А. П. Побегайло

C/C++ для студента

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.068+800.92С/С++
ББК 32.973.26-018.1
П41

Побегайло А. П.

П41 С/С++ для студента. — СПб.: БХВ-Петербург, 2006. — 528 с.: ил.

ISBN 5-94157-647-1

Подробно рассматриваются языки программирования С и С++. Описаны типы данных, функции, классы, шаблоны, а также библиотеки стандартных функций. Язык программирования С++ рассматривается как объектно-ориентированное расширение языка С, что позволяет последовательно изучить процедурное программирование, объектно-ориентированное программирование и обобщенное программирование. Изложение материала отличается краткостью и снабжено большим количеством простых примеров и листингов, которые поясняют технику программирования на языках С и С++.

Для студентов и программистов

УДК 681.3.068+800.92С/С++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Андрей Смышляев*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.05.06.
Формат 60×90$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 33.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-647-1

Оглавление

Введение 17

ЧАСТЬ I. ЯЗЫК ПРОГРАММИРОВАНИЯ C 21

Глава 1. Структура языка C 23

1.1. Элементы языка C 23
1.2. Символы 25
1.3. Ключевые слова 26
1.4. Идентификаторы 27
1.5. Константы 27
1.6. Инструкции 29
1.7. Комментарии 30

Глава 2. Встроенные типы данных и переменные 31

2.1. Базовые типы данных 31
2.2. Модификаторы типов 32
2.3. Спецификаторы типов 33
2.4. Переменные 35
2.5. Фундаментальные типы данных 36
2.6. Квалификаторы типов 37

Глава 3. Операторы и выражения 39

3.1. L-value и R-value 39
3.2. Арифметические операторы 40
3.3. Побитовые операторы 41
3.4. Операторы сравнения 43
3.5. Логические операторы 44

3.6. Выражения....45
3.7. Приведение типов в выражениях....47
3.8. Оператор преобразования типов....48
3.9. Оператор присваивания....49
3.10. Составные операторы присваивания....51
3.11. Операторы инкремента и декремента....52
3.12. Условный оператор....53
3.13. Оператор "запятая"....55
3.14. Побочные эффекты....56
3.15. Точки последовательности....57

Глава 4. Управляющие инструкции....59

4.1. Инструкции выбора *if* и *if...else*....59
4.2. Инструкция выбора *switch*....60
4.3. Инструкции цикла *while* и *do...while*....62
4.4. Инструкция цикла *for*....63
4.5. Инструкция перехода *break*....64
4.6. Инструкция перехода *continue*....64
4.7. Метки инструкций и инструкция перехода *goto*....65

Глава 5. Указатели и массивы....67

5.1. Указатели....67
5.2. Преобразование типов указателей....68
5.3. Операторы определения адреса и обращения по адресу....69
5.4. Указатели на константы и константные указатели....70
5.5. Одномерные массивы....71
5.6. Строки....74
5.7. Арифметические действия с указателями....74
5.8. Многомерные массивы....75

Глава 6. Функции....78

6.1. Объявление функции....78
6.2. Определение функции....80
6.3. Стандартные функции....81
6.4. Вызов функции....82
6.5. Передача массивов в функции....84
6.6. Указатели на функции (функторы)....86

Глава 7. Структура программы на языке C....89

7.1. Область видимости идентификатора....89
7.2. Время существования переменных и функций....90

7.3. Связывание идентификаторов....91
7.4. Спецификаторы классов памяти....92
7.5. Структура программы....96
7.6. Функция *main*....97

Глава 8. Типы данных, определяемые программистом....100

8.1. Введение....100
8.2. Перечисления....101
8.3. Структуры....103
8.4. Объединения....107
8.5. Битовые поля....110
8.6. Передача структур в функции....111
8.7. Объявление *typedef*....114
8.8. Оператор *sizeof*....116
8.9. Сравнимые типы....119

Глава 9. Директивы препроцессора....122

9.1. Препроцессор....122
9.2. Директивы *#define* и *#undef*....123
9.3. Директивы *#if*, *#elsif*, *#else* и *#endif*....124
9.4. Директивы *#ifdef* и *#ifndef*....126
9.5. Директива *#include*....127
9.6. Директивы *#error*....129
9.7. Директива *#line*....129
9.8. Директива *#pragma*....130
9.9. Пустая директива....130
9.10. Оператор *#*....130
9.11. Оператор *##*....131
9.12. Предопределенные макрокоманды....132

ЧАСТЬ II. ЯЗЫК ПРОГРАММИРОВАНИЯ C++....133

Глава 10. Дополнения к типам данных языка C....135

10.1. Тип данных *bool*....135
10.2. Тип данных *wchar_t*....135
10.3. Анонимные объединения....137
10.4. Ссылки....138
10.5. Новые операторы явного преобразования типов....140
10.6. Типизированные операторы распределения памяти....143

Глава 11. Дополнение к функциям языка C 146

11.1. Передача аргументов по умолчанию 146
11.2. Встроенные функции 147
11.3. Передача аргументов и возврат значения через ссылки 148
11.4. Перегрузка функций 149
11.5. Искажение имен функций 153

Глава 12. Пространства имен 155

12.1. Определение пространства имен 155
12.2. Анонимные пространства имен 156
12.3. Стандартное пространство имен 157
12.4. Оператор разрешения области видимости 157
12.5. Объявление *using* 160
12.6. Директива *using* 161
12.7. Псевдонимы 162
12.8. Искажение имен переменных 163

Глава 13. Обработка исключений 165

13.1. Механизм обработки исключений 165
13.2. Обработка нескольких исключений 169
13.3. Перехват всех исключений 170
13.4. Вложенные исключения 171
13.5. Реализация исключений 174
13.6. Спецификация исключений 175

Глава 14. Классы 177

14.1. Введение 177
14.2. Определение класса 178
14.3. Объекты, доступ к членам класса 180
14.4. Указатель *this* 181
14.5. Спецификаторы доступа к членам класса 181
14.6. Друзья класса 183
14.7. Встроенные функции-члены класса 187
14.8. Функции-члены класса с квалификаторами *const* и *volatile* 188
14.9. Данные-члены класса с квалификатором *mutable* 190
14.10. Статические члены класса 190
14.11. Указатели на нестатические члены класса 192
14.12. Вложенные классы 194
14.13. Локальные классы 196

Глава 15. Конструкторы и деструкторы 197

15.1. Конструктор класса 197
15.2. Список инициализации 198
15.3. Конструктор по умолчанию 199
15.4. Конструктор копирования 200
15.5. Явный вызов конструкторов 202
15.6. Деструкторы 203
15.7. Исключения в деструкторах 205

Глава 16. Перегрузка операторов 206

16.1. Общие правила 206
16.2. Унарные операторы 209
16.3. Оператор присваивания 210
16.4. Составные операторы присваивания 212
16.5. Оператор индексирования *[ ]* 213
16.6. Оператор вызова функции 214
16.7. Оператор доступа к членам класса 215
16.8. Операторы инкремента и декремента 216
16.9. Бинарные операторы 217
16.10. Перегрузка операторов >> и << для ввода и вывода 218
16.11. Операторы преобразования типов (конвертеры) 219
16.12. Операторы *new* и *delete* 219

Глава 17. Наследование классов 222

17.1. Определение наследования 222
17.2. Доступ к членам, наследуемым от базового класса 225
17.3. Конструкторы, деструкторы и наследование 226
17.4. Наследование и присваивание 229
17.5. Виртуальные функции 230
17.6. Полиморфизм и позднее связывание 233
17.7. Передача аргументов по умолчанию в виртуальные функции 234
17.8. Виртуальные деструкторы 235
17.9. Абстрактные классы 236
17.10. Множественное наследование 238
17.11. Виртуальное наследование 239

Глава 18. Шаблоны функций 241

18.1. Определение шаблона функции 241
18.2. Конкретизация шаблона функции 243
18.3. Вывод аргументов шаблона функции 246

18.4. Явная специализация шаблона функции 249
18.5. Модели компиляции шаблонов 250
18.6. Перегрузка шаблонов функций 252
18.7. Шаблоны функций как члены класса 255

Глава 19. Шаблоны классов 257

19.1. Определение шаблона класса 257
19.2. Конкретизация шаблона класса 259
19.3. Использование ключевого слова *typename* 262
19.4. Явная специализация шаблона класса 264
19.5. Частичная специализация шаблона класса 265
19.6. Статические члены шаблона класса 267
19.7. Шаблоны как члены шаблона класса 268
19.8. Объявление друзей в шаблоне класса 269
19.9. Шаблоны класса и наследование 273
19.10. Шаблоны класса как параметры шаблонов функций 274
19.11. Шаблоны класса как параметры шаблона класса 275

ЧАСТЬ III. СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА ПРОГРАММИРОВАНИЯ С 277

Глава 20. Стандартные определения <stddef.h> 279

20.1. Типы данных 279
20.2. Макрос *NULL* 280
20.3. Макрос *offsetof* 280

Глава 21. Стандартные функции <stdlib.h> 282

21.1. Динамическое распределение памяти 282
21.2. Динамические массивы 284
21.3. Стандартные функции сортировки и поиска 287
21.4. Взаимодействие с системой 290
21.5. Преобразования строк в числа 294
21.6. Арифметические функции 298
21.7. Генерация случайных чисел 299
21.8. Обработка длинных символов 301

Глава 22. Диагностика ошибок <assert.h> 305

Глава 23. Функции с переменным количеством параметров <stdarg.h> 307

Глава 24. Диапазоны целочисленных данных <limits.h> 311

Глава 25. Диапазоны чисел с плавающей точкой <float.h>.....313

Глава 26. Обработка ошибок <errno.h> 316

Глава 27. Математические функции <math.h>.....318

27.1. Обработка ошибок 318
27.2. Тригонометрические функции.....319
27.3. Экспоненциальная и логарифмическая функции 320
27.4. Гиперболические функции.....321
27.5. Степень и квадратный корень.....321
27.6. Экспонента и мантисса.....322
27.7. Целая и дробная часть числа с плавающей точкой 323
27.8. Целые границы числа с плавающей точкой.....323
27.9. Остаток от деления 324
27.10. Абсолютное значение числа.....324

Глава 28. Функции классификации символов <ctype.h> 325

28.1. Определение типа символа.....325
28.2. Отображения прописных и строчных символов 326

Глава 29. Функции для работы со строками <string.h>.....328

29.1. Сравнение строк.....328
29.2. Копирование строк.....329
29.3. Соединение строк.....330
29.4. Поиск символа в строке.....331
29.5. Поиск подстроки в строке 333
29.6. Разбор строки на лексемы 334
29.7. Определение длины строки.....335
29.8. Функции для работы с памятью.....336
29.9. Сравнение строк, содержащих локальные символы 337
29.10. Получение строки, описывающей код ошибки 339

