умолчанию, будет выведено:

```
13 d 15
```

Параметризованные манипуляторы

Ниже перечислены манипуляторы, требующие указания аргумента. Для их использования требуется подключить к программе заголовочный файл `<iomanip>`.

- `setbase(int n)` — задает основание системы счисления ($n = 8, 16, 10$ или 0). 0 является основанием по умолчанию (десятичное, кроме случаев, когда вводятся 8- или 16-ричные числа);
- `resetiosflags(long)` — сбрасывает флаги состояния потока, биты которых установлены в параметре;
- `setiosflags(long)` — устанавливает флаги состояния потока, биты которых в параметре равны 1;
- `setfill(int)` — устанавливает символ-заполнитель с кодом, равным значению параметра;
- `setprecision(int)` — устанавливает максимальное количество цифр в дробной части для вещественных чисел;
- `setw(int)` — устанавливает максимальную ширину поля вывода.

Пример использования параметризованных манипуляторов:

```
#include <iostream.h>
#include <iomanip.h>
int main(){
    double d[] = {1.234, -12.34567, 123.456789, -1.234, 0.00001};
    cout << setfill('.') << setprecision(4)
        << setiosflags(ios::showpoint | ios::fixed);
    for (int i = 0; i < 5; i++)
        cout << setw(12) << d[i] << endl;
    return 0;
}
```

Результат работы программы:

```
.....1.2340
....-12.3457
...123.4568
.....-1.2340
.....0.0000
```

Методы обмена с потоками

В потоковых классах наряду с операциями извлечения `>` и включения `<<` определены методы для неформатированного чтения и записи в поток (при этом преобразования данных не выполняются).

Ниже приведены функции чтения, определенные в классе `istream`.

<code>gcount()</code>	— возвращает количество символов, считанных с помощью последней функции неформатированного ввода;
<code>get()</code>	— возвращает код извлеченного из потока символа или EOF;
<code>get(c)</code>	— возвращает ссылку на поток, из которого выполнялось чтение, и записывает извлеченный символ в <code>c</code> ;
<code>get(buf, num, lim = '\n')</code> ¹	— считывает <code>num-1</code> символов (или пока не встретится символ <code>lim</code>) и копирует их в символьную строку <code>buf</code> . Вместо символа <code>lim</code> в строку записывается признак конца строки (<code>'\0'</code>). Символ <code>lim</code> остается в потоке. Возвращает ссылку на текущий поток;
<code>getline(buf, num, lim = '\n')</code>	— аналогична функции <code>get</code> , но копирует в <code>buf</code> и символ <code>lim</code> ;
<code>ignore(num = 1, lim = EOF)</code>	— считывает и пропускает символы до тех пор, пока не будет прочитано <code>num</code> символов или не встретится разделитель, заданный параметром <code>lim</code> . Возвращает ссылку на текущий поток;
<code>peek()</code>	— возвращает следующий символ без удаления его из потока или EOF, если достигнут конец файла;
<code>putback(c)</code>	— помещает в поток символ <code>c</code> , который становится текущим при извлечении из потока;
<code>read(buf, num)</code>	— считывает <code>num</code> символов (или все символы до конца файла, если их меньше <code>num</code>) в символьный массив <code>buf</code> и возвращает ссылку на текущий поток;
<code>readsome(buf, num)</code>	— считывает <code>num</code> символов (или все символы до конца файла, если их меньше <code>num</code>) в символьный массив <code>buf</code> и возвращает количество считанных символов;
<code>seekg(pos)</code>	— устанавливает текущую позицию чтения в значение <code>pos</code> ;

¹ Напомню, что при обращении к функции параметры, установленные по умолчанию, можно не указывать.

- `seekg(off, org)` — перемещает текущую позицию чтения на `off` байтов, считая от одной из трех позиций, определяемых параметром `org`: `ios::beg` (от начала файла), `ios::cur` (от текущей позиции) или `ios::end` (от конца файла);
- `tellg()` — возвращает текущую позицию чтения потока;
- `unset()` — помещает последний прочитанный символ в поток и возвращает ссылку на текущий поток.

В классе `ostream` определены аналогичные функции для неформатированного вывода:

- `flush()` — записывает содержимое потока вывода на физическое устройство;
- `put(c)` — выводит в поток символ `c` и возвращает ссылку на поток;
- `seekg(pos)` — устанавливает текущую позицию записи в значение `pos`;
- `seekg(off, org)` — перемещает текущую позицию записи на `off` байтов, считая от одной из трех позиций, определяемых параметром `org`: `ios::beg` (от начала файла), `ios::cur` (от текущей позиции) или `ios::end` (от конца файла);
- `tellg()` — возвращает текущую позицию записи потока;
- `write(buf, num)` — записывает в поток `num` символов из массива `buf` и возвращает ссылку на поток.

Пример 1. Программа считывает строки из входного потока в символьный массив.

```
#include "iostream.h"
int main(){
    const int N = 20, Len = 100;
    char str[N][Len];
    int i = 0;
    while (cin.getline(str[i], Len, '\n') && i < N){
        // -
        i++;
    }
    return 0;
}
```

Пример 2. Программа записывает в файл (файловые потоки рассматриваются в следующем разделе) число с плавающей точкой и строку символов, а затем считывает их из файла и выводит на экран:

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main(){
    // Запись в файл
    ofstream out("test");
```

```
if(!out){
    cout << "Cannot open file 'test' for writing" << endl;
    return 1;
}
double num = 100.45;
char str[ ] = "This is a test.";
out.write(reinterpret_cast<char *>(&num), sizeof(double));
out.write(str, strlen(str));
out.close();
// Чтение из файла
ifstream in("test");
if(!in){
    cout << "Cannot open file 'test' for reading" << endl;
    return 1;
}
double check_num;
char check_str[60];
in.read(reinterpret_cast<char *>(&check_num), sizeof(double));
in.read(check_str, 60);
int lstr = in.gcount(); // количество прочитанных символов
check_str[lstr] = 0;    // занести ноль-символ в конец строки
cout << check_num << ' ' << check_str << endl;
in.close();
return 0;
}
```

Приведение типа `reinterpret_cast<char *>` в вызове функций `write()` и `read()` необходимо в тех случаях, когда параметр не является символьным массивом.

Пример 3. В приведенной ниже программе формируется файл `test`, в который выводится три строки.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main(){
    // Запись в файл
    ofstream out("test");
    if(!out) {
        cout << "Cannot open file 'test' for writing" << endl;
        return 1;
    }
    char *str[ ] = {"This is the first line.",
                   "This is the second line.",
                   "This is the third line."};
    for (int i = 0; i<3; ++i){
        out.write(str[i], strlen(str[i]));
        out.put('\n');
    }
}
```

```

    out.close();
    // Чтение из файла
    ifstream in("test");
    if(!in){
        cout << "Cannot open file 'test' for reading" << endl;
        return 1;
    }
    char check_str[3][60];
    for (i = 0; i<3; ++i){
        in.get(check_str[i], 60);
        in.get();}
    // Контрольный вывод
    for (i = 0; i<3; ++i) cout << check_str[i] << endl;
    in.close();
    return 0;
}

```

После выполнения функции `get(check_str[i], 60)` символ-разделитель строк `'\n'` остается во входном потоке, поэтому необходим вызов `get()` для пропуска одного символа. Альтернативным способом является использование вместо функции `get` функции `getline`, которая извлекает символ-ограничитель из входного потока.

Пример 4. Функции `peek()` и `putback()` позволяют упростить управление, когда неизвестен тип вводимой в каждый конкретный момент времени информации. Следующая программа иллюстрирует это. В ней из файла (файловые потоки рассматриваются в следующем разделе) считываются либо строки, либо целые. Строки и целые могут следовать в любом порядке.

```

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>
int main(){
    char ch;
    // Подготовка файла
    ofstream out("test");
    if(!out) {
        cout << "Cannot open file 'test' for writing" << endl;
        return 1;
    }
    char str[80], *p;
    out << 123 << "this is a test" << 23;
    out << "Hello there!" << 99 << "bye" << endl;
    out.close();
    // Чтение файла
    ifstream in("test");
    if(!in){
        cout << "Cannot open file 'test' for reading" << endl;
        return 1;
    }
}

```

```
}
do{
    p = str;
    ch = in.peek(); // выяснение типа следующего символа
    if(isdigit(ch)){
        while(isdigit(*p = in.get())) p++; // считывание целого
        in.putback(*p); // возврат символа в поток
        *p = '\0'; // заканчиваем строку нулем
        cout << "Number: " << atoi(str);
    }
    else if(isalpha(ch)){ // считывание строки
        while(isalpha(*p = in.get())) p++;
        in.putback(*p); // возврат символа в поток
        *p = '\0'; // заканчиваем строку нулем
        cout << "String: " << str;
    }
    else in.get(); // пропуск
    cout << endl;
} while(!in.eof());
in.close();
return 0;
}
```

Результат работы программы:

```
Number: 123
String: this
```

```
String: is
```

```
String: a
```

```
String: test
```

```
Number: 23
```

```
String: Hello
```

```
String: there
```

```
Number: 99
```

```
String: bye
```

При организации диалогов с пользователем программы при помощи потоков необходимо учитывать буферизацию. Например, при выводе приглашения к вводу мы не можем гарантировать, что оно появится раньше, чем будут считаны данные из входного потока, поскольку приглашение появится на экране только при заполнении буфера вывода:

```
cout << "Введите x";
cin >> x;
```


Для решения этой проблемы в `basic_ios` определена функция `tie()`, которая связывает потоки `istream` и `ostream` с помощью вызова вида `cin.tie(&cout)`. После этого вывод очищается (то есть выполняется функция `cout.flush()`) каждый раз, когда требуется новый символ из потока ввода.

Использовать в одной программе потоки и функции библиотеки C, описанные в `<cstdio>` или `<stdio.h>`, не рекомендуется. Если это по каким-либо причинам необходимо, то до выполнения первой операции с потоками следует вызвать описанную в `ios_base` функцию `sync_with_stdio()`, которая обеспечит использование общих буферов. Вызов `sync_with_stdio(false)` разъединяет буфера (это может привести к увеличению производительности).

Ошибки потоков

В базовом классе `ios` определено поле `state`, которое представляет собой состояние потока в виде совокупности битов:

```
enum io_state{
    goodbit = 0x00,    // Нет ошибок
    eofbit = 0x01,     // Достигнут конец файла
    failbit = 0x02,    // Ошибка форматирования или преобразования
    badbit = 0x04,     // Серьезная ошибка, после которой
                      // пользоваться потоком невозможно
    hardfail = 0x08    // Неисправность оборудования
};
```

Состоянием потока можно управлять с помощью перечисленных ниже методов и операций:

- `int rdstate()` — возвращает текущее состояние потока;
- `int eof()` — возвращает ненулевое значение, если установлен флаг `eofbit`;
- `int fail()` — возвращает ненулевое значение, если установлен один из флагов `failbit`, `badbit` или `hardfail`;
- `int bad()` — возвращает ненулевое значение, если установлен один из флагов `badbit` или `hardfail`;
- `int good()` — возвращает ненулевое значение, если сброшены все флаги ошибок;
- `void clear(int = 0)` — параметр принимается в качестве состояния ошибки, при отсутствии параметра состояние ошибки устанавливается 0;
- `operator void*()` — возвращает нулевой указатель, если установлен хотя бы один бит ошибки;
- `operator !()` — возвращает ненулевой указатель, если установлен хотя бы один бит ошибки.

Далее приведены часто используемые операции с флагами состояния потока.

```
// Проверить, установлен ли флаг flag:
if(stream_obj.rdstate() & ios::flag)
// Сбросить флаг flag:
stream_obj.clear(rdstate() & ~ios::flag)
// Установить флаг flag:
stream_obj.clear(rdstate() | ios::flag)
// Установить флаг flag и сбросить все остальные:
stream_obj.clear( ios::flag)
// Сбросить все флаги:
stream_obj.clear()
```

Операция `void*()` неявно вызывается всякий раз, когда поток сравнивается с 0. Это позволяет записывать циклы вида:

```
while (stream_obj){
    // Все в порядке, можно производить ввод/вывод
}
```

В приведенном ниже примере показана работа функции `rdstate()`. Программа выводит на экран содержимое текстового файла, имя которого задается в командной строке. При наличии ошибки функция сообщает об этом посредством `CheckStatus()`.

```
#include <iostream.h>
#include <fstream.h>
void CheckStatus(istream &in):
int main(int argc, char *argv[ ]){
    if(argc != 2){
        cout << "Usage: <program_name> <file_name>" << endl;
        return 1;
    }
    ifstream in(argv[1]):
    if(!in){
        cout << "Cannot open file" << argv[1] << endl;
        return 1;
    }
    char c;
    while(in.get(c)){
        cout << c; CheckStatus(in);
    }
    CheckStatus(in); // контроль финального состояния
    in.close();
    return 0;
}
void CheckStatus(istream &in){
    int i;
    i = in.rdstate();
    if(i & ios::eofbit)
        cout << "EOF is occurred" << endl;
    else if(i & ios::failbit)
```

```
cout << "Not fatal input/output error" << endl;  
else if(i & ios::badbit)  
    cout << "Fatal input/output error" << endl;
```

Файловые потоки

Под файлом обычно подразумевается именованная информация на внешнем носителе, например, на жестком или гибком магнитном диске. Логически файл можно представить как конечное количество последовательных байтов, поэтому такие устройства, как дисплей, клавиатуру и принтер также можно рассматривать как частные случаи файлов.

По способу доступа файлы можно разделить на *последовательные*, чтение и запись в которых производятся с начала байт за байтом, и *файлы с произвольным доступом*, допускающие чтение и запись в указанную позицию.

Стандартная библиотека содержит три класса для работы с файлами:

ifstream — класс входных файловых потоков;

ofstream — класс выходных файловых потоков;

fstream — класс двунаправленных файловых потоков.

Эти классы являются производными от классов istream, ostream и iostream соответственно, поэтому они наследуют перегруженные операции << и >, флаги форматирования, манипуляторы, методы, состояние потоков и т. д.

Использование файлов в программе предполагает следующие операции:

- ☐ создание потока;
- ☐ открытие потока и связывание его с файлом;
- ☐ обмен (ввод/вывод);
- ☐ закрытие файла.

Каждый класс файловых потоков содержит конструкторы, с помощью которых можно создавать объекты этих классов различными способами.

- ☐ Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом:

```
ifstream();  
ofstream();  
fstream();
```

- ☐ Конструкторы с параметрами создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

```
ifstream(const char *name, int mode = ios::in);  
ofstream(const char *name, int mode = ios::out | ios::trunc);  
fstream(const char *name, int mode = ios::in | ios::out);
```

Вторым параметром конструктора является режим открытия файла. Если установленное по умолчанию значение не устраивает программиста, можно указать другое, составив его из битовых масок, определенных в классе `ios`:

```
enum open_mode{
    in  = 0x01,      // Открыть для чтения
    out = 0x02,      // Открыть для записи
    ate = 0x04,      // Установить указатель на конец файла
    app = 0x08,      // Открыть для добавления в конец
    trunc = 0x10,    // Если файл существует, удалить
    nocreate = 0x20, // Если файл не существует, выдать ошибку
    noreplace = 0x40, // Если файл существует, выдать ошибку
    binary = 0x80,   // Открыть в двоичном режиме
};
```

В таблице 10.2 приведено соответствие между битовыми масками класса `ios` и режимами открытия файла, описанными в `<stdio.h>`.

Таблица 10.2. Режимы открытия файла

Комбинация флагов ios					Эквивалент stdio
binary	in	out	trunc	app	
		+			"w"
		+		+	"a"
		+	+		"w"
	+				"r"
	+	+			"r+"
	+	+	+		"w+"
+		+			"wb"
+		+		+	"ab"
+		+	+		"wb"
+	+				"rb"
+	+	+			"r+b"
+	+	+	+		"w+b"

Открыть файл в программе можно с использованием либо конструкторов, либо метода `open`, имеющего такие же параметры, как и в соответствующем конструкторе, например:

```
ifstream inpf ("input.txt"); // Использование конструктора
if (!inpf){
    cout << "Невозможно открыть файл для чтения": return 1;
```

```

    }
    ostream f;
    f.open("output.txt", ios::out);    // Использование метода open
    if (!f){
        cout << "Невозможно открыть файл для записи";
        return 1;
    }

```

Чтение и запись выполняются либо с помощью операций чтения и извлечения, аналогичных потоковым классам, либо с помощью методов классов.

Пример использования методов (программа выводит на экран содержимое файла):

```

#include <fstream.h>
int main(){
    char text[81], buf[81];
    cout << "Введите имя файла:";
    cin >> text;
    ifstream f(text);
    if (!f){
        cout << "Ошибка открытия файла"; return 1;
    }
    while (!f.eof()){
        f.getline(buf, 81);
        cout << buf << endl;
    }
    return 0;
}

```

Для закрытия потока определен метод `close()`, но поскольку он неявно выполняется деструктором, явный вызов необходим только тогда, когда требуется закрыть поток раньше конца его области видимости.

Строковые потоки

Строковые потоки позволяют считывать и записывать информацию из областей оперативной памяти так же, как из файла, с консоли или на дисплей. В стандартной библиотеке определено три класса строковых потоков¹:

`istream` — входные строковые потоки;
`ostream` — выходные строковые потоки;
`stringstream` — двунаправленные строковые потоки.

Эти классы определяются в заголовочном файле `<sstream>`² и являются производными от классов `istream`, `ostream` и `iostream` соответственно, поэтому они на-

¹ В старых версиях компиляторов эти классы могут иметь другие названия.

² В старых версиях компиляторов заголовочный файл может иметь имя `<strstrea.h>`.

следуют перегруженные операции << и >>, флаги форматирования, манипуляторы, методы, состояние потоков и т. д.

Участки памяти, с которыми выполняются операции чтения и извлечения, по стандарту определяются как строки C++ (класс string). Строковый класс рассматривается на с. 286. Строковые потоки создаются и связываются с этими участками памяти с помощью конструкторов:

```
explicit1 istream(int mode = ios::in);
explicit istream(const string& name, int mode = ios::in);
explicit ostream(int mode = ios::out);
explicit ostream(const string& name, int mode = ios::out);
explicit stringstream(int mode = ios::in | ios::out);
explicit stringstream(const string& name, int mode = ios::in | ios::out);
```

Строковые потоки являются некоторым аналогом функций `sscanf` и `sprintf` библиотеки C и могут применяться для преобразования данных, когда они заносятся в некоторый участок памяти, а затем считываются в величины требуемых типов. Эти потоки могут применяться также для обмена информацией между модулями программы.

В строковых потоках описан метод `str`, возвращающий копию строки или устанавливающий ее значение:

```
string str() const;
void str(const string & s);
```

Проверять строковый поток на переполнение не требуется, поскольку размер строки изменяется динамически.

В приведенном ниже примере строковый поток используется для формирования сообщения, включающего текущее время и передаваемый в качестве параметра номер:

```
#include <sstream>
#include <string>
#include <iostream>
#include <ctime>
using namespace std;
string message( int i){
    ostream os;
    time_t t;
    time(&t);
    os << " time: " << ctime(&t) << " number: " << i << endl;
    return os.str();
}
int main(){
    cout << message(22);
    return 0;
}
```

¹ см. с. 197.

Потоки и типы, определенные пользователем

Для ввода и вывода в потоках используются перегруженные для всех стандартных типов операции чтения и извлечения `<<` и `>>`. При этом выбор конкретной операции определяется типом фактических параметров. Для того чтобы вводить и выводить величины типов, определенных пользователем, требуется перегрузить эти операции. Это бинарные операции, левым операндом которых является объект-поток, а правым — объект, который требуется извлечь или поместить в этот поток. Возвращаемое значение должно быть ссылкой на поток, чтобы можно было организовывать цепочки операций, как и в случае стандартных типов.

Пусть, например, в программе определен класс `MyClass`:

```
class MyClass{
    int   x;
    float y;
}
```

Для того чтобы вводить и выводить объекты этого класса, требуется определить в классе `MyClass` операции следующего вида:

```
// Вывод:
friend ostream& operator << (ostream& out, MyClass& C){
    return out << "x = " << C.x << " y = " << C.y;
}
// Ввод:
friend istream& operator >> (istream& in, MyClass& C){
    cout << "Введите x: "; in >> C.x;
    cout << "Введите y: "; in >> C.y;
    return in;
}
```

После этого в программе можно использовать объекты класса `MyClass` в операциях ввода и вывода наряду с величинами стандартных типов:

```
#include <iostream.h>
class MyClass{
    int   x;
    float y;
public:
    MyClass(int nx = 1, float ny = 0.01){x = nx; y = ny;}
    friend ostream& operator<< (ostream& out, MyClass& C){
        return out << "x = " << C.x << " y = " << C.y;
    }
    friend istream& operator>> (istream& in, MyClass& C){
        cout << "Введите x: "; in >> C.x;
        cout << "Введите y: "; in >> C.y;
        return in;
    }
}
```

```
    }  
};  
int main(){  
    MyClass C;  
    cout << C << endl;  
    MyClass C1(100, 100);  
    cout << C1 << endl;  
    MyClass C2;  
    cin >> C2; cout << C2 << endl;  
    return 0;  
}
```

Экземпляр C класса MyClass создается с параметрами конструктора по умолчанию, поэтому на экран будет выведено:

```
x = 1 y = 0.01
```

Экземпляр C1 класса MyClass создается с параметрами 100, 100:

```
x = 100 y = 100
```

После создания экземпляра C2 будет выведено приглашение ко вводу x и y, а затем введенные с клавиатуры значения будут выведены на экран.

ГЛАВА 11

Строки

*Стандартные классы просто так
в состав C++ не добавляются.*

Г. Шилдт

C++ не содержит стандартного типа данных «строка». Вместо этого он поддерживает массивы символов, завершаемые нуль-символом. Библиотека содержит функции для работы с такими массивами, унаследованные от C и описанные в заголовочном файле `<string.h>` (`<cstring>`). Об этих функциях рассказывалось в разделе «Функции работы со строками и символами», с. 91. Они позволяют достичь высокой эффективности, но весьма неудобны и небезопасны в использовании, поскольку выход за границы строки не проверяется.

Тип данных `string` стандартной библиотеки лишен этих недостатков, но может проигрывать массивам символов в эффективности. Основные действия со строками выполняются в нем с помощью операций и методов, а длина строки изменяется динамически в соответствии с потребностями. Для использования класса необходимо подключить к программе заголовочный файл `<string>`¹. Рассмотрим пример:

```
#include <cstring>
#include <string>
#include <iostream>
using namespace std;
int main (){
    char c1[80], c2[80], c3[80]; // Строки с завершающим нулем
    string s1, s2, s3;
    // Присваивание строк
    strcpy(c1, "old string one");
    strcpy(c2, c1);
```

¹ В старых версиях компиляторов заголовочный файл для подключения класса `string` может иметь имя `cstring.h` или `bstring.h`, а заголовочный файл для строк старого стиля — `string.h`.

```
s1 = "new string one";  
s2 = s1;  
// Конкатенация строк  
strcpy(c3, c1);  
strcpy(c3, c2);  
s3 = s1 + s2;  
// Сравнение строк.  
if (strcmp(c2, c3) < 0 ) cout << c2;  
else cout << c3;  
if (s2 < s3) cout << s2;  
else cout << s3;  
}
```

Как видно из примера, выполнение любых действий со строками старого стиля требует использования функций и менее наглядно. Кроме того, необходимо проверять, достаточно ли места в строке-приемнике при копировании, то есть фактически код работы со строками старого стиля должен быть еще более длинным.

Строки типа `string` защищены от выхода информации за их границы и с ними можно работать так же, как с любым встроенным типом данных, то есть с помощью операций. Рассмотрим основные особенности и приемы работы со строками.

Конструкторы и присваивание строк

В классе `string` определено несколько конструкторов. Ниже в упрощенном виде приведены заголовки наиболее употребительных:

```
string();  
string(const char *);  
string(const char *, int n);  
string(string &);
```

Первый конструктор создает пустой объект типа `string`. Второй создает объект типа `string` на основе строки старого стиля, третий создает объект типа `string` и записывает туда `n` символов из строки, указанной первым параметром. Последний конструктор является конструктором копирования, который создает новый объект как копию объекта, переданного ему в качестве параметра.

В классе `string` определены три операции присваивания:

```
string& operator=(const string& str);  
string& operator=(const char* s);  
string& operator=(char c);
```

Как видно из заголовков, строке можно присваивать другую строку типа `string`, строку старого стиля или отдельный символ, например:

```
string s1;  
string s2("Вася");  
string s3(s2);  
s1 = 'X';  
s1 = "Вася";  
s2 = s3;
```

Операции

Ниже приведены допустимые для объектов класса `string` операции:

Операция	Действие	Операция	Действие
<code>-</code>	присваивание	<code>></code>	больше
<code>+</code>	конкатенация	<code>>=</code>	больше или равно
<code>=</code>	равенство	<code>[]</code>	индексация
<code>!=</code>	неравенство	<code><<</code>	вывод
<code><</code>	меньше	<code>>></code>	ввод
<code><=</code>	меньше или равно	<code>+=</code>	добавление

Синтаксис операций и их действие очевидны. Размеры строк устанавливаются автоматически так, чтобы объект мог содержать присваиваемое ему значение. Надо отметить, что для строк типа `string` не соблюдается соответствие между адресом первого элемента строки и именем, как это было в случае строк старого стиля, то есть `&s[0]` не равно `s`.

Кроме операции индексации, для доступа к элементу строки определена функция `at`:

```
string s("Вася");
cout << s.at(1); // Будет выведен символ а
```

Если индекс превышает длину строки, порождается исключение `out_of_range`.

Для работы со строками целиком этих операций достаточно, а для обработки частей строк (например, поиска подстроки, вставки в строку, удаления символов) в классе `string` определено множество разнообразных методов (функций), наиболее употребительные из которых рассматриваются в следующем разделе.

Функции

Функции класса `string` для удобства рассмотрения можно разбить на несколько категорий: присваивание и добавление частей строк, преобразования строк, поиск подстрок, сравнение и получение характеристик строк.

Присваивание и добавление частей строк

Для присваивания части одной строки другой служит функция `assign`:

```
assign(const string& str);
assign(const string& str, size_type pos, size_type n);
assign(const char* s, size_type n);
```

Первая форма функции присваивает строку `str` вызывающей строке, при этом действие функции эквивалентно операции присваивания:

```
string s1("Вася"), s2;
s2.assign(s1); // Равносильно s2 = s1;
```

Вторая форма присваивает вызывающей строке часть строки `str`, начиная с позиции `pos`¹. Если `pos` больше длины строки, порождается исключение `out_of_range`. Вызывающей строке присваивается `n` символов, либо, если `pos + n` больше, чем длина строки `str`, все символы до конца строки `str`.

Третья форма присваивает вызывающей строке `n` символов строки `s` старого типа.

Для добавления части одной строки к другой служит функция `append`:

```
append(const string& str);  
append(const string& str, size_type pos, size_type n);  
append(const char* s, size_type n);
```

Первая форма функции добавляет строку `str` к концу вызывающей строки, при этом действие функции эквивалентно операции конкатенации (+).

Вторая форма добавляет к вызывающей строке часть строки `str`, начиная с позиции `pos`. Если `pos` больше длины строки, порождается исключение `out_of_range`. К вызывающей строке добавляется `n` символов, либо, если `pos + n` больше, чем длина строки `str`, все символы до конца строки `str`. Если длина результата больше максимально допустимой длины строки, порождается исключение `length_error`.

Третья форма добавляет к вызывающей строке `n` символов строки `s` старого типа.

Преобразования строк

Для вставки в одну строку части другой строки служит функция `insert`:

```
insert(size_type pos1, const string& str);  
insert(size_type pos1, const string& str, size_type pos2, size_type n);  
insert(size_type pos, const char* s, size_type n);
```

Первая форма функции вставляет строку `str` в вызывающую строку, начиная с позиции `pos1` вызывающей строки. Иными словами, вызывающая строка замещается строкой, которая состоит из первых `pos1` символов вызывающей строки, за которыми следует строка `str` целиком, а после нее располагаются остальные символы вызывающей строки. Если `pos1` больше длины строки, порождается исключение `out_of_range`. Если длина результата больше максимально допустимой длины строки, порождается исключение `length_error`.

Вторая форма функции вставляет в вызывающую строку часть строки `str`, начиная с позиции `pos1` вызывающей строки. Вызывающая строка замещается строкой, которая состоит из первых `pos1` символов вызывающей строки, за которыми следуют `n` элементов строки `str`, начиная с позиции `pos2`, а после них располагаются остальные символы вызывающей строки. Если `n` больше длины строки `str`, копируется весь остаток строки `str`. Если `pos1` или `pos2` больше длины соответствующей строки, порождается исключение `out_of_range`. Если длина результата

¹ `size_type` представляет собой беззнаковый целый тип; достаточный для хранения размера самого большого объекта для данной модели.

больше максимально допустимой длины строки, порождается исключение `length_error`.

Третья форма функции вставляет в вызывающую строку `n` элементов строки `s` старого типа, начиная с позиции `pos` вызывающей строки.

Для удаления части строки служит функция `erase`:

```
erase(size_type pos = 0, size_type n = npos);
```

Она удаляет из вызывающей строки `n` элементов, начиная с позиции `pos`. Если `pos` не указано, элементы удаляются с начала строки. Если не указано `n`, удаляется весь остаток строки.

ВНИМАНИЕ

Величина `npos` является статическим членом класса `string` и представляет собой самое большое положительное число типа `size_type` (все единицы в битовом представлении).

Очистку всей строки можно выполнить с помощью функции `clear`:

```
void clear();
```

Для замены части строки служит функция `replace`:

```
replace(size_type pos1, size_type n1, const string& str);  
replace(size_type pos1, size_type n1, const string& str, size_type pos2,  
size_type n2);
```

Здесь `pos1` — позиция вызывающей строки, начиная с которой выполняется замена, `n1` — количество удаляемых элементов, `pos2` — позиция строки `str`, начиная с которой она вставляется в вызывающую строку, `n2` — количество вставляемых элементов строки `str`. Если `pos1` или `pos2` больше длины соответствующей строки, порождается исключение `out_of_range`. Если длина результата больше максимально допустимой длины строки, порождается исключение `length_error`.

Третья форма функции замены позволяет заменить `n1` символов вызывающей строки на `n2` символов строки старого стиля `s`:

```
replace(size_type pos1, size_type n1, const char* s, size_type n2);
```

Для обмена содержимого двух строк служит функция `swap`:

```
swap(string& s);
```

Для выделения части строки служит функция `substr`:

```
string substr(size_type pos = 0, size_type n = npos) const;
```

Эта функция возвращает подстроку вызываемой строки длиной `n`, начиная с позиции `pos`. Если `pos` больше длины строки, порождается исключение `out_of_range`. Если `n` больше длины строки, возвращается весь остаток строки.

Иногда требуется преобразовывать объекты типа `string` в строки старого стиля. Для этого предназначена функция `c_str`:

```
const char* c_str() const;
```

Она возвращает константный указатель на оканчивающуюся нуль-символом строку. Эту строку нельзя пытаться изменить. Указатель, который на нее ссылается, может стать некорректным после любой операции над строкой-источником.

Аналогично работает функция `data`, за тем исключением, что не добавляет в конец строки нуль-символ:

```
const char* data() const;
```

Функция `copy` копирует в массив `s` `n` элементов вызывающей строки, начиная с позиции `pos`. Нуль-символ в результирующий массив не заносится. Функция возвращает количество скопированных элементов:

```
size_type copy(char* s, size_type n, size_type pos = 0) const;
```

Пример использования функций изменения содержимого строк:

```
#include <string>
#include <iostream>
using namespace std;
int main (){
    string s1("прекрасная королева"), s2("ле"), s3("корова");
    cout << "s1= " << s1 << endl;
    cout << "s2= " << s2 << endl;
    cout << "s3= " << s3 << endl;
    // Применение функции insert:
    cout << "после insert:" << endl;
    cout << "s3= " << s3.insert(4, s2) << endl;
    cout << "s3= " << s3.insert(7, "к") << endl;
    // Применение функции erase:
    s1.erase(0,3);
    cout << "после erase:" << endl;
    cout << "s1= " << s1.erase(12,2) << endl;
    // Применение функции replace:
    cout << "после replace:" << endl;
    cout << "s1= " << s1.replace(0,3, s3, 4,2) << endl;
}
```

Результат работы программы:

```
s1= прекрасная королева
s2= ле
s3= корова
после insert:
s3= королева
s3= королевка
после erase:
s1= красная корова
после replace:
s1= лесная корова
```

Поиск подстрок

Для поиска в классе `string` предусмотрено большое разнообразие функций. Ниже приведены основные:

```
size_type find(const string& str, size_type pos = 0) const;
```

Ищет самое левое вхождение строки `str` в вызывающую строку, начиная с позиции `pos`, и возвращает позицию строки или `npos`, если строка не найдена.

```
size_type find(char c, size_type pos = 0) const;
```

Ищет самое левое вхождение символа `c` в вызывающую строку, начиная с позиции `pos`, и возвращает позицию символа или `npos`, если символ не найден.

```
size_type rfind(const string& str, size_type pos = npos) const;
```

Ищет самое правое вхождение строки `str` в вызывающую строку, до позиции `pos`, и возвращает позицию строки или `npos`, если строка не найдена.

```
size_type rfind(char c, size_type pos = npos) const;
```

Ищет самое правое вхождение символа `c` в вызывающую строку, до позиции `pos`, и возвращает позицию символа или `npos`, если символ не найден.

```
size_type find_first_of(const string& str, size_type pos = 0) const;
```

Ищет самое левое вхождение любого символа строки `str` в вызывающую строку, начиная с позиции `pos`, и возвращает позицию символа или `npos`, если вхождение не найдено.

```
size_type find_first_of(char c, size_type pos = 0) const;
```

Ищет самое левое вхождение символа `c` в вызывающую строку, начиная с позиции `pos`, и возвращает позицию символа или `npos`, если вхождение не найдено.

```
size_type find_last_of(const string& str, size_type pos = npos) const;
```

Ищет самое правое вхождение любого символа строки `str` в вызывающую строку, начиная с позиции `pos`, и возвращает позицию символа или `npos`, если вхождение не найдено.

```
size_type find_last_of(char c, size_type pos = npos) const;
```

Ищет самое правое вхождение символа `c` в вызывающую строку, начиная с позиции `pos`, и возвращает позицию символа или `npos`, если вхождение не найдено.

```
size_type find_first_not_of(const string& str, size_type pos = 0) const;
```

Ищет самую левую позицию, начиная с позиции `pos`, для которой ни один символ строки `str` не совпадает с символом вызывающей строки.

```
size_type find_first_not_of(char c, size_type pos = 0) const;
```

Ищет самую левую позицию, начиная с позиции `pos`, для которой символ `c` не совпадает с символом вызывающей строки.

```
size_type find_last_not_of(const string& str, size_type pos = npos) const;
```

Ищет самую правую позицию до позиции `pos`, для которой ни один символ строки `str` не совпадает с символом вызывающей строки.

```
size_type find_last_not_of(char c, size_type pos = npos) const;
```

Ищет самую правую позицию до позиции `pos`, для которой символ `c` не совпадает с символом вызывающей строки.

Для каждой функции существует вариант, позволяющий искать в заданной строке подстроки старого стиля.

Пример применения функций поиска:

```
#include <string>
#include <iostream>
using namespace std;
int main (){
    string s1("лесная королева"), s2("ле");
    cout << "s1= " << s1 << endl;
    cout << "s2= " << s2 << endl;
    int i = s1.find(s2);
    int j = s1.rfind(s2);
    cout << "первое s2 в s1 " << i << endl;
    cout << "последнее s2 в s1 " << j << endl;
    cout << "первое 'o' в s1 " << s1.find('o') << endl;
    cout << "последнее 'o' в s1 " << s1.rfind('o') << endl;
    cout << "первое в s1 " << s1.find_first_of("adcde") << endl;
    cout << "последнее в s1 " << s1.find_last_of("adcde") << endl;
}
```

Результат работы программы:

```
s1= лесная королева
s2= ле
первое s2 в s1 0
последнее s2 в s1 11
первое 'o' в s1 8
последнее 'o' в s1 10
первое в s1 1
последнее в s1 14
```

Сравнение частей строк

Для сравнения строк целиком применяются перегруженные операции отношения, а если требуется сравнивать части строк, используется функция `compare`:

```
int compare(const string& str) const;
int compare(size_type pos1, size_type n1, const string& str) const;
int compare(size_type pos1, size_type n1,
            const string& str, size_type pos2, size_type n2) const;
```

Первая форма функции сравнивает две строки целиком и возвращает значение, меньшее 0, если вызывающая строка лексикографически меньше `str`, равное нулю, если строки одинаковы, и большее нуля — если вызывающая строка больше. Эта форма является аналогом функции сравнения строк `strcmp` библиотеки C.

Вторая форма функции выполняет аналогичные действия, но сравнивает со строкой `str` `n1` символов вызывающей строки, начиная с `pos1`.

Третья форма функции сравнивает `n1` символов вызывающей строки, начиная с `pos1`, с подстрокой строки `str` длиной `n2` символов, начиная с `pos2`.

Аналогичные формы функций существуют и для сравнения строк типа `string` со строками старого стиля.

Пример использования функции сравнения строк:

```
#include <string>
#include <iostream>
using namespace std;
int main (){
    string s1("лесная королева"), s2("ле"), s3("корова");
    cout << "s1= " << s1 << endl;
    cout << "s2= " << s2 << endl;
    cout << "s3= " << s3 << endl;
    if (s2.compare(s3) > 0) cout << "s2 > s3 " << endl;
    if (s1.compare(7, 4, s3) < 0)
        cout << "s1[7-10] < s3 " << endl;
    if (s1.compare(7, 4, s3, 0, 4) == 0)
        cout << "s1[7-10] == s3[0-3] " << endl;
}
```

Результат работы программы:

```
s1= лесная королева
s2= ле
s3= корова
s2 > s3
s1[7-10] < s3
s1[7-10] == s3[0-3]
```

Получение характеристик строк

В классе `string` определено несколько функций-членов, позволяющих получить длину строки и объем памяти, занимаемый объектом:

<code>size_type size() const;</code>	// Количество элементов строки
<code>size_type length() const;</code>	// Количество элементов строки
<code>size_type max_size() const;</code>	// Максимальная длина строки
<code>size_type capacity() const;</code>	// Объем памяти, занимаемый строкой
<code>bool empty() const;</code>	// Истина, если строка пустая

ГЛАВА 12

Контейнерные классы

Контейнерные классы — это классы, предназначенные для хранения данных, организованных определенным образом. Примерами контейнеров могут служить массивы, линейные списки или стеки. Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере, поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована с помощью шаблонов классов, поэтому часть библиотеки C++, в которую входят контейнерные классы, а также алгоритмы и итераторы, о которых будет рассказано в следующих разделах, называют *стандартной библиотекой шаблонов* (STL — Standard Template Library).

Использование контейнеров позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.

Естественно, эти преимущества не даются даром: универсальность и безопасность использования контейнерных классов не могут не отражаться на быстродействии программы. Снижение быстродействия в зависимости от реализации компилятора может быть весьма значительным. Кроме того, для эффективного использования контейнеров требуется затратить усилия на вдумчивое освоение библиотеки.

STL содержит контейнеры, реализующие основные структуры данных, используемые при написании программ — *векторы, двусторонние очереди, списки и их разновидности, словари и множества*. Контейнеры можно разделить на два типа: последовательные и ассоциативные.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности. К ним относятся векторы (vector), двусторонние очереди (deque) и списки (list), а также так называемые адаптеры, то есть варианты контейнеров — стеки (stack), очереди (queue) и очереди с приоритетами (priority_queue).

Каждый вид контейнера обеспечивает свой набор действий над данными. Выбор вида контейнера зависит от того, что требуется делать с данными в программе.

Например, при необходимости часто вставлять и удалять элементы из середины последовательности следует использовать список, а если включение элементов выполняется главным образом в конец или начало — двустороннюю очередь.

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров: словари (*map*), словари с дубликатами (*multimap*), множества (*set*), множества с дубликатами (*multiset*) и битовые множества (*bitset*).

Программист может создавать собственные контейнерные классы на основе имеющихся в стандартной библиотеке.

Центральным понятием STL является *шаблон*, поэтому перед тем, как приступить к изучению материала этой главы, рекомендуется убедиться, что это понятие не представляет для читателя загадку (см. «Шаблоны функций», с. 85, и «Шаблоны классов», с. 211). Также необходимо знать, что такое пространства имен (с. 99), перегрузка функций (с. 83) и перегрузка операций (с. 189).

Контейнерные классы обеспечивают стандартизованный интерфейс при их использовании. Смысл одноименных операций для различных контейнеров одинаков, основные операции применимы ко всем типам контейнеров. Стандарт определяет только интерфейс контейнеров, поэтому разные реализации могут сильно отличаться по эффективности.

Практически в любом контейнерном классе определены поля перечисленных ниже типов:

Поле	Пояснение
<code>value_type</code>	Тип элемента контейнера
<code>size_type</code>	Тип индексов, счетчиков элементов и т. д.
<code>iterator</code>	Итератор
<code>const_iterator</code>	Константный итератор
<code>reverse_iterator</code>	Обратный итератор
<code>const_reverse_iterator</code>	Константный обратный итератор
<code>reference</code>	Ссылка на элемент
<code>const_reference</code>	Константная ссылка на элемент
<code>key_type</code>	Тип ключа (для ассоциативных контейнеров)
<code>key_compare</code>	Тип критерия сравнения (для ассоциативных контейнеров)

Итератор является аналогом указателя на элемент. Он используется для просмотра контейнера в прямом или обратном направлении. Все, что требуется от итератора — уметь ссылаться на элемент контейнера и реализовывать операцию перехода к его следующему элементу. Константные итераторы используются тогда, когда значения соответствующих элементов контейнера не изменяются (более подробно от итераторах рассказывается в разделе «Итераторы», с. 328).

При помощи итераторов можно просматривать контейнеры, не заботясь о фактических типах данных, используемых для доступа к элементам. Для этого в каждом контейнере определено несколько методов, перечисленных в следующей таблице.

Метод	Пояснение
<code>iterator begin()</code> , <code>const_iterator begin() const</code>	Указывают на первый элемент
<code>iterator end()</code> , <code>const_iterator end() const</code>	Указывают на элемент, следующий за последним
<code>reverse_iterator rbegin()</code> , <code>const_reverse_iterator rbegin() const</code>	Указывают на первый элемент в обратной последовательности
<code>reverse_iterator rend()</code> , <code>const_reverse_iterator rend() const</code>	Указывают на элемент, следующий за последним, в обратной последовательности

В каждом контейнере эти типы и методы определяются способом, зависящим от их реализации.

Во всех контейнерах определены методы, позволяющие получить сведения об их размере:

Метод	Пояснение
<code>size()</code>	Число элементов
<code>max_size()</code>	Максимальный размер контейнера (порядка миллиарда элементов)
<code>empty()</code>	Булевская функция, показывающая, пуст ли контейнер

Другие поля и методы контейнеров мы рассмотрим по мере необходимости.

STL определяется в 13 заголовочных файлах:

```
algorithm  deque  functional  iterator  list  map
memory     numeric queue set    stack    utility vector
```

Последовательные контейнеры

Векторы (`vector`), двусторонние очереди (`deque`) и списки (`list`) поддерживают разные наборы операций, среди которых есть совпадающие операции. Они могут быть реализованы с разной эффективностью:

Операция	Метод	<code>vector</code>	<code>deque</code>	<code>list</code>
Вставка в начало	<code>push_front</code>	—	+	+
Удаление из начала	<code>pop_front</code>	—	+	+
Вставка в конец	<code>push_back</code>	+	+	+

Операция	Метод	vector	deque	list
Удаление из конца	pop_back	+	+	+
Вставка в произвольное место	insert	(+)	(+)	+
Удаление из произвольного места	erase	(+)	(+)	+
Произвольный доступ к элементу	[]. at	+	+	—

Знак + означает, что соответствующая операция реализуется за постоянное время, не зависящее от количества n элементов в контейнере. Знак (+) означает, что соответствующая операция реализуется за время, пропорциональное n . Для малых n время операций, обозначенных +, может превышать время операций, обозначенных (+), но для большого количества элементов последние могут оказаться очень дорогими.

Как видно из таблицы, такими операциями являются вставка и удаление произвольных элементов очереди и вектора, поскольку при этом все последующие элементы требуется переписывать на новые места.

Итак, *вектор* — это структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца.

Двусторонняя очередь эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов.