Глава 30. Функции для работы с файлами <stdio.h> 341

30.1. Файлы и потоки.....341
30.2. Открытие файла 343
30.3. Перенаправление потока 344
30.4. Закрытие файла 344

30.5. Работа с индикатором ошибки 344
30.6. Работа с индикатором конца файла 345
30.7. Работа с индикатором позиции файла 345
30.8. Блочный ввод/вывод 347
30.9. Ввод/вывод символов 350
30.10. Запись/чтение строк из файла 352
30.11. Форматированный вывод 354
30.12. Форматированный ввод 360
30.13. Форматирование и сканирование строк 364
30.14. Работа с буферами 366
30.15. Стандартные потоки 368
30.16. Удаление файла 369
30.17. Переименование файла 370
30.18. Работа с временными файлами 371

Глава 31. Организация нелокальных переходов <setjmp.h> 373

31.1. Сохранение информации о состоянии стека 373
31.2. Нелокальный переход 374

Глава 32. Обработка исключительных ситуаций <signal.h> 377

32.1. Сигналы 377
32.2. Установка функции обработки сигнала 378
32.3. Возбуждение сигнала 379

Глава 33. Поддержка локализации <locale.h> 382

33.1. Локальность 382
33.2. Локальные категории 383
33.3. Установка локальности 383
33.4. Форматы числовых и денежных данных 384
33.5. Определение текущих форматов данных 387

Глава 34. Работа с датами и временем <time.h> 389

34.1. Арифметические типы для представления времени 389
34.2. Структура для представления календарного времени 389
34.3. Определение времени работы программы 390
34.4. Определение текущего времени 391
34.5. Нахождение разности времен 392
34.6. Преобразование представления времени из числа в структуру 393

34.7. Преобразование представления времени из структуры в число ...395
34.8. Преобразование представления времени из числа в строку.....395
34.9. Преобразование представления времени из структуры в строку..396
34.10. Форматирование представления времени в строке.....397

ЧАСТЬ IV. СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА ПРОГРАММИРОВАНИЯ C++ 401

Глава 35. Структура стандартной библиотеки C++ 403

Глава 36. Обработка исключений <exception> 407

36.1. Классы и функции.....407
36.2. Класс *exception* 408
36.3. Класс *bad_exception* 408
36.4. Аварийное завершение программы.....409
36.5. Обработка не специфицированного исключения.....410
36.6. Определение наличия выброшенного исключения.....412

Глава 37. Классы стандартных исключений <stdexcept> 414

Глава 38. Динамическая идентификация типов <typeinfo>.....418

38.1. Динамическая идентификация типов 418
38.2. Класс *type_info*.....419
38.3. Оператор *typeid* 420
38.4. Обработка исключения *bad_typeid* 421
38.5. Обработка исключения *bad_cast* 423

Глава 39. Работа со строками в C++ <string>.....425

39.1. Характеристики символов.....426
39.2. Строки.....430
39.2.1. Ассоциированные типы 431
39.2.2. Специальное значение символьного типа..... 432
39.2.3. Конструкторы..... 432
39.2.4. Шаблоны конструкторов 434
39.2.5. Операторы присваивания..... 435
39.2.6. Функции, возвращающие итераторы..... 435
39.2.7. Получение ссылки на элемент строки..... 436
39.2.8. Оператор индексирования 437

39.2.9. Добавление символа в конец строки ... 437
39.2.10. Преобразование в строку языка C ... 437
39.2.11. Получение адреса первого элемента строки ... 438
39.2.12. Функции, работающие с длиной строки ... 438
39.2.13. Соединение строк ... 439
39.2.14. Присваивание строк ... 441
39.2.15. Вставка строк и символов в строку ... 442
39.2.16. Удаление последовательности символов из строки ... 443
39.2.17. Очистка строки ... 444
39.2.18. Замена символов и подстрок в строке ... 444
39.2.19. Копирование подстроки в массив символов ... 446
39.2.20. Обмен строк ... 447
39.2.21. Поиск символов и подстрок в строке ... 447
39.2.22. Выделение подстроки ... 449
39.2.23. Сравнение строк ... 450
39.2.24. Получение распределителя памяти для символа ... 451
39.3. Шаблоны операторов ... 451
39.3.1. Соединение строк ... 451
39.3.2. Сравнение строк ... 452
39.3.3. Ввод/вывод строк ... 453
39.4. Шаблоны функций ... 454
39.4.1. Ввод строк ... 454
39.4.2. Обмен строк ... 455

Глава 40. Потоки ввода/вывода в C++ ... 456

40.1. Структура стандартной библиотеки ввода/вывода ... 456
40.2. Базовые классы потоков <ios> ... 458
40.2.1. Класс *ios_base* ... 459
40.2.1.1. Флаги ... 460
40.2.1.2. Состояния ... 462
40.2.1.3. Режимы открытия потока ... 463
40.2.1.4. Направления поиска ... 463
40.2.1.5. События ... 464
40.2.1.6. Конструктор ... 466
40.2.1.7. Оператор присваивания ... 466
40.2.1.8. Класс *failer* ... 466
40.2.1.9. Класс *Init* ... 466
40.2.1.10. Управление разрешением дисплея ... 467
40.2.1.11. Управление шириной поля ввода/вывода ... 467
40.2.1.12. Управление памятью ... 467

40.2.1.13. Управление локальностью 468
40.2.1.14. Синхронизация с потоками языка программирования C 468
40.2.2. Шаблон класса *basic_ios* 469
40.2.2.1. Ассоциированные типы 469
40.2.2.2. Конструкторы 470
40.2.2.3. Деструктор 470
40.2.2.4. Инициализация объектов 471
40.2.2.5. Управление состоянием потока 472
40.2.2.6. Управление исключениями 472
40.2.2.7. Копирование состояния потока 473
40.2.2.8. Перегруженные операторы 473
40.2.2.9. Установка символа заполнителя 474
40.2.2.10. Управление буфером 474
40.2.2.11. Управление потоком вывода 474
40.2.2.12. Управление локальностью 475
40.2.3. Шаблон класса *fops* 475
40.2.4. Манипуляторы 477
40.3. Базовые потоки вывода <ostream> 479
40.3.1. Шаблон класса *basic_ostream* 479
40.3.1.1. Ассоциированные типы 480
40.3.1.2. Конструктор 480
40.3.1.3. Деструктор 480
40.3.1.4. Перегруженный оператор вывода << 480
40.3.1.5. Вывод символа 482
40.3.1.6. Вывод блока памяти 482
40.3.1.7. Освобождение буфера потока 482
40.3.1.8. Управление индикатором позиции 483
40.3.1.9. Класс *sentry* 484
40.3.2. Манипуляторы 485
40.3.3. Перегруженный шаблон оператора вывода << 485
40.4. Базовые потоки ввода и ввода/вывода <istream> 486
40.4.1. Шаблон класса *basic_istream* 487
40.4.1.1. Ассоциированные типы 488
40.4.1.2. Конструктор 488
40.4.1.3. Деструктор 488
40.4.1.4. Перегруженный оператор ввода >> 488
40.4.1.5. Ввод символов 490
40.4.1.6. Ввод строки 492
40.4.1.7. Просмотр символа в потоке 493

40.4.1.8. Возврат символа в буфер потока ввода 494
40.4.1.9. Ввод блока памяти 496
40.4.1.10. Игнорирование символов 496
40.4.1.11. Определение количества прочитанных символов 497
40.4.1.12. Управление индикатором позиции 497
40.4.1.13. Синхронизация потока с файлом 499
40.4.1.14. Класс *sentry* 499
40.4.2. Манипулятор 499
40.4.3. Перегруженный шаблон оператора ввода >> 500
40.4.4. Шаблон класса *basic_iostream* 501
40.5. Стандартные потоки ввода/вывода <iostream> 502
40.6. Базовые потоки ввода/вывода в файлы <fstream> 503
40.6.1. Шаблон класса *basic_ofstream* 504
40.6.2. Шаблон класса *basic_ifstream* 505
40.6.3. Шаблон класса *basic_fstream* 506
40.6.4. Работа с текстовыми файлами 507
40.6.4.1. Создание текстового файла 508
40.6.4.2. Чтение текстового файла 508
40.6.4.3. Модификация текстового файла 509
40.6.5. Работа с бинарными файлами 509
40.6.5.1. Создание бинарного файла 510
40.6.5.2. Чтение бинарного файла 511
40.6.5.3. Модификация бинарного файла 512
40.6.6. Копирование файла 513
40.7. Манипуляторы <iomanip> 514
40.7.1. Простые манипуляторы 514
40.7.2. Параметризованные манипуляторы 515

Список литературы 517

Предметный указатель 518

Рецензия на книгу "Языки программирования C и C++" доцента кафедры технологии программирования, факультета прикладной математики и информатики Белорусского государственного университета, к. т. н. Побегайло Александра Павловича

В рецензируемой книге дано довольно полное изложение языков программирования C и C++. Книга представляет собой учебное пособие по программированию на языках C и C++ для студентов высших учебных заведений, специальностей "Информатика" и "Прикладная математика". В книге полностью представлены конструкции языков программирования C и C++: типы данных, операторы и выражения, функции, классы, шаблоны. Поэтому можно считать, что рецензируемая книга представляет собой полное и замкнутое пособие по языкам программирования C и C++.

Книга "Языки программирования C и C++" может использоваться как начинающими, так и подготовленными программистами. А также может быть полезной профессиональным программистам.

Книга имеет несколько особенностей. Во-первых, изложение отличается краткостью и простотой. Во-вторых, все вопросы рассматриваются подробно и полностью. Каждое положение снабжено простым примером, что позволяет студентам быстро перейти к практическому программированию. В-третьих, процедурно-ориентированная часть языка программирования C++ излагается на базе и в сравнении с языком программирования C. Это позволяет построить курс программирования следующим образом: процедурное программирование, объектно-ориентированное программирование, обобщенное программирование.

Считаю, что книга "Языки программирования C и C++" может быть рекомендована к изданию и использоваться в учебном процессе.

Зам. генерального директора
Объединенного института проблем информатики НАН
Республики Беларусь
доктор физико-математических наук, профессор *Тузиков А. В.*

Рецензия на книгу "Языки программирования C и C++" доцента кафедры технологии программирования, факультета прикладной математики и информатики Белорусского государственного университета, к. т. н. Побегайло Александра Павловича

В рецензируемой книге рассмотрены актуальные в настоящее время языки программирования C и C++. Изложение материала отличается простотой и логической стройностью, что позволяет изучать процедурное и объектно-ориентированное программирование последовательно, обращая внимание на расширение языка программирование C конструкциями для объектно-ориентированного программирования.

Книга "Языки программирования C и C++" состоит из четырех частей:

Часть I. Язык программирования C.

Часть II. Язык программирования C++.

Часть III. Стандартная библиотека языка программирования C.

Часть IV. Стандартная библиотека языка программирования C++.

Материалы, представленные в частях I и II книги, соответствуют программе курса "Программирование" для студентов специальностей "Информатика" и "Прикладная математика" университетов и поэтому могут использоваться в качестве основы курса лекций по данному предмету. Части III и IV позволяют использовать книгу в качестве справочного пособия по программированию, так как содержат большое количество справочного материала по стандартным библиотекам языков программирования C и C++.

Рекомендую книгу "Языки программирования C и C++" к печати и использованию в учебном процессе.

Зав. кафедрой технологии программирования
факультета прикладной математики и информатики
Белорусского государственного университета
доктор технических наук, профессор *Курбацкий А. Н.*

Введение

Язык программирования C был разработан в период с 1969 по 1973 годы. Прежде чем переходить к изучению языка программирования C, скажем немного о целях, которые преследовали создатели этого языка, американские программисты Кен Томпсон и Деннис Ритчи. Язык C разрабатывался как язык системного программирования и предназначался для кодирования операционной системы Unix, которая была бы переносима на различные аппаратные платформы. Поэтому в языке программирования C так много возможностей для программирования на низком уровне. Вследствие этого язык программирования C часто также рассматривают как язык ассемблера высокого уровня. Следует отметить, что язык C получился очень мощным и компактным. С годами стало понятно, что разработка этого языка явилась выдающимся достижением в области информационных технологий.
В 1999 году разработчики этого языка были удостоены Национальной медали США за достижения в области технологии.

Язык программирования C++ был разработан с 1980 по 1983 годы Бьерном Страуструпом, сотрудником AT&T Bell Laboratories. Цель разработки этого языка программирования состояла в добавлении в язык программирования C конструкций для объектно-ориентированного программирования, а именно — классов. В 1986 году Бьерн Страуструп опубликовал книгу "Язык программирования C++", которая стала классическим руководством по программированию на языке C++ и переиздавалась не-

сколько раз. Первая спецификация языка программирования C++ была опубликована Бьерном Страуструпом и Маргарет Еллис в 1990 году и называлась "Annotated C++ Reference Manual" или сокращенно ARM. В дальнейшем эта спецификация стала основой для стандартизации языка программирования C++. Международный стандарт языка C++ был утвержден в 1998 году. По этому стандарту язык программирования C++ базируется на стандарте языка программирования C, утвержденном в 1990 году. Эти версии языков программирования C и C++ и рассматриваются в данном пособии.

Настоящая книга представляет собой курс по изучению языков программирования C и C++ для студентов университетов. Содержание и форма изложения курса обусловлены временем, отводимым программами университетов на изучение данного курса.

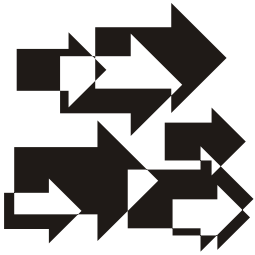
Сначала кратко коснемся содержания курса. Первые две части книги содержат описание языков программирования C и C++. Предполагается, что вопросы, изложенные в этих частях, рассматриваются и обсуждаются на лекциях. Третья и четвертая части разбираются студентами самостоятельно при выполнении лабораторных работ, т. к. на изучение стандартных библиотек языков программирования C и C++ времени в программе не хватает.

Теперь перейдем к форме изложения курса. Я считаю, что на лекциях нужно излагать концепции и технику программирования, а не решать алгоритмические задачи. Для этого предназначены другие курсы. Поэтому форма изложения книги базировалась на стандартах языков программирования C и C++, при этом каждое положение сопровождается максимально простым и понятным работающим примером. Вопросы, касающиеся практического программирования, как, например, работа со структурами, обработка строк, организация файлов, рассматриваются на семинарах и при выполнении лабораторных работ.

Особенно отмечу, что курс разбивается на две части: процедурное программирование и объектно-ориентированное программирование, которые условно совпадают с первыми двумя частями книги. В последнее время можно часто слышать, что изучать про-

граммирование надо сразу с объектно-ориентированных концепций. Мой опыт показывает, что это примерно то же самое, что начинать изучение математики с теории множеств и математической логики. Пока студент четко не усвоил технику и принципы процедурного программирования, он не сможет даже понять, почему класс имеет такой интерфейс, а не другой, не говоря уже о грамотной реализации этого интерфейса. Поэтому сначала нужно обратить внимание на основы и технику программирования, особенно при работе с такими языками программирования, как C и C++, и только затем переходить к объектно-ориентированному программированию.

Представленный материал предназначен для первого ознакомления с предметом, после усвоения изложенного материала можно переходить к более продвинутым пособиям, которые приведены в списке литературы. Изложение материала краткое, но довольно полное. Все приведенные в книге программы были проверены на работоспособность компилятором Microsoft Visual C++ 7.0 (.NET).



ЧАСТЬ I

Язык программирования C

Глава 1. Структура языка C

Глава 2. Встроенные типы данных и переменные

Глава 3. Операторы и выражения

Глава 4. Управляющие инструкции

Глава 5. Указатели и массивы

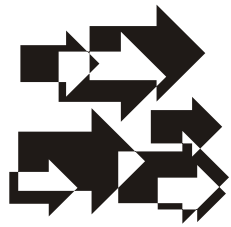
Глава 6. Функции

Глава 7. Структура программы на языке C

Глава 8. Типы данных, определяемые программистом

Глава 9. Директивы препроцессора

ГЛАВА 1



Структура языка С

1.1. Элементы языка С

Как и любой другой, язык программирования С включает следующие элементы: символы, слова и предложения. *Символы* — это базовые графические элементы языка, из которых строятся слова. *Слово* — это последовательность символов, которая имеет смысл в данном языке. Слова, используемые в языке программирования С, разбиваются на три группы: ключевые слова, идентификаторы и константы. Подробнее об этих группах слов будет рассказано в следующих разделах этой главы. Из слов и символов формируются предложения, которые в языке программирования С называются инструкциями. То есть, *инструкция* — это последовательность слов и символов языка программирования С, которая имеет смысл в этом языке. Последовательность инструкций языка С является *программой* на языке С. Программа на языке С преобразуется компилятором в последовательность машинных команд, которые исполняются микропроцессором. Схематически алгоритм работы компилятора может быть представлен блок-схемой, которая показана на рис. 1.1.

Вообще, *компилятор* — это такая программа, входными данными для которой является программа, написанная на определенном языке программирования, а выходными данными — программа в кодах команд микропроцессора, которая функционально соответствует входной программе. Другими словами можно сказать, что

компилятор переводит программу с некоторого языка программирования высокого уровня на машинный код. В соответствии с приведенной схемой процесс компиляции исходной программы можно разбить на следующие этапы:

- ❒ лексический анализ;
- ❒ синтаксический анализ;
- ❒ генерацию кода.

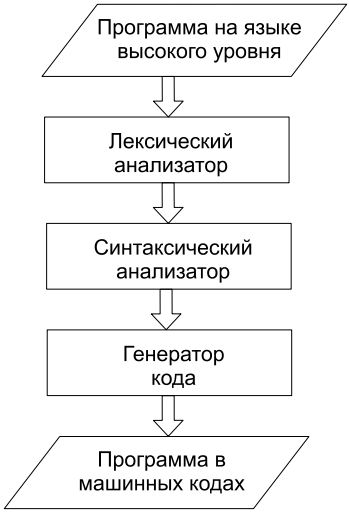


Рис. 1.1. Структурная схема работы компилятора

На этапе лексического анализа компилятор выделяет те элементы языка, из которых строятся предложения (к таким элементам относятся слова и символы). При этом символы должны иметь самостоятельное назначение и использоваться для специальных целей. К таким символам, например, относятся символы для обозначения знаков препинания и операторов. Элементы языка C, которые распознаются на этапе лексического анализа, называются *лексемами* (или *токенами*) (lexical elements или tokens). При этом компилятор проверяет правильность написания слов языка C. То есть определяется, состоят ли слова из допустимых символов языка и правильно ли эти символы используются.

На этапе синтаксического анализа компилятор проверяет, правильно ли составлены инструкции из лексем. А на этапе генерации кода в соответствие каждой инструкции ставится последовательность кодов команд микропроцессора. Если компилятор может выполнять оптимизацию кода, то генератор кода имеет более сложную структуру. В этом случае отображение инструкций языка программирования в коды команд микропроцессора не такое прямолинейное.

1.2. Символы

Каждый язык допускает только определенный набор символов. Не является исключением и язык программирования С, в котором могут использоваться только следующие символы:

- строчные латинские буквы: `a ... z`;
- прописные латинские буквы: `A ... Z`;
- цифры: `0 1 2 3 4 5 6 7 8 9`;
- специальные символы: `. , : ; ' " # { } [ ] ( ) < > & | ^ ! _ + - * / \ % = ? ~ ;`
- символ "пробел";
- нулевой символ или "пусто" (NULL).

Некоторые из символов могут обозначаться специальным образом:

- `\?` — знак вопроса;
- `\'` — апостроф;
- `\"` – кавычки;
- `\\` — обратная косая черта;
- `\0` — пусто.

Кроме того, каждый символ может быть обозначен своим кодом в восьмеричной или шестнадцатеричной системе счисления:

- `\ddd` — где буквы `ddd` обозначают код символа в восьмеричной системе счисления;

- `\xdd` — где буквы `dd` обозначают код символа в шестнадцатеричной системе счисления.

Каждый из символов языка C имеет свое назначение. Буквы и цифры используются главным образом для написания идентификаторов и литералов. Как следует из самого названия специальных символов, они имеют специальное назначение и используются главным образом для обозначения операторов.

Дополнительно в языке программирования C используются *управляющие символы*. Это такие символы, при вставке которых в текст происходит некоторое действие. К ним относятся следующие символы, которые обозначаются специальным образом:

- `\a` — сигнал тревоги;
- `\b` — возврат на шаг;
- `\f` — переход на следующую страницу;
- `\n` — переход на следующую строку;
- `\r` — переход на первую позицию текущей строки;
- `\t` — горизонтальная табуляция;
- `\v` — вертикальная табуляция.

В заключение этого раздела отметим символы, которые используются для разделения слов языка C, такие символы называются *символами разделителями* или *пробельными символами*. К этим символам относятся: пробел, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`.