Список эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

Пример работы с вектором. В файле находится произвольное количество целых чисел. Программа считывает их в вектор и выводит на экран в том же порядке.

```
#include <fstream>
#include <vector>
using namespace std;
int main(){
    ifstream in ("inpnun.txt");
    vector<int> v;
    int x;
    while ( in >> x, !in.eof())
        v.push_back(x);
    for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
        cout << *i << " ";
}
```

Поскольку файл содержит целые числа, используется соответствующая специализация шаблона `vector` — `vector<int>`. Для создания вектора `v` используется конструктор по умолчанию. Организуется цикл до конца файла, в котором из него считывается очередное целое число. С помощью метода `push_back` оно заносится в вектор, размер которого увеличивается автоматически¹.

¹ Размер вектора не изменяется каждый раз при добавлении элемента, это было бы нерационально. Он увеличивается по определенному алгоритму, которым можно управлять (см. с. 301).

Для прохода по всему вектору вводится переменная `i` как итератор соответствующего типа (напомню, что операция `::` обозначает доступ к области видимости, то есть здесь объявляется переменная `i` типа «итератор для конкретной специализации шаблона»). С помощью этого итератора осуществляется доступ ко всем по порядку элементам контейнера, начиная с первого. Метод `begin()` возвращает указатель на первый элемент, метод `end()` — на элемент, следующий за последним. Реализация гарантирует, что этот указатель определен.

Сравнивать текущее значение с граничным следует именно с помощью операции `!=`, так как операции `<` или `<=` могут быть для данного типа не определены. Операция инкремента (`i++`) реализована так, чтобы после нее итератор указывал на следующий элемент контейнера в порядке обхода. Доступ к элементу вектора выполняется с помощью операции разадресации, как для обычных указателей.

В данном примере вместо вектора можно было использовать любой последовательный контейнер путем простой замены слова `vector` на `deque` или `list`. При этом изменилось бы внутреннее представление данных и набор доступных операций, а в поведении программы никаких изменений не произошло бы.

Однако если вместо цикла `for` вставить фрагмент

```
for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
```

в котором использована операция доступа по индексу `[ ]`, программа не будет работать для контейнера типа `list`, поскольку в нем эта операция не определена.

Векторы (vector)

Для создания вектора можно воспользоваться следующими конструкторами (приведена упрощенная запись):

```
explicit vector(); // 1
explicit vector(size_type n, const T& value = T()); // 2
template <class InputIter> // 3
    vector(InputIter first, InputIter last);
vector(const vector<T>& x); // 4
```

Ключевое слово `explicit` используется для того, чтобы при создании объекта запретить выполняемое неявно преобразование при присваивании значения другого типа (см. также с. 197).

Конструктор 1 является конструктором по умолчанию.

Конструктор 2 создает вектор длиной `n` и заполняет его одинаковыми элементами — копиями `value`.

Поскольку изменение размера вектора обходится дорого, при его создании задавать начальный размер весьма полезно. При этом для встроенных типов выполняется инициализация каждого элемента значением `value`. Если оно не указано, элементы глобальных векторов инициализируются нулем.

Если тип элемента вектора определен пользователем, начальное значение формируется с помощью конструктора по умолчанию для данного типа. На месте второго параметра можно написать вызов конструктора с параметрами, создав таким образом вектор элементов с требуемыми свойствами (см. пример далее).

ПРИМЕЧАНИЕ

Элементы любого контейнера являются копиями вставляемых в него объектов. Поэтому для них должны быть определены конструктор копирования и операция присваивания.

Конструктор 3 создает вектор путем копирования указанного с помощью итераторов диапазона элементов. Тип итераторов должен быть «для чтения».

Конструктор 4 является конструктором копирования.

Примеры конструкторов:

```
// Создается вектор из 10 равных единице элементов:
vector<int> v2 (10, 1);
// Создается вектор, равный вектору v1:
vector<int> v4 (v1);
// Создается вектор из двух элементов, равных первым двум элементам v1:
vector<int> v3 (v1.begin(), v1.begin() + 2);
// Создается вектор из 10 объектов класса monstr (см. с. 183)
// (работает конструктор по умолчанию):
vector<monstr> m1 (10);
// Создается вектор из 5 объектов класса monstr с заданным именем
// (работает конструктор с параметром char*):
vector<monstr> m2 (5, monstr("Вася"));
```

В шаблоне vector определены операция присваивания и функция копирования:

```
vector<T>& operator=(const vector<T>& x);
void assign(size_type n, const T& value);
template <class InputIter>
void assign(InputIter first, InputIter last);
```

Здесь через T обозначен тип элементов вектора. Вектора можно присваивать друг другу точно так же, как стандартные типы данных или строки. После присваивания размер вектора становится равным новому значению, все старые элементы удаляются.

Функция assign в первой форме аналогична по действию конструктору 2, но применяется к существующему объекту. Функция assign во второй форме предназначена для присваивания элементам вызывающего вектора значений из диапазона, определяемого итераторами first и last, аналогично конструктору 3, например:

```
vector<int> v1, v2;
// Первым 10 элементам вектора v1 присваивается значение 1:
v1.assign(10, 1);
// Первым 3 элементам вектора v2 присваиваются значения v2[5], v2[6], v2[7]:
v2.assign(v2.begin() + 5, v2.begin() + 8);
```

Итераторы класса vector перечислены в табл. 12.

Доступ к элементам вектора осуществляется с помощью следующих операций и методов:

```
reference      operator[](size_type n);
const_reference operator[](size_type n) const;
```

```

const_reference at(size_type n) const;
reference       at(size_type n);
reference       front();
const_reference front() const;
reference       back();
const_reference back() const;

```

Операция [] осуществляет доступ к элементу вектора по индексу без проверки его выхода за границу вектора. Функция `at` выполняет такую проверку и порождает исключение `out_of_range` в случае выхода за границу вектора. Естественно, что функция `at` работает медленнее, чем операция [], поэтому в случаях, когда диапазон определен явно, предпочтительнее пользоваться операцией:

```
for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
```

В противном случае используется функция `at` с обработкой исключения:

```

try{
    // _
    v.at(i) = v.at(_);
}
catch(out_of_range) { _ }

```

Операции доступа возвращают значение ссылки на элемент (`reference`) или константной ссылки (`const_reference`) в зависимости от того, применяются ли они к константному объекту или нет.

Методы `front` и `back` возвращают ссылки соответственно на первый и последний элементы вектора (это не то же самое, что `begin` — указатель на первый элемент и `end` — указатель на элемент, следующий за последним). Пример:

```

vector<int> v(5, 10);
v.front() = 100;   v.back() = 100;
cout << v[0] << " " << v[v.size() - 1]; // Вывод: 100 100

```

Функция `capacity` определяет размер оперативной памяти, занимаемой вектором:

```
size_type capacity() const;
```

Память под вектор выделяется динамически, но не под один элемент в каждый момент времени (это было бы расточительным расходом ресурсов), а сразу под группу элементов, например, 256 или 1024. Перераспределение памяти происходит только при превышении этого количества элементов, при этом объем выделенного пространства удваивается. После перераспределения любые итераторы, ссылающиеся на элементы вектора, становятся недействительными, поскольку вектор может быть перемещен в другой участок памяти, и нельзя ожидать, что связанные с ним ссылки будут обновлены автоматически.

Существует также функция выделения памяти `reserve`, которая позволяет задать, сколько памяти требуется для хранения вектора:

```
void reserve(size_type n);
```

Пример применения функции:

```

vector<int> v;
v.reserve(1000); // Выделение памяти под 1000 элементов

```


После выполнения этой функции значение функции `capacity` будет равно по меньшей мере `n`. Функцию `reserve` полезно применять тогда, когда размер вектора известен заранее.

Для изменения размеров вектора служит функция `resize`:

```
void resize(size_type sz, T c = T());
```

Эта функция увеличивает или уменьшает размер вектора в зависимости от того, больше задаваемое значение `sz`, чем значение `size()`, или меньше. Вторым параметром задает значение, которое присваивается всем новым элементам вектора. Они помещаются в конец вектора. Если новый размер меньше, чем значение `size()`, из конца вектора удаляется `size() - sz` элементов.

Определены следующие методы для изменения объектов класса `vector`:

```
void push_back(const T& value);
void pop_back();
iterator insert(iterator position, const T& value);
void insert(iterator position, size_type n, const T& value);
template <class InputIter>
    void insert(iterator position, InputIter first, InputIter last);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void swap();
void clear(); // Очистка вектора
```

Функция `push_back` добавляет элемент в конец вектора, функция `pop_back` — удаляет элемент из конца вектора.

Функция `insert` служит для вставки элемента в вектор. Первая форма функции вставляет элемент `value` в позицию, заданную первым параметром (итератором), и возвращает итератор, ссылающийся на вставленный элемент. Вторая форма функции вставляет в вектор `n` одинаковых элементов. Третья форма функции позволяет вставить несколько элементов, которые могут быть заданы любым диапазоном элементов подходящего типа, например:

```
vector<int> v(2), v1(3,9);
int m[3] = {3, 4, 5};
v.insert(v.begin(), m, m + 3); // Содержимое v: 3 4 5 0 0
v1.insert(v1.begin() + 1, v.begin(), v.begin() + 2);
// Содержимое v1: 9 3 4 9 9
```

Вставка в вектор занимает время, пропорциональное количеству сдвигаемых на новые позиции элементов. При этом, если новый размер вектора превышает объем занимаемой памяти, происходит перераспределение памяти. Это — плата за легкость доступа по индексу. Если при вставке перераспределения не происходит, все итераторы сохраняют свои значения. В противном случае они становятся недействительными.

Функция `erase` служит для удаления одного элемента вектора (первая форма функции) или диапазона, заданного с помощью итераторов (вторая форма):

```
vector<int> v;
for (int i = 1; i < 6; i++) v.push_back(i);
```

```
// Содержимое v: 1 2 3 4 5
v.erase(v.begin());           // Содержимое v: 2 3 4 5
v.erase(v.begin(), v.begin() + 2); // Содержимое v: 4 5
```

Обратите внимание, что третьим параметром задается не последний удаляемый элемент, а элемент, следующий за ним.

Каждый вызов функции `erase` так же, как и в случае вставки, занимает время, пропорциональное количеству сдвигаемых на новые позиции элементов. Все итераторы и ссылки «правее» места удаления становятся недействительными.

Функция `swap` служит для обмена элементов двух векторов одного типа, но не обязательно одного размера:

```
vector<int> v1, v2;

v1.swap(v2); // Эквивалентно v2.swap(v1);
```

Для векторов определены операции сравнения `=`, `!=`, `<`, `<=` и `=`. Два вектора считаются равными, если равны их размеры и все соответствующие пары элементов. Один вектор меньше другого, если первый из элементов одного вектора, не равный соответствующему элементу другого, меньше него (то есть сравнение лексикографическое). Пример:

```
#include <vector>
using namespace std;
vector<int> v7, v8;
int main(){
    for (int i = 0; i<6; i++)v7.push_back(i);
    cout << "v7: ";
    for (int i = 0; i<6; i++) cout << v7[i] << " ";
    cout << endl;
    for (int i = 0; i<3; i++)v8.push_back(i+1);
    cout << "v8: ";
    for (int i = 0; i<3; i++) cout << v8[i] << " ";
    cout << endl;
    if (v7 < v8 ) cout << " v7 < v8" << endl;
    else cout << " v7 > v8" << endl;
}
```

Результат работы программы:

```
v7: 0 1 2 3 4 5
v8: 1 2 3
v7 < v8
```

Для эффективной работы с векторами в стандартной библиотеке определены шаблоны функций, называемые алгоритмами. Они включают в себя поиск значений, сортировку элементов, вставку, замену, удаление и другие операции. Алгоритмы описаны в разделе «Алгоритмы» на с. 343.

Векторы логических значений (`vector<bool>`)

Специализация шаблона `vector<bool>` определена для оптимизации размещения памяти, поскольку можно реализовать вектор логических значений так, чтобы

его элемент занимал 1 бит. При этом адресация отдельных битов выполняется программно. Итератор такого вектора не может быть указателем. В остальном векторы логических значений аналогичны обычным и реализуют тот же набор операций и методов. В дополнение к ним определены методы инвертирования бита и вектора в целом (`flip`).

Ссылка на элемент вектора логических значений реализована в виде класса `reference`, моделирующего обычную ссылку на элемент:

```
class reference{
    friend class vector;
    reference();
public:
    ~reference();
    operator bool() const;
    reference& operator=(const bool x);
    reference& operator=(const reference& x);
    void flip();
};
```

Пример (с клавиатуры вводятся в вектор 10 значений 0 или 1, после чего они выводятся на экран).

```
#include <vector>
#include <iostream>
using namespace std;
vector<bool> v(10);
int main(){
    for(int i = 0; i<v.size(); i++)cin >> v[i];
    for (vector<bool>:: const_iterator p = v.begin(); p!=v.end(); ++p)
        cout << *p;
}
```

Двусторонние очереди (deque)

Двусторонняя очередь — это последовательный контейнер, который, наряду с вектором, поддерживает произвольный доступ к элементам и обеспечивает вставку и удаление из обоих концов очереди за постоянное время. Те же операции с элементами внутри очереди занимают время, пропорциональное количеству перемещаемых элементов. Распределение памяти выполняется автоматически.

Рассмотрим схему организации очереди (рис. 12.1). Для того чтобы обеспечить произвольный доступ к элементам за постоянное время, очередь разбита на блоки, доступ к каждому из которых осуществляется через указатель. На рисунке закрашенные области соответствуют занятым элементам очереди. Если при добавлении в начало или в конец блок оказывается заполненным, выделяется память под очередной блок (например, после заполнения блока 4 будет выделена память под блок 5, а после заполнения блока 2 — под блок 1). При заполнении крайнего из блоков происходит перераспределение памяти под массив указателей так, чтобы использовались только средние элементы. Это не занимает много времени.

Таким образом, доступ к элементам очереди осуществляется за постоянное время, хотя оно и несколько больше, чем для вектора.

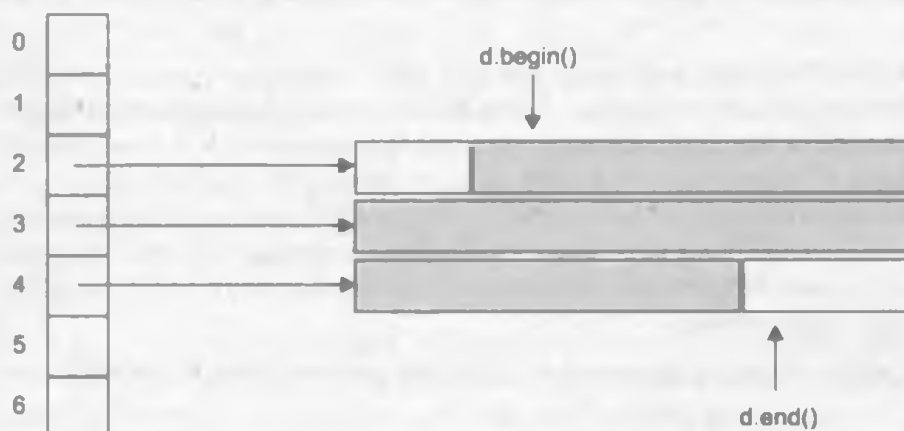


Рис. 12.1. Организация двусторонней очереди

Для создания двусторонней очереди можно воспользоваться следующими конструкторами (приведена упрощенная запись), аналогичными конструкторам вектора:

```
explicit deque(); // 1
explicit deque(size_type n, const T& value = T()); // 2
template <class InputIter> // 3
    deque(InputIter first, InputIter last);
deque(const vector<T>& x); // 4
```

Конструктор 1 является конструктором по умолчанию.

Конструктор 2 создает очередь длиной n и заполняет ее одинаковыми элементами — копиями $value$.

Конструктор 3 создает очередь путем копирования указанного с помощью итераторов диапазона элементов. Тип итераторов должен быть «для чтения».

Конструктор 4 является конструктором копирования.

Примеры конструкторов:

```
// Создается очередь из 10 равных единице элементов:
deque<int> d2(10, 1);
// Создается очередь, равная очереди v1:
deque<int> d4(v1);
// Создается очередь из двух элементов, равных первым двум
// элементам вектора v1 из предыдущего раздела:
deque<int> d3(v1.begin(), v1.begin() + 2);
// Создается очередь из 10 объектов класса monstr (см. с. 183)
// (работает конструктор по умолчанию):
deque<monstr> m1(10);
// Создается очередь из 5 объектов класса monstr с заданным именем
// (работает конструктор с параметром char*):
deque<monstr> m2(5, monstr("Вася в очереди"));
```

В шаблоне `deque` определены операция присваивания, функция копирования, итераторы, операции сравнения, операции и функции доступа к элементам и изменения объектов, аналогичные соответствующим операциям и функциям вектора.

Вставка и удаление так же, как и для вектора, выполняются за пропорциональное количеству элементов время. Если эти операции выполняются над внутренними элементами очереди, все значения итераторов и ссылок на элементы очереди становятся недействительными. После операций добавления в любой из концов все значения итераторов становятся недействительными, а значения ссылок на элементы очереди сохраняются. После операций выборки из любого конца становятся недействительными только значения итераторов и ссылок, связанных с этими элементами.

Кроме перечисленных, определены функции добавления и выборки из начала очереди:

```
void push_front(const T& value);  
void pop_front();
```

При выборке элемент удаляется из очереди.

Для очереди не определены функции `capacity` и `reserve`, но есть функции `resize` и `size`.

К очередям можно применять алгоритмы стандартной библиотеки, описанные в главе 14 «Алгоритмы» на с. 343.

Списки (list)

Список не предоставляет произвольного доступа к своим элементам, зато вставка и удаление выполняются за постоянное время. Класс `list` реализован в STL в виде двусвязного списка, каждый узел которого содержит ссылки на последующий и предыдущий элементы. Поэтому операции инкремента и декремента для итераторов списка выполняются за постоянное время, а передвижение на n узлов требует времени, пропорционального n .

После выполнения операций вставки и удаления значения всех итераторов и ссылок остаются действительными.

Список поддерживает конструкторы, операцию присваивания, функцию копирования, операции сравнения и итераторы, аналогичные векторам и очередям.

Доступ к элементам для списков ограничивается следующими методами:

```
reference      front();  
const_reference front() const;  
reference      back();  
const_reference back() const;
```

Для занесения в начало и конец списка определены методы, аналогичные соответствующим методам очереди:

```
void push_front(const T& value);  
void pop_front();  
void push_back(const T& value);
```

```
void pop_back();
```

Кроме того, действуют все остальные методы для изменения объектов `list`, аналогичные векторам и очередям:

```
iterator insert(iterator position, const T& value);
void insert(iterator position, size_type n, const T& value);
template <class InputIter>
    void insert(iterator position, InputIter first, InputIter last);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void swap();
void clear();
```

Для списка не определена функция `capacity`, поскольку память под элементы отводится по мере необходимости. Можно изменить размер списка, удалив или добавив элементы в конец списка (аналогично двусторонней очереди):

```
void resize(size_type sz, T c = T());
```

Кроме перечисленных, для списков определено несколько специфических методов. **Сцепка списков** (`splice`) служит для перемещения элементов из одного списка в другой без перераспределения памяти, только за счет изменения указателей:

```
void splice(iterator position, list<T>& x);
void splice(iterator position, list<T>& x, iterator i);
void splice(iterator position, list<T>& x, iterator first, iterator last);
```

Оба списка должны содержать элементы одного типа. Первая форма функции вставляет в вызывающий список перед элементом, позиция которого указана первым параметром, все элементы списка, указанного вторым параметром, например:

```
list<int> L1, L2;
_ // Формирование списков
L1.splice(L1.begin() + 4, L2);
```

Второй список остается пустым. Нельзя вставить список в самого себя.

Вторая форма функции переносит элемент, позицию которого определяет третий параметр, из списка `x` в вызывающий список. Допускается переносить элемент в пределах одного списка.

Третья форма функции аналогичным образом переносит из списка в список несколько элементов. Их диапазон задается третьим и четвертым параметрами функции. Если для одного и того же списка первый параметр находится в диапазоне между третьим и четвертым, результат не определен. Пример:

```
#include <list>
using namespace std;
int main(){
    list<int> L1;
    list<int>::iterator i, j, k;
    for (int i = 0; i<5; i++)L1.push_back(i + 1);
    for (int i = 12; i<14; i++)L1.push_back(i);
```



```

cout << "Исходный список: ";
for (i = L1.begin(); i != L1.end(); ++i) cout << *i << " ";
cout << endl;
i = L1.begin(); i++;
k = L1.end();
j = --k; k++; j--;
L1.splice( i, L1, j, k);
cout << "Список после сцепки: ";
for ( i = L1.begin(); i != L1.end(); ++i)
    cout << *i << " ";
}

```

Результат работы программы:

Исходный список: 1 2 3 4 5 12 13

Список после сцепки: 1 12 13 2 3 4 5

Перемещенные элементы выделены полужирным шрифтом. Обратите внимание, что для итераторов списков не определены операции сложения и вычитания, то есть нельзя написать $j = k - 1$, поэтому пришлось воспользоваться допустимыми для итераторов списков операциями инкремента и декремента. В общем случае для поиска элемента в списке используется функция `find` (см. с. 346).

Для удаления элемента по его значению применяется функция `remove`:

```
void remove(const T& value);
```

Если элементов со значением `value` в списке несколько, все они будут удалены.

Можно удалить из списка элементы, удовлетворяющие некоторому условию. Для этого используется функция `remove_if`:

```
template <class Predicate> void remove_if(Predicate pred);
```

Параметром является класс-предикат, задающий условие, накладываемое на элемент списка. О предикатах см. с. 336.

Для упорядочивания элементов списка используется метод `sort`:

```
void sort();
template <class Compare> void sort(Compare comp);
```

В первом случае список сортируется по возрастанию элементов (в соответствии с определением операции `<` для элементов), во втором — в соответствии с функциональным объектом `Compare` (о функциональных объектах рассказывалось в разделе «Перегрузка операции вызова функции» на с. 195). Функциональный объект имеет значение `true`, если два передаваемых ему значения должны при сортировке остаться в прежнем порядке, и `false` — в противном случае.

Порядок следования элементов, имеющих одинаковые значения, сохраняется. Время сортировки пропорционально $N \log_2 N$, где N — количество элементов в списке.

Метод `unique` оставляет в списке только первый элемент из каждой серии идущих подряд одинаковых элементов. Первая форма метода имеет следующий формат:

```
void unique();
```

Вторая форма метода `unique` использует в качестве параметра бинарный предикат (см. с. 336), что позволяет задать собственный критерий удаления элементов списка. Предикат имеет значение `true`, если критерий соблюден, и `false` — в противном случае. Аргументы предиката имеют тип элементов списка:

```
template <class BinaryPredicate>
    void unique(BinaryPredicate binary_pred);
```

Для слияния списков служит метод `merge`:

```
void merge(list<T>& x);
template <class Compare> void merge(list<T>& x, Compare comp);
```

Оба списка должны быть упорядочены (в первом случае в соответствии с определением операции `<` для элементов, во втором — в соответствии с функциональным объектом `Compare`). Результат — упорядоченный список. Если элементы в вызывающем списке и в списке-параметре совпадают, первыми будут располагаться элементы из вызывающего списка.

Метод `reverse` служит для изменения порядка следования элементов списка на обратный (время работы пропорционально количеству элементов):

```
void reverse();
```

Пример работы со списком:

```
#include <fstream>
#include <list>
using namespace std;
void show (const char *str, const list<int> &L){
    cout << str << ":" << endl;
    for (list<int>::const_iterator i = L.begin(); i != L.end(); ++i)
        cout << *i << " ";
    cout << endl;
}
int main(){
    list<int> L;
    list<int>::iterator i;
    int x;
    ifstream in("inprnum");
    while ( in >> x, !in.eof()) L.push_back(x);
    show("Исходный список", L);
    L.push_front(1);
    i = L.begin(); L.insert(++i, 2);
    show("После вставки 1 и 2 в начало", L);
    i = L.end(); L.insert(--i, 100);
    show("После вставки 100 перед последним", L);
    i = L.begin(); x = *i; L.pop_front();
    cout << "Удалили из начала" << x << endl;
    i = L.end(); x = *--i; L.pop_back();
    cout << "Удалили с конца" << x << endl;
    show("Список после удаления", L);
    L.remove(76);
```

```

    show("После удаления элементов со значением 76", L);
    L.sort();
    show("После сортировки", L);
    L.unique();
    show("После unique", L);
    list<int> L1 (L);
    L.reverse();
    show("После reverse", L);
}

```

Результат работы программы:

```

Исходный список:
56 34 54 0 76 23 51 11 51 11 76 88
После вставки 1 и 2 в начало:
1 2 56 34 54 0 76 23 51 11 51 11 76 88
После вставки 100 перед последним:
1 2 56 34 54 0 76 23 51 11 51 11 76 100 88
Удалили из начала 1
Удалили с конца 88
Список после удаления:
2 56 34 54 0 76 23 51 11 51 11 76 100
После удаления элементов со значением 76:
2 56 34 54 0 23 51 11 51 11 100
После сортировки:
0 2 11 11 23 34 51 51 54 56 100
После unique:
0 2 11 23 34 51 54 56 100
После reverse:
100 56 54 51 34 23 11 2 0

```

К спискам можно применять алгоритмы стандартной библиотеки, описанные в разделе «Алгоритмы» (с. 343).

Стеки (stack)

Как известно, в стеке (определение см. раздел «Стеки», с. 119) допускаются только две операции, изменяющие его размер — добавление элемента в вершину стека и выборка из вершины. Стек можно реализовать на основе любого из рассмотренных контейнеров: вектора, двусторонней очереди или списка. Таким образом, стек является не новым типом контейнера, а вариантом имеющихся, поэтому он называется *адаптером* контейнера. Другие адаптеры (очереди и очереди с приоритетами) будут рассмотрены в следующих разделах.

В STL стек определен по умолчанию на базе двусторонней очереди:

```

template <class T, class Container = deque<T> >
class stack {
protected:
    Container c;

```

```

public:
    explicit stack(const Container& = Container());
    bool      empty() const      {return c.empty();}
    size_type size() const      {return c.size();}
    value_type& top()            {return c.back();}
    const value_type& top() const {return c.back();}
    void push(const value_type& x) {c.push_back(x);}
    void pop()                   {c.pop_back();}
};

```

Из приведенного описания (оно дано с сокращениями) видно, что метод занесения в стек `push` соответствует методу занесения в конец `push_back`, метод выборки из стека `pop` — методу выборки с конца `pop_back`, кроме того, добавлен метод `top` для получения или изменения значения элемента на вершине стека. Конструктору класса `stack` передается в качестве параметра ссылка на базовый контейнер, который копируется в защищенное поле данных `c`.

При работе со стеком нельзя пользоваться итераторами и нельзя получить значение элемента из середины стека иначе, чем выбрав из него все элементы, лежащие выше. Для стека, как и для всех рассмотренных выше контейнеров, определены операции сравнения (см. 303).

Пример использования стека (программа вводит из файла числа и выводит их на экран в обратном порядке):

```

#include <fstream>
#include <vector>
#include <stack>
using namespace std;
int main(){
    ifstream in ("inpnun");
    stack <int, vector<int> > s;
    int x;
    while ( in >> x. !in.eof()) s.push(x);
    while (! s.empty()){
        x = s.top(); cout << x << " ";
        s.pop();
    }
}

```

Содержимое файла `inpnun`:

```
56 34 54 0 76 23 51 11 51 11 76 88
```

Результат работы программы:

```
88 76 11 51 11 51 23 76 0 54 34 56
```

Очереди (queue)

Для очереди (определение см. раздел «Очереди», с. 121) допускаются две операции, изменяющие ее размер — добавление элемента в конец и выборка из начала. Очередь является адаптером, который можно реализовать на основе двусторон-

ней очереди или списка (вектор не подходит, поскольку в нем нет операции выборки из начала).

В STL очередь определена по умолчанию на базе двусторонней очереди:

```
template <class T, class Container = deque<T> >
class queue {
protected:
    Container c;
public:
    explicit queue(const Container& = Container());
    bool empty() const {return c.empty();}
    size_type size() const {return c.size();}
    value_type& front() {return c.front();}
    const value_type& front() const {return c.front();}
    value_type& back() {return c.back();}
    const value_type& back() const {return c.back();}
    void push(const value_type& x) {c.push_back(x);}
    void pop() {c.pop_front();}
};
```

Методы `front` и `back` используются для получения значений элементов, находящихся соответственно в начале и в конце очереди (при этом элементы остаются в очереди).

Пример работы с очередью (программа вводит из файла числа в очередь и выполняет выборку из нее, пока очередь не опустеет):

```
#include <fstream>
#include <list>
#include <queue>
using namespace std;
int main(){
    ifstream in ("inprnum");
    queue<int, list<int> > q;
    int x;
    while ( in >> x, !in.eof()) q.push(x);
    cout << "q.front(): " << q.front() << " ";
    cout << "q.back(): " << q.back() << endl;
    while (! q.empty()){
        q.pop();
        cout << "q.front(): " << q.front() << " ";
        cout << "q.back(): " << q.back() << endl;
    }
}
```

Содержимое файла `inprnum`:

```
56 34 54 0 76 23 51 11 51 11 76 88
```

Результат работы программы:

```
q.front(): 56   q.back(): 88
q.front(): 34   q.back(): 88
```

```

q.front(): 54   q.back(): 88
q.front(): 0    q.back(): 88
q.front(): 76   q.back(): 88
q.front(): 23   q.back(): 88
q.front(): 51   q.back(): 88
q.front(): 11   q.back(): 88
q.front(): 51   q.back(): 88
q.front(): 11   q.back(): 88
q.front(): 76   q.back(): 88
q.front(): 88   q.back(): 88
q.front(): 0    q.back(): 0

```

К стекам и очередям можно применять алгоритмы стандартной библиотеки, описанные в разделе «Алгоритмы» (с. 343).

Очереди с приоритетами (priority_queue)

В очереди с приоритетами каждому элементу соответствует приоритет, определяющий порядок выборки из очереди. По умолчанию он определяется с помощью операции <; таким образом, из очереди каждый раз выбирается максимальный элемент.

Для реализации очереди с приоритетами подходит контейнер, допускающий произвольный доступ к элементам, то есть, например, вектор или двусторонняя очередь. Тип контейнера передается вторым параметром шаблона (первый, как обычно, тип элементов). Третьим параметром указывается функция или функциональный объект (см. с. 195 и 334), с помощью которых выполняется определение приоритета:

```

template <class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
class priority_queue {
protected:
    Container c;
    Compare comp;
public:
    explicit priority_queue(const Compare& x = Compare(),
                           const Container& = Container());
    template <class InputIter>
    priority_queue(InputIter first, InputIter last,
                  const Compare& x = Compare(), const Container& = Container());
    bool empty() const {return c.empty();}
    size_type size() const {return c.size();}
    const value_type& top() const {return c.front();}
    void push(const value_type& x);
    void pop();
};

```

Для элементов с равными приоритетами очередь с приоритетами является простой очередью. Как и для стеков, основными методами являются push, pop и top.

Простой пример:

```
#include <iostream>
#include <vector>
#include <functional>
#include <queue>
using namespace std;
int main(){
    priority_queue <int, vector<int>, less<int> > P;
    int x;
    P.push(13); P.push(51); P.push(200); P.push(17);
    while (!P.empty()){
        x = P.top(); cout << "Выбран элемент: " << x << endl;
        P.pop();
    }
}
```

Результат работы программы:

```
Выбран элемент: 200
Выбран элемент: 51
Выбран элемент: 17
Выбран элемент: 13
```

В этом примере третьим параметром шаблона является шаблон, определенный в заголовочном файле <functional> (см. раздел «Функциональные объекты», с. 334). Он задает операцию сравнения на «меньше». Можно задать стандартные шаблоны greater<тип>, greater_equal<тип>, less_equal<тип>. Если требуется определить другой порядок выборки из очереди, вводится собственный функциональный объект. В приведенном ниже примере выборка выполняется по наименьшей сумме цифр в числе:

```
#include <iostream>
#include <vector>
#include <functional>
#include <queue>
using namespace std;
class CompareSum{
public:
    bool operator()(int x, int y){
        int sx = 0, sy = 0;
        while (x){sx += x % 10; x /= 10;}
        while (y){sy += y % 10; y /= 10;}
        return sx > sy ;
    }
};
int main(){
    priority_queue <int, vector<int>, CompareSum > P;
    int x;
    P.push(13); P.push(51); P.push(200); P.push(17);
    while (!P.empty()){
```

```

        x = P.top(); cout << "Выбран элемент: " << x << endl;
        P.pop();
    }
}

```

Результат работы программы:

```

Выбран элемент: 200
Выбран элемент: 13
Выбран элемент: 51
Выбран элемент: 17

```

Ассоциативные контейнеры

Как уже указывалось, ассоциативные контейнеры обеспечивают быстрый доступ к данным за счет того, что они, как правило, построены на основе сбалансированных деревьев поиска (стандартом регламентируется только интерфейс контейнеров, а не их реализация).

Существует пять типов ассоциативных контейнеров: словари (`map`), словари с дубликатами (`multimap`), множества (`set`), множества с дубликатами (`multiset`) и битовые множества (`bitset`). Словари часто называют также *ассоциативными массивами* или *отображениями*.

Словарь построен на основе пар значений, первое из которых представляет собой ключ для идентификации элемента, а второе — собственно элемент. Можно сказать, что ключ ассоциирован с элементом, откуда и произошло название этих контейнеров. Например, в англо-русском словаре ключом является английское слово, а элементом — русское. Обычный массив тоже можно рассматривать как словарь, ключом в котором служит номер элемента. В словарях, описанных в STL, в качестве ключа может использоваться значение произвольного типа. Ассоциативные контейнеры описаны в заголовочных файлах `<map>` и `<set>`.

Для хранения пары «ключ—элемент» используется шаблон `pair`, описанный в заголовочном файле `<utility>`:

```

template <class T1, class T2> struct pair{
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    pair();
    pair(const T1& x, const T2& y);
    template <class U, class V> pair(const pair<U, V> &p);
};

```

Шаблон `pair` имеет два параметра, представляющих собой типы элементов пары. Первый элемент имеет имя `first`, второй — `second`. Определено два конструктора: один должен получать два значения для инициализации элементов, второй (конструктор копирования) — ссылку на другую пару. Конструктора по умолчанию у пары нет, то есть при создании объекта ему требуется присвоить значение явным образом.

Для пары определены проверка на равенство и операция сравнения на меньше (все остальные операции отношения генерируются в STL автоматически на основе этих двух операций). Пара `p1` меньше пары `p2`, если `p1.first < p2.first` или `p1.first == p2.first && p1.second < p2.second`.

Для присваивания значения паре можно использовать функцию `make_pair`:

```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

Пример формирования пар:

```
#include <iostream>
#include <utility>
using namespace std;
int main(){
    pair<int, double> p1(10, 12.3), p2(p1);
    p2 = make_pair(20, 12.3); // Эквивалентно p2 = pair <int, double >(20, 12.3)
    cout << "p1: " << p1.first << " " << p1.second << endl;
    cout << "p2: " << p2.first << " " << p2.second << endl;
    p2.first -= 10;
    if (p1 == p2) cout << "p1 == p2\n";
    p1.second -= 1;
    if (p2 > p1) cout << "p2 > p1\n";
}
```

Результат работы программы:

```
p1: 10 12.3
p2: 20 12.3
p1 == p2
p2 > p1
```

Заголовочный файл `<utility>` при использовании `<map>` или `<set>` подключается автоматически.

Словари (map)

В словаре (`map`), в отличие от словаря с дубликатами (`multimap`), все ключи должны быть уникальны. Элементы в словаре хранятся в отсортированном виде, поэтому для ключей должно быть определено отношение «меньше». Шаблон словаря содержит три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение «меньше» (функциональные объекты рассматривались на с. 195):

```
template <class Key, class T, class Compare = less<Key> >
class map{
public:
    typedef pair <const Key, T> value_type;
    explicit map(const Compare& comp = Compare());
    template <class InputIter>
```

```

        map(InputIter first, InputIter last, const Compare& comp =
Compare()):
    map(const map<Key, T, Compare>& x);

};

```

Как видно из приведенного описания (оно дано с сокращениями), тип элементов словаря `value_type` определяется как пара элементов типа `Key` и `T`.

Первый конструктор создает пустой словарь, используя указанный функциональный объект. Второй конструктор создает словарь и записывает в него элементы, определяемые диапазоном указанных итераторов. Время работы этого конструктора пропорционально количеству записываемых элементов, если они упорядочены, и квадрату количества элементов, если нет. Третий конструктор является конструктором копирования.

Как и для всех контейнеров, для словаря определены деструктор, операция присваивания и операции отношения. Итераторы словаря перечислены в табл. 12.

Для доступа к элементам по ключу определена операция `[ ]`:

```
T& operator[](const Key & x);
```

С помощью этой операции можно не только получать значения элементов, но и добавлять в словарь новые. Не буду отступать от традиций и в качестве примера словаря приведу телефонную книгу, ключом в которой служит фамилия, а элементом — номер телефона:

```

#include <fstream>
#include <iostream>
#include <string>
#include <map>
using namespace std;
typedef map<string, long, less<string> > map_sl; // 1
int main(){
    map_sl m1;
    ifstream in("phonebook");
    string str;
    long num;
    while (in >> num, !in.eof()){
        in.get(); // Чтение номера
        getline(in, str); // Пропуск пробела
        m1[str] = num; // Чтение фамилии
        cout << str << " " << num << endl; // Занесение в словарь
    }
    m1["Petya P."] = 2134622; // Дополнение словаря
    map_sl::iterator i;
    cout << "m1:" << endl; // Вывод словаря
    for (i = m1.begin(); i != m1.end(); i++)
        cout << (*i).first << " " << (*i).second << endl;
    i = m1.begin(); i++; // Вывод второго элемента
    cout << "Второй элемент: ";
    cout << (*i).first << " " << (*i).second << endl;
}

```

```
cout << "Vasia: " << m1["Vasia"] << endl;    // Вывод элемента по ключу
return 0;
}
```

Для улучшения читаемости программы введено более короткое обозначение типа словаря (оператор, помеченный // 1). Сведения о каждом человеке расположены в файле phonebook на одной строке: сначала идет номер телефона, затем через пробел фамилия:

```
1001002 Petya K.
3563398 Ivanova N.M.
1180316 Vovochka
2334476 Vasia
```

Для итераторов словаря допустимы операции инкремента и декремента, но не операции + и -. Ниже приведен результат работы программы (обратите внимание, что словарь выводится в упорядоченном виде):

```
Petya K. 1001002
Ivanova N.M. 3563398
Vovochka 1180316
Vasia 2334476
m1:
Ivanova N.M. 3563398
Petya K. 1001002
Petya P. 2134622
Vasia 2334476
Vovochka 1180316
Второй элемент: Petya K. 1001002
Vasia: 2334476
```

Для поиска элементов в словаре определены следующие функции:

iterator	find(const key_type& x);
const_iterator	find(const key_type& x) const;
iterator	lower_bound(const key_type& x);
const_iterator	lower_bound(const key_type& x) const;
iterator	upper_bound(const key_type& x);
const_iterator	upper_bound(const key_type &x) const;
size_type	count(const key_type& x) const;

Функция find возвращает итератор на найденный элемент в случае успешного поиска или end() в противном случае.

Функция upper_bound возвращает итератор на первый элемент, ключ которого не меньше x, или end(), если такого нет (если элемент с ключом x есть в словаре, будет возвращен итератор на него).

Функция lower_bound возвращает итератор на первый элемент, ключ которого больше x, или end(), если такого нет.

Добавим в приведенный выше пример операторы

```
getline(cin, str);
if (m1.find(str) != m1.end())
```

```

    cout << m1 [str];
else{
    cout << (*m1.upper_bound(str)).first << " " ;
    cout << (*m1.lower_bound(str)).first << " " ;
}

```

Если ввести с клавиатуры фамилию, которая есть в словаре, будет выведен соответствующий номер телефона, а иначе — два раза подряд первая из фамилий, которая по алфавиту следует за введенной, например:

```

Petya M. // Подчеркиванием обозначен ввод пользователя
Petya P. Petya P.

```

Функция `count` возвращает количество элементов, ключ которых равен `x` (таких элементов может быть 0 или 1).

Для вставки и удаления элементов определены функции:

```

pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
template <class InputIter>
void insert(InputIter first, InputIter last);
void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);
void clear();

```

Первая форма функции используется для вставки в словарь пары «ключ—значение». Функция возвращает пару, состоящую из итератора, указывающего на вставленное значение, и булевого признака результата операции: `true`, если записи с таким ключом в словаре не было (только в этом случае происходит добавление), и `false` в противном случае (итератор указывает на существующую запись). Время работы функции пропорционально логарифму количества элементов в словаре.

Таким образом, скорректировать существующую запись, используя функцию вставки, нельзя. Это делается с помощью операции доступа по индексу.

Ниже приведено несколько примеров вставки в словарь, тип которого описан в предыдущем листинге:

```

map_s1 m2; // Создание пустого словаря
m2.insert(map_s1::value_type("Lena", 3157725));
str = "Anna";
num = 5536590;
m2.insert(make_pair(str, num));
num = 5530000;
// Попытка вставки существующей записи:
m2.insert(make_pair(str, num));
i = m1.begin();
m2.insert(*i); // Вставка в m2 первого элемента словаря m1
m2["Lena"] = 2222222; // Корректировка элемента
for (i = m2.begin(); i != m2.end(); i++) // Вывод словаря
    cout << (*i).first << " " << (*i).second << endl;

```


Результат работы программы:

```
Anna 5536590
Ivanova N.M. 3563398
Lena 2222222
```

Вторая форма функции `insert` применяется для ускорения процесса вставки. С этой целью ей передается первым параметром позиция словаря, начиная с которой требуется осуществлять поиск места вставки¹. Вставка выполняется только в случае отсутствия значения `x` в словаре. Функция возвращает итератор на элемент словаря с ключом, содержащимся в `x`.

Например, если известно, что элементы будут помещаться в словарь в порядке возрастания, можно передавать первым параметром в функцию вставки позицию предыдущего элемента (в этом случае время вставки является константой):

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
typedef map<string, long, less<string> > map_sl;
typedef pair<string, long> pair_sl;
int main(){
    pair_sl p[3] = { pair_sl("Anna", 123123),
                    pair_sl("Maria", 234234),
                    pair_sl("Teresa", 345345)};

    map_sl m1;
    map_sl::iterator i = m1.begin();
    for (int k = 0; k<3; k++)
        i = m1.insert(i, p[k]); // sic!
    for (i = m1.begin(); i != m1.end(); i++)
        cout << (*i).first << " " << (*i).second << endl;
    return 0;
}
```

Третья форма функции `insert` используется для вставки группы элементов, определяемой диапазоном итераторов. Функции удаления элементов и очистки словаря аналогичны одноименным функциям других контейнеров: первая форма функции `erase` удаляет элемент словаря из позиции, заданной итератором, вторая — по заданному ключу, а третья удаляет диапазон элементов.

Операции вставки в словарь не приводят к порче связанных с ними итераторов и ссылок, а операции удаления делают недействительными только итераторы и ссылки, связанные с удаляемыми элементами.

Для обмена всех элементов двух словарей применяется функция `swap`:

```
template <class Key, class T, class Compare>
void swap(map<Key, T, Compare>& x, map<Key, T, Compare>& y);
```

¹ Если указанная позиция находится после места, в которое требуется вставить элемент, вставка будет все равно выполнена верно.

Функция `equal_range` возвращает пару итераторов (`lower_bound(x)`, `upper_bound(x)`) для переданного ей значения `x`:

```
pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator>
    equal_range(const key_type& x) const;
```

После вызова функции оба итератора будут указывать на элемент с заданным ключом, если он присутствует в словаре, или на первый элемент, больший него, в противном случае.

Словари с дубликатами (multimap)

Как уже упоминалось, словари с дубликатами (`multimap`) допускают хранение элементов с одинаковыми ключами. Поэтому для них не определена операция доступа по индексу `[ ]`, а добавление с помощью функции `insert` выполняется успешно в любом случае. Функция возвращает итератор на вставленный элемент.

Элементы с одинаковыми ключами хранятся в словаре в порядке их занесения. При удалении элемента по ключу функция `erase` возвращает количество удаленных элементов. Функция `equal_range` возвращает диапазон итераторов, определяющий все вхождения элемента с заданным ключом. Функция `count` может вернуть значение, большее 1. В остальном словари с дубликатами аналогичны обычным словарям.