1.3. Ключевые слова

Ключевые слова языка программирования C это такие слова, которые имеют предопределенное назначение в этом языке и не могут использоваться для других целей. Ниже перечислены все ключевые слова языка C:

```
auto      double  int       struct
break     else    long      switch
case      enum    register  typedef
char      extern  return    union
```

```
const     float   short     unsigned
continue  for     signed    void
default   goto    sizeof    volatile
do        if      static    while
```

Иногда ключевые слова также называют *зарезервированными словами* языка программирования.

1.4. Идентификаторы

Идентификатор — это такое слово языка С, которое может использоваться для обозначения имени переменной, имени функции, имени типа или метки инструкции. Идентификаторы могут включать только алфавитно-цифровые символы языка программирования С, а также символ подчеркивания "_". Кроме того, при написании идентификаторов должны учитываться следующие правила:

- идентификатор должен отличаться от ключевых слов языка С;
- идентификатор не должен начинаться с цифры;
- допускается любая длина идентификатора, но компилятор различает только первые 31 символ;
- в идентификаторе прописные и строчные буквы считаются различными.

Кроме того, не рекомендуется использовать символ "_" в качестве первого символа идентификатора, т. к. этот символ часто используется для именования системных переменных и функций.

В заключение этого раздела скажем, что стиль программирования языка С предлагает для имен переменных, функций и типов использовать строчные буквы, а для имен макросов — прописные буквы.

1.5. Константы

Константами или *литералами* называются некоторые фиксированные значения данных, т. е. такие значения, которые не могут изменяться.

В языке программирования C различаются четыре типа констант:

- целые константы;
- плавающие константы;
- символьные константы;
- строковые константы.

Целая константа может быть записана в десятичной, восьмеричной или шестнадцатеричной системе счисления. В десятичной системе целая константа записывается как обычное десятичное число, при условии, что первая цифра не является нулем. Например, следующие числа являются целыми десятичными целыми константами:

```
12, 234, 1009
```

В восьмеричной системе счисления целая константа записывается восьмеричными цифрами и должна начинаться с нуля. Примерами восьмеричных констант являются следующие числа:

```
012, 0234, 01007
```

В шестнадцатеричной системе счисления целая константа записывается шестнадцатеричными цифрами и должна начинаться с символов `0x` или `0X`. При этом для обозначения шестнадцатеричных цифр от `10` до `15` могут использоваться как строчные буквы `a`, `b`, `c`, `d`, `f`, так и прописные буквы `A`, `B`, `C`, `D`, `F`. Например, следующие целые числа являются целыми шестнадцатеричными константами:

```
0x12, 0X120xABC, 0Xавс
```

Кроме того, в языке программирования C разрешается объявление длинных целых констант, под которые компилятор отводит в два раза больше памяти, чем под целые константы. Для этой цели в конце целой константы ставится буква `l` или `L`. При этом заметим, что если заданное значение целой константы превышает диапазон целого типа данных, то она автоматически представляется длинной целой константой.

Константа с плавающей точкой представляет некоторое действительное число и имеет следующий вид:

```
[целая часть].[дробная часть][E|e[+|-]экспонента]
```

где целая часть, дробная часть и экспонента записываются при помощи десятичных цифр. В определении константы с плавающей точкой должна присутствовать, по крайней мере, одна из частей, заключенных во внешние квадратные скобки. Чтобы получить действительное число, которое представляется константой с плавающей точкой, необходимо целую и дробную часть этой константы умножить на десять в степени, которая задается экспонентой этой константы. Ниже приведены примеры констант с плавающей точкой:

```
3., .14        3.14, 0.314e1, 314e-2.
```

Символьная константа состоит из одного символа, который заключается в апострофы. Ниже приведены примеры символьных констант:

```
'c', 'y', '5', '\101'
```

Сам символ апостроф, используемый в качестве символьной константы, нужно обозначать как `\'`. Отметим, что символьные константы могут содержать символы, не входящие в язык программирования С, например, русские буквы.

Строковая константа представляет собой последовательность символов, заключенную в кавычки. По стандарту длина строковой константы не может превышать 509 символов. Ниже приведены примеры строковых констант.

```
"This is a string.", "Это строка." , "a", "1"
```

Сам символ кавычки, используемый в строковой константе, нужно обозначать как `\"`. Отметим, что строковые константы также могут включать символы, не принадлежащие языку программирования С. Кроме того, в конце каждой строковой константы компилятор помещает нулевой символ `\0`, который отмечает конец строки. В языке программирования С строковые константы обычно называются *строковыми литералами*.

1.6. Инструкции

Инструкцией называется любое синтаксически правильно составленное предложение языка программирования С. Инструкция должна заканчиваться символом `;`. Инструкции описывают неко-

торые действия, которые должна выполнять программа. В языке программирования C допускается пустая инструкция, которая состоит только из символа `;` и не выполняет никаких действий.

Любое количество инструкций, заключенное в фигурные скобки `{` и `}`, называется *составной инструкцией* или *блоком*. Особенно отметим, что после блока точку с запятой ставить не нужно.

1.7. Комментарии

Комментарий — это предложение на естественном языке, которое поясняет ход выполнения программы. Компилятор игнорирует комментарии. В языке программирования C комментарии начинаются парой символов `/*` и заканчиваются парой символов `*/`. Комментарии могут содержать символы, не принадлежащие языку программирования C. Комментарии разрешается применять везде, где используются пробелы. Например, комментарий может выглядеть следующим образом:

```
/* подсчет количества вариантов */
```

Замечание

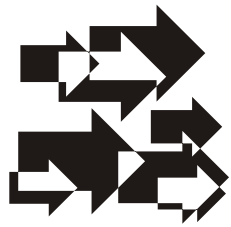
В языке программирования C++ комментарий может начинаться символами `//`. В этом случае весь текст до конца строки считается комментарием.

Например, в языке программирования C++ этот же комментарий можно написать следующим образом:

```
// подсчет количества вариантов
```

В заключение этого раздела сделаем следующее замечание. Так как программы могут использоваться десятилетиями, то поддержка и модификация программного обеспечения может обходиться значительно дороже, чем его разработка. Поэтому считается, что программа, не содержащая комментариев, ничего не стоит и не может быть принята в эксплуатацию.

ГЛАВА 2



Встроенные типы данных и переменные

2.1. Базовые типы данных

Тип данных определяется как множество значений и множество операций, допустимых над этими значениями. В табл. 2.1 приведены *базовые типы данных*, предопределенные в языке программирования C.

Таблица 2.1. *Базовые типы данных*

Тип данных	Длина в байтах	Диапазон значений
`char`	1	0...255 или -128 ... 127
`int`	Не менее 2	Зависит от длины
`float`	4	-3.4E-38 ... 3.4E+38
`double`	8	-1.8E-308 ... 1.8E+308

В стандарте языка C длина в байтах и, соответственно, диапазон типа данных `int` не определены точно и зависят от системы, на которой реализован компилятор. Под системой понимается микропроцессор и операционная система, на которой функционирует компилятор. Однако предполагается, что этот тип данных, по крайней мере, содержит диапазон целых чисел от `-32768` до `32767`.

Диапазоны типов данных `float` и `double` приведены с округлением до первого знака после точки. Точные границы этих диапазонов указаны в заголовочном файле float.h.

Кроме того, в языке программирования C определен тип данных `void`, который определяет пустое множество значений. Этот тип данных используется только в следующих случаях:

- для определения функций, которые не возвращают значения;
- для определения функций, которые не имеют параметров;
- для определения родовых указателей.

Все эти случаи будут подробно рассмотрены в соответствующих разделах.

2.2. Модификаторы типов

Для более точного выбора диапазона значений базового типа данных используются *модификаторы типов*, которые делятся на две группы: модификаторы длины и модификаторы знака. *Модификаторы длины* задаются следующими ключевыми словами:

- `short` — короткое представление;
- `long` — длинное представление.

Модификаторы знака задаются следующими ключевыми словами:

- `signed` — значение данного типа имеет знак;
- `unsigned` — значение данного типа не имеет знака.

Использование модификаторов типов данных подчиняется следующим правилам:

- с типом данных `char` используются только модификаторы знака;
- с типом данных `int` могут использоваться как модификаторы длины, так и модификаторы знака;
- с типом данных `float` по стандарту языка C модификаторы не используются, хотя некоторые компиляторы воспринимают тип `long float` как тип `double`;
- с типом данных `double` используется только модификатор длины `long`, использование с типом данных `double` модификатора длины `short` вызовет ошибку.

2.3. Спецификаторы типов

Комбинация модификаторов типа данных и базового типа данных называется *спецификатором типа данных*. При этом отметим следующие два момента.

Во-первых, в некоторых случаях для краткого обозначения спецификатора типа данных может использоваться только ключевое слово, обозначающее модификатор типа данных.

Во-вторых, при определении спецификатора типа порядок перечисления ключевых слов, обозначающих модификаторы и тип, не имеет значения. Перечислим спецификаторы типов данных, принимая во внимание второе замечание и учитывая правила использования модификаторов с простыми типами данных.

Спецификаторы символьных типов данных приведены в табл. 2.2.

Таблица 2.2. *Спецификаторы символьных типов данных*

Спецификатор типа	Длина в байтах	Диапазон значений
`char`	1	0 ... 255 или -128 ... 127
`signed char`	1	-128 ... 127
`unsigned char`	1	0 ... 255

Спецификаторы целочисленных типов данных приведены в табл. 2.3.

Таблица 2.3. *Спецификаторы целочисленных типов данных*

Спецификатор типа	Длина в байтах	Диапазон значений
`short int`	2	-32768 ... 32767
`short`	2	-32768 ... 32768
`signed short int`	2	-32768 ... 32767
`signed short`	2	-32768 ... 32767

Таблица 2.3 *(окончание)*

Спецификатор типа	Длина в байтах	Диапазон значений
`unsigned short int`	2	0 ... 65535
`unsigned short`	2	0 ... 65535
`signed int`	Не менее 2	Зависит от длины
`signed`	Не менее 2	Зависит от длины
`int`	Не менее 2	Зависит от длины
`unsigned int`	Не менее 2	Зависит от длины
`unsigned`	Не менее 2	Зависит от длины
`long int`	4	–2147483648 ... 2147483647
`long`	4	–2147483648 ... 2147483647
`signed long int`	4	–2147483648 ... 2147483647
`signed long`	4	–2147483648 ... 2147483647
`unsigned long int`	4	0 ... 4294967295
`unsigned long`	4	0 ... 4294967295

Спецификаторы типов данных с плавающей точкой приведены в табл. 2.4.

Таблица 2.4. *Спецификаторы типов данных с плавающей точкой*

Спецификатор типа	Длина в байтах	Диапазон значений
`float`	4	-3.4E-38 ... 3.4E+38
`double`	8	-1.8E-308 ... 1.8E+308
`long double`	10	-1.2E-4932 ... 1.2E4932

Спецификаторы типов данных определяют полный набор типов данных, *встроенных* в язык программирования C. Встроенные типы данных также называются *предопределенными*, или *стандартными*, или *фундаментальными* типами данных.

Типы данных, построенные на основе встроенных типов данных, называются *сложными* (compound). Поэтому встроенные типы данных часто также называются *простыми* (simple). К сложным типам данных относятся указатели, массивы, перечисления, структуры и объединения. Все эти типы данных будут рассмотрены в дальнейшем.

2.4. Переменные

Переменная или *объект* — это поименованная область памяти, в которой хранятся данные. Для именования переменных используются идентификаторы. Прежде чем переменная может использоваться в программе, она должна быть объявлена. Объявление переменной выглядит следующим образом:

```
спецификатор_типа список_идентификаторов;
```

где `спецификатор_типа` является одним из спецификаторов, рассмотренных в предыдущем разделе, а `список_идентификаторов` представляет собой один или несколько идентификаторов, отделенных друг от друга запятыми. Например, переменные могут быть объявлены следующим образом:

```
float d;
int   i, j, k;
long double   b;
unsigned long u;
```

Не может быть объявлена переменная типа `void`, т. к. компилятору неизвестно, сколько памяти нужно отводить под такую переменную.

Замечание

В языке программирования C переменные должны объявляться в начале блока. В языке программирования C++ переменные могут объявляться там, где допустимо использование инструкции.

При своем объявлении переменным могут присваиваться начальные значения или, другими словами, переменные могут инициализироваться. Для инициализации переменных используется опе-

ратор присваивания, который в языке программирования C обозначается символом =. При этом в правой части оператора присваивания пишется выражение, которое включает константы или уже проинициализированные переменные. Язык программирования C допускает присваивание переменной значения, тип которого отличается от типа переменной. В этом случае выполняется неявное преобразование типов. Но об этом будет рассказано в следующем разделе. А теперь покажем, как инициализируются переменные.

```
float  f = 3.14f;
double d = 3.14, g = f;
int    i, j = 10, k = j + 5;
```

В заключение этого раздела отметим, что нужно придерживаться следующего правила: все объявляемые переменные должны быть инициализированы. Следование этому правилу упрощает отладку программы.

2.5. Фундаментальные типы данных

Как уже говорилось в предыдущем разделе, язык программирования C допускает преобразование типов данных. Для упрощения терминологии, связанной с этим вопросом, встроенные в язык программирования C фундаментальные типы данных разбиваются на три группы: интегральный тип, тип с плавающей точкой и пустой тип.

Интегральный тип данных содержит все типы, спецификатор которых содержит хотя бы одно из ключевых слов `char`, `short`, `int` или `long`, исключая тип `long double`. Другими словами можно сказать, что интегральный тип содержит символьный и все целочисленные типы данных языка программирования C. Слово "интегральный" говорит о том, что каждый из типов, входящих в эту группу, содержит непрерывное множество значений.

Тип данных с *плавающей точкой* содержит типы `float`, `double` и `long double`. В отличие от интегрального типа множество значений любого типа с плавающей точкой не является непрерывным.

Иногда тип с плавающей точкой называют просто *плавающим* типом.

Пустой тип данных содержит только тип данных `void`, который в свою очередь не содержит ни одного значения.

Интегральный тип и тип данных с плавающей точкой объединяются в *арифметический* или *числовой* тип данных.

2.6. Квалификаторы типов

Квалификатор типа данных это ключевое слово, которое управляет доступом к переменной. Поэтому квалификаторы типов также иногда называются *модификаторами доступа*. В языке программирования C существует два квалификатора типов: `const` и `volatile`. Квалификатор типа может использоваться как перед спецификатором типа, так и после него.

Квалификатор `const` говорит компилятору о том, что идентификатор является именем константы. Поэтому такую константу часто также называют *именованной константой*. Значение именованной константы нельзя изменить. При этом заметим, что именованная константа всегда должна быть инициализирована при своем объявлении. Приведем пример объявления именованной константы.

```
const int i = 1;
```

Константа именуется для упрощения модификации программы в случае изменения значения этой константы.

Квалификатор `volatile` говорит компилятору о том, что значение переменной может быть изменено при исполнении кода, который не принадлежит программе, в которой объявлена переменная. Например, программой обработки некоторого сигнала. Использование квалификатора `volatile` гарантирует, что все программы обратятся к одной и той же переменной. Если в таких случаях не использовать квалификатор `volatile`, то может возникнуть следующая ситуация. Компилятор может оптимизировать код и поместить в одной программе значение переменной в регистр, а в другой — в какую-то область памяти. В результате разные про-

граммы фактически будут обращаться к разным переменным. Поэтому переменные, объявленные с квалификатором `volatile`, всегда хранятся в памяти. Учитывая вышесказанное, такие переменные часто также называют *нестабильными* или *изменчивыми* переменными. Приведем пример объявления нестабильной переменной:

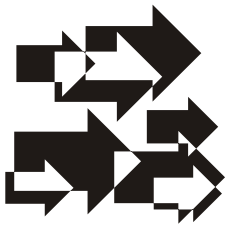
```
volatile int i = 1;
```

Отметим, что допустимо совместное использование квалификаторов `const` и `volatile`. Например:

```
volatile const int i = 1;
```

В этом случае именованная константа определяет область памяти, содержимое которой не может быть изменено внутри программы, но может быть изменено другой программой.

ГЛАВА 3



Операторы и выражения

3.1. L-value и R-value

Данные в языке программирования C могут представляться литералами и переменными. И те и другие хранят значения данных в областях памяти. Это хранимое значение данных называется R-value, что является сокращением от английского названия read value — т. е. значение, которое можно прочитать. Фундаментальное различие между литералами и переменными заключается в том, что переменная именует область памяти, в которой хранится R-value (т. е. фактически переменная представляет адрес этой области), а литерал — нет. Значение адреса области памяти для хранения значения переменной называется L-value, что является сокращением от английского названия location value, т. е. значение, которое определяет местоположение. Таким образом, переменная определяет как R-value, так и L-value, а литерал определяет только R-value.

Понятия L-value и R-value вводятся для того, чтобы точнее определить, какие значения могут быть операндами операторов и какое значение получается в результате выполнения оператора. Так как L-value определяет адрес, то по этому адресу может быть записано новое значение переменной. Литерал же не может получить нового значения. Однако отметим, что если L-value представляет именованную константу, то по этому адресу также не может быть записано новое значение данных. Говорят, что

L-value является *модифицируемым*, если по адресу, определяемому этим значением, может быть записано новое значение данных.

3.2. Арифметические операторы

Над данными арифметического типа определены следующие *унарные* операторы:

- `+` — знак числа не изменяется;
- `-` — изменение знака числа на противоположный знак.

И *бинарные* операторы (которые также называются *арифметическими*):

- `+` — сложение двух чисел;
- `-` — вычитание двух чисел;
- `*` — умножение двух чисел;
- `/` — деление двух чисел.

При этом отметим, что если оператор деления выполняется над данными интегрального типа, то результатом является частное от деления.

Например, над двумя целыми числами можно выполнить следующие арифметические операторы:

```
int x = 1, y = 2, z;
z = + x;        /* z = 1  */
z = x + y;      /* z = 3  */
z = - x;        /* z = -1 */
z = x - y;      /* z = -1 */
z = x * y;      /* z = 2  */
z = x / y;      /* z = 0  */
```

Над данными интегрального типа также определен бинарный арифметический оператор:

- `%` — деления двух чисел по модулю, который дает в результате остаток от деления.

Например:

```
int x = 11, y = 3, z;
z = x % y;    /* z = 2 */
```

Отметим, что операндами арифметических операторов могут быть как переменные, так и литералы арифметических типов, а результатом является R-value.

3.3. Побитовые операторы

Над данными интегрального типа определены следующие *побитовые операторы*:

- `<<` — бинарный оператор сдвига влево;
- `>>` — бинарный оператор сдвига вправо;
- `~` — унарный оператор дополнения до 1 или отрицания;
- `&` — бинарный побитовый оператор И;
- `|` — бинарный побитовый оператор ИЛИ;
- `^` — бинарный побитовый оператор исключающее ИЛИ.

Операндами побитовых операторов могут быть как переменные, так и литералы, а результатом является R-value. Опишем подробнее работу побитовых операторов.

Побитовая операция `x<<n` сдвигает двоичное представление числа `x` на `n` разрядов влево. При этом биты, освобождающиеся справа, заполняются нулями. Например,

```
short x = 0x0001, y;
y = x << 1;    /* y = 0x0002 */
```

Отметим, что оператор левого сдвига не обеспечивает обработку ситуации переполнения левого операнда. В случае если переполнения нет, то сдвиг влево на один разряд эквивалентен умножению левого операнда на 2.

Побитовая операция `x>>n` сдвигает двоичное представление числа `x` на `n` разрядов вправо. Если число без знака, то освобождающиеся слева биты заполняются нулями. Если же число `x` имеет знак,

то освобождающиеся слева биты заполняются копией знакового бита. Например:

```
short x = 0x0004, y;
y = x >> 2;     /* y = 0x0001 */
```

Сдвиг вправо на один бит эквивалентен делению левого операнда на 2.

Замечание

Если второй операнд оператора сдвига является отрицательным числом, то результат оператора сдвига не определен.

Побитовая операция `~x` инвертирует битовое представление числа `x`. То есть над битами числа `x` выполняются следующие преобразования:

- `~0 = 1`;
- `~1 = 0`.

Например:

```
short x = 0x000F, y;
y = ~x;       /* y = 0xFFF0 */
```

Побитовая операция `x&y` выполняет над соответствующими битами чисел `x` и `y` логическую операцию И. То есть над соответствующими битами операндов выполняются следующие действия:

- `0 & 0 = 0`;
- `0 & 1 = 0`;
- `1 & 0 = 0`;
- `1 & 1 = 1`.

Например:

```
int x = 0x001F, y = 0x00F0, z;
z = x & y;       /* z = 0x0010 */
```

Побитовая операция `x|y` выполняет над соответствующими битами чисел `x` и `y` логическую операцию ИЛИ. То есть над соответствующими битами операндов выполняются следующие действия:

- `0 | 0 = 0`;
- `0 | 1 = 1`;
- `1 | 0 = 1`;
- `1 | 1 = 1`.

Например:

```
int x = 0x000F, y = 0x00F0, z;
z = x | y;          /* z = 0x00FF */
```

Побитовая операция `x^y` выполняет над соответствующими битами чисел `x` и `y` логическую операцию исключающее ИЛИ. То есть над соответствующими битами операндов выполняются следующие действия:

- `0 ^ 0 = 0`;
- `0 ^ 1 = 1`;
- `1 ^ 0 = 1`;
- `1 ^ 1 = 0`.