Множества (set)

Множество — это ассоциативный контейнер, содержащий только значения ключей, то есть тип `value_type` соответствует типу `Key`. Значения ключей должны быть уникальны. Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение «меньше»:

```
template <class Key, class Compare = less<Key> >
class set{
public:
    typedef Key          key_type;
    typedef Key          value_type;
    explicit set(const Compare& comp = Compare());
    template <class InputIter>
        set(InputIter first, InputIter last,
            const Compare& comp = Compare());
    set(const set<Key, Compare>& x);
    pair<iterator, bool> insert(const value_type& x);
    iterator insert(iterator position, const value_type& x);
    template <class InputIter>
        void insert(InputIter first, InputIter last);
    void          erase(iterator position);
    size_type     erase(const key_type& x);
    void          erase(iterator first, iterator last);
```

```

void swap(set<Key, Compare>&);
void clear();
iterator find(const key_type& x) const;
size_type count(const key_type& x) const;
iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x) const;
pair<iterator, iterator> equal_range(const key_type& x) const;

};

```

Из описания, приведенного с сокращениями, видно, что интерфейс множества аналогичен интерфейсу словаря. Ниже приведен простой пример, в котором создаются множества целых чисел:

```

#include <iostream>
#include <set>
using namespace std;
typedef set<int, less<int> > set_i;
set_i::iterator i;
int main(){
    int a[4] = {4, 2, 1, 2};
    set_i s1;           // Создается пустое множество
    set_i s2(a, a + 4); // Множество создается копированием массива
    set_i s3(s2);       // Работает конструктор копирования
    s2.insert(10);      // Вставка элементов
    s2.insert(6);
    for ( i = s2.begin(); i != s2.end(); i++) // Вывод
        cout << *i << " ";
    cout << endl;
    // Переменная для хранения результата equal_range:
    pair <set_i::iterator, set_i::iterator > p;
    p = s2.equal_range(2);
    cout << *(p.first) << " " << *(p.second) << endl;
    p = s2.equal_range(5);
    cout << *(p.first) << " " << *(p.second) << endl;
    return 0;
}

```

Результат работы программы:

```

1 2 4 6 10
2 4
6 6

```

Как и для словаря, элементы в множестве хранятся отсортированными. Повторяющиеся элементы в множество не заносятся.

Для работы с множествами в стандартной библиотеке определены алгоритмы, описанные на с. 364.

Множества с дубликатами (multiset)

Во множествах с дубликатами ключи могут повторяться, поэтому операция вставки элемента всегда выполняется успешно, и функция `insert` возвращает итератор на вставленный элемент. Элементы с одинаковыми ключами хранятся в словаре в порядке их занесения. Функция `find` возвращает итератор на первый найденный элемент или `end()`, если ни одного элемента с заданным ключом не найдено.

При работе с одинаковыми ключами в `multiset` часто пользуются функциями `count`, `lower_bound`, `upper_bound` и `equal_range`, имеющими тот же смысл, что и для словарей с дубликатами.

Битовые множества (bitset)

Битовое множество представляет собой шаблон для представления и обработки длинных последовательностей битов¹. Фактически `bitset` — это битовый массив, для которого обеспечиваются операции произвольного доступа, изменения отдельных битов и всего массива. Биты нумеруются справа налево, начиная с 0.

Шаблон битового множества определен в заголовочном файле `<bitset>`. Параметром шаблона является длина битовой последовательности, которая должна быть константой:

```
template<size_t N> class bitset { ... };
```

Для адресации отдельного бита в `bitset` введен класс `reference`:

```
class reference {
    friend class bitset;
    reference();
public:
    ~reference();
    reference& operator=(bool x);           // для b[i] = x;
    reference& operator=(const reference&); // для b[i] = b[j];
    bool operator~() const;                 // инверсия b[i]
    operator bool() const;                  // для x = b[i];
    reference& flip();                      // для инверсии b[i];
};
```

Конструкторы позволяют создать битовое множество из всех нулей, из значения типа `long` или из строки типа `string`²:

```
bitset();           // 1
bitset(unsigned long val); // 2
```

¹ Может оказаться, что в зависимости от реализации битовые последовательности, для размещения которых недостаточно переменной типа `int`, но достаточно `long`, более эффективно обрабатывать с помощью битовых операций над целыми числами (см. 31). Короткие последовательности, уместяющиеся в одном слове, могут обрабатываться с помощью битового множества более эффективно.

² Описание конструкторов приведено с сокращениями.

```
explicit bitset(const string& str,      // 3
               string::size_type pos = 0,
               string::size_type n = string::npos);
```

Первый конструктор создает битовое множество из нулей, второй принимает значение типа `long` и инициализирует каждый бит множества соответствующим битом внутреннего представления этого значения. Третий конструктор принимает строку, которая должна состоять из нулей и единиц (если это не так, порождается исключение `invalid_argument`) и инициализирует каждый бит множества в соответствии со значением символа строки. Второй и третий параметры конструктора задают позиции начала строки и количества символов, которые используются для инициализации. По умолчанию используется вся строка.

Примеры создания битовых множеств:

```
bitset<100> b1;                // сто нулей
bitset<16> b2 (0xf0f);          // 0000111100001111
bitset<16> b3 ("0000111100001111"); // 0000111100001111
bitset<5> b4 ("00110011", 3);   // 10011
bitset<3> b5 ("00110101", 1, 3); // 011
```

С битовыми множествами можно выполнять следующие операции:

```
bool operator==(const bitset<N>& rhs) const;
bool operator!=(const bitset<N>& rhs) const;
bitset<N>& operator&=(const bitset<N>& rhs);
bitset<N>& operator|=(const bitset<N>& rhs);
bitset<N>& operator^=(const bitset<N>& rhs);
bitset<N> operator<<(size_t pos) const;
bitset<N> operator>>(size_t pos) const;
bitset<N>& operator<<=(size_t pos);
bitset<N>& operator>>=(size_t pos);
bitset<N>& set();
bitset<N>& set(size_t pos, int val = true);
bitset<N>& reset();
bitset<N>& reset(size_t pos);
bitset<N> operator~() const;
bitset<N>& flip();
bitset<N>& flip(size_t pos);
reference operator[](size_t pos);    // b[i];
```

Множества можно сравнивать на равенство (`=`) и неравенство (`!=`). Операции `<<` и `>>` создают битовые наборы, сдвинутые на `pos` бит влево или вправо соответственно. При сдвиге освобождающиеся позиции заполняются нулями. Операция `set` устанавливает все биты множества в 1, `reset` — в 0. Операция `~` создает дополнительный набор. С помощью `flip` можно инвертировать значение каждого бита или бита, заданного параметром `pos`.

Доступ к отдельному биту можно выполнять с помощью операции индексации. Если значение индекса выходит за границы набора, порождается исключение `out_of_range`.

```
explicit bitset(const string& str,      // 3
               string::size_type pos = 0,
               string::size_type n = string::npos);
```

Первый конструктор создает битовое множество из нулей, второй принимает значение типа `long` и инициализирует каждый бит множества соответствующим битом внутреннего представления этого значения. Третий конструктор принимает строку, которая должна состоять из нулей и единиц (если это не так, порождается исключение `invalid_argument`) и инициализирует каждый бит множества в соответствии со значением символа строки. Вторым и третьим параметрами конструктора задают позиции начала строки и количества символов, которые используются для инициализации. По умолчанию используется вся строка.

Примеры создания битовых множеств:

```
bitset<100> b1;                // сто нулей
bitset<16> b2 (0xf0f);          // 0000111100001111
bitset<16> b3 ("0000111100001111"); // 0000111100001111
bitset<5> b4 ("00110011", 3);   // 10011
bitset<3> b5 ("00110101", 1, 3); // 011
```

С битовыми множествами можно выполнять следующие операции:

```
bool operator==(const bitset<N>& rhs) const;
bool operator!=(const bitset<N>& rhs) const;
bitset<N>& operator&=(const bitset<N>& rhs);
bitset<N>& operator|=(const bitset<N>& rhs);
bitset<N>& operator^=(const bitset<N>& rhs);
bitset<N> operator<<(size_t pos) const;
bitset<N> operator>>(size_t pos) const;
bitset<N>& operator<<=(size_t pos);
bitset<N>& operator>>=(size_t pos);
bitset<N>& set();
bitset<N>& set(size_t pos, int val = true);
bitset<N>& reset();
bitset<N>& reset(size_t pos);
bitset<N> operator~() const;
bitset<N>& flip();
bitset<N>& flip(size_t pos);
reference operator[](size_t pos); // b[i];
```

Множества можно сравнивать на равенство (`==`) и неравенство (`!=`). Операции `<<` и `>>` создают битовые наборы, сдвинутые на `pos` бит влево или вправо соответственно. При сдвиге освобождающиеся позиции заполняются нулями. Операция `set` устанавливает все биты множества в 1, `reset` — в 0. Операция `~` создает дополнительный набор. С помощью `flip` можно инвертировать значение каждого бита или бита, заданного параметром `pos`.

Доступ к отдельному биту можно выполнять с помощью операции индексации. Если значение индекса выходит за границы набора, порождается исключение `out_of_range`.

В шаблоне `bitset` определены методы преобразования в длинное целое и в строку, а также анализа значений множества:

```
unsigned long to_ulong() const:    // в unsigned long
string to_string() const:         // в string
size_t count() const:             // количество битовых 1
size_t size() const:              // количество битов
bool test(size_t pos) const:      // true, если b[pos] == 1
bool any() const:                 // true, если хотя бы один бит равен 1
bool none() const:                // true, если ни один бит не равен 1
```

Определены также обычные операции **ввода и вывода** `<<` и `>>`. Биты множества выводятся с помощью символов '0' и '1' слева направо, самый старший бит слева.

В битовом множестве не определены итераторы, поэтому оно не является контейнером в чистом виде, поскольку не полностью обеспечивает стандартный интерфейс контейнеров.

Пример использования контейнеров

Приведенная ниже программа формирует для заданного текстового файла указатель, то есть упорядоченный по алфавиту список встречающихся в нем слов, для каждого из которых показаны номера строк, содержащих это слово. Если слово встречается в строке более одного раза, номер строки выводится один раз.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <set>
#include <map>
using namespace std;
typedef set<int, less<int> > set_i;
typedef map<string, set_i, less<string> > map_ss;
bool wordread(ifstream &in, string &word, int &num){
    char ch;
    // Пропуск до первой буквы:
    for (;;){
        in.get(ch);
        if (in.fail()) return false;
        if (isalpha(ch) || ch == '_') break;
        if (ch == '\n') num++;
    }
    word = "";
    // Поиск конца слова:
    do{
        word += tolower(ch);
        in.get(ch);
    }while (!in.fail() && (isalpha(ch) || ch == '_'));
    if (in.fail()) return false;
```

```

    in.putback(ch); // Если символ - '\n'
    return true;
}
int main(){
    map_ss m;
    map_ss::iterator im;
    set_i::iterator is, isbegin, isend;
    string word;
    int num = 1;
    ifstream in ("some_file");
    if (!in){cout << "Cannot open input file.\n"; exit(1);
    }
    while (wordread(in, word, num)){
        im = m.find(word);
        if (im == m.end())
            im = m.insert(map_ss::value_type(word, set_i())).first;
        (*im).second.insert(num);
    }
    for (im = m.begin(); im != m.end(); im++){
        cout << setiosflags(ios::left) << setw(15) << (*im).first.c_str();
        isbegin = (*im).second.begin();
        isend = (*im).second.end();
        for (is = isbegin; is != isend; is++)
            cout << " " << *is;
        cout << endl;
    }
    return 0;
}

```

Допустим, входной файл `some_file` содержит следующий текст:

```

class value_compare:
public binary_function <value_type, value_type, bool> {
friend class map; protected:
Compare comp;
value_compare(Compare c) : comp(c) {} public:
bool operator() (const value_type& x, const value_type& y) const {
    return comp(x.first, y.first);}
};

```

В этом случае программа выведет на экран:

```

binary_function 2
bool            2 6
c               5
class           1 3
comp            4 5 8
compare         4 5
const           7
first           8

```

friend	3
map	3
operator	6
protected	3
public	2 5
return	8
value_compare	1 5
value_type	2 7
x	7 8
y	7 8

Рассмотрим работу программы подробнее. Функция `wordread` считывает очередное слово из входного файла, считая, что в слово могут входить алфавитно-цифровые символы и знак подчеркивания, а также формирует номер текущей строки.

Создаваемый указатель хранится в словаре, состоящем из пар «строка – множество». Строка является ключом, в ней содержится отдельное слово из входного файла. Множество целых чисел хранит номера строк, в которых встречается данное слово. И множество, и словарь без нашего участия обеспечивают быстрый поиск и упорядоченное хранение элементов без дубликатов.

Для каждого слова с помощью функции `find` проверяется, содержится ли оно в словаре. Если нет (функция возвратила `end()`), в словарь с помощью функции `insert` добавляется пара, состоящая из этого слова и пустого множества (вызов конструктора `set_i()`). Функция `insert` возвращает пару, первым элементом которой (`first`) является итератор на вставленный элемент. Он присваивается переменной `im`.

В следующем операторе происходит добавление с помощью функции `insert` нового элемента в множество, которое является вторым (`second`) элементом словаря.

Далее выполняется вывод словаря. Для каждого слова в цикле выводятся элементы множества номеров строк.

ГЛАВА 13

Итераторы и функциональные объекты

Итераторы и функциональные объекты широко используются в стандартной библиотеке. Понятие итератора было введено в предыдущем разделе при рассмотрении контейнерных классов, с функциональными объектами мы встречались при рассмотрении очередей с приоритетами. Рассмотрим их более подробно для того, чтобы применять в алгоритмах библиотеки, которые описываются далее в разделе «Алгоритмы» (с. 343).

Итераторы

Представим себе данные как некую абстрактную последовательность. Вне зависимости от способа ее организации и типа данных нам требуются средства последовательного просмотра последовательности и доступа к каждому ее элементу. Итератор обеспечивает эти средства просмотра.

Итератор — это обобщение понятия указателя для работы с различными структурами данных стандартным способом. Для того, чтобы можно было реализовать алгоритмы, корректно и эффективно работающие с данными различной структуры, стандарт определяет не только интерфейс, но и требования ко времени доступа с помощью итераторов.

Поскольку итератор является обобщением понятия «указатель», семантика у них одинаковая, и все функции, принимающие в качестве параметра итераторы, могут также работать и с обычными указателями.

В стандартной библиотеке итераторы используются для работы с контейнерными классами, потоками и буферами потоков.

В итераторах используются понятия «текущий указываемый элемент» и «указать на следующий элемент». Доступ к текущему элементу последовательности выполняется аналогично указателям с помощью операций * и ->. Переход к сле-

его элемент является *1* бег. При этом адресация отдельная беглая выполняющая функцию. Поэтому только автор не может быть указан. В остальных случаях авторская функция является функцией и реализуется при же авторской функции. В дальнейшем в этом направлении работы по лингвистике беглая и авторская в тексте [11-12].

Следует на основе авторской функции рассмотреть реализацию в авторской функции. Авторская функция является функцией и реализуется в авторской функции.

author reference

authorial reference

authorial

author

authorial

authorial (authorial)

authorial (authorial) (authorial)

authorial (authorial) (authorial)

authorial (authorial)

Примеры 1 и 2 являются частными случаями более общего результата, который будет доказан в § 3.

```

#include <vector>
#include <iostream>
using namespace std;
vector<double> v(10);
int main()
{
    for(int i = 0; i < v.size(); i++) v[i] = i;
    for (vector<double>::const_iterator i = v.begin(); i < v.end(); i++)
        cout << *i << " ";
}

```

Двусторонние очереди (deque)

Другой важная особенность — это исключительная малость количества, которую в настоящее время, по крайней мере, можно считать достаточной для осуществления исследований. То же относится к количеству объектов, которые могут быть использованы в качестве объектов исследования. Таким образом, исследование в области информатики имеет следующие особенности:

[illegible]

дующему элементу — с помощью операции инкремента `++`. Для всех итераторов определены также присваивание, проверка на равенство и неравенство.

Данные могут быть организованы различным образом — например, в виде массива, списка, вектора или дерева. Для каждого вида последовательности требуется свой тип итератора, поддерживающий различные наборы операций. В соответствии с набором обеспечиваемых операций *итераторы делятся на пять категорий*, описанных в табл. 13.1. Приведенные в таблице операции обеспечиваются за постоянное время.

Пусть `i` и `j` — итераторы одного вида, `x` — переменная того же типа, что и элемент последовательности, `n` — целая величина. Тогда допустимы выражения:

`i++` `++i` `i = j` `i - j` `i != j`

Таблица 13.1. Категории итераторов

Категория итератора	Операции	Контейнеры
входной (input)	<code>x = *i</code>	все
выходной (output)	<code>*i = x</code>	все
прямой (forward)	<code>x = *i</code> , <code>*i = x</code>	все
двунаправленный (bidirectional)	<code>x = *i</code> , <code>*i = x</code> , <code>--i</code> , <code>i--</code>	все
произвольного доступа (random access)	<code>x = *i</code> , <code>*i = x</code> , <code>--i</code> , <code>i--</code> <code>i + n</code> , <code>i - n</code> , <code>i += n</code> , <code>i -= n</code> <code>i < j</code> , <code>i > j</code> , <code>i <= j</code> , <code>i >= j</code>	все, кроме <code>list</code>

Как видно из таблицы, *прямой* итератор поддерживает все операции входных и выходных итераторов и может использоваться везде, где требуются входные или выходные итераторы. *Двунаправленный* итератор поддерживает все операции прямого, а также декремента, и может использоваться везде, где требуется прямой итератор. Итератор *произвольного доступа* поддерживает все операции двунаправленного, а кроме того, переход к произвольному элементу последовательности и сравнение итераторов.

Можно сказать, что итераторы образуют иерархию, на верхнем уровне которой находятся итераторы произвольного доступа. Чем выше уровень итератора, тем более высокие функциональные требования предъявляются к контейнеру, для которого используется итератор. Например, для списков итераторами произвольного доступа пользоваться нельзя, поскольку список не поддерживает требуемый набор операций итератора.

Итераторные классы и функции описаны в заголовочном файле `<iterator>`. При использовании стандартных контейнеров этот файл подключается автоматически.

Итераторы могут быть константными. Константные итераторы используются тогда, когда изменять значения соответствующих элементов контейнера нет необходимости.

Итератор может быть *действительным* (когда он указывает на какой-либо элемент) или *недействительным*. Итератор может стать недействительным в следующих случаях:

- ☐ итератор не был инициализирован;
- ☐ контейнер, с которым он связан, изменил размеры или уничтожен;
- ☐ итератор указывает на конец последовательности.

Конец последовательности представляется как указатель на элемент, следующий за последним элементом последовательности. Этот указатель всегда существует. Такой подход позволяет не рассматривать пустую последовательность как особый случай. Понятия «нулевой итератор» не существует.

Поскольку итераторы используются для работы с объектами, на которые они указывают (например, для получения значения элемента контейнера), требуется определить соответствующие типы. Для этого в заголовочном файле `<iterator>` определен шаблон `iterator_traits` (trait в переводе с английского — характерная черта).

```
template<class Iter> struct iterator_traits{
    typedef typename Iter::difference_type difference_type;
    typedef typename Iter::value_type value_type;
    typedef typename Iter::pointer pointer;
    typedef typename Iter::reference reference;
    typedef typename Iter::iterator_category iterator_category;
};
```

Ключевое слово `typename` необходимо для того, чтобы компилятор мог опознать `Iter` как имя типа. `iterator_category` — это тип итератора, который определяет, какие он поддерживает операции. Тип `difference_type` служит для выражения разности между двумя итераторами.

В заголовочном файле описаны следующие типы итераторов:

```
struct input_iterator_tag{};
struct output_iterator_tag{};
struct forward_iterator_tag: public input_iterator_tag{};
struct bidirectional_iterator_tag:
    public forward_iterator_tag{};
struct random_access_iterator_tag:
    public bidirectional_iterator_tag{};
```

Тип итератора можно указать в списке параметров шаблона функции, использующей итератор, для того чтобы на этапе компиляции можно было выбрать тип итератора, обеспечивающий наиболее эффективную реализацию, поскольку один и тот же алгоритм для различных видов итераторов может быть реализован с разной эффективностью.

Для обычных указателей в целях повышения эффективности определены специализации шаблона `iterator_traits` (о специализациях шаблонов рассказывалось на с. 220):

```
template<class T> struct iterator_traits<T*>{
    typedef ptrdiff_t difference_type;
```

```

typedef T value_type;
typedef T* pointer;
typedef T& reference;
typedef random_access_iterator_tag iterator_category;
};

```

Разность между двумя указателями имеет стандартный тип `ptrdiff_t`, определенный в `<ptrdiff.h>`. Аналогичная специализация шаблона введена и для типа `const T*`.

Поскольку только итераторы произвольного доступа поддерживают операции `+` и `-`, в библиотеке описаны функции `distance` и `advance`. Функция `distance` служит для определения расстояния между элементами контейнера:

```
distance(InputIterator first, InputIterator last);
```

Она возвращает значение типа `difference_type`, представляющее собой разность между двумя итераторами. Эта функция для всех итераторов, кроме итераторов произвольного доступа, выполняется с использованием операции `++` за время, пропорциональное расстоянию между элементами контейнера, то есть довольно ресурсоемка. Аналогично работает и функция `advance`, которая применяется для реализации операции `i += n`:

```
void advance(InputIterator& i, Distance n);
```

Величина `n` может быть отрицательной только для двунаправленных итераторов и итераторов произвольного доступа.

Обратные итераторы

Для двунаправленных итераторов и итераторов произвольного доступа определены разновидности, называемые *адаптерами итераторов*. Адаптер, просматривающий последовательность в обратном направлении, называется `reverse_iterator`:

```

template <class It>
class reverse_iterator : public
    iterator<typename iterator_traits<It>::iterator_category,
    typename iterator_traits<It>::value_type,
    typename iterator_traits<It>::difference_type,
    typename iterator_traits<It>::pointer,
    typename iterator_traits<It>::reference>{
protected:
    It current;
public:
    typedef It      iterator_type;
    reverse_iterator();
    explicit reverse_iterator(It x);
    template <class U> reverse_iterator(const reverse_iterator<U>& u);
    It base() const;
    reference operator*() const;
    pointer operator->() const;
};

```

```
reverse_iterator& operator++();
reverse_iterator operator++(int);
reverse_iterator& operator--();
reverse_iterator operator--(int);
reverse_iterator operator+ (difference_type n) const;
reverse_iterator& operator+=(difference_type n);
reverse_iterator operator- (difference_type n) const;
reverse_iterator& operator--(difference_type n);
reference operator[](difference_type n) const;
```

Обратный итератор имеет защищенное поле данных, называемое текущим итератором — `current`. Операция инкремента реализуется путем декремента этого итератора:

```
template <class It>
reverse_iterator<It>& reverse_iterator<It> operator++(){
    --current;
    return *this;
}
```

Итератор может указывать на все элементы контейнера, включая следующий за последним, но для обратного итератора таким элементом будет элемент, стоящий перед первым, а его не существует. Поэтому `current` на самом деле указывает на элемент, следующий за тем, на который указывает обратный итератор. Соотношение между прямым и обратным итератором выражается следующим образом:

```
&*(reverse_iterator(i)) == &(i - 1).
```

Для обратных итераторов поддерживаются обычные операции отношения `==`, `!=`, `<`, `<=`, и `=`.

Обратные итераторы описаны в контейнерных классах для просмотра их элементов в обратном порядке. Кроме того, описаны методы `rbegin()` и `rend()`, возвращающие `reverse_iterator`. Например, чтобы просмотреть вектор в обратном порядке, можно воспользоваться следующим циклом:

```
vector<int> v;

for (vector<int> reverse_iterator i = v.rbegin(); i != v.rend; ++i)
    cout << *i << " ";
```

Если контейнер объявлен как `const` (например, в списке передаваемых функции параметров), то требуется использовать итератор с префиксом `const` — `const_reverse_iterator`.

Итераторы вставки

Итераторы вставки так же, как и обратные, являются адаптерами итераторов. Они предназначены для добавления новых элементов в начало, конец или произвольное место контейнера. В стандартной библиотеке определено три шаблона классов итераторов вставки, построенных на основе выходных итераторов: `back_insert_iterator`, `front_insert_iterator` и `insert_iterator`.

Кроме того, определены три функции вставки:

```
template <class C> back_insert_iterator<C>
    back_inserter(C& x);
template <class C> front_insert_iterator<C>
    front_inserter(C& x);
template <class C, class Iter> insert_iterator<C>
    inserter(C& x, Iter i);
```

Здесь *C* — контейнер, в который требуется вставить элементы. Функция *back_inserter* вставляет элементы в конец контейнера, *front_inserter* — в начало, а *inserter* — перед элементом, на который ссылается ее аргумент-итератор. Эти функции возвращают итераторы соответствующего типа и часто используются в качестве аргументов стандартных алгоритмов (см. 355), где определяют режим работы — вставка или замена элементов контейнера.

Потоковые итераторы

Потоковые итераторы введены для того чтобы стандартные алгоритмы, которые рассматриваются в следующем разделе, могли непосредственно использовать потоки ввода/вывода. Потоки представляются в виде последовательностей. Определено два шаблона классов потоковых итераторов: итератор входного потока *istream_iterator* и итератор выходного потока *ostream_iterator*.

Итератор входного потока читает элементы из потока, для которого он был создан, после чего к ним можно обращаться обычным для итератора образом, то есть через операцию разадресации. Например, для чтения целого числа из файла с именем *temp* можно использовать следующий фрагмент:

```
istream in("temp");
istream_iterator<int> i(in);
int buf = *i;
```

Очередное значение из входного потока считывается при выполнении операции инкремента, например:

```
++i;
int buf1 = *i;
```

При достижении конца входного потока итератор принимает значение конца ввода. Это же значение имеет конструктор итератора по умолчанию, поэтому цикл чтения из файла можно организовать следующим образом:

```
while ( i != istream_iterator<int> () )
    cout << *i++ << " ";
```

Для итераторов входного потока определены операции сравнения на равенство и неравенство. Все итераторы, равные концу ввода, равны между собой. Сравнить не равные концу ввода итераторы можно при условии, что они сконструированы для одного и того же потока.

Особенность итераторов входного потока заключается в том, что они не сохраняют равенство после инкрементации, то есть если *i == j*, то не обязательно, что *++i == ++j*. Поэтому их рекомендуется использовать только в однопроходных ал-

горитмах. Если алгоритм требует нескольких проходов, предпочтительнее использовать структуры данных в оперативной памяти.

Итератор выходного потока записывает с помощью операции << элементы в выходной поток, для которого он был сконструирован. Если вторым аргументом конструктора была строка символов, она выводится после каждого выводимого значения:

```
ostream_iterator<int> os(cout, " кг");
*os = 100;           // Будет выведено: 100 кг
++os; *os = 2;       // Будет выведено: 2 кг
```

Рассмотрим реализацию операции присваивания:

```
ostream_iterator& operator=(const T& value){
    *out_stream << value;
    if(delim != 0) *out_stream << delim;
    return (*this);
}
```

Здесь `out_stream` — ссылка на выходной поток, заданный первым параметром конструктора итератора, `delim` — строка, указанная во втором параметре конструктора.

Пример применения потоковых итераторов вместе с алгоритмами библиотеки приведен на с. 350.

Функциональные объекты

Функциональным объектом называется класс, в котором определена операция вызова функции (см. с. 195). Мы уже встречались с функциональными объектами при рассмотрении очередей с приоритетами (с. 313). Чаще всего эти объекты используются в качестве параметров стандартных алгоритмов для задания пользовательских критериев сравнения объектов или способов их обработки.

В тех алгоритмах, где в качестве параметра можно использовать функциональный объект, можно использовать и указатель на функцию. При этом применение функционального объекта может оказаться более эффективным, поскольку операцию `()` можно определить как встроенную.

Стандартная библиотека предоставляет множество функциональных объектов, необходимых для ее эффективного использования и расширения. Они описаны в заголовочном файле `<functional>`. Среди этих объектов можно выделить объекты, возвращающие значения типа `bool`. Такие объекты называются *предикатами*. Предикатом называется также и обычная функция, возвращающая `bool`.

В качестве **базовых классов**, которые вводят стандартные имена для типов аргументов, в библиотеке определены **шаблоны унарной и бинарной функции**:

```
template <class Arg, class Result>
struct unary_function{
    typedef Arg    argument_type;
    typedef Result result_type;
};
```



```
template <class Arg1, class Arg2, class Result>
struct binary_function{
    typedef Arg1    first_argument_type;
    typedef Arg2    second_argument_type;
    typedef Result  result_type;
};
```

Функциональные объекты библиотеки являются потомками этих базовых объектов-функций. Определение стандартных имен типов для аргументов и результата необходимо для того, чтобы функциональные объекты можно было использовать совместно с адаптерами и другими средствами библиотеки.

Адаптером функции называют функцию, которая получает в качестве аргумента функцию и конструирует из нее другую функцию. На месте функции может быть также функциональный объект.

Стандартная библиотека содержит описание нескольких типов адаптеров:

- ❑ *связыватели* для использования функционального объекта с двумя аргументами как объекта с одним аргументом;
- ❑ *отрицатели* для инверсии значения предиката;
- ❑ адаптеры указателей на функцию;
- ❑ *адаптеры методов* для использования методов в алгоритмах.

Синтаксис и применение перечисленных адаптеров рассматриваются в следующих разделах.

Арифметические функциональные объекты

В стандартной библиотеке определены шаблоны функциональных объектов для всех арифметических операций, определенных в языке C++.

Таблица 13.2. Арифметические функциональные объекты

Имя	Тип	Результат
plus	бинарный	$x + y$
minus	бинарный	$x - y$
multiplies	бинарный	$x * y$
divides	бинарный	x / y
modulus	бинарный	$x \% y$
negate	унарный	$-x$

Ниже приведен шаблон объекта plus (остальные объекты описаны аналогичным образом):

```
template <class T> struct plus : binary_function<T, T, T>{
    T operator()(const T& x, const T& y) const{
        return x + y;
    }
};
```

Поскольку функциональные объекты используются большей частью вместе с алгоритмами стандартной библиотеки, примеры применения арифметических функциональных объектов приведены в разделе «Алгоритмы» на с. 356, 370.

Предикаты

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения и логических операций, определенных в языке C++. Они возвращают значение типа `bool`, то есть являются предикатами.

Таблица 13.3. Предикаты стандартной библиотеки

Имя	Тип	Результат
<code>equal_to</code>	бинарный	$x == y$
<code>not_equal_to</code>	бинарный	$x != y$
<code>greater</code>	бинарный	$x > y$
<code>less</code>	бинарный	$x < y$
<code>greater_equal</code>	бинарный	$x \geq y$
<code>less_equal</code>	бинарный	$x \leq y$
<code>logical_and</code>	бинарный	$x \&\& y$
<code>logical_or</code>	бинарный	$x \ \ y$
<code>logical_not</code>	унарный	$!\ x$

Ниже приведен шаблон объекта `equal_to` (остальные объекты описаны аналогичным образом):

```
template <class T> struct equal_to :
    binary_function<T, T, bool>{
    bool operator()(const T& x, const T& y) const{
        return x == y;
    }
};
```

Программист может описать собственные предикаты для определения критериев сравнения объектов. Это необходимо, когда контейнер состоит из элементов пользовательского типа. Например, для описанного на с. 183 класса `monstr` предикат, сравнивающий два объекта по значению поля `ammo`, имеет вид:

```
struct monstr_less_ammo:
    public binary_function<monstr, monstr, bool>{
    bool operator()(monstr & m1, monstr & m2){
        return m1.get_ammo() < m2.get_ammo();
    }
};
```

Примеры применения предикатов приведены на с. 338, 360, 363.

Отрицатели

Отрицатели `not1` и `not2` применяются для получения противоположного унарного и бинарного предиката соответственно. Использовать отрицатели очень просто. Например, для того чтобы получить инверсию предиката `less<int>()`, нужно записать выражение `not2(less<int>())`. Оно эквивалентно `greater_equal<int>`.

Отрицатели применяются для инвертирования предикатов, заданных пользователем, поскольку для стандартных предикатов библиотека содержит соответствующие им противоположные объекты.

Рассмотрим реализацию отрицателей. В библиотеке описаны функциональные объекты `unary_negate` и `binary_negate`, являющиеся потомками `unary_function` и `binary_function` соответственно. Эти классы принимают в качестве параметра конструктора объект, являющийся предикатом требуемого типа:

```
template <class Predicate>
class unary_negate: public
    unary_function<typename Predicate::argument_type, bool>{
public:
    explicit unary_negate(const Predicate& pred):
        bool operator()
        (const typename Predicate::argument_type& x) const;
};
```

Операция `()` возвращает инверсию предиката. `not1` и `not2` определены как шаблоны функций, принимающих предикат и возвращающих функциональный объект:

```
template <class Predicate>
    unary_negate<Predicate> not1(const Predicate& pred);
template <class Predicate>
    binary_negate<Predicate> not2(const Predicate& pred);
```

Связыватели

С помощью бинарных предикатов можно сравнивать два различных объекта. Часто требуется сравнить объект не с другим объектом, а с константой. Чтобы использовать для этого тот же самый предикат, требуется связать один из двух его аргументов с константой. Для этого используются связыватели `bind2nd` и `bind1st`, позволяющие связать с конкретным значением соответственно второй и первый аргумент бинарной функции.

Связыватели реализованы в стандартной библиотеке как шаблоны функций, принимающих первым параметром функциональный объект `f` с двумя аргументами, а вторым — привязываемое значение `value`. Результатом вызова функции является функциональный объект, созданный из входного объекта `f` путем «подстановки» `value` в его первый или второй аргумент.

Для описания типа возвращаемого функционального объекта в библиотеке описаны шаблоны классов `binder2nd` и `binder1st`. Ниже приведены шаблоны связывателей:

```
template <class Op, class T>
    binder2nd<Op> bind2nd(const Op& op, const T& x);
```

```
template <class Op, class T>
    binder1st<Op> bind1st(const Op& op, const T& x);
```

Здесь `Op` — тип функционального объекта, `T` — тип привязываемого значения.

Допустим, требуется вычислить количество элементов целочисленного массива, меньших 30:

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
int main(){
    int m[8] = {45, 65, 36, 25, 674, 2, 13, 35};
    cout << count_if(m, m + 8, bind2nd(less<int>(), 40));
    return 0;
}
```

В этой программе для подсчета количества элементов применяется алгоритм стандартной библиотеки `count_if` (см. с. 345). Для его использования требуется подключить заголовочный файл `<functional>`. Первыми двумя параметрами `count_if` должны быть итераторы, определяющие начало и конец обрабатываемой последовательности. В качестве итераторов массива используются указатели на его элементы. Третий параметр функции `count_if` должен быть бинарной функцией или функциональным объектом. Функция `bind2nd` превращает условие сравнения $x < y$ в условие $x < 40$.

Адаптеры указателей на функции

Для того чтобы применять связыватели к обычным указателям на функции, требуются специальные преобразователи, или адаптеры. Стандартная библиотека определяет два функциональных объекта — указатель на унарную функцию `pointer_to_unary_function` и указатель на бинарную функцию `pointer_to_binary_function`, а также две функции-адаптера `ptr_fun` с одним и двумя аргументами, которые преобразуют переданный им в качестве параметра указатель на функцию в функциональный объект.

Ниже приведено описание шаблона класса и функции для унарных и бинарных функций:

```
template <class Arg, class Result>
    class pointer_to_unary_function : public
        unary_function<Arg, Result>{
    public:
        explicit pointer_to_unary_function(Result (*f)(Arg));
        Result operator()(Arg x) const; // Возвращает f(x)
    };
template <class Arg, class Result>
    pointer_to_unary_function<Arg, Result>
        ptr_fun(Result (*f)(Arg));
template <class Arg1, class Arg2, class Result>
```

```

class pointer_to_binary_function : public
    binary_function<Arg1, Arg2, Result>{
public:
    explicit pointer_to_binary_function(Result (*f)(Arg1, Arg2));
    Result operator()(Arg1 x, Arg2 y) const;
};

template <class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1, Arg2, Result>
        ptr_fun(Result (*f)(Arg1, Arg2));

```

Пример применения адаптера функции:

```

#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
struct A{
    int x, y;
};
bool lss(A a1, A a2){return a1.x < a2.x;}
int main(){
    A ma[5] = {{2, 4}, {3, 1}, {2, 2}, {1, 2}, {1, 2}};
    A elem = {3, 0};
    cout << count_if(ma, ma + 5, bind2nd(ptr_fun(lss), elem));
    return 0;
}

```

Функция `lss` задает правило сравнения элементов структуры `A`. Алгоритм стандартной библиотеки `count_if` вычисляет количество элементов структуры `ma`, удовлетворяющих условию, заданному третьим параметром. Этим параметром является функциональный объект, созданный связывателем `bind2nd` из функционального объекта, полученного из функции `lss` с помощью адаптера `ptr_fun`, и переменной, подставляемой на место второго параметра функции. В результате будет вычислено количество элементов структуры `A`, поле `x` которых меньше 3 (результат — 4).

В качестве второго примера рассмотрим вектор из объектов класса `monstr`, введенного ранее (см. с. 183). Приведенная ниже программа с помощью алгоритма стандартной библиотеки `count_if` (см. с. 345) вычисляет в векторе количество монстров, здоровье которых пошатнулось (допустим, что этот печальный факт имеет место при значении поля `health < 30`). Для краткости приведены только необходимые методы класса:

```

#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>
using namespace std;
enum color {red, green, blue};
class monstr{
    int health, ammo;

```

```

    color skin;
    char *name;
public:
    monstr(int he = 100, int am = 10);
    monstr(color sk);
    monstr(char * nam);
    monstr(const monstr &M);
    ~monstr() {delete [] name;}
    operator int(){return health;}
    int get_health(){return health;}
    friend ostream&
        operator <<(ostream & out, monstr & m){
            return out << "monstr: " << " ammo = " << m.ammo
                << " health = " << m.health << endl;}
};

monstr::monstr(int he, int am):
    health(he), ammo(am), skin(red), name(0){}

monstr::monstr(const monstr &M){
    if (M.name){ name = new char [strlen(M.name) + 1];
        strcpy(name, M.name);}
    else name = 0;
    health = M.health; ammo = M.ammo; skin = M.skin;
}

monstr::monstr(color sk){
    switch (sk){
    case red :
        health = 1; ammo = 10; skin = red; name = 0; break;
    case green:
        health = 2; ammo = 20; skin = green; name = 0; break;
    case blue:
        health = 3; ammo = 40; skin = blue; name = 0; break;
    }}

monstr::monstr(char * nam){
    name = new char [strlen(nam) + 1]; strcpy(name, nam);
    health = 200; ammo = 10; skin = red;
}

bool less_health (monstr m1, monstr m2){
    return m1.get_health() < m2.get_health() ;}

int main(){
    vector <monstr> m;
    monstr M (10, 30);
    m.push_back(M);

```



```

m.push_back(monstr("vasia"));
m.push_back(monstr(red));
cout << " MONSTRY:" << endl;
for (int i = 0; i < m.size(); i++) cout << m[i] << " ";
cout << endl;
cout << " COUNT_IF: ";
cout << count_if(m.begin(), m.end(),
                 bind2nd(ptr_fun(less_health), 20));
return 0;
}

```

Булева функция `less_health` сравнивает поля `health` у двух переданных ей аргументов класса `monstr`. Чтобы сравнить поле `health` с константой, требуется связать с ней второй аргумент этой функции. Связыватель `bind2nd` должен получать в качестве первого аргумента функциональный объект, который формируется с помощью адаптера `ptr_fun`. Результат работы программы (больных монстров — 2 штуки):

```

MONSTRY:
monstr: ammo = 30 health = 10
monstr: ammo = 10 health = 200
monstr: ammo = 10 health = 1

```

```

COUNT_IF: 2

```

Адаптеры методов

При хранении в контейнерах объектов пользовательских классов данных часто возникает задача применить ко всем элементам контейнера один и тот же метод класса. Для просмотра элементов контейнера в библиотеке есть алгоритм `for_each`. В него можно передать указатель на функцию, которую требуется вызвать для каждого просматриваемого элемента контейнера, например:

```

#include <iostream>
#include <algorithm>
using namespace std;
void show(int a){ cout << a << endl;}
int main(){
    int m[4] = {3, 5, 9, 6};
    for_each(m, m + 4, show);
    return 0;
}

```

Если вместо обычного массива использовать контейнер, содержащий объекты некоторого класса, то записать в аналогичной программе на месте функции `show` вызов метода класса не удастся, поскольку метод требуется вызывать с указанием конкретного объекта класса.

Адаптеры методов позволяют использовать методы классов в качестве аргументов стандартных алгоритмов. Адаптер метода так же, как отрицатель и связыва-

тель, получает функцию и конструирует из нее другую функцию. В стандартной библиотеке определено несколько адаптеров для методов с различным числом аргументов:

Таблица 13.4. Адаптеры методов классов

Имя	Тип объекта	Действие
<code>mem_fun</code>	<code>mem_fun_t</code>	Вызывает безаргументный метод через указатель
<code>mem_fun</code>	<code>const_mem_fun_t</code>	Вызывает безаргументный константный метод через указатель
<code>mem_fun</code>	<code>mem_fun1_t</code>	Вызывает унарный метод через указатель
<code>mem_fun_ref</code>	<code>mem_fun_ref_t</code>	Вызывает безаргументный метод через ссылку
<code>mem_fun_ref</code>	<code>const_mem_fun_ref_t</code>	Вызывает безаргументный константный метод через ссылку
<code>mem_fun_ref</code>	<code>mem_fun1_ref_t</code>	Вызывает унарный метод через ссылку
<code>mem_fun_ref</code>	<code>const_mem_fun1_ref_t</code>	Вызывает унарный константный метод через ссылку
<code>mem_fun_ref</code>	<code>const_mem_fun1_t</code>	Вызывает унарный константный метод через указатель

В качестве примера опишем в классе `monstr` метод `dead`, определяющий, жив ли монстр:

```
bool dead(){return !health;}
```

Подсчитать с помощью уже упоминавшегося алгоритма `count_if` количество потерь в векторе, описанном, например, как

```
vector<monstr> ostrich(100);
```

можно с помощью оператора:

```
cout << count_if(ostrich.begin(), ostrich.end(),
    mem_fun_ref(&monstr::dead));
```

ГЛАВА 14

Алгоритмы

Алгоритмы STL предназначены для работы с контейнерами и другими последовательностями. Каждый алгоритм реализован в виде шаблона или набора шаблонов функции, поэтому может работать с различными видами последовательностей и данными разнообразных типов. Для настройки алгоритма на конкретные требования пользователя применяются функциональные объекты (см. с. 334).

Использование стандартных алгоритмов, как и других средств стандартной библиотеки, избавляет программиста от написания, отладки и документирования циклов обработки последовательностей, что уменьшает количество ошибок в программе, снижает время ее разработки и делает ее более читаемой и компактной.

Объявления стандартных алгоритмов находятся в заголовочном файле `<algorithm>`, стандартных функциональных объектов — в файле `<functional>`.

Все алгоритмы STL можно разделить на четыре категории:

- немодифицирующие операции с последовательностями;
- модифицирующие операции с последовательностями;
- алгоритмы, связанные с сортировкой;
- алгоритмы работы с множествами и пирамидами;

Кроме того, библиотека содержит обобщенные численные алгоритмы, объявления которых находятся в файле `<numeric>` (см. главу 15 «Средства для численных расчетов», с. 369).

В качестве параметров алгоритму передаются итераторы, определяющие начало и конец обрабатываемой последовательности. Вид итераторов определяет типы контейнеров, для которых может использоваться данный алгоритм. Например, алгоритм сортировки (`sort`) требует для своей работы итераторы произвольного доступа, поэтому он не будет работать с контейнером `list`. Алгоритмы не выполняют проверку выхода за пределы последовательности.

Таблицы, приведенные в начале следующих разделов, дают представление о возможностях стандартных алгоритмов STL. Далее приведено описание каждого алгоритма. Следует учитывать, что для последовательностей, содержащих пользовательские типы данных, можно задавать собственные критерии.

При описании параметров шаблонов алгоритмов используются следующие сокращения:

In — итератор для чтения;
 Out — итератор для записи;
 For — прямой итератор;
 Bi — двунаправленный итератор;
 Ran — итератор произвольного доступа;
 Pred — унарный предикат (условие);
 BinPred — бинарный предикат;
 Comp — функция сравнения;
 Op — унарная операция;
 BinOp — бинарная операция.

Немодифицирующие операции с последовательностями

Алгоритмы этой категории просматривают последовательность, не изменяя ее. Они используются для получения информации о последовательности или для определения положения элемента.

Таблица 14.1. Немодифицирующие операции с последовательностями

Алгоритм	Выполняемая функция
adjacent_find	Нахождение пары соседних значений
count	Подсчет количества вхождений значения в последовательность
count_if	Подсчет количества выполнений условия в последовательности
equal	Попарное равенство элементов двух последовательностей
find	Нахождение первого вхождения значения в последовательность
find_end	Нахождение последнего вхождения одной последовательности в другую
find_first_of	Нахождение первого значения из одной последовательности в другой
find_if	Нахождение первого соответствия условию в последовательности
for_each	Вызов функции для каждого элемента последовательности
mismatch	Нахождение первого несовпадающего элемента в двух последовательностях
search	Нахождение первого вхождения одной последовательности в другую
search_n	Нахождение n-го вхождения одной последовательности в другую

Рассмотрим эти алгоритмы подробнее.

adjacent_find

Алгоритм `adjacent_find` выполняет нахождение пары соседних значений.

```
template<class For> For adjacent_find(For first, For last);
template<class For, class BinPred>
    For adjacent_find(For first, For last, BinPred pred);
```

Первая форма алгоритма находит в последовательном контейнере пару соседних одинаковых значений и возвращает итератор на первое из них или конец последовательности (итератор на элемент, следующий за последним).

Вторая форма находит соседние элементы, удовлетворяющие условию, заданному предикатом `pred` в виде функции или функционального объекта.

Пример (программа находит самую левую пару одинаковых элементов целочисленного массива и пару элементов структуры, у которых равна сумма полей):

```
#include <algorithm>
using namespace std;
struct A{ int x, y;};
bool f(A &a1, A& a2){
    return a1.x + a1.y == a2.x + a2.y;}
int main (){
    int m[8] = {45, 60, 60, 25, 25, 2, 13, 35};
    cout << *(adjacent_find(m, m + 8));           // Вывод: 60
    A ma[5] = {{2,4}, {3,1}, {2,2}, {1,2}, {1,2}};
    cout << (*adjacent_find(ma, ma + 5, f)).x << endl; // Вывод:3
    return 0;
}
```

count, count_if

Алгоритм `count` выполняет подсчет количества вхождений значения в последовательность:

```
template<class In, class T>
    typename iterator_traits<In>::difference_type
    count(In first, In last, const T& value);
```

Эта форма алгоритма вычисляет в последовательном контейнере количество вхождений заданного значения `value`. Результат имеет тип разности между двумя итераторами `difference_type` (см. с. 330).

Алгоритм `count_if` выполняет подсчет количества выполнений условия в последовательности:

```
template<class In, class Pred>
    typename iterator_traits<In>::difference_type
    count_if(In first, In last, Pred pred);
```

Эта форма алгоритма вычисляет в последовательном контейнере количество элементов, удовлетворяющих условию, заданному предикатом `pred` в виде функции или функционального объекта.

Пример использования функции `count_if` приведен на с. 338.

equal

Алгоритм `equal` выполняет попарное сравнение элементов двух последовательностей. Пользователь может задать предикат, определяющий, что считать равенством:

```
template <class In1, class In2>
    bool equal(In1 first1, In1 last1, In2 first2);
template <class In1, class In2, class BinPred>
    bool equal(In1 first1, In1 last1, In2 first2,
               BinPred pred);
```

find, find_if

Алгоритмы семейства `find` осуществляют поиск в последовательности.

Алгоритм `find` выполняет поиск заданного значения `value`:

```
template<class In, class T>
    In find(In first, In last, const T& value);
```

Алгоритм `find_if` выполняет поиск значения, соответствующего заданному предикату `pred`:

```
template<class In, class Pred>
    In find_if(In first, In last, Pred pred);
```

Эти алгоритмы возвращают итератор на самое левое найденное значение в случае успешного поиска и на конец последовательности — в противном случае.

В качестве примера рассмотрим нахождение значений в векторе, элементы которого считываются из файла (в качестве предиката используется функциональный объект):

```
#include <fstream>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class In_10_50{
public:
    bool operator()(int x) {return x > 10 && x < 50;}
};
int main(){
    ifstream in ("inpnun");
    vector<int> v;
    int x;
    while ( in >> x, !in.eof()){
        v.push_back(x);}
    for (int i = 0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;
    // Поиск элемента, равного 51:
    cout << *find(v.begin(), v.end(), 51) << endl;
```



```
// Поиск элемента, удовлетворяющего условию 10<x<50:
cout << *find_if(v.begin(), v.end(), In_10_50()) << endl;
return 0;
}
```

Результат работы программы:

```
56 34 54 0 76 23 51 11 51 11 76 88
51
34
```

find_first_of

Алгоритм `find_first_of` находит первое вхождение в первую последовательность элемента из второй последовательности:

```
template<class For1, class For2>
For1 find_first_of(For1 first1, For1 last1,
                  For2 first2, For2 last2);
template<class For1, class For2, class BinPred>
For1 find_first_of(For1 first1, For1 last1,
                  For2 first2, For2 last2, BinPred pred);
```

Границы последовательностей задаются с помощью итераторов. Первая форма алгоритма ищет вхождение любого элемента, а вторая — элемента, для которого выполняется бинарный предикат, анализирующий соответствующие элементы первой и второй последовательности. В случае неудачного поиска возвращается `last1`.

find_end

Алгоритм `find_end` находит первое вхождение в первую последовательность второй последовательности (с анализом предиката или без) и возвращает итератор на последний совпадающий элемент:

```
template<class For1, class For2>
For1 find_end(For1 first1, For1 last1,
              For2 first2, For2 last2);
template<class For1, class For2, class BinPred>
For1 find_end(For1 first1, For1 last1,
              For2 first2, For2 last2, BinPred pred);
```

В случае неудачного поиска возвращается `last1`.

for_each

Алгоритм вызывает для каждого элемента последовательности заданную функцию:

```
template<class In, class Function>
Function for_each(In first, In last, Function f);
```

Пример применения алгоритма приведен на с. 341.

mismatch

Алгоритм `mismatch` ищет первую пару несовпадающих элементов двух последовательностей и возвращает итераторы на эту пару:

```
template<class In1, class In2>
    pair<In1, In2> mismatch(In1 first1, In1 last1,
                           In2 first2);
template <class In1, class In2, class BinPred>
    pair<In1, In2> mismatch(In1 first1, In1 last1,
                           In2 first2, BinPred pred);
```

Длина второй последовательности считается большей или равной длине первой. Пользователь может задать предикат, определяющий, что считать несовпадением.

search, search_n

Алгоритм `search` находит первое вхождение в первую последовательность второй последовательности (с анализом предиката или без) и возвращает итератор на первый совпадающий элемент:

```
template<class For1, class For2>
    For1 search(For1 first1, For1 last1,
                For2 first2, For2 last2);
template<class For1, class For2, class BinPred>
    For1 search(For1 first1, For1 last1,
                For2 first2, For2 last2, BinPred pred);
```

В случае неудачного поиска возвращается `last1`.

Алгоритм `search_n` находит в последовательности подпоследовательность, состоящую из по крайней мере `n` значений `value` (с анализом предиката или без) и возвращает итератор на первый совпадающий элемент:

```
template<class For, class Size, class T>
    For search_n(For first, For last,
                 Size count, const T& value);
template <class For, class Size, class T, class BinPred>
    For search_n(For first, For last,
                 Size count, const T& value, BinPred pred);
```

Модифицирующие операции с последовательностями

Алгоритмы этой категории тем или иным образом изменяют последовательность, с которой они работают. Они используются для копирования, удаления, замены и изменения порядка следования элементов последовательности.

Таблица 14.2. Модифицирующие операции с последовательностями

Алгоритм	Выполняемая функция
copy	Копирование последовательности, начиная с первого элемента
copy_backward	Копирование последовательности, начиная с последнего элемента
fill	Замена всех элементов заданным значением
fill_n	Замена первых n элементов заданным значением
generate	Замена всех элементов результатом операции
generate_n	Замена первых n элементов результатом операции
iter_swap	Обмен местами двух элементов, заданных итераторами
random_shuffle	Перемещение элементов в соответствии со случайным равномерным распределением
remove	Перемещение элементов с заданным значением
remove_copy	Копирование последовательности с перемещением элементов с заданным значением
remove_copy_if	Копирование последовательности с перемещением элементов при выполнении предиката
remove_if	Перемещение элементов при выполнении предиката
replace	Замена элементов с заданным значением
replace_copy	Копирование последовательности с заменой элементов с заданным значением
replace_copy_if	Копирование последовательности с заменой элементов при выполнении предиката
replace_if	Замена элементов при выполнении предиката
reverse	Изменение порядка элементов на обратный
reverse_copy	Копирование последовательности в обратном порядке
rotate	Циклическое перемещение элементов последовательности
rotate_copy	Циклическое копирование элементов
swap	Обмен местами двух элементов
swap_ranges	Обмен местами элементов двух последовательностей
transform	Выполнение заданной операции над каждым элементом последовательности
unique	Удаление равных соседних элементов
unique_copy	Копирование последовательности с удалением равных соседних элементов

Рассмотрим эти алгоритмы подробнее.

copy, copy_backward

Алгоритм `copy` выполняет копирование начиная с первого элемента последовательности, границы которой задаются итераторами `first` и `last`, в выходную последовательность, для которой задается итератор начала `result`:

```
template<class In, class Out>
    Out copy(In first, In last, Out result);
```

Алгоритм `copy_backward` выполняет копирование, начиная с последнего элемента заданной последовательности. Третий параметр должен указывать на элемент, следующий за последним элементом приемника, поскольку его значение уменьшается на шаг перед операцией копирования каждого элемента:

```
template<class Bi1, class Bi2>
    Bi2 copy_backward(Bi1 first, Bi1 last, Bi2 result);
```

Последовательности могут перекрываться. При копировании нужно следить за тем, чтобы не выйти за границы выходной последовательности.

Пример копирования последовательности:

```
#include <iostream>
#include <algorithm>
using namespace std;
int main(){
    int b[4], a[5] = {1, 2, 3, 4, 5}, i;
    copy (a + 1, a + 5, b);
    for (i = 0; i < 4; i++) cout << b[i]; // 2 3 4 5
    cout << endl;
    copy (a + 1, a + 5, a);
    for (i = 0; i < 5; i++) cout << a[i]; // 2 3 4 5 5
    cout << endl;
    copy_backward (b, b + 3, b + 4);
    for (i = 0; i < 4; i++) cout << b[i]; // 2 2 3 4
    cout << endl;
    return 0;
}
```

Алгоритм `copy` можно применять также для ввода и вывода последовательности. Для этого третьим параметром задается потоковый итератор (см. с. 333):

```
#include <iostream>
#include <algorithm>
using namespace std;
int main(){
    const int N = 5;
    int a[N] = {1, 2, 3, 4, 5};
    copy (a, a + N, ostream_iterator<int>(cout, " "));
    cout << endl;
    return 0;
}
```

fill, fill_n

Алгоритм `fill` выполняет замену всех элементов последовательности, определенной с помощью итераторов `first` и `last`, заданным значением `value`. Алгоритм `fill_n` выполняет замену `n` элементов заданным значением:

```
template<class For, class T>
    void fill(For first, For last, const T& value);
template<class Out, class Size, class T>
    void fill_n(Out first, Size n, const T& value);
```

Рассмотрим пример заполнения целочисленного массива:

```
#include <iostream>
#include <algorithm>
using namespace std;
int main(){
    int a[5], i;
    fill (a, a + 5, 1);
    for (i = 0; i < 5; i++) cout << a[i]; // 1 1 1 1 1
    cout << endl;
    fill_n (a + 2, 2, 0);
    for (i = 0; i < 5; i++) cout << a[i]; // 1 1 0 0 1
    cout << endl;
    return 0;
}
```

Обратите внимание на то, что для списков мы не можем пользоваться выражением типа `fill (a, a + 5, 1);`, поскольку операция сложения для итераторов списка не определена. Если нам не известен итератор на конец заполняемой последовательности, можно воспользоваться второй формой алгоритма.

generate, generate_n

Алгоритм `generate` выполняет замену всех элементов результатом операции. Это позволяет заполнить контейнер не одинаковыми значениями, а вычисленными с помощью функции или функционального объекта `gen`, заданного третьим параметром.

```
template<class For, class Generator>
    void generate(For first, For last,
        Generator gen);
template<class Out, class Size, class Generator>
    void generate_n(Out first, Size n, Generator gen);
```

Простой пример:

```
#include <iostream>
#include <algorithm>
using namespace std;
int f(){
    static int i = 1;
    return (++i) * 3;
```

```

}
int main(){
    int a[5], i;
    generate(a, a + 5, f);
    for (i = 0; i < 5; i++) cout << a[i] << " "; // 6 9 12 15 18
    return 0;
}

```

iter_swap, swap, swap_ranges

Алгоритм `iter_swap` выполняет обмен местами двух элементов, заданных итераторами:

```

template<class For1, class For2>
void iter_swap(For1 a, For2 b);

```

Алгоритм `swap` выполняет обмен местами двух элементов:

```

template<class T> void swap(T& a, T& b);

```

Алгоритм `swap_ranges` выполняет обмен местами элементов в двух указанных диапазонах (для второго диапазона задано только его начало):

```

template<class For1, class For2>
For2 swap_ranges(For1 first1, For1 last1, For2 first2);

```

random_shuffle

Алгоритм `random_shuffle` выполняет перемещение (перетасовку) элементов в соответствии со случайным равномерным распределением. Третьим параметром алгоритма можно задать генератор случайных чисел. Это позволяет получать разные результаты при каждом запуске программы. Генератор может быть функцией или функциональным объектом, получающим аргумент `n` типа `int` и возвращающим целое число в диапазоне от 0 до `n`.

```

template<class Ran>
void random_shuffle(Ran first, Ran last);
template<class Ran, class RandomNumberGenerator>
void random_shuffle(Ran first, Ran last,
    RandomNumberGenerator& rand);

```

В примере в генераторе использована функция `rand`, объявленная в заголовочном файле `<time.h>`:

```

#include <iostream>
#include <algorithm>
#include <time.h>
using namespace std;
struct random_gen{
    random_gen(){srand((unsigned int)time(NULL)); }
    int operator()(int n){return rand() % n;}
};
int main(){

```



```

int a[5] = {1, 2, 3, 4, 5}; i:
random_shuffle(a, a + 5, random_gen());
for (i = 0; i < 5; i++) cout << a[i] << " "; // 5 3 4 1 2
cout << endl;
return 0;
}

```

remove, remove_if, remove_copy, remove_copy_if

Алгоритмы семейства `remove` выполняют перемещение в конец последовательности элементов с заданным значением `value` или по предикату `pred`. При этом оставшиеся элементы перемещаются в начало последовательности с сохранением их относительного порядка. Алгоритм возвращает границу их размещения. Элементы, расположенные после границы, не удаляются, размер последовательности не изменяется. Формы алгоритма, содержащие слово `copy`, перед обработкой копируют последовательность на место, заданное итератором `Out`, и обрабатывают копию последовательности.

```

template<class For, class T>
    For remove(For first, For last, const T& value);
template<class For, class Pred>
    For remove_if(For first, For last, Pred pred);
template<class In, class Out, class T>
    Out remove_copy(In first, In last,
                    Out result, const T& value);
template<class In, class Out, class Pred>
    Out remove_copy_if(In first, In last,
                       Out result, Pred pred);

```

Пример применения `remove` для удаления элементов вектора, значения которых равны 2:

```

#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main(){
    vector<int> a;
    int i;
    for (i = 0; i < 5; i++) a.push_back(i);
    for (i = 0; i < 5; i++) a.push_back(i);
    for (i = 0; i < a.size(); i++) cout << a[i];
    cout << endl;
    vector<int>::iterator k, p = remove(a.begin(), a.end(), 2);
    for (i = 0; i < a.size(); i++) cout << a[i];
    cout << endl;
    for (k = a.begin(); k != p; k++) cout << *k;
    return 0;
}

```

Результат работы программы:

```
0123401234
0134013434
01340134
```

Пример применения `remove_if` совместно с методом `erase` для удаления элементов вектора, значения которых лежат в пределах от 10 до 50:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool In_10_50 (int x) {return x > 10 && x < 50;}
int main(){
    vector<int> a;
    int i;
    for (i = 1; i<10; i++) a.push_back(i*10);
    for (i = 0; i<a.size(); i++) cout << a[i] << " ";
    cout << endl;
    vector<int>::iterator new_end = remove_if(a.begin(), a.end(), In_10_50);
    a.erase(new_end, a.end());
    for (i = 0; i<a.size(); i++) cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

Результат работы программы:

```
10 20 30 40 50 60 70 80 90
10 50 60 70 80 90
```

replace, replace_if, replace_copy, replace_copy_if

Алгоритмы семейства `replace` выполняют замену элементов с заданным значением на новое значение или в соответствии с предикатом. Формы алгоритма, содержащие слово `copy`, перед обработкой копируют последовательность на место, заданное итератором `Out`, и обрабатывают копию последовательности.

```
template<class For, class T>
void replace(For first, For last,
             const T& old_value, const T& new_value);
template<class For, class Pred, class T>
void replace_if(For first, For last,
                Pred pred, const T& new_value);
template<class In, class Out, class T>
Out replace_copy(In first, In last, Out result,
                 const T& old_value, const T& new_value);
template<class Iterator, class Out, class Pred, class T>
Out replace_copy_if(Iterator first, Iterator last,
                    Out result, Pred pred, const T& new_value);
```

Рассмотрим пример, в котором выполняется копирование вектора *a* в новый вектор *b* (используется итератор вставки) с заменой всех элементов, лежащих в диапазоне от 10 до 50, на значение 33:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool In_10_50(int x){
    return x >10 && x < 50;
}
int main(){
    vector<int> a, v;
    vector<int>::iterator i;
    for (int k = 1; k < 10; k++)a.push_back(k * 10);
    for (i = a.begin(); i != a.end(); i++) cout << *i << " ";
    cout << endl;
    replace_copy_if(a.begin(), a.end(),
                    inserter(v, v.begin()), In_10_50, 33);
    for (i = v.begin(); i != v.end(); i++) cout << *i << " ";
    cout << endl;
    return 0;
}
```

Результат работы программы (в первой строке исходный вектор, во второй — результирующий):

```
10 20 30 40 50 60 70 80 90
10 33 33 33 50 60 70 80 90
```

reverse, reverse_copy

Алгоритм *reverse* изменяет порядок следования элементов последовательности на обратный, а *reverse_copy* выполняет копирование исходной последовательности в результирующую в обратном порядке:

```
template<class Bi>
void reverse(Bi first, Bi last);
template<class Bi, class Out>
Out reverse_copy(Bi first, Bi last, Out result);
```

rotate, rotate_copy

Алгоритм *rotate* выполняет циклическое перемещение элементов последовательности, а *rotate_copy* — копии последовательности:

```
template<class For>
void rotate(For first, For middle, For last);
template<class For, class Out>
Out rotate_copy(For first, For middle, For last,
                Out result);
```

Перемещение выполняется до тех пор, пока элемент, на который указывает второй параметр, не станет первым в последовательности. Первый и третий параметры задают начало и конец обрабатываемой последовательности. Например, для целочисленного массива `int a[5] = {1, 2, 3, 4, 5}` вызов `rotate(a, a+2, a+5)` приведет к тому, что элементы будут следовать в порядке 34512. Если после этого вызвать `rotate(a, a+3, a+5)`, массив примет первоначальный вид.

transform

Алгоритм `transform` выполняет заданную операцию над каждым элементом последовательности. Первая форма алгоритма выполняет унарную операцию, заданную функцией или функциональным объектом `op`, и помещает результат в место, заданное итератором `result`:

```
template<class In, class Out, class Op>
    Out transform(In first, In last, Out result, Op op);
```

Вторая форма алгоритма выполняет бинарную операцию над парой соответствующих элементов двух последовательностей и помещает результат в место, заданное итератором `result`:

```
template<class In1, class In2, class Out, class BinaryOperation>
    Out transform(In1 first1, In1 last1,
                  In2 first2, Out result, BinaryOperation binary_op);
```

Функциональный объект может быть стандартным или заданным пользователем. Пользовательский функциональный объект должен быть потомком `unary_function` или `binary_function` (см. с. 334).

В приведенном ниже примере первый вызов `transform` выполняет преобразование массива `a` по формуле: $a_i = a_i^2 - b_i^2$, второй вызов меняет знак у элементов массива `b` с помощью стандартного функционального объекта `negate`.

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
struct preobr: binary_function<double, double, double>{
    double operator()(double x, double y) const{
        return x * x - y * y;}
};
int main(){
    const int m = 5;
    double a[m] = {5, 3, 2, 3, 1},
           b[m] = {1, 10, -3, 2, -4};
    transform(a, a + m, b, a, preobr());
    transform(b, b + m, b, negate<double>());
    int i;
    for (i = 0; i < m; i++) cout << a[i] << " ";
    cout << endl;
    for (i = 0; i < m; i++) cout << b[i] << " ";
```

```

    cout << endl;
    return 0;
}

```

Результат работы программы:

```

24 -91 -5 5 -15
-1 -10 3 -2 4

```

unique, unique_copy

Алгоритм `unique` выполняет удаление из последовательности соседних элементов, равных друг другу или удовлетворяющих критерию, заданному с помощью бинарного предиката `pred`. Размер последовательности при этом не изменяется¹. Алгоритм возвращает итератор на новый логический конец данных.

```

template<class For>
    For unique(For first, For last);
template<class For, class BinPred>
    For unique(For first, For last, BinPred pred);

```

Алгоритм `unique_copy` выполняет те же действия с копией последовательности:

```

template<class In, class Out>
    Out unique_copy(In first, In last, Out result);
template<class In, class Out, class BinPred>
    Out unique_copy(In first, In last,
                    Out result, BinPred pred);

```

Алгоритмы, связанные с сортировкой

Алгоритмы этой категории упорядочивают последовательности, выполняют поиск элементов, слияние последовательностей, поиск минимума и максимума, лексикографическое сравнение, перестановки и т. п.

Таблица 14.3. Алгоритмы, связанные с сортировкой

Алгоритм	Выполняемая функция
<code>binary_search</code>	Поиск заданного значения
<code>equal_range</code>	Нахождение последовательности элементов с заданным значением
<code>inplace_merge</code>	Слияние отсортированных последовательностей одного диапазона
<code>lexicographical_compare</code>	Лексикографически первая из двух последовательностей
<code>lower_bound</code>	Нахождение первого вхождения заданного значения

продолжение ➤

¹ Для списков предпочтительнее пользоваться одноименным методом класса `list`.

Таблица 14.3 (продолжение)

Алгоритм	Выполняемая функция
<code>max</code>	Большее из двух значений
<code>max_element</code>	Наибольшее значение в последовательности
<code>merge</code>	Слияние отсортированных последовательностей
<code>min</code>	Меньшее из двух значений
<code>min_element</code>	Наименьшее значение в последовательности
<code>next_permutation</code>	Следующая перестановка в лексикографическом порядке
<code>nth_element</code>	Помещение n-го элемента на заданное место
<code>partial_sort</code>	Частичная сортировка
<code>partial_sort_copy</code>	Частичная сортировка с копированием
<code>partition</code>	Перемещение вперед элементов, удовлетворяющих условию
<code>prev_permutation</code>	Предыдущая перестановка в лексикографическом порядке
<code>sort</code>	Сортировка
<code>stable_partition</code>	Перемещение вперед элементов, удовлетворяющих условию, с сохранением их относительного порядка
<code>stable_sort</code>	Сортировка, сохраняющая порядок для одинаковых элементов
<code>upper_bound</code>	Нахождение первого элемента, большего, чем заданное значение

Рассмотрим эти алгоритмы подробнее. Для каждого из алгоритмов существует две формы: одна использует операцию `<`, а другая — функцию сравнения, заданную пользователем. Обратите внимание на то, что многим алгоритмам требуются итераторы произвольного доступа.

binary_search

Алгоритм `binary_search` выполняет поиск значения `value` в *отсортированной* последовательности, заданной итераторами `first` и `last`. Возвращается только факт, найдено искомое значение или нет. Двоичным поиск называется потому, что выполняется путем последовательного деления интервала пополам («как поймать льва в пустыне»).

```
template<class For, class T>
bool binary_search(For first, For last,
                  const T& value);
template<class For, class T, class Compare>
bool binary_search(For first, For last,
                  const T& value, Compare comp);
```


equal_range

Алгоритм `equal_range` выполняет нахождение границ последовательности элементов, в любое место которой можно вставить заданное значение без нарушения порядка. Последовательность должна быть отсортирована. При задании функционального объекта алгоритм находит границы, в пределах которых для каждого значения итератора `k` выполняется условие `comp(*k, value) == false && comp(value, *k) == false`.

```
template<class For, class T>
    pair<For, For> equal_range(For first, For last,
        const T& value);
template<class For, class T, class Compare>
    pair<For, For> equal_range(For first, For last,
        const T& value, Compare comp);
```

Например, для последовательности 2 4 5 5 7 9 12 18 вызов `equal_range` с `value = 8` даст в результате пару итераторов, указывающих на элементы 9 и 9, а вызов с `value = 5` — на первый из элементов, равный 5, и 7.

inplace_merge

Алгоритм `inplace_merge` выполняет слияние двух отсортированных частей одной последовательности. Границы первой части задаются двумя первыми параметрами, начало второй части — третьим параметром.

```
template<class Bi>
    void inplace_merge(Bi first, Bi middle, Bi last);
template<class Bi, class Compare>
    void inplace_merge(Bi first, Bi middle, Bi last,
        Compare comp);
```

lexicographical_compare

Алгоритм `lexicographical_compare` выполняет поэлементное сравнение двух последовательностей либо с использованием операции `<`, либо с помощью заданной функции `comp`. Возвращается `true`, если первая последовательность лексикографически меньше второй (то есть очередной элемент первой последовательности оказался меньше соответствующего элемента второй), и `false` в противном случае. Если длины последовательностей не совпадают, недостающие элементы считаются меньшими соответствующих элементов другой последовательности.

```
template<class In1, class In2>
    bool lexicographical_compare
        (In1 first1, In1 last1, In2 first2, In2 last2);
template<class In1, class In2, class Compare>
    bool lexicographical_compare
        (In1 first1, In1 last1, In2 first2, In2 last2,
        Compare comp);
```

Таблица 14.3 (продолжение)

Алгоритм	Выполняемая функция
max	Большее из двух значений
max_element	Наибольшее значение в последовательности
merge	Слияние отсортированных последовательностей
min	Меньшее из двух значений
min_element	Наименьшее значение в последовательности
next_permutation	Следующая перестановка в лексикографическом порядке
nth_element	Помещение n-го элемента на заданное место
partial_sort	Частичная сортировка
partial_sort_copy	Частичная сортировка с копированием
partition	Перемещение вперед элементов, удовлетворяющих условию
prev_permutation	Предыдущая перестановка в лексикографическом порядке
sort	Сортировка
stable_partition	Перемещение вперед элементов, удовлетворяющих условию, с сохранением их относительного порядка
stable_sort	Сортировка, сохраняющая порядок для одинаковых элементов
upper_bound	Нахождение первого элемента, большего, чем заданное значение

Рассмотрим эти алгоритмы подробнее. Для каждого из алгоритмов существуют две формы: одна использует операцию `<`, а другая — функцию сравнения, заданную пользователем. Обратите внимание на то, что многим алгоритмам требуются итераторы произвольного доступа.

binary_search

Алгоритм `binary_search` выполняет поиск значения `value` в *отсортированной* последовательности, заданной итераторами `first` и `last`. Возвращается только факт: найдено искомое значение или нет. Двоичным поиск называется потому, что выполняется путем последовательного деления интервала пополам («как поиск льва в пустыне»).

```
template<class For, class T>
bool binary_search(For first, For last,
                  const T& value);
template<class For, class T, class Compare>
bool binary_search(For first, For last,
                  const T& value, Compare comp);
```

equal_range

Алгоритм `equal_range` возвращает итераторы, указывающие на место в последовательности, начиная с которого элементы больше заданного значения. Возвращает пару итераторов, указывающих на место в последовательности, начиная с которого элементы больше заданного значения. Возвращает пару итераторов, указывающих на место в последовательности, начиная с которого элементы больше заданного значения.

```
template<class For, class T>
pair<For, For> equal_range(For first, For last, const T& value, Compare comp);
template<class For, class T>
pair<For, For> equal_range(For first, For last, const T& value);
```

Например, для последовательности `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}` в результате пары итераторов `equal_range(1, 10, 5)` получим `value = 5` — на первый элемент.

inplace_merge

Алгоритм `inplace_merge` выполняет сортировку последовательности. Грань между элементами первой и второй части находится на месте.

```
template<class Bi>
void inplace_merge(Bi first, Bi last, Compare comp);
template<class Bi, class Compare>
void inplace_merge(Bi first, Bi last, Compare comp);
```

lexicographical_compare

Алгоритм `lexicographical_compare` проверяет, является ли первая последовательность лексически меньше второй (то есть, если для каждого элемента первой последовательности найдется элемент второй последовательности, который больше или равен ему).

```
template<class In1, class In2>
bool lexicographical_compare(In1 first1, In1 last1, In2 first2, In2 last2);
template<class In1, class In2, class Compare>
bool lexicographical_compare(In1 first1, In1 last1, In2 first2, In2 last2, Compare comp);
```


equal_range

Алгоритм `equal_range` выполняет нахождение границ последовательности элементов, в любое место которой можно вставить заданное значение без нарушения порядка. Последовательность должна быть отсортирована. При задании функционального объекта алгоритм находит границы, в пределах которых для каждого значения итератора `k` выполняется условие `comp(*k, value) == false` && `comp(value, *k) == false`.

```
template<class For, class T>
pair<For, For> equal_range(For first, For last,
    const T& value);
template<class For, class T, class Compare>
pair<For, For> equal_range(For first, For last,
    const T& value, Compare comp);
```

Например, для последовательности 2 4 5 5 7 9 12 18 вызов `equal_range` с `value = 8` даст в результате пару итераторов, указывающих на элементы 9 и 9, а вызов с `value = 5` — на первый из элементов, равный 5, и 7.

inplace_merge

Алгоритм `inplace_merge` выполняет слияние двух отсортированных частей одной последовательности. Границы первой части задаются двумя первыми параметрами, начало второй части — третьим параметром.

```
template<class Bi>
void inplace_merge(Bi first, Bi middle, Bi last);
template<class Bi, class Compare>
void inplace_merge(Bi first, Bi middle, Bi last,
    Compare comp);
```

lexicographical_compare

Алгоритм `lexicographical_compare` выполняет поэлементное сравнение двух последовательностей либо с использованием операции `<`, либо с помощью заданной функции `comp`. Возвращается `true`, если первая последовательность лексикографически меньше второй (то есть очередной элемент первой последовательности оказался меньше соответствующего элемента второй), и `false` в противном случае. Если длины последовательностей не совпадают, недостающие элементы считаются меньшими соответствующих элементов другой последовательности.

```
template<class In1, class In2>
bool lexicographical_compare
    (In1 first1, In1 last1, In2 first2, In2 last2);
template<class In1, class In2, class Compare>
bool lexicographical_compare
    (In1 first1, In1 last1, In2 first2, In2 last2,
    Compare comp);
```

Пример:

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
int main(){
    const int m = 5;
    double a[m] = {5, 3, 2, 3, 1},
           b[m] = {5, 3, 2, 3, 2},
           c[m] = {5, 3, 1, 3, 10};
    cout << lexicographical_compare(a, a + m, b, b + m); // 1
    cout << lexicographical_compare(a, a + m, c, c + m); // 0
    cout << lexicographical_compare(a, a + m, b, b + m, greater<int>()); // 0
    return 0;
}
```

lower_bound, upper_bound

Алгоритм `lower_bound` находит итератор на первый, а `upper_bound` — на последний из элементов отсортированной последовательности, перед которым можно вставить заданное значение, не нарушая упорядоченности.

```
template<class For, class T>
    For lower_bound(For first, For last, const T& value);
template<class For, class T, class Compare>
    For lower_bound(For first, For last, const T& value,
                    Compare comp);
template<class For, class T>
    For upper_bound(For first, For last, const T& value);
template<class For, class T, class Compare>
    For upper_bound(For first, For last, const T& value,
                    Compare comp);
```

max, min

Алгоритм `max` ищет наибольшее из двух значений, а `min` — наименьшее, используя или операцию `<`, или собственный критерий сравнения.

```
template<class T> const T& min(const T& a, const T& b);
template<class T, class Compare>
    const T& min(const T& a, const T& b, Compare comp);
template<class T> const T& max(const T& a, const T& b);
template<class T, class Compare>
    const T& max(const T& a, const T& b, Compare comp);
```

max_element, min_element

Алгоритм `max_element` возвращает итератор на наибольшее значение в последовательности, а алгоритм `min_element` — на наименьшее значение.

```

template<class For>
    For min_element(For first, For last);
template<class For, class Compare>
    For min_element(For first, For last, Compare comp);
template<class For>
    For max_element(For first, For last);
template<class For, class Compare>
    For max_element(For first, For last, Compare comp);

```

merge

Алгоритм `merge` выполняет слияние отсортированных последовательностей.

```

template<class In1, class In2, class Out>
    Out merge(In1 first1, In1 last1, In2 first2,
              In2 last2, Out result);
template<class In1, class In2, class Out, class Compare>
    Out merge(In1 first1, In1 last1, In2 first2,
              In2 last2, Out result, Compare comp);

```

В отличие от одноименного метода слияния списков, элементы из исходных последовательностей не удаляются. При равенстве ключей элементы первой последовательности предшествуют элементам второй (это имеет значение для структур данных, содержащих, кроме ключа, информационную часть).

Пример:

```

#include <iostream>
#include <algorithm>
using namespace std;
int main(){
    const int m = 5;
    double a[m] = {3, 4, 8, 17, 20},
           b[m] = {5, 6, 8, 10, 35}, c[m * 2];
    int i;
    merge(a, a + m, b, b + m, c);
    for (i = 0; i < m * 2; i++)
        cout << c[i] << " ";    // 3 4 5 6 8 8 10 17 20 35
    cout << endl;
    return 0;
}

```

next_permutation, prev_permutation

Элементы любой последовательности можно расположить разными способами. Для последовательности длины n таких перестановок существует $n!$ ($1 \cdot 2 \cdot \dots \cdot n$). Алгоритм `next_permutation` производит очередную перестановку в лексикографическом порядке, а алгоритм `prev_permutation` — предыдущую. Алгоритмы возвращают булевское значение `true`, если следующая перестановка существует, и `false` в противном случае.


```

template<class Bi>
    bool next_permutation(Bi first, Bi last);
template<class Bi, class Compare>
    bool next_permutation(Bi first, Bi last, Compare comp);
template<class Bi>
    bool prev_permutation(Bi first, Bi last);
template<class Bi, class Compare>
    bool prev_permutation(Bi first, Bi last, Compare comp);

```

Пример:

```

#include <iostream>
#include <algorithm>
using namespace std;
int main(){
    const int m = 3;
    int a[m]={1, 4, 2}, b[m];
    int i;
    copy(a, a + m, b);
    cout << " next_permutation(a, a + m):" << endl;
    while (next_permutation(a, a + m)){
        for (i = 0; i < m; i++) cout << a[i] << " ";
        cout << endl;
    }
    cout << " prev_permutation(b, b + m):" << endl;
    while (prev_permutation(b, b + m)){
        for (i = 0; i < m; i++) cout << b[i] << " ";
        cout << endl;
    }
    return 0;
}

```

Результат работы программы:

```

next_permutation(a, a + m):
2 1 4
2 4 1
4 1 2
4 2 1
prev_permutation(b, b + m):
1 2 4

```

nth_element

Алгоритм выполняет частичную сортировку массива. После выполнения алгоритма значение элемента, заданного итератором `nth`, будет таким же, как после полной сортировки, то есть все элементы левее этой позиции будут меньше него, а все, что правее, — больше.

```

template<class Ran>
    void nth_element(Ran first, Ran nth, Ran last);
template<class Ran, class Compare>

```

```
void nth_element(Ran first, Ran nth, Ran last,
                Compare comp);
```

partial_sort, partial_sort_copy

Алгоритм `partial_sort` также выполняет частичную сортировку последовательности. После выполнения алгоритма элементы от `first` до `middle` будут располагаться в таком же порядке, как после полной сортировки.

Алгоритм `partial_sort_copy` выполняет те же действия с копией последовательности.

```
template<class Ran>
    void partial_sort(Ran first, Ran middle, Ran last);
template<class Ran, class Compare>
    void partial_sort(Ran first, Ran middle, Ran last,
                    Compare comp);
template<class In, class Ran>
    Ran partial_sort_copy(In first, In last,
                        Ran result_first, Ran result_last);
template<class In, class Ran, class Compare>
    Ran partial_sort_copy(In first, In last,
                        Ran result_first, Ran result_last, Compare comp);
```

Частичная сортировка экономит время в тех случаях, когда нас интересуют только несколько самых больших или самых маленьких значений, например, «горячая десятка».

```
#include <iostream>
#include <algorithm>
using namespace std;
int main(){
    const int m = 8;
    int a[m] = {3, 1, 2, 34, 8, 7, 20, 2};
    int i;
    partial_sort(a, a + 5, a + m, greater<int>());
    for (i = 0; i < m; i++) cout << a[i] << " ";
    cout << endl; // 34 20 8 7 3 1 2 2
    return 0;
}
```

partition, stable_partition

Алгоритм `partition` размещает элементы, удовлетворяющие заданному условию, перед остальными элементами. Алгоритм `stable_partition` выполняет то же самое, но с сохранением относительного порядка элементов. Условие задается с помощью функции или функционального объекта.

```
template<class Bi, class Pred>
    Bi partition(Bi first, Bi last, Pred pred);
template<class Bi, class Pred>
    Bi stable_partition(Bi first, Bi last, Pred pred);
```

sort, stable_sort

Алгоритм `sort` выполняет сортировку эффективную последовательности за время, пропорциональное $N \cdot \log_2 N$. Для сохранения порядка следования одинаковых элементов следует применять алгоритм `stable_sort`. Время его работы пропорционально $N \cdot (\log_2 N)^2$.

```
template<class Ran>
    void sort(Ran first, Ran last);
template<class Ran, class Compare>
    void sort(Ran first, Ran last, Compare comp);
template<class Ran>
    void stable_sort(Ran first, Ran last);
template<class Ran, class Compare>
    void stable_sort(Ran first, Ran last, Compare comp);
```

Обратите внимание на то, что этому алгоритму требуются итераторы произвольного доступа.

Алгоритмы работы с множествами и пирамидами

Алгоритмы этой категории выполняют сортировку множеств и операции с пирамидами. В первом случае отсортированная последовательность рассматривается как *множество*, а операции объединения и пересечения имеют тот же смысл, что в теории множеств. Приведенные алгоритмы не изменяют входные последовательности, а выходные последовательности упорядочены.

*Пирамидой*¹ называется последовательность, для всех элементов которой выполняются условия:

$$\begin{aligned} a[i] &\leq a[2 \cdot i + 1] \\ a[i] &\leq a[2 \cdot i + 2] \end{aligned}$$

Пример пирамиды из 10 целых чисел: 23 20 21 17 19 18 15 12 10 14.

Как следует из определения, максимальный элемент пирамиды расположен первым, поэтому пирамиды удобно использовать для реализации очередей с приоритетами (см. с. 313), так как существуют эффективные алгоритмы извлечения первого элемента и добавления нового с сохранением условия пирамидальности.

Для работы с пирамидой требуются итераторы произвольного доступа.

Так же, как и для других категорий алгоритмов, для каждого алгоритма существует две формы: одна использует операцию `<`, а другая — функцию сравнения, заданную пользователем.

¹ Часто пирамиду изящно называют *кучей* (перевод английского слова *heap*).

Таблица 14.4. Алгоритмы работы с множествами и пирамидами

Алгоритм	Выполняемая функция
<code>includes</code>	Включение одного множества в другое
<code>set_intersection</code>	Создание отсортированного пересечения множеств
<code>set_difference</code>	Создание отсортированной последовательности элементов, входящих только в первую из двух последовательностей
<code>set_symmetric_difference</code>	Создание отсортированной последовательности элементов, входящих только в одну из двух последовательностей
<code>set_union</code>	Создание отсортированного объединения множеств
<code>make_heap</code>	Преобразование последовательности с произвольным доступом в пирамиду
<code>pop_heap</code>	Извлечение элемента из пирамиды
<code>push_heap</code>	Добавление элемента в пирамиду
<code>sort_heap</code>	Сортировка пирамиды

Рассмотрим эти алгоритмы подробнее.

includes

Алгоритм `includes` выполняет проверку включения одной последовательности в другую. Результат равен `true` в том случае, когда каждый элемент последовательности `[first2, last2)` содержится в последовательности `[first1, last1)`¹.

```
template<class In1, class In2>
bool includes(In1 first1, In1 last1,
             In2 first2, In2 last2);
template<class In1, class In2, class Compare>
bool includes (In1 first1, In1 last1,
             In2 first2, In2 last2, Compare comp);
```

set_intersection

Алгоритм `set_intersection` создает отсортированное пересечение множеств (то есть множество, содержащее только элементы, входящие и в первое, и во второе множество):

```
template<class In1, class In2, class Out>
Out set_intersection
(In1 first1, In1 last1, In2 first2, In2 last2,
```

¹ Квадратная скобка означает, что соответствующий элемент принадлежит последовательности, круглая — что не принадлежит (указывает на следующий за ней элемент).

```

    Out result);
template<class In1, class In2, class Out,
        class Compare>
    Out set_intersection (In1 first1, In1 last1,
        In2 first2, In2 last2, Out result, Compare comp);

```

set_difference, set_symmetric_difference

Алгоритм `set_difference` выполняет копирование в `Out` элементов, входящих только в первую из двух последовательностей:

```

template<class In1, class In2, class Out>
    Out set_difference(In1 first1, In1 last1,
        In2 first2, In2 last2, Out result);
template<class In1, class In2, class Out, class Compare>
    Out set_difference (In1 first1, In1 last1,
        In2 first2, In2 last2, Out result, Compare comp);

```

Алгоритм `set_symmetric_difference` выполняет копирование в `Out` элементов, входящих только в одну из двух последовательностей:

```

template<class In1, class In2, class Out>
    Out set_symmetric_difference(In1 first1, In1 last1,
        In2 first2, In2 last2, Out result);
template<class In1, class In2, class Out, class Compare>
    Out set_symmetric_difference(In1 first1, In1 last1,
        In2 first2, In2 last2, Out result, Compare comp);

```

Результирующая последовательность не должна перекрываться ни с одной из исходных.

set_union

Алгоритм `set_union` создает отсортированное объединение множеств (то есть множество, содержащее элементы первого и второго множества без повторов):

```

template<class In1, class In2, class Out>
    Out set_union(In1 first1, In1 last1,
        In2 first2, In2 last2, Out result);
template<class In1, class In2, class Out, class Compare>
    Out set_union(In1 first1, In1 last1,
        In2 first2, In2 last2, Out result, Compare comp);

```

Пример использования алгоритмов работы с множествами

```

#include <iostream>
#include <algorithm>
using namespace std;
void show(const char *s, const int *begin,
        const int *end){

```

```

    cout << s << " ";
    copy(begin, end, ostream_iterator<int>(cout, " "));
    cout << endl;
}

int main(){
    int a[4] = {2, 5, 7, 9}, b[3] = {1, 5, 9},
        Union[7], *p_union,
        isect[4], *p_isect,
        dif[3], *p_dif,
        symdif[7], *p_symdif;
    p_isect = set_intersection(a, a + 4, b, b + 3, isect);
    p_union = set_union(a, a + 4, b, b + 3, Union);
    p_dif = set_difference(a, a + 4, b, b + 3, dif);
    p_symdif = set_symmetric_difference(a, a + 4, b, b + 3,
        symdif);
    show("a: ", a, a + 4);
    show("b: ", b, b + 3);
    show("isect: ", isect, p_isect);
    show("Union: ", Union, p_union);
    show("dif: ", dif, p_dif);
    show("symdif: ", symdif, p_symdif);
    if (includes(a, a + 4, b, b + 3))
        cout << "a включает b.\n";
    else cout << "a не включает b.\n";
    if (includes(Union, p_union, b, b + 3))
        cout << "Union включает b.\n";
    else cout << "Union не включает b.\n";
    return 0;
}

```

Результат работы программы:

```

a:      2 5 7 9
b:      1 5 9
isect:   5 9
Union:   1 2 5 7 9
dif:     2 7
symdif:  1 2 7
a не включает b.
Union включает b.

```

make_heap

Алгоритм `make_heap` выполняет преобразование последовательности с произвольным доступом в пирамиду:

```

template<class Ran>
void make_heap(Ran first, Ran last);
template<class Ran, class Compare>
void make_heap(Ran first, Ran last, Compare comp);

```


pop_heap

Алгоритм `pop_heap` удаляет первый элемент последовательности, а затем восстанавливает условие пирамидальности:

```
template<class Ran>
    void pop_heap(Ran first, Ran last);
template<class Ran, class Compare>
    void pop_heap(Ran first, Ran last, Compare comp);
```

Получение значения извлекаемого элемента из пирамиды требуется выполнить обычным образом. Например, для приведенного выше массива *a* можно записать:

```
x = *a; pop_heap(a, a + m);
```

push_heap

Алгоритм `push_heap` выполняет преобразование последовательности в пирамиду после добавления в нее последнего элемента:

```
template<class Ran>
    void push_heap(Ran first, Ran last);
template<class Ran, class Compare>
    void push_heap(Ran first, Ran last, Compare comp);
```

Обратите внимание, что до вызова `push_heap` требуется добавить элемент в последовательность способом, соответствующим типу контейнера, например:

```
v.push_back(x); push_heap(v.begin(), v.end());
```

sort_heap

Алгоритм `sort_heap` преобразует пирамиду в отсортированную по возрастанию последовательность:

```
template<class Ran>
    void sort_heap(Ran first, Ran last);
template<class Ran, class Compare>
    void sort_heap(Ran first, Ran last, Compare comp);
```

Поскольку этот алгоритм использует свойства пирамиды, он работает быстрее обычной сортировки. Сортировка не сохраняет относительный порядок следования элементов с одинаковыми ключами.

ГЛАВА 15

Средства для численных расчетов

Обобщенные численные алгоритмы

Для использования численных алгоритмов требуется подключить заголовочный файл `<numeric>`.

Таблица 15. Обобщенные численные алгоритмы

Алгоритм	Выполняемая функция
<code>accumulate</code>	Накопление
<code>inner_product</code>	Скалярное произведение
<code>partial_sum</code>	Расчет суммы с накоплением
<code>adjacent_difference</code>	Вычисление разности между смежными элементами

`accumulate`

Первая форма алгоритма `accumulate` используется для накопления суммы элементов последовательности, заданной итераторами `first` и `last`. Начальное значение суммы (обычно это 0) задается третьим параметром. Тип этого параметра определяет тип результата (функция возвращает вычисленную сумму):

```
template <class In, class T>
T accumulate(In first, In last, T init);
```

Вторая форма алгоритма `accumulate` позволяет выполнять над третьим параметром и очередным элементом последовательности заданную операцию:

```
template <class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp binary_op);
```

В приведенном далее примере вычисляется сумма, произведение и сумма квадратов элементов массива:

```

#include <iostream>
#include <numeric>
#include <functional>
using namespace std;
int sumkv( int s, int x){ return s + x * x;}
int main(){
    const int m = 5;
    int a[m] = {3, 2, 5, 1, 6}, sum = 0, mul = 1, sum2 = 0;
    cout << accumulate(a, a + m, sum) << endl;
    cout << accumulate(a, a + m, mul, multiplies<int>()) << endl;
    cout << accumulate(a, a + m, sum2, sumkv) << endl;
    return 0;
}

```

inner_product

Первая форма алгоритма `inner_product` используется для вычисления скалярного произведения двух последовательностей (скалярным произведением последовательностей a и b является выражение $\sum a_i \cdot b_i$). Начальное значение произведения задается третьим параметром. Тип этого параметра определяет тип результата (функция возвращает вычисленное произведение):

```

template <class In1, class In2, class T>
    T inner_product(In1 first1, In1 last1, In2 first2,
                   T init);

```

Вторая форма алгоритма `inner_product` используется для выполнения над двумя последовательностями действий, заданных с помощью двух функций или функциональных объектов. Первый используется вместо операции сложения, второй — вместо умножения:

```

template <class In1, class In2, class T,
          class BinOp1, class BinOp2>
    T inner_product(In1 first1, In1 last1, In2 first2,
                   T init, BinOp1 binary_op1, BinOp2 binary_op2);

```

Следующий вызов вычисляет произведение сумм соответствующих элементов последовательностей a и b :

```

cout << inner_product(a, a + m, b, mul, multiplies<int>(), plus<int>());

```

partial_sum

Алгоритм `partial_sum` формирует последовательности из частичных сумм элементов. Например, для последовательности чисел 3 1 2 3 5 результатом будет 3 4 6 9 14, то есть каждый элемент результата равен предыдущему элементу, увеличенному на текущий элемент исходной последовательности.

Вместо суммы четвертым параметром можно задать другую операцию. Результат можно поместить и в исходную последовательность, для этого третьим парамет-

ром указывается итератор на ее начало. Функция возвращает итератор на элемент, следующий за концом результирующей последовательности.