Например:

```
int x = 0x00FF, y = 0x0FF0, z;
z = x ^ y;        /* z = 0x0F0F */
```

3.4. Операторы сравнения

В языке программирования C нет логических или булевых типов данных. Вместо этого считается, что любое ненулевое значение является истинным, а равное нулю — ложным. Операторы сравнения и логические операторы просто вычисляют значение типа `int`, которое равно `0` или `1`, в зависимости от значений операндов.

В языке программирования C определены следующие *операторы сравнения*:

- `<` — бинарный оператор меньше;
- `>` — бинарный оператор больше;
- `<=` — бинарный оператор не больше;

- `>=` — бинарный оператор не меньше;
- `==` — бинарный оператор равно;
- `!=` — бинарный оператор не равно.

Например, над двумя целыми числами можно выполнить следующие операторы сравнения:

```
int x = -1; y = 0, z;
z = x < y;      /* z = 1 */
z = x > y;      /* z = 0 */
z = x <= y;     /* z = 0 */
z = x >= y;     /* z = 0 */
z = x == y;     /* z = 0 */
z = x != y;     /* z = 1 */
```

Операндами операторов сравнения могут быть переменные и литералы арифметических типов и указатели, о которых будет рассказано далее. Результатом оператора сравнения является R-value.

3.5. Логические операторы

Логические операторы выполняют над своими операндами логические операции НЕ, И и ИЛИ, учитывая при этом, что любое не равное нулю значение считается истинным, а равное нулю значение — ложным. В языке программирования C определены следующие логические операторы:

- `!` — унарный оператор логического отрицания;
- `&&` — бинарный оператор логическое И;
- `||` — бинарный оператор логическое ИЛИ.

Логические операторы работают следующим образом. Перед вычислением значения логического оператора вычисляется логическое значение его операндов. После этого над полученными значениями операндов выполняется логическая операция. Результатом логической операции является R-value.

Например, над целыми числами можно выполнить следующие логические операции:

```
int x = -1, y = 0, z;
z = !x;         /* z = 0 */
z = x && y;     /* z = 0 */
z = x || y;     /* z = 1 */
```

3.6. Выражения

Выражением называется синтаксически правильно построенная последовательность операторов и их операндов. Под словами синтаксически правильная последовательность понимается, что каждому оператору поставлены в соответствие операнды, которые в свою очередь тоже могут быть выражениями. Из этого определения следует, что выражения строятся рекурсивно и могут содержать несколько операторов. Ниже приведены примеры правильного и ошибочного выражений:

```
x + y * z     /* правильное выражение */
x + y %% z    /* ошибочное выражение  */
```

Приоритетом оператора (operator precedence) называется очередность вычисления оператора в выражении, содержащем несколько операторов. Операторы, рассмотренные в предыдущих разделах, имеют следующие приоритеты, которые расположены по убыванию:

- `! ~ - +` — унарные операторы;
- `* / %` — бинарные мультипликативные арифметические операторы;
- `+ -` — бинарные аддитивные арифметические операторы;
- `<< >>` — бинарные операторы сдвига влево и вправо;
- `< > <= >=` — бинарные операторы сравнения на неравенство;
- `== !=` — бинарные операторы сравнения на равенство;
- `&` — бинарный побитовый оператор И;
- `^` — бинарный побитовый оператор исключающее ИЛИ;
- `|` — бинарный побитовый оператор ИЛИ;
- `&&` — бинарный логический оператор И;
- `||` — бинарный логический оператор ИЛИ.

Если выражение содержит несколько операторов, которые имеют одинаковый приоритет, то порядок вычисления этих операторов не определен.

Для того чтобы задать порядок вычисления операторов, в выражении используются круглые скобки. Операторы, заключенные в круглые скобки со своими операндами, вычисляются до операторов, которые находятся вне этих круглых скобок. Например, в следующем выражении оператор + вычисляется до оператора *.

```
int x = 1, y =2, z;
z = (x + y) * y;     /* z = 6 */
```

При программировании выражений лучше расставлять скобки, это значительно упрощает понимание выражения.

Если в выражении нет круглых скобок, то порядок вычисления нескольких подряд идущих одинаковых операторов определяется *ассоциативностью операторов*. Унарные операторы вычисляются справа налево, а бинарные наоборот — слева направо. Например, выражение:

```
x + y + z,
```

эквивалентно выражению:

```
(x + y) + z.
```

Кроме того, при разборе выражения компилятор распознает лексемы наибольшей длины. Например, выражение:

```
y = ++x;,
```

интерпретируется как:

```
y = (++x);,
```

а не как:

```
y = +(+x);.
```

Принимая во внимание, что символы ++ обозначают оператор инкремента, который будет рассмотрен далее.

3.7. Приведение типов в выражениях

Приведением типов называется преобразование значения одного типа к значению другого типа, при этом возможно как сохранение величины этого значения, так и изменение этой величины. Часто приведение типов также называют *преобразованием типов*.

Если в выражении встречаются переменные и константы разных типов, то все они приводятся к типу с наибольшим диапазоном значений. Такое приведение типов называется *неявным*. Перечислим типы по убыванию их диапазонов значений:

```
long double
double
float
unsigned long
long
unsigned int
int
```

Как видим, в этой иерархии отсутствуют типы, диапазон значений которых не превышает диапазон значений типа `int`. Причина этого заключается в том, что если выражение содержит операнды только интегральных типов, диапазон которых не превышает диапазон типа `int`, то перед вычислением это выражение всегда преобразуется компилятором в эквивалентное выражение, все операнды которого имеют тип `int`. Это делается для того, чтобы повысить скорость работы программы.

Приведение типов внутри интегрального типа данных называется *интегральным расширением* (integral promotion).

Приведение типов внутри плавающего типа данных называется *расширением с плавающей точкой* (float-point promotion).

Можно сказать, что интегральным расширением и расширением с плавающей точкой являются такие приведения типов внутри интегрального и плавающего типов соответственно, при которых не теряется числовое значение данных приводимого типа.

Стиль программирования на языке C поощряет неявное преобразование типов данных. Ниже приведен пример неявного преобразования типа в выражении:

```
int x = 1;
double y = 1.5, z;
z = x + y;  /* тип переменной x приводится к типу double */
```

В заключение этого раздела отметим, что неявное преобразование типов данных в выражениях определяет полиморфизм встроенных операторов, который называется *принудительным полиморфизмом* (coercion polymorphism).

3.8. Оператор преобразования типов

Для явного преобразования типа выражения используется *оператор преобразования типа*, который имеет вид:

```
(тип) выражение
```

Этот оператор приводит значение `выражения` к типу, который указан в круглых скобках. Например, следующее выражение приводится к типу `int`:

```
(int)(1.5 / 0.3)
```

А в следующем выражении к типу `int` приводится только числитель, а тип всего выражения равен `double`:

```
(int)1.5 / 0.3
```

Заметим, что приоритет оператора преобразования типа ниже приоритета унарных операторов, но выше приоритета бинарных мультипликативных арифметических операторов. Результатом оператора преобразования типа является R-value.

Замечание

В языке программирования C++ результат оператора преобразования типа может дать в результате L-value, если тип, к которому приводится выражение, является ссылкой.

Учитывая замечание, в языке программирования C++ можно написать следующее выражение:

```
char* p;
(int&)p = 10;
```

Оператор преобразования типа позволяет выполнить любое преобразование арифметических типов данных. С помощью этого оператора также возможно преобразовать любой тип данных в тип данных `void`. Однако при помощи этого оператора нельзя преобразовать тип `void` ни в какой другой тип данных.

3.9. Оператор присваивания

Оператор присваивания обозначается символом `=` и имеет следующий вид:

```
операнд_1 = операнд_2;
```

В результате выполнения оператора присваивания `операнду_1` присваивается значение `операнда_2`. При этом `операнд_1` должен быть модифицируемым L-value, а `операнд_2` представлять R-value.

Примечание

В связи с этим фактом часто полагают, что L-value является кратким обозначением для left value т. е. значение, которое может использоваться слева от оператора присваивания, а R_value — кратким обозначением для right value, т. е. значение, которое может использоваться только справа от оператора присваивания. Это не совсем корректно, но такая трактовка наглядно показывает, что собой представляют L- и R-value.

Результатом оператора присваивания является R-value, значение которого равно значению `операнда_2`. Приоритет оператора присваивания ниже приоритета всех бинарных операторов. Поэтому `операнд_2` в операторе присваивания может быть выражением.

При присваивании допускаются разные типы операндов. В этом случае полученное значение выражения, которое находится справа от оператора присваивания, неявно приводится к типу переменной, которая находится слева от оператора присваивания. Допускаются любые преобразования между арифметическими типами данных. Но если диапазон типа левого операнда меньше диапазона типа значения выражения, то компилятор может выдать предупреждение. Например:

```
int x = 1;
double d;
d = x + 2;  /* d = 3.0 */
```

Но в этих случаях нужно быть внимательным, т. к. возможны ошибки следующего типа:

```
int x = 1;
double d;
d = x / 2;  /* d = 0 */
```

В последнем примере не учитывается, что выражение сначала вычисляется, а потом преобразуется к типу левого операнда оператора присваивания. Чтобы избежать этой ошибки, нужно использовать плавающий тип в выражении справа от оператора равно. Например:

```
int  x = 1;
double  d;
d = x / 2.0;  /* d = 0.5 */
```

Так как оператор присваивания возвращает R-value, то возможно последовательное использование операторов присваивания. Например, в языке программирования C возможна запись следующего выражения:

```
int x, y, z;
x = y = z = 1;     /* x = 1, y = 1, z =1 */
```

Оно эквивалентно выражению:

```
x = (y = (z = 1));     /* x=1, y = 1, z =1 */
```

То есть последовательные операторы присваивания вычисляются справа налево.

Замечание

В языке программирования C++ оператор присваивания возвращает L-value.

Учитывая это замечание, в языке программирования можно написать следующее выражение:

```
int n;
++(n = 10);  // n = 11
```

3.10. Составные операторы присваивания

В языке программирования C также используются *составные* операторы присваивания, которые объединяют арифметический или побитовый оператор с оператором присваивания. В общем случае составной оператор присваивания имеет следующий вид:

```
операнд_1  оператор=  операнд_2
```

Здесь `оператор` обозначает один из арифметических или побитовых операторов. Далее перечислены все составные операторы присваивания, определенные в языке программирования C:

- `+=` — прибавление числа и присваивание;
- `-=` — вычитание числа и присваивание;
- `*=` — умножение на число и присваивание;
- `/=` — деление на число и присваивание;
- `%=` — деление по модулю на число и присваивание;
- `<<=` — сдвиг влево числа и присваивание;
- `>>=` — сдвиг вправо числа и присваивание;
- `&=` — побитовое И и присваивание;
- `^=` — побитовое исключающее ИЛИ и присваивание;
- `|=` — побитовое ИЛИ и присваивание.