```
template t<class In, class Out>
    Out partial_sum(In first, In last, Out result);
template t<class In, class Out, class BinOp>
    Out partial_sum(In first, In last, Out result,
        BinOp binary_op);
```

adjacent_difference

Алгоритм `adjacent_difference` выполняет вычисление разности между смежными элементами, то есть $d_i = a_i - a_{i-1}$. Вместо разности четвертым параметром можно задать другую операцию. Функция возвращает итератор на элемент, следующий за концом результирующей последовательности.

```
template t<class In, class Out>
    Out adjacent_difference(In first, In last, Out result);
template t<class In, class Out, class BinOp>
    Out adjacent_difference(In first, In last, Out result,
        BinOp binary_op);
```

Этот алгоритм является обратным предыдущему, то есть вызов для одной и той же последовательности сначала одного, а потом другого алгоритма приведет к исходной последовательности.

Класс `valarray`

Для эффективной работы с массивами чисел в стандартной библиотеке определен шаблонный класс `valarray`. Операции с этим классом реализованы с расчетом на их поддержку в архитектурах высокопроизводительных систем. В библиотеке описаны также четыре вспомогательных класса, позволяющих получить различные подмножества `valarray`: `slice_array`, `gslice_array`, `mask_array` и `indirect_array`. Все эти классы реализованы как шаблоны классов с параметром, представляющим собой тип элемента массива.

Для использования класса `valarray` и связанных с ним средств требуется подключить к программе заголовочный файл `<valarray>`. В нем описаны, кроме перечисленных шаблонов классов, классы `slice` и `gslice`, задающие подмножества индексов массива, а также заголовки функций, предназначенных для работы с этими шаблонами и классами.

Шаблон `slice_array` представляет собой строку, столбец или другое регулярное подмножество элементов `valarray` (например, его четные элементы). С помощью этого шаблона можно представлять `valarray` как матрицу произвольной размерности.

Шаблон `gslice_array` позволяет задавать более сложные законы формирования подмножества массива, а `mask_array` — произвольные подмножества с помощью битовой маски. Шаблон `indirect_array` содержит не сами элементы массива, и их индексы.

ром указывается итератор на ее начало. Функция возвращает итератор на элемент, следующий за концом результирующей последовательности.

```
template t<class In, class Out>
    Out partial_sum(In first, In last, Out result);
template t<class In, class Out, class BinOp>
    Out partial_sum(In first, In last, Out result,
        BinOp binary_op);
```

adjacent_difference

Алгоритм `adjacent_difference` выполняет вычисление разности между смежными элементами, то есть $d_i = a_i - a_{i-1}$. Вместо разности четвертым параметром можно задать другую операцию. Функция возвращает итератор на элемент, следующий за концом результирующей последовательности.

```
template t<class In, class Out>
    Out adjacent_difference(In first, In last, Out result);
template t<class In, class Out, class BinOp>
    Out adjacent_difference(In first, In last, Out result,
        BinOp binary_op);
```

Этот алгоритм является обратным предыдущему, то есть вызов для одной и той же последовательности сначала одного, а потом другого алгоритма приведет к исходной последовательности.

Класс `valarray`

Для эффективной работы с массивами чисел в стандартной библиотеке определен шаблонный класс `valarray`. Операции с этим классом реализованы с расчетом на их поддержку в архитектурах высокопроизводительных систем. В библиотеке описаны также четыре вспомогательных класса, позволяющих получить различные подмножества `valarray`: `slice_array`, `gslice_array`, `mask_array` и `indirect_array`. Все эти классы реализованы как шаблоны классов с параметром, представляющим собой тип элемента массива.

Для использования класса `valarray` и связанных с ним средств требуется подключить к программе заголовочный файл `<valarray>`. В нем описаны, кроме перечисленных шаблонов классов, классы `slice` и `gslice`, задающие подмножества индексов массива, а также заголовки функций, предназначенных для работы с этими шаблонами и классами.

Шаблон `slice_array` представляет собой строку, столбец или другое регулярное подмножество элементов `valarray` (например, его четные элементы). С помощью этого шаблона можно представлять `valarray` как матрицу произвольной размерности.

Шаблон `gslice_array` позволяет задавать более сложные законы формирования подмножества массива, а `mask_array` — произвольные подмножества с помощью битовой маски. Шаблон `indirect_array` содержит не сами элементы массива, и их индексы.

В классе `valarray` определены **конструкторы**, позволяющие создавать пустой массив и массив, заполненный одинаковыми или заданными значениями:

```
template<class T> class valarray{
public:
    // Массив нулевого размера:
    valarray();
    // Массив из n элементов (к каждому применяется
    // конструктор по умолчанию:
    explicit valarray(size_t n);
    // Массив из n элементов со значением v:
    valarray(const T& v, size_t n);
    // Массив из n элементов со значениями из массива m:
    valarray(const T* m, size_t n);
    // Конструктор копирования
    valarray(const valarray&);
    // Массив со значениями из среза:
    valarray(const slice_array<T>&);
    // Массив со значениями из обобщенного среза:
    valarray(const gslice_array<T>&);
    // Массив со значениями из подмножества:
    valarray(const mask_array<T>&);
    // Массив со значениями из indirect_array:
    valarray(const indirect_array<T>&);
}
```

Примеры создания массивов:

```
valarray<double> v1;           // Пустой массив
valarray<int> v2(100);         // Массив из 100 элементов,
                                // каждый по умолчанию обнулен
valarray<int> v3(5, 100);       // Массив из 100 элементов, равных 5
const int m = {4, 4, 3, 6, 2};
valarray<int> v4(m, 5);         // Массив из 5 элементов: 4,4,3,6,2
```

В массивах определены операции доступа по индексу и присваивания:

```
T operator[](size_t) const;
T& operator[](size_t);
// Копирование массива:
valarray<T>& operator=(const valarray<T>&);
// Присваивание одного и того же значения всем элементам:
valarray<T>& operator=(const T&);
// Присваивание подмножеств массиву valarray:
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
```

Нумерация элементов массива `valarray` начинается с 0. Проверка выхода за пределы диапазона не производится.

В классе `valarray` определены **конструкторы**, позволяющие создавать пустой массив и массив, заполненный одинаковыми или заданными значениями:

```
template<class T> class valarray{
public:
    // Массив нулевого размера:
    valarray();
    // Массив из n элементов (к каждому применяется
    // конструктор по умолчанию:
    explicit valarray(size_t n);
    // Массив из n элементов со значением v:
    valarray(const T& v, size_t n);
    // Массив из n элементов со значениями из массива m:
    valarray(const T* m, size_t n);
    // Конструктор копирования
    valarray(const valarray&);
    // Массив со значениями из среза:
    valarray(const slice_array<T>&);
    // Массив со значениями из обобщенного среза:
    valarray(const gslice_array<T>&);
    // Массив со значениями из подмножества:
    valarray(const mask_array<T>&);
    // Массив со значениями из indirect_array:
    valarray(const indirect_array<T>&);
};
```

Примеры создания массивов:

```
valarray<double> v1;           // Пустой массив
valarray<int> v2(100);         // Массив из 100 элементов,
                                // каждый по умолчанию обнулен
valarray<int> v3(5, 100);      // Массив из 100 элементов, равных 5
const int m = {4, 4, 3, 6, 2};
valarray<int> v4(m, 5);        // Массив из 5 элементов: 4,4,3,6,2
```

В массивах определены операции **доступа по индексу и присваивания**:

```
T operator[](size_t) const;
T& operator[](size_t);
// Копирование массива:
valarray<T>& operator=(const valarray<T>&);
// Присваивание одного и того же значения всем элементам:
valarray<T>& operator=(const T&);
// Присваивание подмножеств массиву valarray:
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
```

Нумерация элементов массива `valarray` начинается с 0. Проверка выхода за пределы диапазона не производится.

Кроме приведенных выше, определены операции обобщенной индексации, или извлечения подмассивов из `valarray`¹:

```
valarray<T>      operator[](slice) const;
slice_array<T>  operator[](slice);
valarray<T>      operator[](const gslice&) const;
gslice_array<T> operator[](const gslice&);
valarray<T>      operator[](const valarray<bool>&) const;
mask_array<T>   operator[](const valarray<bool>&);
valarray<T>      operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);
```

Формы операций с `const` возвращают подмножество как новый массив типа `valarray`, а формы без `const` — как массив соответствующего типа.

Определены унарные операции `+`, `-`, `-` и `!`, бинарные операции `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `<<`, `>>`, а также соответствующие сложные присваивания (`+=`, `*=` и т. д.), вторым операндом которых может быть как число, так и массив, например:

```
valarray<T>& operator+= (const T&);
valarray<T>& operator+= (const valarray<T>&);
```

В первом случае каждый элемент массива увеличивается на заданное число, а во втором выполняется почленное сложение соответствующих элементов двух массивов. Операции применимы к массивам, для элементов которых они являются допустимыми. Если размеры массивов, участвующих в операции, не совпадают, результат не определен. Операции `<<` и `>>` обозначают поразрядный сдвиг каждого элемента (то есть не сдвиг элементов и не ввод/вывод).

Примеры:

```
v4 += 1;      // Результат – 5,5,4,7,2 (v4 описан выше)
v2 += v3;     // v2[i] = v2[i] + v3[i] для i = 1, ..., 100
v4 >>= 1;     // Результат – 2,2,2,3,1
```

В классе `valarray` определены следующие методы:

```
size_t size() const; // Размер массива
T sum() const;       // Сумма элементов
T min() const;       // Наименьшее значение
T max() const;       // Наибольшее значение
// Логический сдвиг на i элементов (влево, если i>0):
valarray<T> shift(int i) const;
// Циклический сдвиг на i элементов (влево, если i>0):
valarray<T> cshift(int i) const;
// Применение функции func к каждому элементу массива:
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
// повторная инициализация (все элементы заменяются на
// значение по умолчанию, размер вектора – sz:
void resize(size_t sz, T c = T());
```

¹ Пояснения действия этих операций даны в следующем разделе.

Функции сдвига `shift` и `cshift` и применение функции `apply` возвращают новый массив, при этом исходный остается без изменений. При логическом сдвиге освободившиеся значения заполняются значением по умолчанию для данного типа элементов.

Повторная инициализация с помощью `resize` не предназначена для того, чтобы динамически изменять размер существующего массива. Предполагается, что `valarray` используется для быстрой работы с числами и его размер остается постоянным в течение всего времени его использования.

К массивам можно применять функции `abs`, `acos`, `asin`, `atan`, `atan2`, `cos`, `cosh`, `exp`, `log`, `log10`, `pow`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, например:

```
v2 = cos(v3); // v2[i] = cos(v3[i]) для i = 1. ..., 100
```

Поскольку элементы массива `valarray` представляют собой последовательность, к нему применимы все стандартные алгоритмы и итераторы, рассмотренные в предыдущих разделах. Но более эффективно работать с массивом через срезы и подмножества, рассматриваемые в следующем разделе.

Для `valarray` определены логические операции `<`, `>`, `==`, `!=` и т. д., дающие в результате массив типа `bool`, каждый элемент которого равен результату операции, примененной к паре соответствующих элементов сравниваемых массивов, например:

```
template<class T> valarray<bool>
    operator== (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool>
    operator!= (const valarray<T>&, const valarray<T>&);
```

Если размеры массивов, участвующих в операции, не совпадают, результат не определен.

Вспомогательные классы

Класс `slice` (**срез**) позволяет задать подмножество индексов элементов массива:

```
class slice{
public:
    slice();
    slice(size_t start, size_t size, size_t stride);
    size_t start() const;    // индекс первого элемента
    size_t size() const;    // число элементов
    size_t stride() const;  // шаг
};
```

Шаг — это расстояние между двумя элементами среза. Например, `slice(3, 4, 2)` задает срез, состоящий из элементов массива с номерами 3, 5, 7, 9, а `slice(1, 3, 4)` — с номерами 1, 5, 9. Таким образом, с помощью срезов можно представить массив как матрицу, задав в нем строку или столбец.

Теперь можно привести пример операции индексации подмножеств массива `valarray`, описанной в предыдущем разделе: оператор `a[slice(1, 5, 3)] = b` выполнит присваивание массива `b` только элементам массива `a`, заданным срезом.

Для удобной работы с подмножеством элементов массива введен шаблон класса `slice_array`. Самостоятельные объекты этого класса не создаются, все обращения к ним на самом деле переадресуются на исходный массив:

```
slice_array<double>& s1 = v2[slice(0, v.size() / 10, 10)];
// Увеличить на 5 первый столбец матрицы 10 X 10:
s1 += 5;
slice_array<double>& s2 = v2[slice(0, v.size() / 10, 1)];
// Увеличить в 2 раза каждый элемент первой строки матрицы 10 X 10:
s2 *= 2;
```

В шаблоне `slice_array` определены сложные операции присваивания (`+=`, `*=`, `>=` и т. д.), например:

```
void operator+= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
```

Они выполняют указанные действия с аргументом и элементами исходного массива, заданными срезом, на которые ссылается `slice_array`. Кроме того, определена операция присваивания скалярного значения, которая выполняет присваивание всем элементам исходного массива, входящим в срез.

Стандартная библиотека не предоставляет средств для работы с матрицами, но с помощью абстракций, задаваемых срезами, можно эффективно работать с массивом `valarray` как с матрицей произвольной размерности.

Если требуется работать с подмножеством, которое нельзя задать одним срезом, используется шаблон `gslice` — **обобщенный срез**:

```
class gslice{
public:
    gslice():
    gslice(size_t s, const valarray<size_t>& len,
           const valarray<size_t>& d);
    size_t start() const; // индекс первого элемента
    valarray<size_t> size() const; // число элементов в измерении
    valarray<size_t> stride() const; // шаги для индекса[0]. индекса[1]...
};
```

Обобщенный срез задает несколько срезов. Индекс начального элемента указывается первым параметром `s`, размеры срезов передаются через массив `len`, а шаги — через массив `d`. Размерности массивов, определяющие количество задаваемых срезов, должны совпадать.

Например, если задать параметры

```
start = 0
len = {2, 3}
d = {4, 1}
```

получим последовательность индексов массива

```
k = (0,1) × 4 + (0,1,2) × 1
```

которые дадут в результате матрицу 2×3 из левого верхнего угла матрицы 3×4, расположенной в памяти по столбцам (первый срез с длиной 2 и шагом 4 описы-

```
valarray<int> v7 = v4[index]:  
// v7[0] = v4[3], v7[1] = v4[2], v7[2] = v4[4], v7[3] = v4[0], v7[4] = v4[1]
```

Число элементов массива, используемого в качестве индекса, не должно превышать число элементов в индексируемом массиве.

Перечисленные классы — `slice_array`, `gslice_array`, `mask_array` и `indirect_array` — нельзя создавать напрямую и копировать. Они прозрачны для пользователя и создаются в результате использования в качестве индекса массива `valarray` срезов, обобщенных срезов, логических массивов `valarray` и массивов типа `size_t` соответственно.

ПРИЛОЖЕНИЕ 6

Функции стандартной библиотеки

Приложение содержит две части. В первой части функции сгруппированы по заголовочным файлам, в которых они описаны, во второй части приведен алфавитный перечень функций с описанием.

Состав заголовочных файлов

Заголовочный файл <ctype.h> (<cctype>) — функции классификации и преобразования символов

isalnum	Проверяет, является ли символ буквой или цифрой
isalpha	Проверяет, является ли символ буквой
iscntrl	Проверяет, является ли символ управляющим
isdigit	Проверяет, является ли символ цифрой
isgraph	Проверяет, является ли символ видимым
islower	Проверяет, является ли символ буквой нижнего регистра
isprint	Проверяет, является ли символ печатаемым
ispunct	Проверяет, является ли символ символом пунктуации
isspace	Проверяет, является ли символ разграничительным
isupper	Проверяет, является ли символ буквой верхнего регистра
iswalnum	Проверяет, является ли символ буквой или цифрой
iswalpha	Проверяет, является ли символ буквой
iswcntrl	Проверяет, является ли символ управляющим
iswctype	Проверяет многобайтный символ

Упражнения к части III

Стандартные потоки

Выполнить задания из раздела «Циклические вычислительные процессы» (см. упражнения к первой части) с использованием стандартных потоков, манипуляторов и форматирующих методов.

Файловые потоки

Выполнить задания из раздела «Функции и файлы» (см. упражнения к первой части) с использованием файловых потоков и методов обработки ошибок.

Потоки и типы, определенные пользователем

Дополнить программы из раздела «Классы» (см. упражнения ко второй части) перегруженными операциями ввода/вывода в поток для созданных классов.

Строки

Выполнить задания из раздела «Функции библиотеки для работы со строками и символами» (см. упражнения к первой части) с использованием класса `string`.

Контейнерные классы

Выполнить задания из разделов «Одномерные и двумерные массивы», «Структуры» и «Динамические структуры данных» (см. упражнения к первой части) с использованием соответствующих контейнерных классов библиотеки.

Алгоритмы

Выполнить задания из раздела «Одномерные массивы» (см. упражнения к первой части) с использованием контейнерных классов и алгоритмов библиотеки.

Класс `valarray`

Выполнить задания из разделов «Одномерные массивы» и «Двумерные массивы» (см. упражнения к первой части) с использованием класса `valarray` и срезов.

ГЛАВА 16

Другие средства стандартной библиотеки

Кроме рассмотренных, стандартная библиотека C++ содержит средства поддержки языка, диагностики, локализации, работы с комплексными числами, более 200 функций, унаследованных из стандартной библиотеки C, а также типов для их поддержки и макросов.

Средства поддержки языка включают описания функций и типов, которые должны присутствовать при выполнении программы. Они включают в себя поддержку запуска и завершения программы, операций `new` и `delete`, описание свойств встроенных типов данных, динамическую идентификацию типов, обработку исключений и другие средства времени выполнения. Эти средства представлены в заголовочных файлах `<cstdlib>`, `<limits>`, `<climits>`, `<float>`, `<cstdlib>`, `<new>`, `<typeinfo>`, `<exception>`, `<csdarg>`, `<setjmp>`, `<ctime>`, `<csignal>`.

В библиотеке (файл `<limits>`) определен шаблон класса `numeric_limits`, в котором определяются свойства типа данных, зависящие от реализации: диапазон, точность, наличие знака и т. д. Для каждого из основных типов данных определена специализация шаблона `numeric_limits`. Эти же характеристики определяются и в константах, описанных в заголовочных файлах `<climits>` и `<float>`, унаследованных из библиотеки C, например:

`FLT_DIG` — количество верных значащих цифр в величине типа `float`;

`FLT_MIN` — минимальное нормализованное число с плавающей точкой, которое может быть представлено типом `float`;

`UINT_MAX` — максимальное значение, которое может принимать величина типа `unsigned int` (должно быть не менее 65 535).

Полностью эти константы описаны в приложении 5.

Для обработки ошибок при выделении памяти в библиотеке описан класс `bad_alloc`, тип функций-обработчиков ошибок `new_handler` и функция установки нового обработчика `set_new_handler`.

Механизм обработки не может выделить функцию-обработчик, либо по изводного от него, л программы. Функция обработчика вместо операций `new` и `delete`

Для поддержки динамич `type_info`, рассмотрен классы исключений

Все стандартные ис. файле `<exception>` кл:

Средства диагностик используются для ди программы. Все ошибки принципе можно обн: ния (`runtime_error`). В чений, которая должн мистом:

exce

Рис. 16

Ниже кратко описаны:

- `bad_alloc` — ошибка
- `bad_cast` — неправил
- Преобразования ти
- `bad_typeid` — операци
- `bad_exception` — поро: ключений которой о

ГЛАВА 16

Другие средства стандартной библиотеки

Кроме рассмотренных, стандартная библиотека C++ содержит средства поддержки языка, диагностики, локализации, работы с комплексными числами, более 200 функций, унаследованных из стандартной библиотеки C, а также типов для их поддержки и макросов.

Средства поддержки языка включают описания функций и типов, которые должны присутствовать при выполнении программы. Они включают в себя поддержку запуска и завершения программы, операций `new` и `delete`, описание свойств встроенных типов данных, динамическую идентификацию типов, обработку исключений и другие средства времени выполнения. Эти средства представлены в заголовочных файлах `<cstddef>`, `<limits>`, `<climits>`, `<cfloat>`, `<cstdlib>`, `<new>`, `<typeinfo>`, `<exception>`, `<cstdint>`, `<setjmp>`, `<ctime>`, `<csignal>`.

В библиотеке (файл `<limits>`) определен шаблон класса `numeric_limits`, в котором определяются свойства типа данных, зависящие от реализации: диапазон, точность, наличие знака и т. д. Для каждого из основных типов данных определена специализация шаблона `numeric_limits`. Эти же характеристики определяются и в константах, описанных в заголовочных файлах `<limits>` и `<cfloat>`, унаследованных из библиотеки C, например:

`FLT_DIG` — количество верных значащих цифр в величине типа `float`;

`FLT_MIN` — минимальное нормализованное число с плавающей точкой, которое может быть представлено типом `float`;

`UINT_MAX` — максимальное значение, которое может принимать величина типа `unsigned int` (должно быть не менее 65 535).

Полностью эти константы описаны в приложении 5.

Для обработки ошибок при выделении памяти в библиотеке описан класс `bad_alloc`, тип функций-обработчиков ошибок `new_handler` и функция установки нового обработчика `set_new_handler`.

Механизм обработки ошибок выделения памяти следующий: если операция `new` не может выделить требуемое количество динамической памяти, она вызывает функцию-обработчик типа `new_handler`, который должен либо попытаться освободить память, либо породить исключение в виде объекта класса `bad_alloc` или производного от него, либо вызвать функцию `abort` или `exit` для завершения программы. Функция `set_new_handler` используется для установки собственного обработчика вместо стандартного. Как правило, она применяется при перезагрузке операций `new` и `delete`.

Для поддержки динамической идентификации типов в библиотеке определен класс `type_info`, рассмотренный в разделе «Преобразования типов» на с. 231, а также классы исключений `bad_cast` и `bad_typeid`.

Все стандартные исключения являются потомками описанного в заголовочном файле `<exception>` класса `exception`.

Средства диагностики включают в себя стандартные классы и функции, которые используются для диагностики ошибок, возникающих в процессе работы программы. Все ошибки подразделяются на *логические* (`logic_error`), которые в принципе можно обнаружить до запуска программы, и *ошибки времени выполнения* (`runtime_error`). В библиотеке определена простая иерархия классов исключений, которая должна служить основой для исключений, создаваемых программистом:

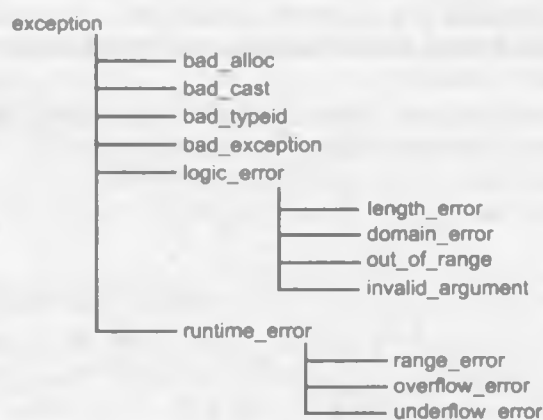


Рис. 16.1. Иерархия классов стандартных исключений

Ниже кратко описаны эти классы:

- ❑ `bad_alloc` — ошибка при динамическом распределении памяти с помощью `new`;
- ❑ `bad_cast` — неправильное использование оператора `dynamic_cast` (см. раздел «Преобразования типов», с. 231);
- ❑ `bad_typeid` — операция `typeid` не может определить тип операнда (см. с. 238);
- ❑ `bad_exception` — порождается, если при вызове функции, в спецификации исключений которой оно указано, произошло неожиданное исключение;

- ❑ `length_error` — попытка создания объекта, большего, чем максимальный размер типа `size_t`;
- ❑ `domain_error` — нарушение внутренних условий перед выполнением действия;
- ❑ `out_of_range` — попытка вызова функции с параметром, не входящим в допустимые значения;
- ❑ `invalid_argument` — попытка вызова функции с неверным параметром;
- ❑ `range_error` — неправильный результат выполнения;
- ❑ `overflow_error` — арифметическое переполнение (результат слишком большой для указанного типа данных);
- ❑ `underflow_error` — исчезновение порядка (результат слишком маленький для указанного типа данных).

Для использования этих классов необходимо подключить заголовочный файл `<stdexcept>`.

Средства локализации включают в себя упорядоченную по категориям информацию, специфичную для разных стран, например, представление даты и времени, формат национальной валюты, представление чисел и символов. Категории представлены классами. Программист может использовать их для обеспечения универсальности программы при ее работе в различных странах. Описания средств локализации находятся в заголовочных файлах `<locale>` и `<ctype>`.

Для работы с комплексными числами библиотека предоставляет шаблон класса `complex`, его специализации для типов `float`, `double` и `long double`, а также многочисленные функции. Все это богатство становится доступным при подключении заголовочного файла `<complex>`.

Функции библиотеки описаны в приложении 6, константы, макросы и типы — в приложении 5, заголовочные файлы — в приложении 4.

Упражнения к части III

Стандартные потоки

Выполнить задания из раздела «Циклические вычислительные процессы» (см. упражнения к первой части) с использованием стандартных потоков, манипуляторов и форматирующих методов.

Файловые потоки

Выполнить задания из раздела «Функции и файлы» (см. упражнения к первой части) с использованием файловых потоков и методов обработки ошибок.

Потоки и типы, определенные пользователем

Дополнить программы из раздела «Классы» (см. упражнения ко второй части) перегруженными операциями ввода/вывода в поток для созданных классов.

Строки

Выполнить задания из раздела «Функции библиотеки для работы со строками и символами» (см. упражнения к первой части) с использованием класса `string`.

Контейнерные классы

Выполнить задания из разделов «Одномерные и двумерные массивы», «Структуры» и «Динамические структуры данных» (см. упражнения к первой части) с использованием соответствующих контейнерных классов библиотеки.

Алгоритмы

Выполнить задания из раздела «Одномерные массивы» (см. упражнения к первой части) с использованием контейнерных классов и алгоритмов библиотеки.

Класс `valarray`

Выполнить задания из разделов «Одномерные массивы» и «Двумерные массивы» (см. упражнения к первой части) с использованием класса `valarray` и срезов.

Послесловие

В этой книге большое внимание уделялось описанию основ языка C++. Владение ими является базой, необходимой для написания программ. Но создать эффективную, легко читаемую и расширяемую программу невозможно без знания механизмов реализации возможностей языка и способов их взаимодействия. Кроме того, как утверждает Скотт Майерс, «работа на C++ *при правильном его использовании* способна доставить удовольствие». Неполное же понимание функционирования языка, напротив, приводит к созданию программ, полных ошибок и не поддающихся сопровождению.

Поведение языка определяют цели, поставленные при его разработке. Для C++ основными целями являлись эффективность, применимость для широкого спектра реальных задач, а также совместимость с языком C, традиционными инструментами и окружением.

История создания языка и причины включения в него тех возможностей, которыми он обладает в настоящий момент, подробно описаны в книге Б. Страуструпа «Дизайн и эволюция языка C++» [18]. Изучение этой книги поможет понять, отчего средства языка реализованы именно так, а не иначе. Полезные практические советы по грамотному написанию эффективных программ содержатся в книгах [17], [13] и [9]. Для получения полной справочной информации о языке и стандартной библиотеке лучше всего пользоваться стандартом ISO/IEC 14882:1998 (его предварительные варианты, мало отличающиеся от окончательного, можно найти в Интернете).

Желаю всем читателям дальнейших успехов в освоении и использовании C++!

Литература

1. Аммерааль Л. STL для программистов на C++. — М.: ДМК, 2000. — 240 с.
2. Бабз Б. Просто и ясно о Borland C++. — М.: БИНОМ, 1995. — 400 с.
3. Багдд Т. Объектно-ориентированное программирование в действии. — СПб.: Питер, 1997. — 464 с.
4. Баженова И. Ю. Visual C++ 6.0. Уроки программирования. — М.: Диалог-МИФИ, 1999. — 416 с.
5. Дэвис С. C++ для «чайников». — К.: Диалектика, 1996. — 304 с.
6. Брукс Ф. Мифический человеко-месяц, или как создаются программные комплексы. — М.: «Символ-Плюс», 2000. — 304 с.
7. Буч Г. Объектно-ориентированный анализ и проектирование с примерами на C++. — М.: БИНОМ, 1998. — 560 с.
8. Вирт Н. Алгоритмы + структуры данных = программы. — М.: Мир, 1985. — 406 с.
9. Голуб А. И. С и C++. Правила программирования. — М.: БИНОМ, 1996. — 272 с.
10. Карпов Б., Баранова Т. C++: специальный справочник. — СПб.: Питер, 2000. — 480 с.
11. Ласло М. Вычислительная геометрия и компьютерная графика на C++. — М.: БИНОМ, 1997. — 304 с.
12. Луис Д. С и C++. Справочник. — М.: Восточная книжная компания, 1997. — 592 с.
13. Мейерс С. Эффективное использование C++. — М.: ДМК, 2000. — 240 с.
14. Подбельский В. В. Язык Си++; Учебное пособие. — М.: Финансы и статистика, 1996. — 560 с.
15. Пол И. Объектно-ориентированное программирование с использованием C++. — К.: ДияСофт, 1995. — 480 с.
16. Романовская Л. М. и др. Программирование в среде Си для ПЭВМ ЕС. — М.: Финансы и статистика, 1992. — 352 с.
17. Страуструп Б. Язык программирования C++. — СПб.: БИНОМ, 1999. — 991 с.
18. Страуструп Б. Дизайн и эволюция языка C++. — М.: ДМК, 2000. — 448 с.
19. Шилдт Г. Теория и практика C++. — СПб.: BHV, 1996. — 416 с.
20. Неформальное введение в C++ и Turbo Vision. — Галерея «Петрополь», 1992. — 381 с.

ПРИЛОЖЕНИЕ 1

Операции языка C++

Операции приведены в порядке убывания приоритета. Операции с разными приоритетами разделены чертой.

Операция	Краткое описание
Унарные операции	
::	доступ к области видимости
~	выбор
>>	выбор
[]	индексация
{ }	вызов функции
<тип>{ }	конструирование
++	постфиксный инкремент
--	постфиксный декремент
typeid	идентификация типа
dynamic_cast	преобразование типа с проверкой на этапе выполнения
static_cast	преобразование типа с проверкой на этапе компиляции
reinterpret_cast	преобразование типа без проверки
const_cast	константное преобразование типа
sizeof	размер объекта или типа
--	префиксный декремент
++	префиксный инкремент
-	поразрядное отрицание
!	логическое отрицание

ПРИЛОЖЕНИЕ 1

Операции языка C++

Операции приведены в порядке убывания приоритета. Операции с разными приоритетами разделены чертой.

Операция	Краткое описание
Унарные операции	
[]	доступ к области видимости
()	выбор
->	выбор
[]	индексация
()	вызов функции
<тип>()	конструирование
++	постфиксный инкремент
--	постфиксный декремент
typeid	идентификация типа
dynamic_cast	преобразование типа с проверкой на этапе выполнения
static_cast	преобразование типа с проверкой на этапе компиляции
reinterpret_cast	преобразование типа без проверки
const_cast	константное преобразование типа
sizeof	размер объекта или типа
--	префиксный декремент
++	префиксный инкремент
~	поразрядное отрицание
!	логическое отрицание

Операция	Краткое описание
-	арифметическое отрицание (унарный минус)
+	унарный плюс
&	взятие адреса
*	разадресация
new	выделение памяти
delete	освобождение памяти
(<тип>)	преобразование типа
*	выбор
->*	выбор
Бинарные и тернарные операции	
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
=	равно
!=	не равно
&	поразрядная конъюнкция (И)
^	поразрядное исключающее ИЛИ
	поразрядная дизъюнкция (ИЛИ)
&&	логическое И
	логическое ИЛИ
? :	условная операция (тернарная)
=	присваивание
*=	умножение с присваиванием
/=	деление с присваиванием

Операция	Краткое описание
-	арифметическое отрицание (унарный минус)
+	унарный плюс
&	взятие адреса
*	разадресация
new	выделение памяти
delete	освобождение памяти
(<тип>)	преобразование типа
*	выбор
->*	выбор
Бинарные и тернарные операции	
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
=	равно
!=	не равно
&	поразрядная конъюнкция (И)
^	поразрядное исключающее ИЛИ
	поразрядная дизъюнкция (ИЛИ)
&&	логическое И
	логическое ИЛИ
? :	условная операция (тернарная)
=	присваивание
*=	умножение с присваиванием
/=	деление с присваиванием

Операция	Краткое описание
%=	остаток от деления с присваиванием
+=	сложение с присваиванием
-=	вычитание с присваиванием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное И с присваиванием
=	поразрядное ИЛИ с присваиванием
^=	поразрядное исключающее ИЛИ с присваиванием
throw	исключение
»	последовательное вычисление

ПРИЛОЖЕНИЕ 2

Спецификации формата для функций семейства printf

Спецификация	Пояснения
c	аргумент рассматривается как отдельный символ
d, i	аргумент преобразуется к десятичному виду
e, E	аргумент, рассматриваемый как переменная типа float или double, преобразуется в десятичную форму в виде [-]m.nnnnnne[+-]xx, где длина строки из n определяется указанной точностью (см. след. табл.). Точность по умолчанию равна 6
f	аргумент, рассматриваемый как переменная типа float или double, преобразуется в десятичную форму в виде [-]mmm.nnnnn, где длина строки из n определяется указанной точностью (см. след. табл.). Точность по умолчанию равна 6
g, G	используется формат %e или %f, который короче; незначащие нули не печатаются
o	аргумент преобразуется в беззнаковую восьмеричную форму (без лидирующего нуля)
p	вывод указателя в шестнадцатеричном формате (эта спецификация не входит в стандарт, но она существует практически во всех реализациях)
s	аргумент является строкой: символы строки печатаются до тех пор, пока не будет достигнут нулевой символ или не будет напечатано количество символов, указанное в спецификации точности
u	аргумент преобразуется в беззнаковую десятичную форму
x, X	аргумент преобразуется в беззнаковую шестнадцатеричную форму (без лидирующих 0x)
%	выводится символ %

Модификаторы формата

Модификаторы формата применяются для управления шириной поля, отводимого для размещения значения. Модификаторы — это одно или два числа, первое из которых задает минимальное количество позиций, отводимых под число, а второе — сколько из этих позиций отводится под дробную часть числа (точность). Если указанного количества позиций для размещения значения недостаточно, автоматически выделяется большее количество позиций:

- %-minC или %minC;
- %-min.precisionC или %min.precisionC.

Здесь C — спецификация формата из приведенной выше таблицы, min — число, задающее минимальную ширину поля. Смысл модификатора precision, также задаваемого десятичным числом, зависит от спецификации формата, с которой он используется:

- при выводе строки (спецификация %s) precision указывает максимальное число символов для вывода;
- при выводе вещественного числа (спецификации %f или %e) precision указывает количество цифр после десятичной точки;
- при выводе целого числа (спецификации %d или %i), precision указывает минимальное количество выводимых цифр. Если число представляется меньшим числом цифр, чем указано в precision, выводятся ведущие (начальные) нули.

Символ минус (-) указывает на то, что значение выравнивается по левому краю и, если нужно, дополняется пробелами справа. При отсутствии минуса значение выравнивается по правому краю и дополняется пробелами слева.

Перед спецификацией могут использоваться префиксы l и h, например, %lf, %hu.

Префикс h с типами d, i, o, x и X указывает на то, что тип аргумента short int, а с типом u — short unsigned int.

Префикс l с типами d, i, o, x и X указывает на то, что тип аргумента long int, с типом u — long unsigned int, а с типами e, E, f, g и G — что тип аргумента double, а не float.

Пример:

```
#include <stdio.h>
int main(){
    int int1 = 45, int2 = 13;
    float f = 3.621;
    double dbl = 2.23;
    char ch = 'z', *str = "ramambahari";
    printf("int1 = %d| int2 = %3d| int2 = %-4d|\n", int1, int2, int2);
    printf("int1 = %X| int2 = %3x| int2 = %4o|\n", int1, int2, int2);
    printf("f = %f| f = %4.2f| f = %6.1f|\n", f, f, f);
    printf("f = %g| f = %e| f = %E|\n", f, f, f);
    printf("dbl = %5.2lf| dbl = %e| dbl = %4.1G|\n", dbl, dbl, dbl);
    printf("ch = %c| ch = %3c|\n", ch, ch);
```

```
printf("str = %14s|\nstr = %-14s|\nstr = %s|\n", str, str, str);  
return 0;  
}
```

Результат работы программы:

```
int1 = 45| int2 =    13| int2 = 13 |  
int1 = 20| int2 =     d| int2 = 15 |  
f = 3.621000| f = 3.62 | f = 3.6 |  
f = 3.621 | f = 3.621000e+000 | f = +3.621000E+000|  
dbl = 2.23 | dbl = 2.230000e+000 | dbl = 2 |  
ch = z| ch = z |  
str = ramambahari|  
str = ramambahari|  
str = ramambahari|
```

ПРИЛОЖЕНИЕ 3

Арифметические преобразования типов

Преобразования типов выполняются, если операнды, входящие в выражения, имеют различные типы. Ниже приведена последовательность преобразований.

- ❑ Любые операнды типа `char`, `unsigned char` или `short` преобразуются к типу `int` по правилам:
 - `char` расширяется нулем или знаком в зависимости от умолчания для `char`;
 - `unsigned char` расширяется нулем;
 - `signed char` расширяется знаком;
 - `short`, `unsigned short` и `enum` при преобразовании не изменяются.
 - Затем любые два операнда становятся либо `int`, либо `float`, `double` или `long double`.
- ❑ Если один из операндов имеет тип `long double`, то другой преобразуется к типу `long double`.
- ❑ Если один из операндов имеет тип `double`, то другой преобразуется к типу `double`.
- ❑ Если один из операндов имеет тип `float`, то другой преобразуется к типу `float`.
- ❑ Иначе, если один из операндов имеет тип `unsigned long`, то другой преобразуется к типу `unsigned long`.
- ❑ Иначе, если один из операндов имеет тип `long`, то другой преобразуется к типу `long`.
- ❑ Иначе, если один из операндов имеет тип `unsigned`, то другой преобразуется к типу `unsigned`.
- ❑ Иначе оба операнда должны иметь тип `int`.

Тип результата тот же, что и тип участвующих в выражении операндов.

ПРИЛОЖЕНИЕ 4

Заголовочные файлы стандартной библиотеки

Ввод/вывод

<cstdio>	Функции ввода/вывода в стиле C
<cstdlib>	Функции для работы с символами
<wchar>	Функции ввода/вывода в стиле C для многобайтных символов
<fstream>	Файловые потоки
<iomanip>	Манипуляторы
<ios>	Базовые классы потоков ввода/вывода
<iosfwd>	Предварительные объявления средств ввода/вывода
<iostream>	Стандартные объекты и операции с потоками ввода/вывода
<istream>	Входные потоки
<ostream>	Выходные потоки
<sstream>	Строковые потоки
<streambuf>	Буферизация потоков

Строки

<cctype>	Функции классификации символов
<cwctype>	
<wctype>	
<cstring>	Функции для работы со строками в стиле C
<cstdlib>	Функции для работы с символами
<string>	Строковый класс

Контейнерные классы

<bitset>	Битовое множество
<deque>	Двусторонняя очередь
<list>	Двусвязный список
<map>	Словарь, словарь с дубликатами
<queue>	Очередь, очередь с приоритетами
<set>	Множество, множество с дубликатами
<stack>	Стек
<vector>	Одномерный массив

Алгоритмы, итераторы, утилиты

<algorithm>	Алгоритмы
<cstdlib>	Сортировка и поиск из библиотеки C
<ctime>	Дата и время в стиле C
<functional>	Функциональные объекты
<iterator>	Итераторы
<memory>	Распределители памяти для контейнеров
<utility>	Операторы и пары

Диагностика, поддержка языка, локализация

<cassert>	Макрос assert
<cerrno>	Обработка ошибок в стиле C
<cfloat>	Макросы предельных значений в стиле C
<climits>	Макросы предельных значений в стиле C
<locale>	Локализация в стиле C
<csignal>	Обработка сигналов в стиле C
<cstdarg>	Поддержка функций с переменным числом аргументов
<cstddef>	Поддержка языка библиотеки C
<cstdlib>	Завершение программы
<ctime>	Системные часы
<exception>	Поддержка исключений
<limits>	Числовые ограничения
<locale>	Классы локализации
<new>	Работа с динамической памятью
<stdexcept>	Стандартные исключения
<typeinfo>	Динамическая идентификация типов

Численные расчеты

<code><math></code>	Математические функции
<code><complex></code>	Комплексные числа
<code><cstdlib></code>	Случайные числа в стиле C
<code><numeric></code>	Числовые операции
<code><valarray></code>	Классы для численных расчетов

ПРИЛОЖЕНИЕ 5

Константы, макросы и типы данных стандартной библиотеки

Перечисленные в этом приложении константы, макросы и типы используются в основном в функциях стандартной библиотеки, унаследованных из языка C.

assert	<assert.h>

void assert(int expression);

Макрос assert прерывает выполнение программы, если значение выражения expression является ложным (нулевым). При завершении работы выводится сообщение об ошибке вида: Assertion failed: expression, file <имя_файла>, line <номер_строки>.

BUFSIZ	<stdio.h>

Константа BUFSIZ применяется для задания размера в байтах буфера потока данных функциями fopen(), freopen(), setbuf(). При открытии потока данных к нему автоматически присоединяется буфер длиной BUFSIZ. Минимальный размер BUFSIZ — 255 байтов.

CHAR_BIT	<limits.h>

В константе CHAR_BIT хранится количество бит наиболее короткого типа данных (по стандарту — типа char). Минимальное значение CHAR_BIT — 8.

cin, cout, cerr, clog	<iostream.h>

Объекты потоковых классов, связанные со стандартными периферийными устройствами.

Объект	Поток данных	Класс
cin	stdin	istream
cout	stdout	ostream
cerr	stderr	ostream
clog	stderr (буферизованный)	ostream

CHAR_MAX, CHAR_MIN

<limits.h>

В константе CHAR_MAX содержится максимальное значение, которое может иметь тип char. Если в конкретной реализации поддерживается тип signed char, то значение CHAR_MAX идентично значению SCHAR_MAX, в противном случае — UCHAR_MAX.

В константе CHAR_MIN содержится минимальное значение, которое может иметь тип char. Если в конкретной реализации поддерживается тип signed char, то значение CHAR_MIN идентично значению SCHAR_MIN, в противном случае — UCHAR_MIN.

clock_t

<time.h>

Тип данных clock_t предназначен для представления времени:

```
typedef long clock_t;
```

Значение этого типа возвращает функция clock().

CLOCKS_PER_SEC

<time.h>

В константе CLOCKS_PER_SEC содержится количество циклов процессора в секунду. Функция clock() возвращает время, выраженное в циклах процессора. Чтобы перевести его в секунды, его необходимо разделить на CLOCKS_PER_SEC.

DBL...

<float.h>

(см. FLT...)

div_t

<float.h>

Структура div_t служит для определения типа возвращаемого значения функции div(). Поле структуры quot содержит частное, а поле rem — остаток от деления.

```
typedef struct{
    int quot;
    int rem;
}div_t;
```

EDOM

<errno.h>

Макрос EDOM задает отличную от нуля целочисленную константу, что позволяет обнаруживать ошибку области, то есть ошибку, возникающую при выходе аргумента за область значений, для которой определена математическая функция.

EOF

<stdio.h>

Константа EOF сообщает об окончании файла (представляет собой отрицательное целое число).

ERANGE

<errno.h>

Макрос ERANGE задает отличную от нуля целочисленную константу, используемую для фиксации ошибки диапазона. Ошибка диапазона возникает в случае, если результат функции не может быть представлен объектом типа double. В этом случае переменная errno (см. ниже) устанавливается в значение ERANGE.

errno

<errno.h>

Глобальная целочисленная переменная errno используется для обработки ошибок, возникающих при использовании математических функций (может содержать номер ошибки, который дает дополнительную информацию о последней из обнаруженных ошибок).

EXIT_FAILURE

<stdlib.h>

Код состояния, который может использоваться при обращении к библиотечной функции exit() для сообщения об ошибке.

EXIT_SUCCESS

<stdlib.h>

Код состояния, который может использоваться при обращении к библиотечной функции exit() для сообщения об успешном завершении программы.

FILE

<stdio.h>

Тип данных FILE содержит информацию, необходимую для выполнения операций с файлом. Структура этого типа данных зависит от конкретной реализации, ниже приведено одно из возможных описаний.

```
typedef struct{
    int level;           // флаг состояния буфера
    unsigned flags;      // флаг состояния файла
    char fd;             // префикс файла
    unsigned char hold;  // непереданный символ
    int bsize;           // размер внутреннего буфера
    unsigned char_FAR *buffer; /* значение указателя для доступа внутри
буфера: задает начало буфера, начало строки или текущее значение указателя
внутри буфера в зависимости от режима буферизации */
    unsigned char_FAR *curp; /* текущее значение указателя для доступа внутри
буфера: задает текущую позицию в буфере для обмена с программой */
    unsigned istemp;     // флаг временного файла
    short token;         // маркер действительности файла
}FILE;
```

filebuf

<fstream.h>

Специализация шаблона класса `basic_filebuf` для типа данных `char`.

```
typedef basic_filebuf<char> filebuf;
```

FILENAME_MAX

<stdio.h>

В константе `FILENAME_MAX` задается ограничение на длину имени файла.

float_round_style

<stdio.h>

Перечисляемый тип `float_round_style` определяет константы для различных способов округления при сложении чисел с плавающей точкой (см. `numeric_limits`).

```
enum float_round_style{
    round_indeterminate,
    round_toward_zero,
    round_to_nearest,
    round_toward_infinity,
    round_toward_neg_infinity
};
```

FLT_

<float.h>

Следующие константы определены также для типов `double` (`DBL_`) и `long double` (`LDBL_`):

Константа	Описание
FLT_DIG	Количество верных десятичных цифр в объекте типа <code>float</code>
FLT_EPSILON	Минимальное возможное значение переменной типа <code>float</code> , такое, что $1.0 + \text{FLT\_EPSILON} \neq 1.0$
FLT_MANT_DIG	Количество цифр по основанию <code>FLT_RADIX</code> в мантиссе для чисел типа <code>float</code>
FLT_MAX	Максимальное число с плавающей точкой типа <code>float</code>
FLT_MAX_10_EXP	Максимальное целое число, для которого справедливо, что число, полученное при возведении числа 10 в степень <code>FLT_MAX_10_EXP</code> , может быть представлено объектом типа <code>float</code>
FLT_MAX_EXP	Максимальное целое число, для которого справедливо, что число, полученное при возведении числа <code>FLT_RADIX</code> в степень $(\text{FLT\_MAX\_EXP} - 1)$, может быть представлено объектом типа <code>float</code>
FLT_MIN	Минимальное нормализованное число с плавающей точкой, которое может быть представлено типом <code>float</code>
FLT_MIN_10_EXP	Минимальное целое число, для которого справедливо, что число, полученное при возведении числа 10 в степень <code>FLT_MIN_10_EXP</code> , может быть представлено объектом типа <code>float</code>

Константа	Описание
FLT_MIN_EXP	Минимальное целое число, для которого справедливо, что число, полученное при возведении числа FLT_RADIX в степень (FLT_MIN_EXP - 1), может быть представлено объектом типа float

Следующие константы не имеют соответствия для типов double и long double.

Константа	Описание
FLT_RADIX	Основание представления экспоненты для чисел с плавающей запятой
FLT_ROUNDS	Способ округления, используемый при выполнении операций над числами с плавающей запятой

FOPEN_MAX	<stdio.h>

В константе FOPEN_MAX задается максимальное число открытых потоков. Оно должно быть больше или равно 8, при этом каждая программа автоматически открывает стандартные потоки данных stdin, stdout и stderr. Программы, которые используют более пяти потоков, должны с помощью макросов осуществлять проверку, предоставляется ли программе системой необходимое количество потоков.

fpos_t	<stdio.h>

Тип fpos_t используется функциями fgetpos() и fsetpos() для хранения текущей позиции файла:

```
typedef long fpos_t;
```

HUGE_VAL	<math.h>

В макросе HUGE_VAL определяется наибольшее значение, которое может иметь величина типа double.

ifstream	<fstream.h>

Специализация шаблона класса basic_ifstream для типа данных char.

```
typedef basic_ifstream<char> ifstream;
```

INT_MAX, INT_MIN	<limits.h>

В константе INT_MAX содержится наибольшее значение, которое может иметь величина типа int. Значение INT_MAX должно быть больше или равно 32 767.

В константе INT_MIN содержится наименьшее значение, которое может иметь величина типа int. Значение INT_MIN должно быть меньше или равно -32 767.

<code>_IOFBF, _IOLBF, _IONBF</code>	<code><stdio.h></code>

Эти константы применяются для определения режима буферизации в функции `setvbuf()`:

<code>_IOFBF</code>	0	полная буферизация;
<code>_IOLBF</code>	1	построчная буферизация текстового файла;
<code>_IONBF</code>	2	отмена буферизации.

<code>ios</code>	<code><ios.h></code>

Специализация шаблона класса `basic_ios` для типа данных `char`.

```
typedef basic_ios<char> ios;
```

<code>istream</code>	<code><istream.h></code>

Специализация шаблона класса `basic_istream` для типа данных `char`.

```
typedef basic_istream<char> istream;
```

<code>istreamstream</code>	<code><sstream.h></code>

Специализация шаблона класса `basic_istreamstream` для типа данных `char`.

```
typedef basic_istreamstream<char> istringstream;
```

<code>L_tmpnam</code>	<code><stdio.h></code>

Константа `L_tmpnam` задает минимальную длину строки, генерируемой функцией `tmpnam()` в качестве имени файла.

<code>LC_</code>	<code><locale.h></code>

Константы с префиксом `LC_` используются функциями `setlocale()` и `localeconv()` для выбора категорий локализации.

Константы	Описания
<code>LC_ALL</code>	Все категории
<code>LC_COLLATE</code>	Информация для сопоставления текста и сортировки
<code>LC_CTYPE</code>	Информация для обработки символьных записей
<code>LC_MONETARY</code>	Информация для формирования валютных данных
<code>LC_NUMERIC</code>	Территориально-специфическая информация о представлении десятичной точки
<code>LC_TIME</code>	Территориально-специфическая информация о дате и времени

lconv

<locale.h>

Структура lconv содержит территориально-специфическую информацию:

```
typedef struct lconv{
    char *currency_symbol;    // Символ валюты
    char *decimal_point;      // Символ для десятичной точки
    char *frac_digits;        // Разряд после запятой для валюты
    char grouping;            // Группировка цифр
    char *int_curr_symbol;    // Валютный символ ISO
    char *mon_decimal_point;  // Символы для десятичной точки валюты
    char mon_grouping;        // Группировка цифр валютных данных
    char *mon_thousand_sep;   // Разделители тысяч в валюте
    char n_cs_precedes;       // Позиция в валюте знака —
    char n_sep_by_space;      // Пробелы между знаком — и отрицательной
                                // валютной величиной
    char n_sign_posn;         // Разряд и формат символа
                                // для отрицательной валютной величины
    char *negative_sign;      // Знаки для отрицательной валютной величины
    char p_cs_precedes;       // Позиция валютного символа
    char p_sep_by_space;      // Пробелы между валютным символом
                                // и валютной величиной
    char p_sign_posn; /* Позиция и форматирование знака + для положительной
    валютной величины */
    char *positive_sign;      // Знаки для положительной валюты
    char *thousands_sep;     // Разделитель тысяч
};
```

LDBL_

<float.h>

(см. FLT_)

ldiv_t

<stdlib.h>

Структура div_t служит для определения типа возвращаемого значения функции ldiv(). Поле структуры quot содержит частное, а поле rem — остаток от деления.

```
typedef struct{
    int quot;
    int rem;
}div_t;
```

LONG_MAX, LONG_MIN

<limits.h>

В константе LONG_MAX содержится наибольшее значение, которое может иметь величина типа long. Значение LONG_MAX должно быть по меньшей мере 2 147 483 647.

В константе `LONG_MIN` содержится наименьшее значение, которое может иметь величина типа `long`. Значение `LONG_MIN` должно быть не более `-2 147 483 647`.

`MB_CUR_MAX``<stdlib.h>`

В макросе `MB_CUR_MAX` задается максимальное количество байт в одном символе действующего набора символов.

`MB_LEN_MAX``<limits.h>`

Эта константа содержит максимальное количество байт в символе во всех вспомогательных наборах символов, символы которых состоят из более чем одного байта.

`wbstate_t``<wchar.h>`

Определение типа, которое используется различными функциями обработки символьных строк.

`new_handler``<new.h>`

Тип функции для обработки ошибок, применяемый при использовании операций `new` и `new[]`.

```
typedef void (*new_handler)();
```

`NULL``<stddef.h>, <string.h> и т. д.`

Макрос `NULL` определяет указатель на нуль. В соответствии с ANSI-стандартом `NULL` может быть определен как `0` или `0L`.

`offsetof``<stddef.h>`

Макрос `offsetof` возвращает смещение элемента в структуре в байтах. Структура не может содержать ссылки или указатели.

```
size_t offsetof(struct_bez, element_bez);
```

`ofstream``<fstream.h>`

Специализация шаблона класса `basic_ofstream` для типа данных `char`.

```
typedef basic_ofstream<char> ofstream;
```

`ostream``<ostream.h>`

Специализация шаблона класса `basic_ostream` для типа данных `char`.

```
typedef basic_ostream<char> ostream;
```

ostreamstream

<sstream.h>

Специализация шаблона класса `basic_ostreamstream` для типа данных `char`.

```
typedef basic_ostreamstream<char> ostreamstream;
```

ptrdiff_t

<stddef.h>

Тип данных `ptrdiff_t` является целочисленным типом со знаком, предназначенным для хранения разности двух указателей.

```
typedef int ptrdiff_t;
```

RAND_MAX

<stdlib.h>

Содержит наибольшее возможное из псевдослучайных чисел, возвращаемых функцией `rand()`.

SCHAR_MAX, SCHAR_MIN

<limits.h>

В константе `SCHAR_MAX` содержится максимальное значение, которое может принимать величина типа `signed char`. Должно быть не менее 127.

В константе `SCHAR_MIN` содержится минимальное значение, которое может принимать величина типа `signed char`. Должно быть не более -127.

SEEK_CUR, SEEK_END, SEEK_SET

<stdio.h>

Константы используются в качестве аргумента функции `fseek()` и устанавливают точку отсчета для перемещения указателя текущей позиции в файле.

SEEK_SET	0	Сдвиг выполняется от начала файла;
SEEK_CUR	1	Сдвиг выполняется от текущей позиции указателя записи-чтения файла;
SEEK_END	2	Сдвиг выполняется от конца файла.

SHRT_MAX, SHRT_MIN

<limits.h>

В константе `SHRT_MAX` содержится максимальное значение, которое может принимать величина типа `short int`; должно быть не менее 32 767.

В константе `SHRT_MIN` содержится наименьшее значение, которое может принимать величина типа `signed short`; должно быть не более -32 767.

SIG_

<signal.h>

Перечисленные ниже типы сигналов возвращаются функцией `raise()` и обрабатываются функцией `signal()`.

Сигнал	Значение
SIG_DFL	Ссылка на программу-обработчик исключений по умолчанию
SIG_ERR	Возвращаемое значение функции <code>signal()</code> при возникновении ошибки
SIG_IGN	Игнорирование полученного сигнала
SIGABRT	Аварийное завершение программы
SIGFPE	Ошибка арифметической операции над вещественными и целыми числами
SIGILL	Недопустимая машинная инструкция
SIGINT	Прерывание от клавиатуры
SIGSEGV	Нарушение условий защиты памяти
SIGTERM	Запрос о завершении

sig_atomic_t

<signal.h>

Тип данных `sig_atomic_t` предназначен для хранения информации, к которой необходим доступ во время одного из асинхронных прерываний.

```
typedef sig_atomic_t;
```

size_t

<stdio.h>

Тип `size_t` является беззнаковым целочисленным типом, используемым для представления результата операции `sizeof`. Этот тип не может быть длиннее, чем `unsigned long`.

```
typedef unsigned size_t;
```

stdin, stdout, stderr

<stdio.h>

`stdin`, `stdout` и `stderr` представляют собой имена потоков, то есть имена, используемые для доступа к устройствам через функции, унаследованные из библиотеки C.

stderr	Стандартное устройство вывода ошибок. По умолчанию дисплей
stdin	Стандартное устройство ввода. По умолчанию клавиатура
stdout	Стандартное устройство вывода. По умолчанию дисплей

streambuf

<streambuf.h>

Специализация шаблона класса `basic_streambuf` для типа данных `char`.

```
typedef basic_streambuf<char> streambuf;
```


streamoff, streampos

<ios.h>

Типы данных, предназначенные для позиционирования указателя текущей позиции в потоках.

typedef OFF_T streamoff;

typedef POS_T streampos;

streamsize

<ios.h>

Тип данных, предназначенный для хранения количества переданных символов или о количестве имеющихся буферов.

typedef INT_T streamsize;

string

<string.h>

Специализация шаблона класса basic_string для типа данных char.

typedef basic_string<char> string;

stringbuf

<sstream.h>

Специализация шаблона класса basic_stringbuf для типа данных char.

typedef basic_stringbuf<char> stringbuf;

terminate_handler

<exception.h>

Тип функции обработки ошибок, который используется функцией set_terminate() при аварийном завершении программы.

typedef void (*terminate_handler)();

time_t

<time.h>

Тип time_t предназначен для хранения времени. Он содержит календарное время в форме, в которой оно возвращается функцией time() и обрабатывается функцией difftime().

typedef long time_t;

TMP_MAX

<stdio.h>

В константе TMP_MAX хранится максимальное количество различных временных имен файла, которые могут генерироваться функцией tmpnam() в течение одного сеанса работы программы. Значение TMP_MAX не может быть менее 25.

struct tm

<time.h>

Структура **tm** содержит все компоненты календарного времени, используемые функциями **localtime()** и **gmtime()**.

```
struct tm{
    int tm_sec; // секунды от начала минуты (0 ... 59)
    int tm_min; // минуты от начала часа(0 ... 59)
    int tm_hour; // часы до полуночи (0 ... 23)
    int tm_mday; // число месяца (0 ... 31)
    int tm_mon; // месяцы от начала года. (0 ... 11)
    int tm_year; // год после 1900
    int tm_yday; // день от начала года(0 ... 365)
    int tm_isdst; // флаг летнего времени
};
```

UCHAR_MAX

<limits.h>

В константе **UCHAR_MAX** хранится максимальное значение, которое может принимать величина типа **unsigned char**. Значение должно быть не менее 127.

UINT_MAX

<limits.h>

В константе **UINT_MAX** хранится максимальное значение, которое может принимать величина типа **unsigned int**. Значение должно быть не менее 65 535.

ULONG_MAX

<limits.h>

В константе **ULONG_MAX** хранится максимальное значение, которое может принимать величина типа **unsigned long int**. Значение должно быть не менее 4 294 967 295.

USHRT_MAX

<limits.h>

В константе **USHRT_MAX** хранится максимальное значение, которое может принимать величина типа **unsigned short int**. Значение должно быть не менее 65 535.

unexpected_handler

<exception.h>

Тип функции обработки исключений. Он используется функцией **set_unexpected()**, для перехвата исключения, не предусмотренного в списке исключений функции.

```
typedef void (*unexpected_handler) ();
```

va_arg, va_end, va_list

<stdarg.h>

Эти величины применяются для работы с функциями с переменным числом аргументов. Они описаны в заголовочном файле <stdarg.h>. Тип va_list предназначен для хранения указателя на очередной аргумент. Макрос va_start инициализирует этот указатель. Макрос va_arg возвращает значение очередного аргумента, каждый его вызов приводит к продвижению указателя, хранящегося в va_list. После перебора аргументов, но до выхода из функции с переменным числом аргументов необходимо обратиться к макросу va_end.

```
typedef void_FAR *va_list;
```

WCHAR_MAX, WCHAR_MIN

<wchar.h>

В константе WCHAR_MAX хранится наибольшее значение, которое может принимать переменная типа wchar_t.

В константе WCHAR_MIN хранится наименьшее значение, которое может принимать переменная типа wchar_t.

wtrans_t

<wctype.h>

Вспомогательный тип для преобразования символов расширенного набора.

wctype_t

<wctype.h>

В структуре данных wctype_t хранится информация, предназначенная для преобразования и классификации символов.

```
struct wctype_t{
    long tmin;           // Минимальный код символа
    long tmax;           // Максимальный код символа
    unsigned char *index; // Индекс классификации
    unsigned int *type;   // Тип
    long cmin;           // Минимальный код для преобразования
    long cmax;           // Максимальный код для преобразования
    long *code;          // Код преобразования
};
```

WEOF

<wctype.h>

Константа WEOF сообщает об окончании файла (синоним EOF для многобайтных символов).

werr, win, wlog, wout

<iostream.h>

Объекты потоковых классов для потоков данных расширенных символов, связанные со стандартными периферийными устройствами.

Объект	Поток данных	Класс
win	stdin	wistream
wout	stdout	wostream
werr	stderr	wostream
wlog	stderr (буферизованный)	wostream

wfilebuf

<fstream.h>

Специализация шаблона класса `basic_filebuf` для типа данных `wchar_t`.

```
typedef basic_filebuf<wchar_t> wfilebuf;
```

wofstream

<fstream.h>

Специализация шаблона класса `basic_ifstream` для типа данных `wchar_t`.

```
typedef basic_ifstream<wchar_t> wifstream;
```

wint_t

<wctype.h>

Целочисленный тип для хранения символов расширенного набора.

wios

<ios.h>

Специализация шаблона класса `basic_ios` для типа данных `wchar_t`.

```
typedef basic_ios<wchar_t> wios;
```

wistream

<istream.h>

Специализация шаблона класса `basic_istream` для типа данных `wchar_t`.

```
typedef basic_istream<wchar_t> wistream;
```

wstringstream

<sstream.h>

Специализация шаблона класса `basic_istringstream` для типа данных `wchar_t`.

```
typedef basic_istringstream<wchar_t> istringstream;
```

wofstream

<fstream.h>

Специализация шаблона класса `basic_ofstream` для типа данных `wchar_t`.

```
typedef basic_ofstream<wchar_t> wofstream;
```

Объект	Поток данных	Класс
win	stdin	wistream
wout	stdout	wostream
werr	stderr	wostream
wlog	stderr (буферизованный)	wostream

wfilebuf

<fstream.h>

Специализация шаблона класса `basic_filebuf` для типа данных `wchar_t`.

```
typedef basic_filebuf<wchar_t> wfilebuf;
```

wifstream

<fstream.h>

Специализация шаблона класса `basic_ifstream` для типа данных `wchar_t`.

```
typedef basic_ifstream<wchar_t> wifstream;
```

wint_t

<wctype.h>

Целочисленный тип для хранения символов расширенного набора.

wios

<ios.h>

Специализация шаблона класса `basic_ios` для типа данных `wchar_t`.

```
typedef basic_ios<wchar_t> wios;
```

wistream

<istream.h>

Специализация шаблона класса `basic_istream` для типа данных `wchar_t`.

```
typedef basic_istream<wchar_t> wistream;
```

wstringstream

<sstream.h>

Специализация шаблона класса `basic_istringstream` для типа данных `wchar_t`.

```
typedef basic_istringstream<wchar_t> istringstream;
```

wofstream

<fstream.h>

Специализация шаблона класса `basic_ofstream` для типа данных `wchar_t`.

```
typedef basic_ofstream<wchar_t> wofstream;
```


ПРИЛОЖЕНИЕ 6

Функции стандартной библиотеки

Приложение содержит две части. В первой части функции сгруппированы по заголовочным файлам, в которых они описаны, во второй части приведен алфавитный перечень функций с описанием.

Состав заголовочных файлов

Заголовочный файл `<ctype.h>` (`<cctype>`) — функции классификации и преобразования символов

<code>isalnum</code>	Проверяет, является ли символ буквой или цифрой
<code>isalpha</code>	Проверяет, является ли символ буквой
<code>iscntrl</code>	Проверяет, является ли символ управляющим
<code>isdigit</code>	Проверяет, является ли символ цифрой
<code>isgraph</code>	Проверяет, является ли символ видимым
<code>islower</code>	Проверяет, является ли символ буквой нижнего регистра
<code>isprint</code>	Проверяет, является ли символ печатаемым
<code>ispunct</code>	Проверяет, является ли символ символом пунктуации
<code>isspace</code>	Проверяет, является ли символ разграничительным
<code>isupper</code>	Проверяет, является ли символ буквой верхнего регистра
<code>iswalnum</code>	Проверяет, является ли символ буквой или цифрой
<code>iswalpha</code>	Проверяет, является ли символ буквой
<code>iswcntrl</code>	Проверяет, является ли символ управляющим
<code>iswctype</code>	Проверяет многобайтный символ

iswdigit	Проверяет, является ли символ цифрой
iswgraph	Проверяет, является ли символ видимым
iswlower	Проверяет, является ли символ буквой нижнего регистра
iswprint	Проверяет, является ли символ печатаемым
iswpunct	Проверяет, является ли символ символом пунктуации
iswspace	Проверяет, является ли символ разграничительным
iswupper	Проверяет, является ли символ буквой верхнего регистра
iswxdigit	Проверяет, является ли символ символом шестнадцатеричной цифры
isxdigit	Проверяет, является ли символ символом шестнадцатеричной цифры
tolower	Возвращает символ в нижнем регистре
toupper	Возвращает символ в верхнем регистре
towlower	Возвращает символ в нижнем регистре
toupper	Возвращает символ в верхнем регистре

Заголовочный файл <locale.h> (<locale>) — функции локализации

localeconv	Возвращает описание установок локальной среды
setlocale	Устанавливает новую локализацию

Заголовочный файл <math.h> (<cmath>) — математические функции

acos	Возвращает арккосинус аргумента
asin	Возвращает арксинус аргумента
atan	Возвращает арктангенс аргумента
atan2	Возвращает арктангенс отношения аргументов
ceil	Округляет вверх
cos	Вычисляет косинус
cosh	Вычисляет гиперболический косинус
exp	Возвращает степень числа e
fabs	Возвращает модуль числа
floor	Округляет вниз
fmod	Возвращает остаток от деления x на y.
frexp	Выделяет из числа мантиссу и экспоненциальную часть
ldexp	Преобразует мантиссу и показатель степени в число
log	Вычисляет натуральный логарифм
log10	Вычисляет логарифм по основанию 10

modf	Разбивает число на целую и дробную части
pow	Возводит число в степень
sin	Вычисляет синус
sinh	Вычисляет гиперболический синус
sqrt	Вычисляет квадратный корень
tan	Возвращает тангенс аргумента
tanh	Возвращает гиперболический тангенс аргумента

Заголовочный файл <signal.h> (<csignal>) — функции обработки сигналов

raise	Прерывает выполнение программы
signal	Регистрирует обработчик сигналов

Заголовочный файл <stdio.h> (<cstdio>) — функции ввода/вывода в стиле C

clearerr	Очищает флаги ошибок при работе с потоком
fclose	Закрывает поток ввода/вывода
feof	Проверяет достижение конца файла
ferror	Возвращает код ошибки при работе с потоком
fflush	Записывает данные из буфера
fgetc	Читает из потока символ
fgetpos	Возвращает текущую позицию в файле
fgets	Читает из потока в строку p символов
fgetwc	Читает из потока символ
fgetws	Читает из потока в строку p символов
fopen	Открывает поток ввода/вывода
fprintf	Записывает данные в поток
fputc	Записывает символ в поток
fputs	Записывает строку символов в поток
fputwc	Записывает символ в поток
fputws	Записывает строку символов в поток
fread	Читает данные из потока ввода
freopen	Открывает поток ввода/вывода
fscanf	Вводит из файла форматированные данные
fseek	Перемещает позицию в файле относительно текущей
fsetpos	Перемещает текущую позицию в файле относительно его начала

ftell	Возвращает текущую позицию в файле
fwrite	Записывает данные в поток
fwrite	Записывает данные из заданного буфера в поток
fscanf	Читает из потока данные и пишет их по заданным адресам
getc	Читает символ из потока
getchar	Читает символ из стандартного ввода
gets	Читает символы с клавиатуры до появления символа новой строки
getwc	Читает из потока символ
getwchar	Возвращает очередной символ с клавиатуры
perror	Печатает строку вида «s: error сообщение»
printf	Выводит строку параметров в определенном формате
putc	Записывает символ в поток
putchar	Выводит символ на стандартное устройство вывода
puts	Выводит строку на стандартное устройство вывода
putwc	Записывает символ в поток
putwchar	Выводит символ на стандартное устройство вывода
remove	Удаляет файл
rename	Переименовывает файл
rewind	Очищает флаги ошибок при работе с потоком и переходит к началу файла
scanf	Вводит строку параметров в определенном формате
setbuf	Устанавливает буферизацию потока ввода/вывода
setvbuf	Преобразует строки на основе текущей локализации
sprintf	Выводит строку параметров в определенном формате
sscanf	Вводит данные из строки
swprintf	Выводит строку параметров в определенном формате
swscanf	Вводит данные из строки
tmpfile	Открывает поток двоичного ввода/вывода во временный файл
tmpnam	Создает уникальное имя файла
ungetc	Возвращает символ в поток
ungetwc	Возвращает символ в поток
vfprintf	Посылает отформатированный вывод в поток
vfwprintf	Посылает отформатированный вывод в поток

vprintf	Посылает отформатированный вывод в стандартный поток вывода
vsprintf	Выводит строку параметров в определенном формате
vswprintf	Выводит строку параметров в определенном формате
wprintf	Посылает отформатированный вывод в стандартный поток вывода
wprintf	Выводит строку параметров в определенном формате
wscanf	Вводит строку параметров определенном формате

Заголовочный файл <stdlib.h> (<cstdlib>) — различные функции в стиле C

abort	Прерывает выполнение программы
abs	Возвращает модуль числа
atexit	Регистрирует функцию, вызываемую при завершении работы программы
atof	Преобразует строку в вещественное число
atoi	Преобразует строку в целое число
atol	Преобразует строку в длинное целое число
bsearch	Ищет элемент в отсортированном массиве
calloc	Выделяет блок памяти
div	Делит с остатком
exit	Прерывает выполнение программы
free	Освобождает блок памяти
getenv	Возвращает значение переменной окружения
labs	Возвращает модуль числа
ldiv	Делит с остатком
malloc	Выделяет блок памяти
mblen	Определяет размер многобайтных символов
mbstowcs	Преобразует строку многобайтных символов в массив из wchar_t
mbtowc	Преобразует многобайтный символ в wchar_t
qsort	Сортирует заданный массив
rand	Генерирует случайные числа
realloc	Изменяет размеры ранее выделенного блока памяти
srand	Устанавливает начальное псевдослучайное число
strtod	Преобразует строку в число
strtol	Преобразует строку в число с учетом системы счисления
strtoul	Преобразует строку в число с учетом системы счисления

system	Передаёт строку командному процессору ОС
wctod	Преобразует строку в число
wctol	Преобразует строку в число с учётом системы счисления
wctombs	Преобразует wchar_t в строку многобайтных символов
wctoul	Преобразует строку в число с учётом системы счисления
wctomb	Преобразует wchar_t в многобайтный символ

Заголовочный файл <string.h> (<cstring>) — функции работы со строками в стиле C

memchr	Ищет первое вхождение символа в блок памяти
memcmp	Сравнивает блоки памяти
memcpy	Копирует блок памяти
memmove	Переносит блок памяти
memset	Заполняет блок памяти символом
strcat	Складывает строки
strchr	Ищет символ в строке
strcmp	Сравнивает строки
strcoll	Сравнивает строки с учётом установленной локализации
strcpy	Копирует одну строку в другую
strcspn	Ищет один из символов одной строки в другой
strerror	Возвращает указатель на строку с описанием ошибки
strlen	Возвращает длину строки
strncat	Складывает одну строку с p символами другой
strncmp	Сравнивает одну строку с p символами другой
strncpy	Копирует первые p символов одной строки в другую
strpbrk	Ищет один из символов одной строки в другой
strrchr	Ищет символ в строке
strspn	Ищет символ одной строки, отсутствующий в другой
strstr	Ищет подстроку в строке
strtok	Выделяет из строки лексемы
strxfrm	Преобразует строки на основе текущей локализации
wscat	Складывает строки
wcschr	Ищет символ в строке
wscmp	Сравнивает строки
wscoll	Сравнивает строки с учётом установленной локализации

wscpy	Копирует одну строку в другую
wscspn	Ищет один из символов одной строки в другой
wcslen	Возвращает длину строки
wcsncat	Складывает одну строку с n символами другой
wcsncmp	Сравнивает одну строку с n символами другой
wcsncpy	Копирует первые n символов одной строки в другую
wcspbrk	Ищет один из символов одной строки в другой
wcsrchr	Ищет символ в строке
wcsspn	Ищет символ одной строки, отсутствующий в другой
wcsstr	Ищет подстроку в строке
wcstok	Выделяет из строки лексемы
wcstrxfrm	Преобразует строки на основе текущей локализации
wmemcpy	Копирует блок памяти
wmemmove	Переносит блок памяти
wmemset	Заполняет блок памяти символом

Заголовочный файл <time.h> (<ctime>) — функции для работы с датой и временем в стиле C

asctime	Преобразует дату/время в строку
clock	Возвращает время исполнения программы
ctime	Преобразует время в строку
difftime	Возвращает разницу времен
gmtime	Инициализирует структуру tm на основе time_t
localtime	Инициализирует структуру tm на основе time_t
mktime	Заполняет поля дня недели и дня года
strftime	Преобразует время в формате fmt в формат tm
time	Возвращает текущие дату/время в виде time_t
wcsftime	Преобразует время в формате fmt в формат tm

Заголовочный файл <wchar.h> (<swchar>) — функции для работы с многобайтными символами в стиле C

btowc	Преобразует символ в длинное целое число
fwide	Определяет вид потока
wmemchr	Ищет первое вхождение символа в блок памяти
wmemcmp	Сравнивает блоки памяти

Алфавитный перечень функций

Константы, типы и структуры, используемые приведенными ниже функциями, описаны в приложении 5.

abort	Прерывает выполнение программы	<stdlib.h>

```
void abort(void);
```

Функция выводит сообщение «abnormal program termination» и прерывает выполнение программы. При этом буферы не выгружаются и потоки не закрываются. Предпочтительней использовать `exit`. Функция не возвращает управление в вызывающую программу, она возвращает код завершения программы, по умолчанию равный 3.

abs	Возвращает модуль числа	<stdlib.h>

```
int abs(int num);
```

Функция возвращает абсолютное значение числа `num`.

acos	Возвращает арккосинус аргумента	<math.h>

```
double acos(double x);
```

Функция возвращает значение арккосинуса аргумента `x` в радианах.

asctime	Преобразует дату/время в строку	<time.h>

```
char *asctime(const struct tm *ctm);
```

Функция возвращает строку, представляющую символьное представление даты и времени из структуры типа `tm`.

asin	Возвращает арксинус аргумента	<math.h>

```
double asin(double x);
```

Функция возвращает значение арксинуса аргумента `x` в радианах.

atan	Возвращает арктангенс аргумента	<math.h>

```
double atan(double x);
```

Функция возвращает значение арктангенса аргумента `x` в радианах.

atan2	Возвращает арктангенс отношения аргументов	<math.h>

```
double atan2(double y, double x);
```

Функция возвращает значение арктангенса отношения параметров `y/x` в радианах.

atexit	Регистрирует функцию, вызываемую при завершении работы программы	<stdlib.h>

```
void atexit(void (*func)(void));
```

Функция регистрирует функцию `func`, которая будет вызываться при нормальном завершении программы. Она не должна иметь ни параметров, ни возвращаемого значения. Повторные вызовы `atexit` формируют стек функций.

atof	Преобразует строку в вещественное число	<stdlib.h>

```
double atof(const char *str);
```

Функция преобразует строку, содержащую символьное представление целого или вещественного числа, в соответствующее вещественное число с двойной точностью. В качестве признака конца числа служит первый символ, который не может быть интерпретирован как принадлежащий числу. Если преобразование не удалось, возвращает 0.0.

atoi	Преобразует строку в целое число	<stdlib.h>

```
int atoi(const char *str);
```

Функция преобразует строку, содержащую символьное представление целого числа, в соответствующее целое число. В качестве признака конца числа служит первый символ, который не может быть интерпретирован как принадлежащий числу. Если преобразование не удалось, возвращает 0.

atol	Преобразует строку в длинное целое число	<stdlib.h>

```
long atol(const char *str);
```

Функция преобразует строку, содержащую символьное представление целого числа, в соответствующее длинное целое число. В качестве признака конца числа служит первый символ, который не может быть интерпретирован как принадлежащий числу. Если преобразование не удалось, возвращает 0L.

bsearch	Ищет элемент в отсортированном массиве	<stdlib.h>

```
void bsearch(const void *key, const void *buf, size_t num, size_t size,
int e(*compare)(const void *, const void *));
```

Функция производит дихотомический поиск в упорядоченном массиве `buf` элемента, совпадающего с тем, на который указывает `key`, и возвращает указатель на найденный элемент. `num` — количество элементов в массиве `buf`, `size` — размер элемента массива в байтах.

Функция `compare()` должна выполнять сравнение двух элементов массива и возвращать одно из трех значений —1, 0 или 1 в зависимости от того, меньше, равен или больше первый элемент в сравнении со вторым. Несмотря на то, что оба параметра функции `compare` определены как `void*`, реально параметрами функции

являются указатели на элементы, и она должна приводить их к соответствующему типу перед разыменованием (получением значений по адресу).

Если функция `bsearch` нашла искомый элемент, то она возвращает указатель на него, в противном случае возвращается `NULL`.

<code>btowc</code>	Преобразует символ в длинное целое число	<code><wchar.h></code>

```
wint_t btowc(int c);
```

Функция возвращает `WEOF`, если `c` равно `EOF`, иначе преобразует символ `c` в многобайтный аналогично функции `mbtowc`. Если преобразование не удалось, возвращает `WEOF`.

<code>calloc</code>	Выделяет блок памяти	<code><stdlib.h></code>

```
void *calloc(size_t num, size_t size);
```

Функция выделяет блок памяти (`num` элементов по `size` байт) и возвращает указатель на него. Каждый элемент полученного массива инициализируется нулевым значением (в отличие от функции `malloc`). Если выделить блок памяти не удалось, возвращается `NULL`.

<code>ceil</code>	Округляет вверх	<code><math.h></code>

```
double ceil(double x);
```

Функция округляет вещественное значение `x` до ближайшего большего целого и возвращает его как вещественное.

<code>clearerr</code>	Очищает флаги ошибок при работе с потоком	<code><stdio.h></code>

```
void clearerr(FILE *f);
```

Функция очищает флаги ошибок для потока `f` (флаги ошибок не сбрасываются, пока не будет вызвана одна из функций `clearerr`, `fseek`, `fsetpos` или `rewind`).

<code>clock</code>	Возвращает время исполнения программы	<code><time.h></code>

```
clock_t clock(void);
```

Функция возвращает время исполнения программы как значение типа `clock_t` (это значение может быть переведено в секунды делением его на определенную в файле `time.h` константу `CLOCK_PER_SEC`).

<code>cos</code>	Вычисляет косинус	<code><math.h></code>

```
double cos(double x);
```

Функция возвращает значение косинуса угла, равного `x` радиан.

cosh	Вычисляет гиперболический косинус	<math.h>

```
double cosh(double x);
```

Функция возвращает значение гиперболического косинуса угла, равного x радиан. Если значение функции окажется вне представимого диапазона, то функция возвращает значение HUGEVAL, а глобальная переменная errno получает значение ERANGE.

ctime	Преобразует время в строку	<time.h>

```
char *ctime(time_t *t);
```

Функция возвращает строку, представляющую символьное представление даты и времени параметра типа time_t.

difftime	Возвращает разницу времен	<time.h>

```
double difftime(time_t t2, time_t t1);
```

Функция возвращает разницу времен ($t2-t1$) в секундах.

div	Делит с остатком	<stdlib.h>

```
div_t div(int numerator, int denominator);
```

Функция делит целое numerator на целое denominator, возвращая результат в структуре, содержащей частное и остаток:

```
struct div_t{
    int quot;    //частное
    int rem;     //остаток
};
```

exit	Прерывает выполнение программы	<stdlib.h>

```
void exit(int exit_code);
```

Функция вызывает нормальное завершение программы, записывая все буферы в соответствующие файлы, закрывая потоки и вызывая функции, зарегистрированные с помощью atexit, в обратном регистрации порядке.

exp	Возвращает степень числа e	<math.h>

```
double exp(double x);
```

Функция возвращает результат возведения числа e в указанную степень.

fabs	Возвращает модуль числа	<math.h>

```
double fabs(double x);
```

Функция возвращает абсолютное значение числа num.

fclose	Закрывает поток ввода/вывода	<stdio.h>

```
int fclose(FILE *f);
```

Функция закрывает поток ввода/вывода *f*.

feof	Проверяет достижение конца файла	<stdio.h>

```
int feof(FILE *f);
```

Функция возвращает EOF или значение, отличное от 0, если достигнут конец файла, в противном случае возвращает 0.

ferror	Возвращает код ошибки при работе с потоком	<stdio.h>

```
int ferror(FILE *f);
```

Функция возвращает целое, означающее код ошибки; 0 — отсутствие ошибки.

fflush	Записывает данные из буфера	<stdio.h>

```
int fflush(FILE *f);
```

Функция очищает буфер вывода посредством немедленной отправки данных для записи на физическое устройство. При успешном завершении возвращает значение 0, иначе возвращается значение EOF.

fgetc	Читает из потока символ	<stdio.h>

```
int fgetc(FILE *f);
```

Функция возвращает очередной символ в форме int из потока *f*. Если символ не может быть прочитан, то возвращается значение EOF.

fgetpos	Возвращает текущую позицию в файле	<stdio.h>

```
int fgetpos(FILE *f, fpos_t *pos);
```

Функция возвращает текущую позицию в файле, связанном с потоком *f*, и копирует значение по адресу *pos*. Это значение позднее может использоваться функцией *fsetpos*. Возвращаемое значение имеет тип *fpos_t*.

fgets	Читает из потока в строку <i>n</i> символов	<stdio.h>

```
char *fgets(char *s, int n, FILE *f);
```

Функция читает не более *n-1* байт из потока *f* в строку *s*, прекращая чтение при обнаружении символа новой строки или конца файла. Символ новой строки при чтении не отбрасывается, а помещается в конец строки. Прочитанная строка дополняется ограничителем строки ('\0'). При обнаружении ошибки или конца файла возвращается NULL, в противном случае — указатель на строку *s*.

fgetc	Читает из потока символ	<stdio.h>

```
wint_t fgetc(FILE *f);
```

Функция возвращает очередной символ в форме `wint_t` из потока `f`. Если символ не может быть прочитан, то возвращается значение `WEOF`. Является аналогом функции `fgetc` для многобайтных символов.

fgetws	Читает из потока в строку n символов	<stdio.h>

```
wchar_t *fgetws(wchar_t *s, int n, FILE *f);
```

Функция читает не более `n-1` символов из потока `f` в строку `s`, прекращая чтение при обнаружении символа новой строки или конца файла. Символ новой строки при чтении не отбрасывается, а помещается в конец строки. Прочитанная строка дополняется ограничителем строки (`'\0'`). При обнаружении ошибки или конца файла возвращается `NULL`, в противном случае — указатель на строку `s`. Является аналогом функции `fgets` для многобайтных символов.

floor	Округляет вниз	<math.h>

```
double floor(double x);
```

Функция округляет вещественное значение `x` до ближайшего меньшего целого и возвращает его как вещественное.

fmod	Возвращает остаток от деления x на y	<math.h>

```
double fmod(double x, double y);
```

Функция возвращает остаток от деления `x` на `y`. Аналогична операции `%`, но работает с вещественными числами.

fopen	Открывает поток ввода/вывода	<stdio.h>

```
FILE *fopen(const char *fname, const char *mode);
```

Функция открывает файл с именем `fname` для работы в режиме, указанном строкой `mode`. При успешном открытии возвращается указатель на поток (таким образом, полученный поток связывается с файлом), в противном случае — `NULL`.

fprintf	Записывает данные в поток	<stdio.h>

```
int fprintf(FILE *f, const char *fmt, ...);
```

Функция записывает в поток `f` переменные, список которых обозначен многоточием (...), в формате, указанном строкой `fmt`. Возвращает число записанных символов.

fputc	Записывает символ в поток	<stdio.h>

```
int fputc(int ch, FILE *f);
```

Функция записывает символ *ch* в поток *f*. При ошибке возвращает значение EOF, иначе — записанный символ.

fputs	Записывает строку символов в поток	<stdio.h>

```
int fputs(const char *s, FILE *f);
```

Функция записывает строку символов *s* в поток *f*. Символ конца строки в файл не записывается. При ошибке возвращает значение EOF, иначе — неотрицательное число.

fputc	Записывает символ в поток	<stdio.h>

```
w_int fputc(w_int ch, FILE *f);
```

Функция записывает символ *ch* в поток *f*. При ошибке возвращает значение WEOF, иначе — записанный символ. Является аналогом функции *fputc* для многобайтных символов.

fputws	Записывает строку символов в поток	<stdio.h>

```
int fputws(const wchar_t *s, FILE *f);
```

Функция записывает строку символов *s* в поток *f*. Символ конца строки в файл не записывается. При ошибке возвращает значение WEOF, иначе — неотрицательное число. Является аналогом функции *fputs* для многобайтных символов.

free	Освобождает блок памяти	<stdlib.h>

```
void free(void *pointer);
```

Функция освобождает блок памяти, полученный ранее одной из функций *malloc*, *calloc* или *realloc*.

fread	Читает данные из потока ввода	<stdio.h>

```
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

Функция возвращает количество прочитанных элементов, которое может быть меньше *count*, если при чтении произошла ошибка или встретился конец файла.

freopen	Открывает поток ввода/вывода	<stdio.h>

```
FILE *freopen(const char *fname, const char *mode, FILE *f);
```

Функция работает аналогично *fopen*, но предварительно закрывает поток *f*, если тот был ранее открыт.

frexp	Выделяет из числа мантиссу и экспоненциальную часть	<math.h>

```
double frexp(double x, int *exp_ptr);
```

Функция выделяет мантиссу и показатель степени числа x . Возвращает значение мантиссы и копирует экспоненциальную часть по адресу `expptr`.