Работают составные операторы присваивания следующим образом. Сначала выполняется операция над левым и правым операндами составного оператора присваивания, а затем полученный результат присваивается левому операнду. Например:

```
int x = 1, y = 2;
y += x;    /* y = 3 */
```

Это эквивалентно выражению:

```
y = y + x;
```

Отметим, что приоритет составных операторов присваивания совпадает с приоритетом простого оператора присваивания.

Поэтому выражение:

```
a *= b - c;,
```

эквивалентно выражению:

```
a = a * (b - c);
```

Смысл введения составных операторов присваивания в язык программирования C заключается в следующем. В микропроцессорах существуют команды, которые работают аналогично составным операторам присваивания. Причем эти команды работают быстрее, чем команды, которые выполняют операцию над двумя операндами и результат этой операции заносят в третий операнд. Так как язык программирования C разрабатывался как язык ассемблера высокого уровня, то аналоги упомянутых команд микропроцессора и были введены в синтаксис языка C.

3.11. Операторы инкремента и декремента

Над переменными арифметических типов определены также унарные операторы:

- ++ — инкремент, т. е. увеличение значения числа на единицу;
- -- — декремент, т. е. уменьшение значения числа на единицу.

Эти операторы могут быть как *префиксными*, так и *постфиксными*, т. е. записываться как перед операндом, так и после него. Операндом как префиксного, так и постфиксного оператора инкремента или декремента должно быть модифицируемое L-value. Результатом как префиксного, так и постфиксного оператора является R-value. При этом префиксные операторы инкремента и декремента дают в результате измененное значение переменной, а результатом постфиксных операторов инкремента и декремента является старое значение переменной.

Следующий пример демонстрирует разницу между префиксными и постфиксными операторами инкремента и декремента:

```
int a, b = 1;
a = b++;    /* a = 1, b = 2 */
```

```
a = ++b;    /* a = 3, b = 3 */
a = b--;    /* a = 3, b = 2 */
a = --b;    /* a = 1, b = 1 */
```

Приоритет префиксных операторов инкремента и декремента совпадает с приоритетом унарных операторов. А приоритет постфиксных операторов инкремента и декремента выше приоритета унарных операторов и, следовательно, выше приоритета префиксных операторов инкремента и декремента. Поэтому выражение:

```
++x++,
```

вызовет ошибку компиляции, т. к. результатом постфиксного оператора декремента является R-value.

Замечание

В языке программирования C++ результатом префиксного оператора инкремента или декремента является L-value. Поэтому эти операторы могут встречаться слева от оператора присваивания.

Учитывая это замечание, в языке программирования C++ можно написать следующее выражение:

```
int x = 2, y = 3;
++y += x;    /* x = 2, y = 6 */
```

3.12. Условный оператор

Условный оператор является тернарным, обозначается двумя символами: `?` и `:`, и имеет следующий вид:

```
операнд_1 ? операнд_2 : операнд_3
```

Вычисляется условный оператор следующим образом. Сначала вычисляется логическое значение `операнда_1`. Если это значение истинно, то вычисляется `операнд_2`, значение которого и является значением условного оператора. В противном случае вычисляется `операнд_3` и его значение является значением условного оператора. Например, в следующем примере с помощью условного оператора вычисляется минимум из чисел `x` и `y`.

```
int x= 1, y = 2, min;
min = (x < y) ? x : y;  /* min = 1 */
```

Допускаются следующие типы второго и третьего операндов условного оператора:

- оба операнда имеют арифметический тип;
- оба операнда имеют тип `void`;
- оба операнда являются указателями на сравнимые типы;
- оба операнда являются структурами или объединениями одного типа;
- один операнд является указателем, а второй — пустым указателем;
- один операнд является указателем, а второй указателем на тип `void`.

Об указателях и структурах будет рассказано далее. Теперь же отметим, что приоритет условного оператора ниже, чем приоритет бинарных операторов, но выше приоритета оператора присваивания.

Результатом вычисления условного оператора является R-value. При вычислении условного оператора выполняются следующие преобразования типов значений второго и третьего операндов, независимо от того, какой из операндов вычисляется:

- если оба операнда имеют арифметический тип, то результат имеет тип операнда с наибольшим диапазоном значений;
- если один из операндов является указателем на тип `void`, то результат также будет иметь тип указателя на тип `void`;
- если один из операндов является указателем, а другой — пустым указателем, то результат имеет тип не пустого указателя.

Ассоциативны условные операторы справа налево. Поэтому следующее выражение:

```
int x = 1, y = 2, z = 3;
x < y ? y < z ? --x : --y : --z;   /* x = 0, y = 2, z = 3 */
```

эквивалентно выражению:

```
x < y ? (y < z ? --x : --y) : --z; /* x = 0, y = 2, z = 3 */
```

Замечание

В языке программирования C++ результатом условного оператора может быть и L-value, если результатом вычисления как `операнда_2`, так и `операнда_3` являются L-value одного типа. В противном случае результатом вычисления условного оператора является R-value.

Учитывая это замечание, в языке программирования C++ можно написать так:

```
int x = 1, y = 2;
((x < y) ? x : y) = 0;  /* x = 0, y = 2 */
```

но:

```
int x = 1, y = 2;
(x < y) ? x : y = 0;    /* x = 1, y = 2 */
```

т. к. условный оператор имеет приоритет выше, чем оператор присваивания.

3.13. Оператор "запятая"

Оператор "запятая" обозначается символом `,` и имеет следующий вид:

```
операнд_1, операнд_2,
```

где операнды могут быть произвольными выражениями. Работает оператор "запятая" следующим образом. Сначала вычисляется значение `операнда_1`, а затем значение `операнда_2`. Значением оператора `,` является R-value, равное значению `операнда_2`, а значение `операнда_1` игнорируется. Например:

```
int x = 1, y = 2, z;
z = (x++, x - y);    /* z = 0 */
```

но:

```
z = x++, x - y;      /* z = 1 */
```

т. к. оператор "запятая" имеет наименьший приоритет из всех операторов. Оператор `,` имеет ассоциативность слева направо.

Замечание

В языке программирования C++ результатом оператора , может быть и L-value, если оно является результатом вычисления второго операнда.

Учитывая это замечание, в языке программирования C++ можно написать следующее выражение:

```
int x = 1, y = 2, z;
x++, z = x - y;    /* z = 0 */
```

На практике оператор , обычно используется там, где требуется вычислить несколько выражений, хотя по синтаксису допускается запись только одного выражения. Например, цикл `for` может быть инициализирован следующим образом:

```
for (i = 1, j = n; i < j; ++i, --j)
  a[i] = b[j];
```

3.14. Побочные эффекты

Если при вычислении выражения значение переменной, входящей в это выражение, изменяется, то говорят, что произошел *побочный эффект*. Например, вычисление выражения:

```
++x - y
```

дает побочный эффект, т. к. при его вычислении изменяется значение переменной `x`. Но в этом случае результат вычисления выражения предсказуем, т. к. оператор инкремента имеет более высокий приоритет, чем оператор вычитания. Но если операторы имеют одинаковый приоритет, то результат вычисления выражения может быть непредсказуемым, т. к. последовательность вычисления таких операторов не определена. Например, в языке программирования C можно написать следующее выражение:

```
y = x++ + x--;
```

результат вычисления которого не определен. Побочные эффекты также встречаются при использовании оператора индексирования, например:

```
x[i] = i++;
```

И при вызове функций, например:

```
f(i + 1, ++i);
```

Таких ситуаций следует избегать, т. к. в этом случае работа программы будет зависеть от компилятора.

Отметим, что вычисление оператора присваивания всегда дает побочный эффект, т. к. изменяется значение левого операнда этого оператора.

3.15. Точки последовательности

В языке программирования С при вычислении значения выражения определяются такие точки, до перехода через которые все предыдущие вычисления должны быть завершены. Такие точки называются *точками последовательности* (sequence points). Точки последовательности фиксируют побочные эффекты при вычислении значения выражения. Между точками последовательности значение любого объекта должно модифицироваться только один раз, в противном случае значение выражения не определено. В языке программирования С определены следующие точки последовательности при вычислении значения выражения:

- левый операнд логического оператора `&&`, т. е. вычисляется значение левого операнда логического оператора `&&`, и если оно равно 0, то дальнейших вычислений не происходит;
- левый операнд логического оператора `||`, т. е. вычисляется значение левого операнда логического оператора `||`, и если оно равно 1, то дальнейших вычислений не происходит;
- левый операнд оператора `,`(запятая), т. е. перед вычислением правого операнда оператора `,` вычисление его левого операнда полностью заканчивается;
- первый операнд условного оператора, т. е. вычисление этого операнда полностью завершается перед вычислением значений других операндов.

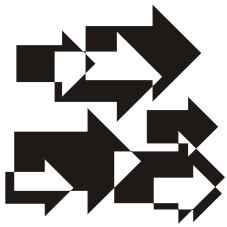
Например, при вычислении значения следующего выражения вызов функции `f` не происходит, т. к. левый операнд логического оператора `&&` дает в результате `0`.

```
int x = 0, y;
y = x && f(x);       /* y = 0 */
```

Однако в следующем случае функция `f` вызывается:

```
int x = 1, y;
y = x && f(x);
```

ГЛАВА 4



Управляющие инструкции

4.1. Инструкции выбора *if* и *if...else*

Инструкция `if` имеет следующий синтаксис:

```
if (выражение)
  инструкция
```

Работает `if` следующим образом. Сначала вычисляется логическое значение `выражения`. При этом отметим, что при вычислении `выражения` завершаются все побочные эффекты. Если это логическое значение истинно, то выполняется `инструкция`. В противном случае `инструкция` пропускается. Например, в следующем примере значение `x` увеличивается на единицу только в том случае, если выполняется условие `x > 0`:

```
if (x > 0)
  ++x;
```

Если в зависимости от логического значения `выражения` необходим выбор одной из двух инструкций, то используется инструкция `if...else`, которая имеет следующий синтаксис:

```
if (условие)
  инструкция_1
else
  инструкция_2
```

В этом случае, если логическое значение `выражения` истинно, то выполняется `инструкция_1`, в противном случае выполняется `инструкция_2`. Например:

```
if (a > 0)
  ++a;
else
  --a;
```

Отметим, что инструкции `if` и `if...else` могут быть вложенными. Например:

```
if (a > 0)
  if (b > 0)
    c = a + b;
  else
    c = a - b;
```

В этом примере ключевое слово `else` относится ко второй инструкции `if`. Чтобы связать это слово с первой инструкцией `if`, нужно заключить вторую инструкцию `if` в блок, т. е. написать:

```
if (a > 0)
{
   if (b > 0)
     c = a + b;
}
else
  c = a - b;
```

4.2. Инструкция выбора *switch*

Инструкция выбора `switch` имеет следующий вид:

```
switch (выражение)
{
case  константа_1:
  инструкции
case  константа_2:
  инструкции
...
```

```
default:
  инструкции
}
```

Работает инструкция `switch` следующим образом. Сначала вычисляется значение `выражения`, при этом завершаются все побочные эффекты. Затем полученное значение `выражения` сравнивается с константами, и управление передается на ту `инструкцию`, для которой это сравнение дает значение "истина". То есть ключевое слово `case` и следующая за ним константа определяют просто метку инструкции, на которую может быть передано управление. Для выхода из блока `case` используется оператор `break` *(см. разд. 4.5)*. Если значение `выражения` не совпадает ни с одной из констант, то управление передается `инструкции` с меткой `default`. А если этой метки нет, то управление передается инструкции, следующей за блоком инструкции `switch`. Инструкция `switch` используется в основном для "разбора случаев". Например:

```
int x = 2, y = 0;
switch(x)
{
case 0:
case 1: ++y;
case 2: y += 2;
case 3: y += 3;
}    /* y = 5 */
```

На инструкцию `switch` накладываются следующие ограничения:

- выражение и константы в инструкции `switch` должны иметь интегральный тип;
- две константы не могут иметь одинаковое значение;
- при сравнении значения выражения с константами используется интегральное продвижение по типам.