<code>fscanf</code>	Вводит из файла форматированные данные	<code><stdio.h></code>

```
int fscanf(FILE *f, const char *fmt [.par1. par2. _]):
```

Функция вводит строку параметров `par1`, `par2` и т. д. в формате, определенном строкой `fmt`, из файла `f`. Возвращает число переменных, которым присвоено значение.

<code>fseek</code>	Перемещает позицию в файле относительно текущей	<code><stdio.h></code>

```
int fseek(FILE *f, long off, int org):
```

Функция перемещает текущую позицию в файле, связанном с потоком `f`, на позицию `off`, отсчитываемую от значения `org`, которое должно быть равно одной из трех констант, определенных в `<stdio.h>`:

`SEEK_CUR` — от текущей позиции указателя;

`SEEK_END` — от конца файла;

`SEEK_SET` — от начала файла.

<code>fsetpos</code>	Перемещает текущую позицию в файле относительно его начала	<code><stdio.h></code>

```
int fsetpos(FILE *f, const fpos_t *pos):
```

Функция перемещает текущую позицию в файле, связанном с потоком `f`, на позицию `*pos`, предварительно полученную с помощью функции `fgetpos`.

<code>ftell</code>	Возвращает текущую позицию в файле	<code><stdio.h></code>

```
long int ftell(FILE *f):
```

Функция возвращает текущую позицию в файле, связанном с потоком `f`, как длинное целое.

<code>fwide</code>	Определяет вид потока	<code><wchar.h></code>

```
int fwide(FILE *stream, int mode):
```

Функция определяет в зависимости от `mode` вид потока — байт-ориентированный (`mode < 0`) или поток из многобайтных символов (`mode > 0`). Возвращает текущий вид потока.

<code>fwprintf</code>	Записывает данные в поток	<code><stdio.h></code>

```
int fwprintf(FILE *f, const wchar_t *fmt, _):
```

Функция записывает в поток *f* переменные, список которых обозначен многоточием, в формате, указанном строкой *fmt*. Возвращает число записанных символов. Является аналогом функции *fprintf* для многобайтных символов.

<code>fwrite</code>	Записывает данные из заданного буфера в поток	<code><stdio.h></code>

```
size_t fwrite(const void *p, size_t size, size_t n, FILE *f);
```

Функция записывает *n* элементов длиной *size* байт из буфера, заданного указателем *p*, в поток *f*. Возвращает число записанных элементов.

<code>fwscanf</code>	Читает из потока данные и пишет их по заданным адресам	<code><stdio.h></code>

```
int fwscanf(FILE *f, const wchar_t *fmt, ...);
```

Функция в соответствии с форматом, указанным строкой *fmt*, читает из потока *f* переменные, список адресов которых обозначен многоточием. Возвращает число прочитанных элементов или *WEOF*, в случае неудачного чтения. Является аналогом функции *fscanf* для многобайтных символов.

<code>getc</code>	Читает символ из потока	<code><stdio.h></code>

```
int getc(FILE *f);
```

Функция возвращает очередной символ в форме *int* из потока *f*. Если символ не может быть прочитан, то возвращается значение *EOF*.

<code>getchar</code>	Читает символ из стандартного ввода	<code><stdio.h></code>

```
int getchar(void);
```

Функция возвращает очередной символ в форме *int* из стандартного ввода. Если символ не может быть прочитан, то возвращается значение *EOF*.

<code>getenv</code>	Возвращает значение переменной окружения	<code><stdlib.h></code>

```
int getenv(const char *var_name);
```

Функция возвращает указатель на строку, которая содержит описание указанной переменной окружения.

<code>gets</code>	Читает символы с клавиатуры до появления символа новой строки	<code><stdio.h></code>

```
char *gets(char *s);
```

Функция читает символы с клавиатуры до появления символа новой строки и помещает их в строку *s* (сам символ новой строки в строку не включается). Возвращает указатель на *s*.

<code>getwc</code>	Читает из потока символ	<code><stdio.h></code>

```
wint_t getwc(FILE *f);
```

Функция возвращает очередной символ в форме `wint_t` из потока `f`. Если символ не может быть прочитан, то возвращается значение `WEOF`. Является аналогом функции `getc` для многобайтных символов.

<code>getwchar</code>	Возвращает очередной символ с клавиатуры	<code><stdio.h></code>

```
wint_t getwchar(void);
```

Функция возвращает очередной символ из стандартного потока ввода (обычно клавиатура). Является аналогом функции `getchar` для многобайтных символов.

<code>gmtime</code>	Инициализирует структуру <code>tm</code> на основе <code>time_t</code>	<code><time.h></code>

```
const struct tm gmtime(const time_t *t);
```

Функция возвращает указатель на временную структуру типа `tm`, инициализированную на основе даты/времени из `t`. Время устанавливается по Гринвичу.

<code>isalnum</code>	Проверяет, является ли символ буквой или цифрой	<code><ctype.h></code>

```
int isalnum(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является буквой или цифрой, или `false` в противном случае.

<code>isalpha</code>	Проверяет, является ли символ буквой	<code><ctype.h></code>

```
int isalpha(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является буквой, или `false` в противном случае.

<code>iscntrl</code>	Проверяет, является ли символ управляющим	<code><ctype.h></code>

```
int iscntrl(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является управляющим символом (типа `line feed`, `del`, табуляции и тому подобных, большинство из которых находятся в диапазоне `0x01–0x1F` (для кодировки ASCII)), или `false` в противном случае.

<code>isdigit</code>	Проверяет, является ли символ цифрой	<code><ctype.h></code>

```
int isdigit(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является цифрой, или `false` в противном случае.

<code>isgraph</code>	Проверяет, является ли символ видимым	<code><ctype.h></code>

```
int isgraph(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является видимым (то есть он не является символом пробела, табуляции и т. д.) или `false` в противном случае.

<code>islower</code>	Проверяет, является ли символ буквой нижнего регистра	<code><ctype.h></code>

```
int islower(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является буквой нижнего регистра, или `false` в противном случае.

<code>isprint</code>	Проверяет, является ли символ печатаемым	<code><ctype.h></code>

```
int isprint(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является печатаемым (`isgraph` + пробел), или `false` в противном случае.

<code>ispunct</code>	Проверяет, является ли символ символом пунктуации	<code><ctype.h></code>

```
int ispunct(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является символом пунктуации (то есть печатаемым, но не буквой, не цифрой, не пробелом), или `false` в противном случае.

<code>isspace</code>	Проверяет, является ли символ разграничительным	<code><ctype.h></code>

```
int isspace(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является символом пробела или табуляцией, или символом новой строки, или символом новой страницы (символом перевода формата), или `false` в противном случае.

<code>isupper</code>	Проверяет, является ли символ буквой верхнего регистра	<code><ctype.h></code>

```
int isupper(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является буквой верхнего регистра, или `false` в противном случае.

<code>iswalnum</code>	Проверяет, является ли символ буквой или цифрой	<code><ctype.h></code>

```
int iswalnum(wint_t ch);
```

Функция возвращает значение `true`, если символ `ch` является буквой или цифрой, или `false` в противном случае. Является аналогом функции `isalnum` для многобайтных символов.

iswalpha	Проверяет, является ли символ буквой	<ctype.h>

```
int iswalpha(wint_t ch);
```

Функция возвращает значение true, если символ ch является буквой, или false в противном случае. Является аналогом функции isalpha для многобайтных символов.

iswcntrl	Проверяет, является ли символ управляющим	<ctype.h>

```
int iswcntrl(wint_t ch);
```

Функция возвращает значение true, если символ ch является управляющим символом (типа line feed, del, табуляции и тому подобных, большинство из которых находятся в диапазоне 0x01–0x1F (для кодировки ASCII)), или false в противном случае. Является аналогом функции iscntrl для многобайтных символов.

iswctype	Проверяет многобайтный символ	<ctype.h>

```
int iswctype(wint_t c, wctype_t desc);
```

Функция возвращает ненулевое значение, если символ c обладает свойством desc, или нулевое в противном случае.

iswdigit	Проверяет, является ли символ цифрой	<ctype.h>

```
int iswdigit(wint_t ch);
```

Функция возвращает значение true, если символ ch является цифрой, или false в противном случае. Является аналогом функции isdigit для многобайтных символов.

iswgraph	Проверяет, является ли символ видимым	<ctype.h>

```
int iswgraph(wint_t ch);
```

Функция возвращает значение true, если символ ch является видимым (то есть он не является символом пробела, табуляции и т. д.) или false в противном случае. Является аналогом функции isgraph для многобайтных символов.

iswlower	Проверяет, является ли символ буквой нижнего регистра	<ctype.h>

```
int iswlower(wint_t ch);
```

Функция возвращает значение true, если символ ch является буквой нижнего регистра, или false в противном случае. Является аналогом функции islower для многобайтных символов.

iswprint	Проверяет, является ли символ печатаемым	<ctype.h>

```
int iswprint(wint_t ch);
```

Функция возвращает значение `true`, если символ `ch` является печатаемым (`iswgraph` + пробел), или `false` в противном случае. Является аналогом функции `iswprint` для многобайтных символов.

<code>iswpunct</code>	Проверяет, является ли символ символом пунктуации	<code><ctype.h></code>

```
int iswpunct(wint_t ch);
```

Функция возвращает значение `true`, если символ `ch` является символом пунктуации (то есть печатаемым, но не буквой, не цифрой, не пробелом), или `false` в противном случае. Является аналогом функции `ispunct` для многобайтных символов.

<code>iswspace</code>	Проверяет, является ли символ разграничительным	<code><ctype.h></code>

```
int iswspace(wint_t ch);
```

Функция возвращает значение `true`, если символ `ch` является символом пробела или табуляцией, или символом новой строки, или символом новой страницы (символом перевода формата), или `false` в противном случае. Является аналогом функции `isspace` для многобайтных символов.

<code>iswupper</code>	Проверяет, является ли символ буквой верхнего регистра	<code><ctype.h></code>

```
int iswupper(wint_t ch);
```

Функция возвращает значение `true`, если символ `ch` является буквой верхнего регистра, или `false` в противном случае. Является аналогом функции `isupper` для многобайтных символов.

<code>iswxdigit</code>	Проверяет, является ли символ символом шестнадцатеричной цифры	<code><ctype.h></code>

```
int iswxdigit(wint_t ch);
```

Функция возвращает значение `true`, если символ `ch` является символом шестнадцатеричной цифры (цифры, а также буквы от А до F в нижнем или верхнем регистрах), или `false` в противном случае. Является аналогом функции `isxdigit` для многобайтных символов.

<code>isxdigit</code>	Проверяет, является ли символ символом шестнадцатеричной цифры	<code><ctype.h></code>

```
int isxdigit(int ch);
```

Функция выделяет младший байт параметра `ch` и возвращает значение `true`, если символ `ch` является символом шестнадцатеричной цифры (цифры, а также буквы от А до F в нижнем или верхнем регистрах), или `false` в противном случае.

<code>labs</code>	Возвращает модуль числа	<code><stdlib.h></code>

```
long int labs(long int num);
```

Функция возвращает абсолютное значение числа `num`.

<code>ldexp</code>	Преобразует мантиссу и показатель степени в число	<code><math.h></code>

```
double ldexp(double x, int exp);
```

Функция получает мантиссу `x` и показатель степени `exp` и возвращает число, равное произведению мантиссы на 2 в степени показатель степени. Противоположна функции `frexp`.

<code>ldiv</code>	Делит с остатком	<code><stdlib.h></code>

```
ldiv_t ldiv(long numerator, long denominator);
```

Функция делит длинное целое `numerator` на длинное целое `denominator`, возвращая результат в структуре, содержащей частное и остаток.

<code>localeconv</code>	Возвращает описание установок локальной среды	<code><locale.h></code>

```
struct lconv *localeconv(void);
```

Функция возвращает структуру, которая содержит детальное описание текущих установок локальной среды, то есть информацию, зависящую от страны.

<code>localtime</code>	Инициализирует структуру <code>tm</code> на основе <code>time_t</code>	<code><time.h></code>

```
struct tm localtime(time_t *t);
```

Функция возвращает указатель на временную структуру типа `tm`, инициализированную на основе даты/времени из `t`. Время устанавливается локальное (в отличие от `gmtime`, где время устанавливается по Гринвичу).

<code>log</code>	Вычисляет натуральный логарифм	<code><math.h></code>

```
double log(double x);
```

Функция возвращает значение натурального логарифма `x`.

<code>log10</code>	Вычисляет логарифм по основанию 10	<code><math.h></code>

```
double log10(double x);
```

Функция возвращает значение логарифма `x` по основанию 10.

<code>malloc</code>	Выделяет блок памяти	<code><stdlib.h></code>

```
void *malloc(size_t size);
```

Функция выделяет блок памяти размером `size` байт и возвращает указатель на него. В случае неудачи возвращаемым значением будет `NULL`. Предпочтительней вместо `malloc` использовать операцию `new`.

mblen	Определяет размер многобайтных символов	<stdlib.h>

```
int mblen(const char *s, size_t n);
```

Функция возвращает размер символа *s, если s не NULL.

mbstowcs	Преобразует строку многобайтных символов в массив из wchar_t	<stdlib.h>

```
size_t mbstowcs(wchar_t *pwc, const char *s, size_t n);
```

Функция преобразует строку многобайтных символов в массив из wchar_t. Если встретился недопустимый символ, функция возвращает -1. В противном случае функция возвращает количество преобразованных символов.

mbtowlc	Преобразует многобайтный символ в wchar_t	<stdlib.h>

```
int mbtowlc(wchar_t *pwc, const char *s, size_t n);
```

Функция преобразует многобайтный символ в wchar_t. Если s указывает на многобайтный символ, функция возвращает -1. Если s указывает на NULL, возвращаемое значение равно 0. В противном случае функция возвращает длину символа в байтах.

memchr	Ищет первое вхождение символа в блок памяти	<string.h>

```
void *memchr(const void *p, int ch, size_t n);
```

Функция возвращает указатель на первое вхождение байта, представленного младшим байтом аргумента ch в блоке памяти p длиной n.

memcmp	Сравнивает блоки памяти	<string.h>

```
int memcmp(const void *p1, const void *p2, size_t n);
```

Функция сравнивает два блока памяти и возвращает значение: меньше нуля, равное нулю или больше нуля — аналогично кодам возврата функции strcmp.

memcpy	Копирует блок памяти	<string.h>

```
void *memcpy(void *dest, const void *src, size_t n);
```

Функция копирует блок памяти длиной n байт из адреса src по адресу dest.

memmove	Переносит блок памяти	<string.h>

```
void *memmove(void *dest, const void *src, size_t n);
```

Функция аналогична memcpy, но блоки dest и src могут перекрываться.

memset	Заполняет блок памяти символом	<string.h>

```
void *memset(const void *p, int ch, size_t n);
```

Функция заполняет блок памяти символом, взятым из младшего байта `ch`.

mktime	Заполняет поля дня недели и дня года	<time.h>

```
struct tm mktime(time_t *t);
```

Функция заполняет поля дня недели и дня года структуры `tm` на основе имеющейся в `t` информации.

modf	Разбивает число на целую и дробную части	<math.h>

```
double modf(double x, double *intptr);
```

Функция разбивает `x` на целую и дробную части, причем дробную часть числа возвращает, а целую часть числа помещает по адресу, определяемому указателем `intptr`.

perror	Печатает строку вида "s: error сообщение"	<stdio.h>

```
void perror(const char *s);
```

Функция печатает строку вида «s: error сообщение», в которой сообщение определяется на основе значения глобальной переменной `errno`. Если значением `s` является `NULL`, то печатается только сообщение об ошибке. Вывод производится в стандартный поток ошибок `stderr`, обычно назначенный на монитор.

pow	Возводит число в степень	<math.h>

```
double pow(double x, double y);
```

Функция вычисляет значение числа `x` в степени `y`.

printf	Выводит строку параметров в определенном формате	<stdio.h>

```
int printf(const char *fmt, ...);
```

Функция выводит на стандартное устройство вывода значения переменных, перечисленных в списке, обозначенном многоточием, в формате, определенном строкой `format`. Спецификации формата приведены в приложении 2.

putc	Записывает символ в поток	<stdio.h>

```
int putc(int ch, FILE *f);
```

Функция записывает символ `ch` в поток `f`. При ошибке возвращает значение `EOF`, иначе — записанный символ. Работает полностью аналогично `fputc`.

putchar	Выводит символ на стандартное устройство вывода	<stdio.h>

```
int putchar(int ch);
```

Функция выводит символ *ch* на стандартное устройство вывода. Если вывод успешен, возвращается значение *ch*, иначе возвращается значение EOF.

puts	Выводит строку на стандартное устройство вывода	<stdio.h>

```
int puts(char *s);
```

Функция выводит строку *s* на стандартное устройство вывода, добавляя в конце символ новой строки. Возвращает неотрицательное значение при успехе или EOF — при ошибке.

putwc	Записывает символ в поток	<stdio.h>

```
wint_t putwc(wint_t ch, FILE *f);
```

Функция записывает символ *ch* в поток *f*. При ошибке возвращает значение WEOF, иначе — записанный символ. Является аналогом функции *putc* для многобайтных символов.

putwchar	Выводит символ на стандартное устройство вывода	<stdio.h>

```
wint_t putwchar(wint_t ch);
```

Функция выводит символ *ch* на стандартное устройство вывода. Если вывод успешен, возвращается значение *ch*, иначе возвращается значение WEOF. Является аналогом функции *putchar* для многобайтных символов.

qsort	Сортирует заданный массив	<stdlib.h>

```
void qsort(void *buf, size_t num, size_t size,
int (compare)(const void *, const void *));
```

Функция выполняет сортировку массивов любых размеров и типов. Сортируемый массив: *buf* — указатель на массив, *num* — число элементов, *size* — размер каждого элемента. Аргумент *compare* — адрес функции, которая должна сравнить два значения, переданных ей функцией *qsort*, и вернуть соответствующий код: меньше нуля (если первый аргумент меньше второго), нуль (если аргументы равны) или больше нуля (если первый аргумент больше второго). Аргументы, передаваемые ей, определены как *void**, то есть внутри функции они должны приводиться к нужному типу.

raise	Прерывает выполнение программы	<signal.h>

```
int raise(int signal);
```

Функция посылает сигнал программе, идентифицируя ошибку во время ее выполнения. Эта функция устарела и оставлена только для совместимости с C, так

как в C++ для таких целей используется механизм исключений. Возвращает нулевое значение в случае успеха и ненулевое — в противном случае.

rand	Генерирует случайные числа	<stdlib.h>

```
int rand(void):
```

Функция возвращает очередное псевдослучайное число в диапазоне от 0 до RAND_MAX.

realloc	Изменяет размеры ранее выделенного блока памяти	<stdlib.h>

```
void *realloc(void *mемblock, size_t size):
```

Функция получает блок памяти (выделенный ранее одной из функций malloc, calloc или realloc), при необходимости перемещает его так, чтобы он соответствовал новому размеру. При этом ранее размещенные в нем данные не теряются. Большинство правил, относящихся к malloc, действительны и для realloc.

Значение, возвращаемое функцией, является указателем на новое положение блока памяти. Оно отличается от старого значения (memblock) только в том случае, если realloc пришлось переместить блок памяти. Если не удалось увеличить блок памяти, то возвращаемое значение равно NULL, причем значение memblock остается доступным программе.

remove	Удаляет файл	<stdio.h>

```
int remove(const char *filename):
```

Функция удаляет существующий файл. В случае успеха возвращает нулевое значение, иначе — ненулевое.

rename	Переименовывает файл	<stdio.h>

```
int rename(const char *oldname, const char *newname):
```

Функция переименовывает существующий файл или папку. В случае успеха возвращает нулевое значение, иначе — ненулевое.

rewind	Очищает флаги ошибок при работе с потоком и переходит к началу файла	<stdio.h>

```
void rewind(FILE *f):
```

Функция очищает флаги ошибок в потоке f и устанавливает текущую позицию на начало файла.

scanf	Вводит строку параметров в определенном формате	<stdio.h>

```
int scanf(const char *fmt [.par1. par2. ...]):
```

Функция вводит строку параметров `par1`, `par2` и т. д. в формате, определенном строкой `fmt`, со стандартного устройства ввода (обычно с клавиатуры). Возвращает число переменных, которым присвоено значение.

<code>setbuf</code>	Устанавливает буферизацию потока ввода/вывода	<code><stdio.h></code>

```
void setbuf(FILE *f, char *p);
```

Функция устанавливает буфер ввода/вывода, заданный указателем `p`. Размер буфера должен быть длиной `BUFSIZ`. Если `p` равен `null`, то буферизация отменяется.

<code>setlocale</code>	Устанавливает новую локализацию	<code><locale.h></code>

```
char *setlocale(int type, const char *locale);
```

Функция устанавливает новую локализацию программы.

<code>setvbuf</code>	Преобразует строки на основе текущей локализации	<code><stdio.h></code>

```
void setvbuf(FILE *f, char *p, int mode, size_t size);
```

Функция устанавливает буфер ввода/вывода, заданный указателем `p`. Параметр `mode` указывает режим, а `size` — размер буфера. Возвращается значение 0 при успешном завершении и ненулевое значение в случае ошибки. Параметр `mode` является целым, принимающим одно из значений: `_IOFBF` (полная буферизация), `_IONBF` (запрет буферизации) и `_IOLBF` (для вывода — построчная буферизация, то есть опустошение буфера при записи в буфер символа новой строки).

<code>signal</code>	Регистрирует обработчик сигналов	<code><signal.h></code>

```
void (*signal(int signum, void (*func)(int)))(int);
```

Функция регистрирует обработчик сигналов. Несмотря на то, что обработчики исключений C++ обеспечивают лучший механизм управления ошибками, функции `signal` и `raise` включены в библиотеку для совместимости с C.

<code>sin</code>	Вычисляет синус	<code><math.h></code>

```
double sin(double x);
```

Функция возвращает значение синуса угла, равного `x` радиан.

<code>sinh</code>	Вычисляет гиперболический синус	<code><math.h></code>

```
double sinh(double x);
```

Функция возвращает значение гиперболического синуса угла, равного `x` радиан. Если значение функции окажется вне представимого диапазона, то функция возвращает значение `HUGEVAL`, а глобальная переменная `errno` получает значение `ERANGE`.

printf	Выводит строку параметров в определенном формате	<stdio.h>

```
int printf(char *buffer, const char *format[, argument. ]);
```

Функция выводит в строку *buffer* значения переменных, перечисленных в списке, обозначенном многоточием, в формате, определенном строкой *format*.

sqrt	Вычисляет квадратный корень	<math.h>

```
double sqrt(double x);
```

Функция возвращает квадратный корень из числа *x*.

srand	Устанавливает начальное псевдослучайное число	<stdlib.h>

```
void srand(unsigned int seed);
```

Функция устанавливает начальное значение последовательности псевдослучайных чисел.

sscanf	Вводит данные из строки	<stdio.h>

```
int sscanf(const char *buf, const char *format [.par1. par2. j]);
```

Функция аналогично функции *scanf* вводит данные, но не с клавиатуры, а из строки символов, переданной ей первым параметром. Аргумент *buf* — строка символов, из которой вводятся значения, *format* — строка формата, в соответствии с которой происходит преобразование данных, а многоточие указывает на наличие необязательных аргументов, соответствующих адресам вводимых значений.

strcat	Складывает строки	<string.h>

```
char *strcat(char *s1, char *s2);
```

Функция добавляет *s2* к *s1* и возвращает *s1*. В конец результирующей строки добавляется нуль-символ.

strchr	Ищет символ в строке	<string.h>

```
char *strchr(char *s, int ch);
```

Функция возвращает указатель на первое вхождение символа *ch* в строку *s*, если его нет, то возвращается *NULL*.

strcmp	Сравнивает строки	<string.h>

```
int strcmp(char *s1, char *s2);
```

Функция сравнивает строки и возвращает отрицательное (если *s1* меньше *s2*), нулевое (если *s1* равно *s2*) или положительное (если *s1* больше *s2*) значение.

strcoll	Сравнивает строки с учетом установленной локализации	<string.h>

```
char *strcoll(char *s1, char *s2);
```

Функция сравнивает строки аналогично strcmp, но учитывает установки локализации (см. функцию setlocale).

strcpy	Копирует одну строку в другую	<string.h>

```
char *strcpy(char *s1, char *s2);
```

Функция копирует s2 в s1 и возвращает s1.

strcspn	Ищет один из символов одной строки в другой	<string.h>

```
size_t strcspn(char *s1, char *s2);
```

Функция возвращает значение индекса любого из символов из s2 в строке s1.

strerror	Возвращает указатель на строку с описанием ошибки	<string.h>

```
char *strerror(size_t n);
```

Функция возвращает указатель на строку с описанием ошибки номер n.

strftime	Преобразует время в формате fmt в формат tm	<time.h>

```
struct tm strftime(char *s, size_t size, const struct tm *ctm);
```

Функция возвращает отформатированную строку с датой и временем на основе формата fmt. Значение функции имеет тип time_t, соответствующий типу tm.

strlen	Возвращает длину строки	<string.h>

```
size_t strlen(char *s);
```

Функция возвращает длину строки (без учета символа завершения строки).

strncat	Складывает одну строку с n символами другой	<string.h>

```
char *strncat(char *s1, char *s2, size_t n);
```

Функция добавляет не более n символов из s2 к s1 и возвращает s1. Первый символ s2 пишется на место завершающего нуль-символа строки s1. Если длина строки s2 меньше n, переписываются все символы s2. К строке s1 добавляется нуль-символ. Если строки перекрываются, поведение не определено.

strncmp	Сравнивает одну строку с n символами другой	<string.h>

```
int strncmp(char *s1, char *s2, size_t n);
```

Функция сравнивает первую строку и первые n символов второй строки и возвращает отрицательное (если $s1$ меньше $s2$), нулевое (если $s1$ равно $s2$) или положительное (если $s1$ больше $s2$) значение.

strncpy	Копирует первые n символов одной строки в другую	<string.h>

```
char *strncpy(char *s1, char *s2, size_t n);
```

Функция копирует не более n символов из $s2$ в $s1$ и возвращает $s1$. Если длина исходной строки превышает или равна n , нуль-символ в конец строки $s1$ не добавляется. В противном случае строка дополняется нуль-символами до n -го символа. Если строки перекрываются, поведение не определено.

strpbrk	Ищет один из символов одной строки в другой	<string.h>

```
char *strpbrk(char *s1, char *s2);
```

Функция возвращает указатель на символ, являющийся первым вхождением любого из символов из $s2$ в строку $s1$, если его нет, возвращается NULL.

strrchr	Ищет символ в строке	<string.h>

```
char *strrchr(char *s, int ch);
```

Функция возвращает указатель на первое вхождение символа ch в строку s справа, если его нет, возвращает NULL.

strspn	Ищет символ одной строки, отсутствующий в другой	<string.h>

```
size_t strspn(char *s1, char *s2);
```

Функция возвращает индекс первого символа в $s1$, отсутствующего в $s2$.

strstr	Ищет подстроку в строке	<string.h>

```
char *strstr(char *s1, char *s2);
```

Функция выполняет поиск первого вхождения подстроки $s2$ в строку $s1$. В случае удачного поиска, возвращает указатель на элемент из $s1$, с которого начинается $s2$, и NULL в противном случае.

strtod	Преобразует строку в число	<stdlib.h>

```
double strtod(const char *str, char **end);
```

Функция преобразует строку символов в числовое значение и возвращает его. При переполнении возвращает $\pm$ HUGE_VAL. При невозможности выполнить преобразование или исчезновении порядка возвращает 0. В обоих последних случаях $errno$ устанавливается в ERANGE. end указывает на символ, на котором преобразование завершается.

strtok	Выделяет из строки лексемы	<string.h>

```
char *strtok(char *s1, char *s2):
```

Функция возвращает следующую лексему из s1, отделенную любым из символов из набора s2.

strtol	Преобразует строку в число с учетом системы счисления	<stdlib.h>

```
double strtol(const char *str, char **end, int radix):
```

Функция преобразует строку символов в числовое значение с учетом указанной системы счисления radix и возвращает полученное число. Функция пропускает возможные начальные пробелы и заканчивает преобразование на первом символе, который не может появиться в образе числа. Параметр end является адресом указателя типа char*; этот указатель будет содержать адрес первого непреобразованного символа.

При переполнении возвращает LONG_MAX или LONG_MIN. При невозможности выполнить преобразование возвращает 0. В обоих последних случаях errno устанавливается в ERANGE.

strtoul	Преобразует строку в число с учетом системы счисления	<stdlib.h>

```
double strtoul(const char *str, char **end, int radix):
```

Функция работает аналогично strtol, но работает с беззнаковым длинным целым. При переполнении возвращает ULONG_MAX.

strxfrm	Преобразует строки на основе текущей локализации	<string.h>

```
size_t strxfrm(char *s1, char *s2, size_t n):
```

Функция преобразует строку из s2 и помещает ее в s1 на основе текущей локализации. Преобразуется не более n символов.

swprintf	Выводит строку параметров в определенном формате	<stdio.h>

```
int swprintf(wchar_t *buffer, const wchar_t *format[, argument. ...]):
```

Функция выводит в строку buffer значения переменных, перечисленных в списке, обозначенном многоточием, в формате, определенном строкой format. Является аналогом функции sprintf для многобайтных символов.

swscanf	Вводит данные из строки	<stdio.h>

```
int swscanf(const wchar_t *buf, const wchar_t *format, ...):
```

Функция аналогично функции scanf вводит данные, но не с клавиатуры, а из строки символов, переданной ей первым параметром. Аргумент buf — строка символов, из которой вводятся значения, format — строка формата, в соответ-

вии с которой происходит преобразование данных, а многоточие указывает на наличие необязательных аргументов, соответствующих адресам вводимых значений. Является аналогом функции `sscanf` для многобайтных символов.

system	Передает строку командному процессору ОС	<stdlib.h>

```
int system(const char *command);
```

Функция передает строку командному процессору операционной системы. Это позволяет исполнить другую программу или утилиту. Возвращаемое значение зависит от реализации, но, как правило, значение 0 указывает на успешное завершение.

tan	Возвращает тангенс аргумента	<math.h>

```
double tan(double x);
```

Функция возвращает значение тангенса аргумента x .

tanh	Возвращает гиперболический тангенс аргумента	<math.h>

```
double tanh(double x);
```

Функция возвращает значение гиперболического тангенса аргумента x .

time	Возвращает текущие дату/время в виде <code>time_t</code>	<time.h>

```
time_t time(time_t *t);
```

Функция возвращает текущие дату/время в виде `time_t`. Если `*t` не равно `NULL`, то значение заносится и по этому адресу.

tmpfile	Открывает поток двоичного ввода/вывода во временный файл	<stdio.h>

```
FILE *tmpfile(void);
```

Функция открывает временный файл в режиме двоичного ввода/вывода и возвращает указатель на поток.

tmpnam	Создает уникальное имя файла	<stdio.h>

```
char *tmpnam(char *s);
```

Функция создает уникальное имя файла, которое может быть успешно использовано как имя временного файла. Различные строки генерируются `TMP_MAX` раз. `TMP_MAX` определена в `<stdio.h>` и равна 65535. Параметр `s` — либо 0, либо указатель на массив, состоящий не менее, чем из `L_tmpnam` символов. `L_tmpnam` определено в `<stdio.h>`. Если `s` — ноль, `tmpnam` оставляет имя генерированного временного файла во внутреннем статическом объекте и возвращает указатель на этот объект.

Если *s* не ноль, *tolower* помещает свой результат в указанный массив, который должен быть длиной не менее *L_toupper* символов, и возвращает *s*.

<code>tolower</code>	Возвращает символ в нижнем регистре	<code><ctype.h></code>

```
int tolower(int ch);
```

Функция получает параметр *ch* и возвращает его в нижнем регистре. В параметре *ch* используется только младший байт.

<code>toupper</code>	Возвращает символ в верхнем регистре	<code><ctype.h></code>

```
int toupper(int ch);
```

Функция получает параметр *ch* и возвращает его в верхнем регистре. В параметре *ch* используется только младший байт.

<code>tolower</code>	Возвращает символ в нижнем регистре	<code><ctype.h></code>

```
int tolower(wint_t ch);
```

Функция получает символ *ch* и возвращает его в нижнем регистре. Является аналогом функции *tolower* для многобайтных символов.

<code>toupper</code>	Возвращает символ в верхнем регистре	<code><ctype.h></code>

```
int toupper(wint_t ch);
```

Функция получает символ *ch* и возвращает его в верхнем регистре. Является аналогом функции *toupper* для многобайтных символов.

<code>ungetc</code>	Возвращает символ в поток	<code><stdio.h></code>

```
int ungetc(int ch, FILE *f);
```

Функция возвращает символ *ch* обратно во входной поток *f*. Возвращает EOF в случае ошибки, в противном случае возвращает *ch*. Гарантируется возвращение только одного символа.

<code>ungetwc</code>	Возвращает символ в поток	<code><stdio.h></code>

```
wint_t ungetwc(wint_t ch, FILE *f);
```

Функция возвращает символ *ch* обратно во входной поток *f*. Возвращает WEOF в случае ошибки, в противном случае возвращает *ch*. Гарантируется возвращение только одного символа. Является аналогом функции *ungetc* для многобайтных символов.

<code>fprintf</code>	Посылает отформатированный вывод в поток	<code><stdio.h></code>

```
int fprintf(FILE *stream, char *format, va_list arglist);
```

Функция аналогична функции `fprintf`, но имеет доступ к указателю на список аргументов, а не к самому списку.

<code>vfwprintf</code>	Посылает отформатированный вывод в поток	<code><stdio.h></code>

```
int vfwprintf(FILE *stream, wchar_t *format, va_list arglist);
```

Функция аналогична функции `fprintf`, но имеет доступ к указателю на список аргументов, а не к самому списку, а также работает с многобайтными символами.

<code>vprintf</code>	Посылает отформатированный вывод в стандартный поток вывода	<code><stdio.h></code>

```
int vprintf(const char *format, va_list arglist);
```

Функция аналогична функции `printf`, но имеет доступ к указателю на список аргументов, а не к самому списку.

<code>vsprintf</code>	Выводит строку параметров в определенном формате	<code><stdio.h></code>

```
int vsprintf(char *string, const char *format, va_list arglist);
```

Функция аналогична функции `sprintf`, но имеет доступ к указателю на список аргументов, а не к самому списку.

<code>vswprintf</code>	Выводит строку параметров в определенном формате	<code><stdio.h></code>

```
int vswprintf(wchar_t *string, const wchar_t *format, va_list arglist);
```

Функция аналогична функции `sprintf`, но имеет доступ к указателю на список аргументов, а не к самому списку, а также работает с многобайтными символами.

<code>wprintf</code>	Посылает отформатированный вывод в стандартный поток вывода	<code><stdio.h></code>

```
int wprintf(const wchar_t *format, va_list arglist);
```

Функция аналогична функции `printf`, но имеет доступ к указателю на список аргументов, а не к самому списку, а также работает с многобайтными символами.

<code>wscat</code>	Складывает строки	<code><string.h></code>

```
wchar_t *wscat(wchar_t *s1, wchar_t *s2);
```

Функция добавляет `s2` к `s1` и возвращает `s1`. Является аналогом функции `strcat` для многобайтных символов.

<code>wcschr</code>	Ищет символ в строке	<code><string.h></code>

```
wchar_t *wcschr(wchar_t *s, int ch);
```

вии с которой происходит преобразование данных, а многоточие указывает на наличие необязательных аргументов, соответствующих адресам вводимых значений. Является аналогом функции `sscanf` для многобайтных символов.

system	Передает строку командному процессору ОС	<stdlib.h>

```
int system(const char *command);
```

Функция передает строку командному процессору операционной системы. Это позволяет исполнить другую программу или утилиту. Возвращаемое значение зависит от реализации, но, как правило, значение 0 указывает на успешное завершение.

tan	Возвращает тангенс аргумента	<math.h>

```
double tan(double x);
```

Функция возвращает значение тангенса аргумента x .

tanh	Возвращает гиперболический тангенс аргумента	<math.h>

```
double tanh(double x);
```

Функция возвращает значение гиперболического тангенса аргумента x .

time	Возвращает текущие дату/время в виде <code>time_t</code>	<time.h>

```
time_t time(time_t *t);
```

Функция возвращает текущие дату/время в виде `time_t`. Если `*t` не равно `NULL`, то значение заносится и по этому адресу.

tmpfile	Открывает поток двоичного ввода/вывода во временный файл	<stdio.h>

```
FILE *tmpfile(void);
```

Функция открывает временный файл в режиме двоичного ввода/вывода и возвращает указатель на поток.

tmpnam	Создает уникальное имя файла	<stdio.h>

```
char *tmpnam(char *s);
```

Функция создает уникальное имя файла, которое может быть успешно использовано как имя временного файла. Различные строки генерируются `TMP_MAX` раз. `TMP_MAX` определена в `<stdio.h>` и равна 65535. Параметр `s` — либо 0, либо указатель на массив, состоящий не менее, чем из `L_tmpnam` символов. `L_tmpnam` определено в `<stdio.h>`. Если `s` — ноль, `tmpnam` оставляет имя генерированного временного файла во внутреннем статическом объекте и возвращает указатель на этот объект.

вии с которой происходит преобразование данных, а многоточие указывает на наличие необязательных аргументов, соответствующих адресам вводимых значений. Является аналогом функции `sscanf` для многобайтных символов.

<code>system</code>	Передает строку командному процессору ОС	<code><stdlib.h></code>

```
int system(const char *command);
```

Функция передает строку командному процессору операционной системы. Это позволяет исполнить другую программу или утилиту. Возвращаемое значение зависит от реализации, но, как правило, значение 0 указывает на успешное завершение.

<code>tan</code>	Возвращает тангенс аргумента	<code><math.h></code>

```
double tan(double x);
```

Функция возвращает значение тангенса аргумента x .

<code>tanh</code>	Возвращает гиперболический тангенс аргумента	<code><math.h></code>

```
double tanh(double x);
```

Функция возвращает значение гиперболического тангенса аргумента x .

<code>time</code>	Возвращает текущие дату/время в виде <code>time_t</code>	<code><time.h></code>

```
time_t time(time_t *t);
```

Функция возвращает текущие дату/время в виде `time_t`. Если `*t` не равно `NULL`, то значение заносится и по этому адресу.

<code>tmpfile</code>	Открывает поток двоичного ввода/вывода во временный файл	<code><stdio.h></code>

```
FILE *tmpfile(void);
```

Функция открывает временный файл в режиме двоичного ввода/вывода и возвращает указатель на поток.

<code>tmpnam</code>	Создает уникальное имя файла	<code><stdio.h></code>

```
char *tmpnam(char *s);
```

Функция создает уникальное имя файла, которое может быть успешно использовано как имя временного файла. Различные строки генерируются `TMP_MAX` раз. `TMP_MAX` определена в `<stdio.h>` и равна 65535. Параметр `s` — либо 0, либо указатель на массив, состоящий не менее, чем из `L_tmpnam` символов. `L_tmpnam` определено в `<stdio.h>`. Если `s` — ноль, `tmpnam` оставляет имя генерированного временного файла во внутреннем статическом объекте и возвращает указатель на этот объект.

вии с которой происходит преобразование данных, а многоточие указывает на наличие необязательных аргументов, соответствующих адресам вводимых значений. Является аналогом функции `sscanf` для многобайтных символов.

system	Передает строку командному процессору ОС	<stdlib.h>

```
int system(const char *command);
```

Функция передает строку командному процессору операционной системы. Это позволяет исполнить другую программу или утилиту. Возвращаемое значение зависит от реализации, но, как правило, значение 0 указывает на успешное завершение.

tan	Возвращает тангенс аргумента	<math.h>

```
double tan(double x);
```

Функция возвращает значение тангенса аргумента x .

tanh	Возвращает гиперболический тангенс аргумента	<math.h>

```
double tanh(double x);
```

Функция возвращает значение гиперболического тангенса аргумента x .

time	Возвращает текущие дату/время в виде <code>time_t</code>	<time.h>

```
time_t time(time_t *t);
```

Функция возвращает текущие дату/время в виде `time_t`. Если `*t` не равно `NULL`, то значение заносится и по этому адресу.

tmpfile	Открывает поток двоичного ввода/вывода во временный файл	<stdio.h>

```
FILE *tmpfile(void);
```

Функция открывает временный файл в режиме двоичного ввода/вывода и возвращает указатель на поток.

tmpnam	Создает уникальное имя файла	<stdio.h>

```
char *tmpnam(char *s);
```

Функция создает уникальное имя файла, которое может быть успешно использовано как имя временного файла. Различные строки генерируются `TMP_MAX` раз. `TMP_MAX` определена в `<stdio.h>` и равна 65535. Параметр `s` — либо 0, либо указатель на массив, состоящий не менее, чем из `L_tmpnam` символов. `L_tmpnam` определено в `<stdio.h>`. Если `s` — ноль, `tmpnam` оставляет имя генерированного временного файла во внутреннем статическом объекте и возвращает указатель на этот объект.

strtok	Выделяет из строки лексемы	<string.h>

```
char *strtok(char *s1, char *s2);
```

Функция возвращает следующую лексему из s1, отделенную любым из символов из набора s2.

strtol	Преобразует строку в число с учетом системы счисления	<stdlib.h>

```
double strtol(const char *str, char **end, int radix);
```

Функция преобразует строку символов в числовое значение с учетом указанной системы счисления radix и возвращает полученное число. Функция пропускает возможные начальные пробелы и заканчивает преобразование на первом символе, который не может появиться в образе числа. Параметр end является адресом указателя типа char*; этот указатель будет содержать адрес первого непреобразованного символа.

При переполнении возвращает LONG_MAX или LONG_MIN. При невозможности выполнить преобразование возвращает 0. В обоих последних случаях errno устанавливается в ERANGE.

strtoul	Преобразует строку в число с учетом системы счисления	<stdlib.h>

```
double strtoul(const char *str, char **end, int radix);
```

Функция работает аналогично strtol, но работает с беззнаковым длинным целым. При переполнении возвращает ULONG_MAX.

strxfrm	Преобразует строки на основе текущей локализации	<string.h>

```
size_t strxfrm(char *s1, char *s2, size_t n);
```

Функция преобразует строку из s2 и помещает ее в s1 на основе текущей локализации. Преобразуется не более n символов.

swprintf	Выводит строку параметров в определенном формате	<stdio.h>

```
int swprintf(wchar_t *buffer, const wchar_t *format[, argument. ...]);
```

Функция выводит в строку buffer значения переменных, перечисленных в списке, обозначенном многоточием, в формате, определенном строкой format. Является аналогом функции sprintf для многобайтных символов.

swscanf	Вводит данные из строки	<stdio.h>

```
int swscanf(const wchar_t *buf, const wchar_t *format, ...);
```

Функция аналогично функции scanf вводит данные, но не с клавиатуры, а из строки символов, переданной ей первым параметром. Аргумент buf — строка символов, из которой вводятся значения, format — строка формата, в соответст-

вии с которой происходит преобразование данных, а многоточие указывает на наличие необязательных аргументов, соответствующих адресам вводимых значений. Является аналогом функции `sscanf` для многобайтных символов.

system	Передает строку командному процессору ОС	<stdlib.h>

```
int system(const char *command);
```

Функция передает строку командному процессору операционной системы. Это позволяет исполнить другую программу или утилиту. Возвращаемое значение зависит от реализации, но, как правило, значение 0 указывает на успешное завершение.

tan	Возвращает тангенс аргумента	<math.h>

```
double tan(double x);
```

Функция возвращает значение тангенса аргумента x .

tanh	Возвращает гиперболический тангенс аргумента	<math.h>

```
double tanh(double x);
```

Функция возвращает значение гиперболического тангенса аргумента x .

time	Возвращает текущие дату/время в виде <code>time_t</code>	<time.h>

```
time_t time(time_t *t);
```

Функция возвращает текущие дату/время в виде `time_t`. Если `*t` не равно `NULL`, то значение заносится и по этому адресу.

tmpfile	Открывает поток двоичного ввода/вывода во временный файл	<stdio.h>

```
FILE *tmpfile(void);
```

Функция открывает временный файл в режиме двоичного ввода/вывода и возвращает указатель на поток.

tmpnam	Создает уникальное имя файла	<stdio.h>

```
char *tmpnam(char *s);
```

Функция создает уникальное имя файла, которое может быть успешно использовано как имя временного файла. Различные строки генерируются `TMP_MAX` раз. `TMP_MAX` определена в `<stdio.h>` и равна 65535. Параметр `s` — либо 0, либо указатель на массив, состоящий не менее, чем из `L_tmpnam` символов. `L_tmpnam` определено в `<stdio.h>`. Если `s` — ноль, `tmpnam` оставляет имя генерированного временного файла во внутреннем статическом объекте и возвращает указатель на этот объект.

Если *s* не ноль, *tmpnam* помещает свой результат в указанный массив, который должен быть длиной не менее *L_tmpnam* символов, и возвращает *s*.

<code>tolower</code>	Возвращает символ в нижнем регистре	<code><ctype.h></code>

```
int tolower(int ch);
```

Функция получает параметр *ch* и возвращает его в нижнем регистре. В параметре *ch* используется только младший байт.

<code>toupper</code>	Возвращает символ в верхнем регистре	<code><ctype.h></code>

```
int toupper(int ch);
```

Функция получает параметр *ch* и возвращает его в верхнем регистре. В параметре *ch* используется только младший байт.

<code>towlower</code>	Возвращает символ в нижнем регистре	<code><ctype.h></code>

```
int tolower(wint_t ch);
```

Функция получает символ *ch* и возвращает его в нижнем регистре. Является аналогом функции *tolower* для многобайтных символов.

<code>toupper</code>	Возвращает символ в верхнем регистре	<code><ctype.h></code>

```
int toupper(wint_t ch);
```

Функция получает символ *ch* и возвращает его в верхнем регистре. Является аналогом функции *toupper* для многобайтных символов.

<code>ungetc</code>	Возвращает символ в поток	<code><stdio.h></code>

```
int ungetc(int ch, FILE *f);
```

Функция возвращает символ *ch* обратно во входной поток *f*. Возвращает EOF в случае ошибки, в противном случае возвращает *ch*. Гарантируется возвращение только одного символа.

<code>ungetwc</code>	Возвращает символ в поток	<code><stdio.h></code>

```
wint_t ungetwc(wint_t ch, FILE *f);
```

Функция возвращает символ *ch* обратно во входной поток *f*. Возвращает WEOF в случае ошибки, в противном случае возвращает *ch*. Гарантируется возвращение только одного символа. Является аналогом функции *ungetc* для многобайтных символов.

<code>vfprintf</code>	Посылает отформатированный вывод в поток	<code><stdio.h></code>

```
int vfprintf(FILE *stream, char *format, va_list arglist);
```

Функция аналогична функции `fprintf`, но имеет доступ к указателю на список аргументов, а не к самому списку.

<code>vfwprintf</code>	Посылает отформатированный вывод в поток	<code><stdio.h></code>

```
int vfwprintf(FILE *stream, wchar_t *format, va_list arglist);
```

Функция аналогична функции `fprintf`, но имеет доступ к указателю на список аргументов, а не к самому списку, а также работает с многобайтными символами.

<code>vprintf</code>	Посылает отформатированный вывод в стандартный поток вывода	<code><stdio.h></code>

```
int vprintf(const char *format, va_list arglist);
```

Функция аналогична функции `printf`, но имеет доступ к указателю на список аргументов, а не к самому списку.

<code>vsprintf</code>	Выводит строку параметров в определенном формате	<code><stdio.h></code>

```
int vsprintf(char *string, const char *format, va_list arglist);
```

Функция аналогична функции `sprintf`, но имеет доступ к указателю на список аргументов, а не к самому списку.

<code>vswprintf</code>	Выводит строку параметров в определенном формате	<code><stdio.h></code>

```
int vswprintf(wchar_t *string, const wchar_t *format, va_list arglist);
```

Функция аналогична функции `sprintf`, но имеет доступ к указателю на список аргументов, а не к самому списку, а также работает с многобайтными символами.

<code>wprintf</code>	Посылает отформатированный вывод в стандартный поток вывода	<code><stdio.h></code>

```
int wprintf(const wchar_t *format, va_list arglist);
```

Функция аналогична функции `printf`, но имеет доступ к указателю на список аргументов, а не к самому списку, а также работает с многобайтными символами.

<code>wcscat</code>	Складывает строки	<code><string.h></code>

```
wchar_t *wcscat(wchar_t *s1, wchar_t *s2);
```

Функция добавляет `s2` к `s1` и возвращает `s1`. Является аналогом функции `strcat` для многобайтных символов.

<code>wcschr</code>	Ищет символ в строке	<code><string.h></code>

```
wchar_t *wcschr(wchar_t *s, int ch);
```

Функция возвращает указатель на первое вхождение символа *ch* в строку *s*, если его нет, то возвращается *NULL*. Является аналогом функции *strchr* для многобайтных символов.

<code>wscmp</code>	Сравнивает строки	<code><string.h></code>

```
int *wscmp(wchar_t *s1, wchar_t *s2);
```

Функция сравнивает строки и возвращает отрицательное (если *s1* меньше *s2*), нулевое (если *s1* равно *s2*) или положительное (если *s1* больше *s2*) значение.

<code>wscoll</code>	Сравнивает строки с учетом установленной локализации	<code><string.h></code>

```
wchar_t *wscoll(wchar_t *s1, wchar_t *s2);
```

Функция сравнивает строки аналогично *strcmp*, но учитывает установки локализации (см. функцию *setlocale*). Является аналогом функции *strcoll* для многобайтных символов.

<code>wscpy</code>	Копирует одну строку в другую	<code><string.h></code>

```
wchar_t *wscpy(wchar_t *s1, wchar_t *s2);
```

Функция копирует *s2* в *s1* и возвращает *s1*. Является аналогом функции *strcpy* для многобайтных символов.

<code>wscspn</code>	Ищет один из символов одной строки в другой	<code><string.h></code>

```
size_t wscspn(wchar_t *s1, char *s2);
```

Функция возвращает значение индекса любого из символов из *s2* в строке *s1*. Является аналогом функции *strcspn* для многобайтных символов.

<code>wcsftime</code>	Преобразует время в формате <i>fmt</i> в формат <i>tm</i>	<code><time.h></code>

```
struct tm wcsftime(wchar_t *s, size_t size, const struct tm *ctm);
```

Функция возвращает отформатированную строку с датой и временем на основе формата *fmt*. Значение функции имеет тип *time_t*, соответствующий типу *tm*. Является аналогом функции *strftime* для многобайтных символов.

<code>wcslen</code>	Возвращает длину строки	<code><string.h></code>

```
size_t wcslen(wchar_t *s);
```

Функция возвращает длину строки (без учета символа завершения строки). Является аналогом функции *strlen* для многобайтных символов.

<code>wcsncat</code>	Складывает одну строку с <i>n</i> символами другой	<code><string.h></code>

```
wchar_t *wcsncat(wchar_t *s1, wchar_t *s2, size_t n);
```

Функция добавляет *s2* к *s1*, но не более *n* символов, и возвращает *s1*. Является аналогом функции *strncat* для многобайтных символов.

wcsncmp	Сравнивает одну строку с <i>n</i> символами другой	<string.h>

```
int wcsncmp(wchar_t *s1, wchar_t *s2, size_t n);
```

Функция сравнивает первую строку и первые *n* символов второй строки, и возвращает отрицательное (если *s1* меньше *s2*), нулевое (если *s1* равно *s2*) или положительное (если *s1* больше *s2*) значение. Является аналогом функции *strncmp* для многобайтных символов.

wcsncpy	Копирует первые <i>n</i> символов одной строки в другую	<string.h>

```
wchar_t *wcsncpy(wchar_t *s1, wchar_t *s2, size_t n);
```

Функция копирует *s2* в *s1*, но не более *n* символов, и возвращает *s1*. Является аналогом функции *strncpy* для многобайтных символов.

wcspbrk	Ищет один из символов одной строки в другой	<string.h>

```
wchar_t *wcspbrk(wchar_t *s1, wchar_t *s2);
```

Функция возвращает указатель на символ, являющийся первым вхождением любого из символов из *s2* в строку *s1*, если его нет, то возвращается NULL. Является аналогом функции *strpbrk* для многобайтных символов.

wcsrchr	Ищет символ в строке	<string.h>

```
char *wcsrchr(wchar_t *s, wchar_t ch);
```

Функция возвращает указатель на первое вхождение символа *ch* в строку *s* справа, если его нет, то возвращает NULL. Является аналогом функции *strrchr* для многобайтных символов.

wcsspn	Ищет символ одной строки, отсутствующий в другой	<string.h>

```
size_t wcsspn(wchar_t *s1, wchar_t *s2);
```

Функция возвращает индекс первого символа в *s1*, отсутствующего в *s2*. Является аналогом функции *strspn* для многобайтных символов.

wcsstr	Ищет подстроку в строке	<string.h>

```
wchar_t *wcsstr(wchar_t *s1, wchar_t *s2);
```

Функция выполняет поиск первого вхождения подстроки *s2* в строку *s1*. В случае удачного поиска, возвращает указатель на элемент из *s1*, с которого начинается *s2*, и NULL в противном случае. Является аналогом функции *strstr* для многобайтных символов.

wcstod	Преобразует строку в число	<stdlib.h>

```
double wcstod(const wchar_t *str, wchar_t **end);
```

Функция преобразует строку символов в числовое значение и возвращает его. Является аналогом функции strtod для многобайтных символов.

wcstok	Выделяет из строки лексемы	<string.h>

```
wchar_t *wcstok(wchar_t *s1, wchar_t *s2);
```

Функция возвращает следующую лексему из s1, отделенную любым из символов из набора s2. Является аналогом функции strtok для многобайтных символов.

wcstol	Преобразует строку в число с учетом системы счисления	<stdlib.h>

```
double wcstol(const wchar_t *str, wchar_t **end, int radix);
```

Функция преобразует строку символов в числовое значение с учетом указанной системы счисления radix и возвращает полученное число. Функция пропускает возможные начальные пробелы и заканчивает преобразование на первом символе, который не может появиться в образе числа. Параметр end является адресом указателя типа char*; этот указатель будет содержать адрес первого непреобразованного символа. Является аналогом функции strtol для многобайтных символов.

wcstombs	Преобразует wchar_t в строку многобайтных символов	<stdlib.h>

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Функция преобразует wchar_t в строку многобайтных символов. Если s указывает на немногобайтный символ, функция возвращает -1. В противном случае функция возвращает количество преобразованных байт.

wcstoul	Преобразует строку в число с учетом системы счисления	<stdlib.h>

```
double wcstoul(const wchar_t *str, wchar_t **end, int radix);
```

Функция работает аналогично strtol, но с беззнаковым длинным целым. Является аналогом функции strtoul для многобайтных символов.

wcstrxfrm	Преобразует строки на основе текущей локализации	<string.h>

```
size_t wcstrxfrm(wchar_t *s1, wchar_t *s2, size_t n);
```

Функция преобразует строку из s2 и помещает ее в s1 на основе текущей локализации. Преобразуется не более n символов. Является аналогом функции strxfrm для многобайтных символов.

wctomb	Преобразует wchar_t в многобайтный символ	<stdlib.h>

```
int wctomb(char *s, wchar_t wc);
```

Функция преобразует wchar_t в многобайтный символ. Если указатель s не NULL, функция возвращает ненулевое значение, если многобайтный символ имеет регионо-зависимое соответствие, ноль в противном случае. Если указатель NULL, wctomb возвращает -1.

wmemchr	Ищет первое вхождение символа в блок памяти	<wchar.h>	<string.h>

```
wchar_t *wmemchr(wchar_t *p, wchar_t c, size_t n);
```

Функция возвращает указатель на первое вхождение аргумента c в блоке памяти p длиной n. Является аналогом функции memchr для многобайтных символов.

wmemcmp	Сравнивает блоки памяти	<wchar.h>

```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Функция сравнивает два блока памяти и возвращает значение: меньше нуля, равное нулю или больше нуля — аналогично кодам возврата функции strcmp. Является аналогом функции memcmp для многобайтных символов.

wmemcpy	Копирует блок памяти	<string.h>

```
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

Функция копирует блок памяти длиной n байт из адреса s2 по адресу s1. Является аналогом функции memcpy для многобайтных символов.

wmemmove	Переносит блок памяти	<string.h>

```
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

Функция аналогична memcpy, но блоки s1 и s2 могут перекрываться. Является аналогом функции memmove для многобайтных символов.

wmemset	Заполняет блок памяти символом	<string.h>

```
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Функция заполняет блок памяти символом c. Является аналогом функции memset для многобайтных символов.

wprintf	Выводит строку параметров в определенном формате	<stdio.h>

```
int printf(const wchar_t *fmt, ...);
```

Функция выводит на стандартное устройство вывода значения переменных, перечисленных в списке, обозначенном многоточием, в формате, определенном строкой `format`. Является аналогом функции `printf` для многобайтных символов.

<code>wscanf</code>	Вводит строку параметров в определенном формате	<code><stdio.h></code>

```
int wscanf(const wchar_t *fmt [.par1. par2. ...]):
```

Функция вводит строку параметров `par1`, `par2` и т. д. в формате, определенном строкой `fmt`, со стандартного устройства ввода (обычно с клавиатуры). Возвращает число переменных, которым присвоено значение. Является аналогом функции `scanf` для многобайтных символов.

ПРИЛОЖЕНИЕ 7

Таблицы кодов ASCII

Символы с кодами 0–127

код	симв.	код	симв.	код	симв.	код	симв.
0	nul	32		64	@	96	`
1	soh	33	!	65	A	97	a
2	stx	34	"	66	B	98	b
3	etx	35	#	67	C	99	c
4	eot	36	\$	68	D	100	d
5	enq	37	%	69	E	101	e
6	ack	38	&	70	F	102	f
7	bel	39	'	71	G	103	g
8	bs	40	(	72	H	104	h
9	ht	41	)	73	I	105	i
10	lf	42	*	74	J	106	j
11	vt	43	+	75	K	107	k
12	ff	44	,	76	L	108	l
13	cr	45	-	77	M	109	m
14	soh	46	.	78	N	110	n
15	si	47	/	79	O	111	o
16	dle	48	0	80	P	112	p
17	dc1	49	1	81	Q	113	q
18	dc2	50	2	82	R	114	r
19	dc3	51	3	83	S	115	s
20	dc4	52	4	84	T	116	t
21	nak	53	5	85	U	117	u
22	syn	54	6	86	V	118	v
23	etb	55	7	87	W	119	w
24	can	56	8	88	X	120	x
25	em	57	9	89	Y	121	y
26	sub	58	:	90	Z	122	z
27	esc	59	;	91	[	123	{
28	fs	60	<	92	\	124	
29	gs	61	=	93	]	125	}
30	rs	62	>	94	^	126	~
31	us	63	?	95	_	127	del

Символы с кодами 128–255 (кодировка таблицы 1251 — Windows)

код	симв.	код	симв.	код	симв.	код	симв.
128		160		192	А	224	а
129		161	Ў	193	Б	225	б
130	,	162	ў	194	В	226	в
131	ѓ	163	Ј	195	Г	227	г
132	„	164	Ѡ	196	Д	228	д
133	...	165	Ѓ	197	Е	229	е
134	†	166		198	Ж	230	ж
135	‡	167	§	199	З	231	з
136	^	168	Ё	200	И	232	и
137	‰	169	©	201	Й	233	й
138	Љ	170	Є	202	К	234	к
139	<	171	•	203	Л	235	л
140	Њ	172	¬	204	М	236	м
141		173		205	Н	237	н
142		174	®	206	О	238	о
143		175	Ї	207	П	239	п
144		176	•	208	Р	240	р
145	‘	177	±	209	С	241	с
146	’	178	І	210	Т	242	т
147	“	179	і	211	У	243	у
148	”	180	г	212	Ф	244	ф
149		181	μ	213	Х	245	х
150	—	182	¶	214	Ц	246	ц
151	—	183	“	215	Ч	247	ч
152	”	184	ё	216	Ш	248	ш
153	™	185	№	217	Щ	249	щ
154	љ	186	є	218	Ъ	250	ъ
155	>	187	»	219	Ы	251	ы
156	њ	188	ј	220	Ь	252	ь
157		189	ѕ	221	Э	253	э
158		190	ѕ	222	Ю	254	ю
159	џ	191	ї	223	Я	255	я

Символы с кодами 128–255 (кодировка 866 — MS DOS)

КОД	СИМВ	КОД	СИМВ	КОД	СИМВ	КОД	СИМВ
128	А	160	а	192	Ќ	224	Р
129	Б	161	б	193	Ћ	225	С
130	В	162	в	194	Ў	226	Т
131	Г	163	г	195	ѐ	227	У
132	Д	164	д	196	—	228	Ф
133	Е	165	е	197	+	229	Х
134	Ж	166	ж	198	ѓ	230	Ц
135	З	167	з	199	І	231	Ч
136	И	168	и	200	Ќ	232	Ш
137	Й	169	й	201	Г	233	Щ
138	К	170	к	202	Ћ	234	Ъ
139	Л	171	л	203	Т	235	М
140	М	172	м	204	ѓ	236	Ь
141	Н	173	н	205	—	237	Э
142	О	174	о	206	†	238	Ю
143	П	175	п	207	Ћ	239	Я
144	Р	176	ѐ	208	Ћ	240	Ё
145	С	177	■	209	т	241	ѐ
146	Т	178	■	210	т	242	Є
147	У	179	І	211	Ќ	243	ё
148	Ф	180	†	212	Ќ	244	Ї
149	Х	181	†	213	Г	245	ї
150	Ц	182	†	214	Г	246	9
151	Ч	183	†	215	†	247	Ń
152	Ш	184	†	216	†	248	•
153	Щ	185	†	217	Ј	249	•
154	Ъ	186	І	218	Г	250	•
155	Ь	187	†	219	■	251	Ї
156	Э	188	Ј	220	■	252	Ї
157	Ю	189	Ј	221	І	253	0
158	Я	190	Ј	222	І	254	•
159		191	†	223	■	255	

Алфавитный указатель

#%&

#define, 94
#elif, 95
#else, 95
#endif, 95
#if, 95
#ifdef, 95
#ifndef, 95
#include, 93
#undef, 96
__cplusplus, 96
__DATE__, 96
__FILE__, 96
__LINE__, 96
__TIME__, 96
_IOFBF, 399
_IOLBF, 399
_IONBF, 399

A

abort, 223, 228, 416
abs, 93, 416
accumulate, алгоритм, 369
acos, 93, 416
adjacent_difference, алгоритм, 371
adjacent_find, алгоритм, 345
adjustfield, ios, 270
advance, 331
any, bitset, 325
app, ios, 281
append, string, 289
apply, valarray, 373
asctime, 416
asin, 93, 416
assert, 394
assign, string, 288
assign, vector, 300
at, string, 288
at, vector, 301
at, контейнеры, 298
atan, 93, 416
atan2, 93, 416

ate, ios, 281
atexit, 417
atof, 92, 417
atoi, 92, 417
atol, 92, 417
auto, 30

B

back, list, 306
back, queue, 312
back, vector, 301
back_insert_iterator, 332
back_inserter, 333
bad, ios, 278
bad_alloc, 230, 378, 379
bad_cast, 233, 379
bad_exception, 379
bad_typeid, 238, 379
badbit, ios, 278
basefield, ios, 270
before, type_info, 239
begin, итератор, 297
binary, ios, 281
binary_function, 335
binary_negate, 337
binary_search, алгоритм, 358
bind1st, связыватели, 337
bind2nd, связыватели, 337
binder1st, 337
binder2nd, 337
bitset, 323
bool, 24
bsearch, 417
btowc, 418
BUFSIZ, 394

C

C_str, string, 290
calloc, 418
capacity, string, 294
capacity, vector, 301
catch, 224, 301
ceil, 93, 418

cerr, 267, 395
char, 24
CHAR_BIT, 394
CHAR_MAX, 395
CHAR_MIN, 395
cin, 267, 395
class, 178
clear, ios, 278
clear, list, 307
clear, map, 319
clear, string, 290
clear, vector, 202
clearerr, 418
clock, 418
clock_t, 395
CLOCKS_PER_SEC, 395
clog, 267, 395
close, ostream, 282
compare, string, 293
complex, 380
const, 29, 53
const_cast, 232
const_iterator begin, контейнеры, 297
const_iterator end, контейнеры, 297
const_iterator, контейнеры, 296
const_reference, контейнеры, 296
const_reverse_iterator rbegin, контейнеры, 297
const_reverse_iterator rend, контейнеры, 297
const_reverse_iterator, контейнеры, 296
copy, string, 291
copy, алгоритм, 350
copy_backward, алгоритм, 350
cos, 93, 418
cosh, 93, 419
count, bitset, 325
count, map, 318
count, set, 322
count, алгоритм, 345
count_if, алгоритм, 338, 345
cout, 267, 395
cshift, valarray, 373
ctime, 419

D

data, string, 291
DBL, 395
dec, ios, 269
dec, манипулятор, 271
deque, 297, 304
difftime, 419
distance, 331
div, 419
div_t, 395, 400
divides, функциональный объект, 335
domain_error, 379
double, 25
dynamic_cast, 233

E

EDOM, 93, 395
empty, priority_queue, 313
empty, queue, 312
empty, stack, 311
empty, string, 294
empty, контейнеры, 297
end, итератор, 297
endl, манипулятор, 271
ends, манипулятор, 271
enum, 66
eof, 298
EOF, 396
eof, ios, 278
eofbit, ios, 278
equal, алгоритм, 346
equal_range, map, 321
equal_range, set, 322
equal_range, алгоритм, 359
equal_to, предикат, 336
ERANGE, 92, 396
erase, list, 307
erase, map, 319
erase, set, 321
erase, string, 290
erase, vector, 302
erase, контейнеры, 298
errno, 92, 396
exception, 230, 379
exit, 419
EXIT_FAILURE, 396
EXIT_SUCCESS, 396
exp, 93, 419
explicit, 299
extern, 30, 74, 98

F

fabs, 93, 419
fail, ios, 278
failbit, ios, 278
false, 24
fclose, 90, 420
feof, 90, 420
ferror, 90, 420
fflush, 420
fgetc, 90, 420
fgetpos, 89, 420
fgets, 90, 420
fgetwc, 421
fgetws, 421
FILE, 88, 396
filebuf, 397
FILENAME_MAX, 397
fill, ios, 270
fill, алгоритм, 351
fill_n, алгоритм, 351
find, map, 318

find, set, 322
find, string, 291
find, алгоритм, 346
find_end, алгоритм, 347
find_first_not_of, string, 292
find_first_of, string, 292
find_first_of, алгоритм, 347
find_if, алгоритм, 346
find_last_not_of, string, 292
find_last_of, string, 292
fixed, ios, 269
flags, ios, 270
flip, bitset, 324
flip, vector_bool, 304
float, 25
float_round_style, 397
floatfield, ios, 270
floor, 93, 421
FLT, 397
FLT_DIG, 378, 397
FLT_EPSILON, 397
FLT_MANT_DIG, 397
FLT_MAX, 397
FLT_MAX_EXP, 397
FLT_MIN, 378, 397
FLT_MIN_10_EXP, 397
FLT_MIN_EXP, 398
FLT_RADIX, 398
FLT_ROUNDS, 398
FLT_MAX_10_EXP, 397
flush, ostream, 274
flush, манипулятор, 271
fmod, 93, 421
fopen, 89, 421
FOPEN_MAX, 398
for_each, алгоритм, 347
fpos_t, 398
fprintf, 90, 421
fputc, 90, 421
fputs, 90, 422
fputwc, 422
fputws, 422
fread, 90, 422
free, 55, 422
freopen, 422
frexp, 422
friend, 187
front, list, 306
front, queue, 312
front, vector, 301
front_insert_iterator, 332
front_inserter, 333
fscanf, 90, 423
fseek, 89, 423
fsetpos, 89, 423
fstream, 280
ftell, 89, 423

fwide, 423
fwprintf, 423
fwrite, 90, 424
fwscanf, 424

G

gcount, istream, 273
generate, алгоритм, 351
generate_n, алгоритм, 351
get, istream, 273
getc, 90, 424
getchar, 90, 424
getenv, 424
getline, istream, 273
gets, 90, 424
getwc, 424
getwchar, 425
gmtime, 425
good, ios, 278
goodbit, ios, 278
greater, предикат, 336
greater_equal, предикат, 336
gslice, 375
gslice_array, 371, 376

H

hardfail, ios, 278
hex, ios, 269
hex, манипулятор, 271
HUGE_VAL, 398

I

istream, 280, 398
ignore, istream, 273
in, ios, 281
includes, алгоритм, 365
indirect_array, 371, 376
inline, 74
inner_product, алгоритм, 370
inplace_merge, алгоритм, 359
insert, list, 307
insert, map, 319
insert, set, 321
insert, string, 289
insert, vector, 302
insert, контейнеры, 298
insert_iterator, 332
inserter, 333
int, 23
INT_MAX, 398
INT_MIN, 398
internal, ios, 269
invalid_argument, 379
ios, 265, 399
 x_flags, 269
isalfa, 92
isalnum, 92, 425
isalpha, 425

isctrl, 92, 425
isdigit, 92, 425
isgraph, 92, 425
islower, 92, 426
isprint, 92, 426
ispunct, 92, 426
isspace, 426
ispace, 92
istream, 265, 399
 методы, 273
istream_iterator, 333
istreamstring, 282, 399
isupper, 92, 426
iswalnum, 426
iswalpha, 427
iswcntrl, 427
iswctype, 427
iswdigit, 427
iswgraph, 427
iswlower, 427
iswprint, 427
iswpunct, 428
iswspace, 428
iswupper, 428
iswxdigit, 428
isxdigit, 92, 428
iter_swap, алгоритм, 352
iterator, 298
iterator begin, контейнеры, 297
iterator end, контейнеры, 297
iterator, контейнеры, 296
iterator_traits, 330

К
key_compare, контейнеры, 296
key_type, контейнеры, 296

L
L_tmpnam, 399
labs, 428
LC_ALL, 399
LC_COLLATE, 399
LC_CTYPE, 399
LC_MONETARY, 399
LC_NUMERIC, 399
LC_TIME, 399
lconv, 400
LDBL, 400
ldexp, 429
ldiv, 429
ldiv_t, 400
left, ios, 269
length, string, 294
length_error, 379
less, предикат, 336
less_equal, предикат, 336
lexicographical_compare, алгоритм, 359
list, 297, 306

localeconv, 429
localtime, 429
log, 93, 429
log10, 93, 429
logic_error, 379
logical_and, предикат, 336
logical_not, предикат, 336
logical_or, предикат, 336
long, 23
long double, 25
LONG_MAX, 400
LONG_MIN, 401
lower_bound, map, 318
lower_bound, set, 322
lower_bound, алгоритм, 360
L-значение, 33

M

main, 26, 87
make_heap, алгоритм, 367
make_pair, pair, 316
malloc, 54, 231, 429
map, 315, 316, 325
mask_array, 371, 376
max, valarray, 373
max, алгоритм, 360
max_element, алгоритм, 361
max_size, string, 294
max_size, контейнеры, 297
MB_CUR_MAX, 401
MB_LEN_MAX, 401
mblen, 430
mbstate_t, 401
mbstowcs, 430
mbtowc, 430
mem_fun, 342
mem_fun_ref, 342
memchr, 430
memcmp, 430
memcpy, 430
memmove, 430
memset, 431
merge, list, 309
merge, алгоритм, 361
min, valarray, 373
min, алгоритм, 360
min_element, алгоритм, 361
minus, функциональный объект, 335
mismatch, алгоритм, 348
mktime, 431
modf, 93, 431
modulus, функциональный объект, 335
multimap, 321
multiplies, функциональный объект, 335
multiset, 323

N

name, type_info, 239
 namespace, 97
 negate, функциональный
 объект, 335
 new, 54, 116, 180
 new_handler, 378, 401
 next_permutation, алгоритм, 362
 nocreate, ios, 281
 none, bitset, 325
 noreplace, ios, 281
 not_equal_to, предикат, 336
 not1, отрицатель, 337
 not2, отрицатель, 337
 nth_element, алгоритм, 362
 NULL, 54, 401
 numeric_limits, 378

O

oct, ios, 269
 oct, манипулятор, 271
 offsetof, 401
 ofstream, 280, 401
 open, fstream, 281
 open_mode, ios, 281
 operator, 190
 ostream, 265, 401
 ostream_iterator, 333
 ostringstream, 282, 402
 out, ios, 281
 out_of_range, 288, 301, 379
 overflow_error, 379

P

pair, 315
 partial_sort, алгоритм, 363
 paqueue, 313
 pop, queue, 313
 partial_sort_copy, алгоритм, 363
 partial_sum, алгоритм, 371
 partition, алгоритм, 363
 peek, istream, 273
 perror, 431
 plus, функциональный объект, 335
 pointer_to_binary_function, 338
 pointer_to_unary_function, 338
 pop, priority_queue, 313
 pop, stack, 311
 pop_back, list, 307
 pop_back, vector, 302
 pop_back, контейнеры, 298
 pop_front, deque, 306
 pop_front, list, 306
 pop_front, контейнеры, 297
 pop_heap, алгоритм, 368
 pow, 93, 431
 precision, ios, 270
 prev_permutation, алгоритм, 362
 printf, 27, 81, 90, 231, 431
 private, 178, 200

priority_queue, 313
 protected, 200
 ptr_fun, 338
 ptrdiff_t, 331, 402
 public, 179, 200, 210
 push, priority_queue, 313
 push, queue, 312
 push, stack, 311
 push_back, list, 306
 push_back, vector, 302
 push_back, контейнеры, 297
 push_front, deque, 306
 push_front, list, 306
 push_front, контейнеры, 297
 push_heap, алгоритм, 368
 put, ostream, 274
 putback, istream, 273
 putc, 90, 431
 putchar, 90, 432
 puts, 90, 432
 putwc, 432
 putwchar, 432

Q

qsort, 432
 queue, 311

R

raise, 432
 rand, 433
 RAND_MAX, 402
 random_shuffle, алгоритм, 352
 range_error, 379
 rbegit, итератор, 297
 rdstate, ios, 278
 read, istream, 273
 readsome, 273
 realloc, 433
 reference, контейнеры, 296
 register, 30
 reinterpret_cast, 238
 remove, 433
 remove, list, 308
 remove, алгоритм, 353
 remove_copy, алгоритм, 353
 remove_copy_if, алгоритм, 353
 remove_if, list, 308
 remove_if, алгоритм, 353
 rename, 433
 rend, итератор, 297
 replace, string, 290
 replace, алгоритм, 354
 replace_copy, алгоритм, 354
 replace_copy_if, алгоритм, 354
 replace_if, алгоритм, 354
 reserve, vector, 301
 reset, bitset, 324
 resetiosflags, манипулятор, 272

resize, list, 307
resize, valarray, 373
resize, vector, 302
return, 76
reverse, list, 309
reverse, алгоритм, 355
reverse_copy, алгоритм, 355
reverse_iterator, 331
reverse_iterator rbegin, контейнеры, 297
reverse_iterator rend, контейнеры, 297
reverse_iterator, контейнеры, 296
rewind, 433
rfind, string, 292
right, ios, 269
rotate, алгоритм, 355
rotate_copy, алгоритм, 355
RTTI, 233, 238
runtime_error, 379

S

scanf, 27, 90, 433
SCHAR_MAX, 402
SCHAR_MIN, 402
scientific, ios, 269
search, алгоритм, 348
search_n, алгоритм, 348
SEEK_CUR, 402
SEEK_END, 402
SEEK_SET, 402
seekg, istream, 273
seekg, ostream, 274
set, 321, 325
set, bitset, 324
set_difference, алгоритм, 366
set_intersection, алгоритм, 365
set_new_handler, 378
set_symmetric_difference, алгоритм, 366
set_terminate, 228
set_unexpected, 228
set_union, алгоритм, 366
setbase, манипулятор, 272
setbuf, 89, 434
setf, ios, 270
setfill, манипулятор, 272
setiosflags, манипулятор, 272
setlocale, 434
setprecision, манипулятор, 272
setvbuf, 89, 434
setw, манипулятор, 272
shift, valarray, 373
short, 23
showbase, ios, 269
showpoint, ios, 269
showpos, ios, 269
SHRT_MAX, 402
SHRT_MIN, 402
sig_atomic_t, 403
SIG_DFL, 403
SIG_ERR, 403
SIG_IGN, 403
SIGABRT, 403
SIGFPE, 403
SIGILL, 403
SIGINT, 403
signal, 434
signed, 23
SIGSEGV, 403
SIGTERM, 403
sin, 93, 434
sinh, 93, 434
size, bitset, 325
size, priority_queue, 313
size, queue, 312
size, stack, 311
size, string, 294
size, valarray, 373
size, контейнеры, 297
size_t, 192, 403
size_type, контейнеры, 296
sizeof, 231
skipws, ios, 269
slice, 374
slice_array, 371, 375
sort, list, 308
sort, алгоритм, 364
sort_heap, алгоритм, 368
splice, list, 307
sprintf, 90, 435
sqrt, 93, 435
srand, 435
sscanf, 90, 435
stable_partition, алгоритм, 363
stable_sort, алгоритм, 364
stack, 310
state, ios, 278
static, 30, 74, 185
static_cast, 237
std, 100, 264
stdaux, 89
stderr, 89, 403
stdin, 89, 403
stdio, ios, 269
stdout, 89, 403
stdprn, 89
STL, 295
str, stringstream, 283
strcat, 91, 435
strchr, 91, 435
strcmp, 91, 435
strcoll, 436
strcpy, 91, 436
strcspn, 436
streambuf, 265, 403
streamoff, 404

streamsize, 404
 strerror, 436
 strftime, 436
 string, 286, 404
 конструкторы, 287
 stringbuf, 404
 stringstream, 282
 strlen, 91, 436
 strncat, 91, 436
 strncmp, 91, 436
 strncpy, 91, 437
 strpbrk, 91, 437
 strchr, 91, 437
 strspn, 437
 strstr, 91, 437
 strtod, 437
 strtok, 438
 strtol, 438
 strtoul, 438
 struct, 67, 209
 strxfrm, 438
 substr, string, 290
 sum, valarray, 373
 swap, list, 307
 swap, map, 320
 swap, set, 322
 swap, string, 290
 swap, vector, 302
 swap, алгоритм, 352
 swap_ranges, алгоритм, 352
 swprintf, 438
 swscanf, 438
 sync_with_stdio, ios_base, 278
 system, 439

T

tan, 93, 439
 tanh, 93, 439
 tellg, istream, 274
 tellg, ostream, 274
 template, 215
 terminate, 223, 228
 terminate_handler, 404
 test, bitset, 325
 this, 181
 throw, 223, 227
 tie, basic_ios, 278
 time, 439
 time_t, 404
 tm, 405
 TMP_MAX, 404
 tmpfile, 439
 tmpnam, 439
 to_string, bitset, 325
 to_ulong, bitset, 325
 tolower, 93, 440
 top, priority_queue, 313

top, stack, 311
 toupper, 93, 440
 tolower, 440
 towupper, 440
 transform, алгоритм, 356
 true, 24
 trunc, ios, 281
 try, 223, 301
 type_info, 238
 before, 239
 name, 239
 typedef, 65, 220

U

UCHAR_MAX, 405
 UINT_MAX, 378, 405
 ULONG_MAX, 405
 unary_function, 334
 unary_negate, 337
 underflow_error, 379
 unexpected, 228
 unexpected_handler, 405
 unget, istream, 274
 ungetc, 440
 ungetwc, 440
 union, 69, 209
 unique, list, 308
 unique, алгоритм, 357
 unique_copy, алгоритм, 357
 unitbuf, ios, 269
 unsetf, ios, 270
 unsigned, 23
 upper_bound, map, 318
 upper_bound, set, 322
 upper_bound, алгоритм, 360
 uppercase, ios, 269
 USHRT_MAX, 405

V

va_arg, 406
 va_list, 406
 va_start, 406
 valarray, 371
 value_type, контейнеры, 296
 vector, 297, 299
 vector_bool, 303
 vfprintf, 440
 vfwprintf, 441
 virtual, 206
 void, 26
 vprintf, 441
 vptr, 207
 vsprintf, 441
 vswprintf, 441
 vtbl, 207
 vwprintf, 441

W

WCHAR_MAX, 406
wchar_t, 24
wscat, 441
wcschr, 441
wcscmp, 442
wcscoll, 442
wcsncpy, 442
wcscspn, 442
wcsftime, 442
wcslen, 442
wcsncat, 442
wcsncmp, 443
wcsncpy, 443
wcpbrk, 443
wcsrchr, 443
wcssp, 443
wcsstr, 443
wcstod, 444
wcstok, 444
wcstol, 444
wcstombs, 444
wcstoul, 444
wcstrxfrm, 444
wctomb, 445
wctype_t, 406
WEOF, 406
werr, 407
wfilebuf, 407
whar_t, 266
width, ios, 270
wifstream, 407
win, 407
wint_t, 407
wios, 407
wistream, 266, 407
wistreamstream, 407
wlog, 407
wmemchr, 445
wmemcmp, 445
wmemcpy, 445
wmemmove, 445
wmemset, 445
wofstream, 266, 407
wostream, 266, 408
wostringstream, 408
wout, 407
wprintf, 445
write, ostream, 274
ws, манипулятор, 271
wscanf, 446
wstreambuf, 408
wstreamoff, 408
wstring, 408
wstringbuf, 408
wtrans_t, 406

X

x_fill, ios, 270
x_flags, 269
x_precision, ios, 270
x_width, ios, 270

A

абстрактный класс, 208
адаптер
 итератора, 331
 контейнера, 295, 310
 метода, 341
 указателя на функцию, 338
 функции, 335
алгоритмы STL, 343
алфавит, 17
ассоциативные контейнеры, 296, 315

Б

базовые конструкции, 38
базовые классы, 200
бинарное дерево, 122
битовое множество, 323
битовые поля, 69
блок, 16
буфер, 89, 265

В

вектор, 298
ветвление, 39
виртуальные методы, 206
включение в поток, 265
время жизни, 29
встроенная функция, 74
входной итератор, 329
выражение, 15, 28, 37
выходной итератор, 329
генерация исключений, 224

Д

двунаправленный итератор, 329
двусторонняя очередь, 298, 304
декремент, 33
деструктор, 188
 наследование, 205
динамические переменные, 53
динамические структуры данных, 114
динамический массив, 60
директива препроцессора, 93
дружественные
 классы, 188
 функции, 187

операции (продолжение)

вызов функции ()
 перегрузка, 195
 выбора . (точка), 68, 181
 выбора ->, 68, 181
 декремент --, 33
 деления /, 34
 доступа к области
 видимости, 180
 извлечения из потока, 267
 индексирования, 196
 инкремент ++, 33
 логические &&, ||, 36
 логическое отрицание !, 34
 остатка от деления %, 34
 отношения, 35
 получения адреса &, 53
 поразрядное отрицание ~, 34
 поразрядные &, |, ^, 35
 приведения типа
 перегрузка, 195
 приведения типа, 56, 231
 присваивания
 +=, -=, *=, /=, %=, 36
 наследование, 204
 перегрузка, 191
 разадресация *, 55
 сдвига, 35
 унарный минус -, 34
 условная (? :), 36
 определение переменной, 31
 отрицатели, 337
 очередь, 121, 311
 очередь с приоритетами, 313

П

последовательность, 328
 парадигма, 175
 параметрический полиморфизм, 221
 перегрузка
 операций, 189
 << и >>, 284
 функций, 83
 передача
 по адресу, 77
 по значению, 77
 по ссылке, 78
 перекрестное преобразование, 233
 переменная, 28
 пирамида, 364
 повышающее преобразование, 233
 позднее связывание, 206
 поименованная область, 97, 99
 полиморфизм, 176
 полиморфные функции, 208
 полиморфный объект, 208
 поля класса, 178
 наследование, 205

понижающее преобразование, 233
 последовательные контейнеры, 295
 потоки, 265
 форматирование, 269
 потоковые итераторы, 333
 предикат, 334
 преобразование типа, 38
 препроцессор, 16, 93
 пробельные символы, 18
 программа, 16
 производные классы, 200
 произвольного доступа итератор, 329
 простое наследование, 201
 пространство имен, 97
 прототип функции, 73
 прямой итератор, 329
 пустой оператор, 40

Р

размерность массива, 58
 раннее связывание, 206
 раскручивание стека, 223
 рекурсивная функция, 82

С

связыватели, 337
 сигнатура функции, 73
 следование, 38
 словари с дубликатами, 321
 словарь, 315
 событийно-управляемая модель, 175
 сообщение, 175
 составной оператор, 16
 специализация шаблона
 класса, 220
 функций, 86
 спецификации формата, 388
 список, 211, 298, 306
 список инициализаторов, 184
 средства диагностики, 379
 средства локализации, 380
 средства поддержки языка, 378
 срез, 374
 ссылка, 58
 стандартная библиотека шаблонов, 295
 стандартные
 исключения, 230
 потоки, 265
 статические
 методы, 186
 поля, 186
 стек, 119, 310
 строка, 63
 строковый литерал, 21
 структурное программирование, 13, 14
 структуры, 210

Т

таблица виртуальных методов, 207

тип данных

void, 26

тип данных, 22

bool, 23

char, 23

double, 23

float, 23

int, 23

wchar_t, 23

тип функции, 74

типизированная константа, 29

У

узел списка, 211

указатель, 51

вывод, 268

на метод класса, 197

на поле класса, 198

потока, 89

управляющая последовательность, 21

Ф

файловые потоки, 280

флаги, потоки, 269

форматирующие методы, потоки, 270

функции, 73

main, 87

встроенные, 74

объявление, 73

определение, 73

перегрузка, 83

рекурсивные, 82

шаблоны, 85

функции-операции, 189

функциональный класс, 195

функциональный объект, 334

Ц-Ч

цикл, 39

чисто виртуальный метод, 206

Ш

шаблоны классов, 211

шаблоны функций, 85

Э

экземпляр класса, 180

элемент динамической структуры данных, 115

элементы класса, 178

Павловская Татьяна Александровна
C/C++. Программирование на языке высокого уровня

Заведующий редакцией *А. Кривош*
Ведущий редактор *Ю. Суркис*
Литературный редактор *Е. Ваулина*
Художник *Н. Биржаков*
Верстка *П. Быстров*
Корректор *И. Смирнова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;

95 3005 — литература учебная.

Подписано в печать 06.08.12. Формат 70х100/16. Усл. п. л. 37,410. Доп. тираж. Заказ 6252.

Отпечатано по технологии СР в ИПК ООО «Ленинградское издательство»

194044, Санкт-Петербург, ул. Менделеевская, д. 9

Телефон факс: (812) 495-56-10.

Нет времени ходить по магазинам?



наберите:



www.piter.com



Здесь вы найдете:

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге — отзывы, рецензии, отрывки
Старые книги — в библиотеке и на CD



**И наконец, вы нигде не купите
наши книги дешевле!**

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?

Вы ведете блог?

Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?

ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!



Зарегистрируйтесь на нашем сайте в качестве партнера по адресу www.piter.com/ePartners



Получите свой персональный уникальный номер партнера



Выбирайте книги на сайте www.piter.com, размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт www.piter.com)

ВНИМАНИЕ! В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:

<http://www.piter.com/book.phtml?978538800282> – обычная ссылка

<http://www.piter.com/book.phtml?978538800282&refer=0000> – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробнее о Партнерской программе
ИД «Питер» читайте на сайте
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ
 **ПИТЕР®**
WWW.PITER.COM

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

✍ Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: spb@piter.com

✍ Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

✍ Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

✍ Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225



САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

АНТИВИРУСНЫЕ
ПРОГРАММНЫЕ ПРОДУКТЫ

Задача этой книги — дать краткое и четкое изложение языка C++ в соответствии со стандартом ISO/IEC 14882 (1998) без легковесности и фрагментарности серии «Для чайников» и без пространных рассуждений о нюансах, важность которых становится очевидной только после приобретения опыта программирования. Книга предназначена для студентов, изучающих язык «с нуля», но будет полезна и более искушенным в программировании.

Контрольные задания по ключевым темам представлены в 20 вариантах!

Темы, рассмотренные в книге:

- основные принципы структурного написания программ;
- конструкции языка и технология создания программ;
- принципы объектно-ориентированного программирования и их реализация на C++;
- базовые средства языка — классы, наследование, шаблоны, исключения;
- средства стандартной библиотеки: потоки, контейнеры, итераторы, алгоритмы;
- справочная информация, необходимая при практическом программировании;
- задания для проведения лабораторных работ в учебной группе.

ПИТЕР®

Заказ книг:

197198, Санкт-Петербург, а/я 127
тел.: (812) 703-73-74, postbook@piter.com
61093, Харьков-93, а/я 9130
тел.: (057) 758-41-45, 751-10-02, piter@kharkov.piter.com

www.piter.com — вся информация о книгах и веб-магазин

ISBN 978-5-496-00031-4



9 785496 000314