Относительно метки `default` справедливы следующие замечания:

- вариант `default` не обязателен;
- если вариант `default` присутствует, то он не обязательно должен быть последним.

В языке программирования C переменные могут объявляться только в начале блока `switch`, а в языке программирования C++ и внутри блока. Кроме того, в языке программирования C++ нельзя инициализировать переменные, объявленные в блоке `switch`.

4.3. Инструкции цикла *while* и *do...while*

Инструкция цикла `while` имеет следующий синтаксис:

```
while (выражение)
  инструкция
```

Работает инструкция `while` следующим образом. Сначала вычисляется значение `выражения`, при этом завершаются все побочные эффекты. Результат вычисления `выражения` должен иметь арифметический тип или указатель. Если полученное значение истинно, то выполняется `инструкция`. Так повторяется до тех пор, пока значение `выражения` не станет ложным. Например:

```
int a = -2;
while (a)
  ++a;      /* a = 0 после выхода их цикла */
```

`Выражение` после ключевого слова `while` часто называют *условием продолжения цикла*. Если это условие выполняется, то цикл продолжается. В противном случае цикл завершается.

Инструкция цикла `do...while` имеет следующий синтаксис:

```
do
  инструкция
while (выражение);
```

Работает инструкция `do...while` следующим образом. `Инструкция` выполняется до тех пор, пока значение `выражения` остается истинным. Но в отличие от инструкции `while`, значение `выражения` вычисляется всякий раз после выполнения `инструкции` и это значение должно иметь арифметический тип или быть указателем.

То есть в случае do...while инструкция отработает хотя бы один раз, независимо от значения выражения.

Например:

```
int a = -2;
do
  ++a;
while(a);  /* a = 0 после выхода из цикла */
```

4.4. Инструкция цикла *for*

Инструкция цикла for имеет следующий синтаксис:

```
for (выражение_1; выражение_2; выражение_3)
  инструкция
```

Работает инструкция for следующим образом. Перед началом цикла один раз вычисляется выражение_1. Значение выражения_2 вычисляется каждый раз перед выполнением инструкции. Если это значение равно истинна, то выполняется инструкция, в противном случае цикл заканчивается. После каждой итерации цикла вычисляется выражение_3. При вычислении выражений завершаются все побочные эффекты. Например, следующий цикл находит сумму цифр от 1 до 9:

```
int i, s = 0;
for ( i = 1; i < 10; ++i)
  s += i;   /* s = 45 после выхода из цикла */
```

Замечание

В языке программирования C++ выражение_1 может содержать объявления переменных. В этом случае объявленная переменная видима только в цикле for.

Учитывая это замечание, в языке программирования C++ этот же цикл можно записать следующим образом:

```
int s = 0;
for ( int i = 1; i < 10; ++i)
  s += i;
```

Относительно выражений в инструкции `for` нужно сделать следующее замечание. Как каждое в отдельности, так и все вместе выражения в инструкции `for` могут отсутствовать. Если отсутствует `выражение_2`, то считается, что оно постоянно истинно. Поэтому инструкция:

```
for (;;)
  инструкция
```

описывает бесконечный цикл.

4.5. Инструкция перехода *break*

Инструкция `break` прекращает работу циклов `while`, `do...while` и `for`, а также выполняет выход из блока `switch`. Например:

```
int x = 1, y = 0;
switch(x)
{
case 0:
  --y;
  break;
case 1:
  ++y;
  break;
default:
  y = 10;
}      /* y = 1 */
```

Инструкция `break` чаще всего используется в том случае, если внутри цикла получено такое значение некоторого выражения, которое делает продолжение цикла невозможным. Конечно, лучше вставлять такую проверку в условие продолжения цикла. Но это не всегда возможно, если проверка условия продолжения цикла требует нетривиальных предварительных вычислений.

4.6. Инструкция перехода *continue*

Инструкция `continue` завершает текущую итерацию циклов `while`, `do...while`, `for` и передает управление на вычисление условия

цикла. То есть инструкция `continue` прерывает исполнение тела цикла и передает управление на начало этого цикла.

Например:

```
int x = -10, y = 0;
while (x)
{
  ++x;
  if (!(x + y))
    continue;
  --y;
}       /* x = 0, y = -10 после выхода из цикла */
```

Как и в случае с инструкцией `break`, если это возможно, то инструкцию цикла `continue` лучше не использовать для продолжения цикла.

4.7. Метки инструкций и инструкция перехода *goto*

Метка инструкции состоит из идентификатора, за которым следует символ `:`, и может стоять перед любой инструкцией. Например:

```
err: s = 0;
```

Метка инструкции используется для передачи управления на эту инструкцию при помощи инструкции `goto`, которая имеет следующий вид:

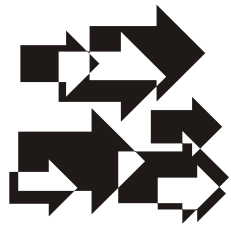
```
goto метка;
```

Областью видимости `метки` является функция, в которой эта `метка` определена. Поэтому передачу управления посредством инструкции `goto` можно выполнять только внутри функции. При этом отметим, что инструкция `goto` чаще всего используется для передачи управления наверх из вложенных управляющих инструкций в случае возникновения ошибки. Иначе, программа считается неструктурной. Ниже показан пример использования инструкции `goto`.

```
  int s = 0;
  if (!s)
    goto err;
  ++s;
err:--s;      /* s = -1 */
```

В настоящее время в программировании инструкция `goto` применяется крайне редко (и не рекомендуется к использованию), а для обработки ошибки во вложенных блоках используется механизм обработки исключений.

ГЛАВА 5



Указатели и массивы

5.1. Указатели

Указатель — это переменная или именованная константа, которая содержит адрес (в памяти) другой переменной, именованной константы или функции. В этом разделе будут рассмотрены только указатели на переменные и именованные константы. Для объявления указателя нужно перед именем переменной, которая объявляется как указатель, поставить символ `*`. В этом случае переменная является указателем на данные, тип которых задан спецификатором типа. Например:

```
int  *ip;     /* указатель на целое число */
char *cp;     /* указатель на символ */
double *dp;   /* указатель на плавающее число */
```

Часто при объявлении указателей символ `*` ставят не перед переменной, а после типа, на который ссылается указатель. Например:

```
int* ip;
```

Но в этом случае следует избегать ошибки, которая возможна, если несколько указателей объявляются в одну строку. Например:

```
int* p1, p2; /* p1 - указатель, p2 - переменная типа int */
```

Чтобы в одной инструкции объявить несколько указателей на один тип, нужно перед каждым указателем ставить символ `*`. Например:

```
int *p1, *p2;  /* p1, p2 - указатели */
```

При объявлении указателя желательно сразу же его проинициализировать, что значительно облегчает отладку программы. Если при объявлении указателя его начальное значение неизвестно, то указателю нужно присвоить значение символической константы NULL, которая равна нулю. Например:

```
int  *p = NULL;
```

Отметим, что нулевые адреса памяти, как правило, используются операционной системой и не распределяются программе. Поэтому если значение указателя равно NULL, то при отладке программы сразу видно, что этому указателю забыли присвоить нужное значение. Если же указатель не инициализировать при объявлении, то в нем будет находиться "неопределенное значение". В этом случае обращение к памяти через такой указатель является ошибкой, которую довольно сложно обнаружить.

Часто для числовых типов данных и указателей используют общее название — *скалярные типы данных*.

5.2. Преобразование типов указателей

В языке программирования C можно преобразовывать типы указателей. Явное и неявное преобразование типов указателей подчиняется следующим правилам:

- указатель на любой тип данных может быть приведен к указателю на любой другой тип данных. Если приведение типов указателей неявное — то компилятор может выдать предупреждение;
- указатель типа void* неявно приводится к указателю на любой тип данных и наоборот, причем эти операции взаимно обратимы;
- указатель может быть преобразован в интегральный тип данных и наоборот, при этом указатель рассматривается как число без знака; если преобразование выполняется неявно, то компилятор может выдать предупреждение;

- выравнивание адресов на границы, кратные длине типа данных, не учитывается, поэтому операция преобразования типов указателей может дать неопределенный результат;
- указатель, значение которого равно `NULL`, преобразовывается в целое число, значение которого равно 0, и наоборот.

В языке программирования C++ приведение типов указателей подчиняется следующим правилам:

- указатель на любой тип данных может быть приведен к указателю на любой другой тип данных только явно;
- указатель типа `void*` можно только явно привести к указателю на любой тип данных; указатель на любой тип данных неявно приводится к указателю типа `void*`;
- указатель может быть преобразован в интегральный тип и наоборот только явно.

Остальные два правила остаются без изменения.

5.3. Операторы определения адреса и обращения по адресу

Для получения адреса переменной или именованной константы используется оператор `&`, который называется *оператором определения адреса* и имеет следующий вид:

```
&операнд
```

Здесь `операндом` должно быть L-value, которое удовлетворяет двум ограничениям:

- не указывает на битовое поле;
- не имеет спецификатор класса памяти `register`.

Например:

```
int n;
int *p = &n;  /* p указывает на переменную n */
```

Получить значение переменной или именованной константы, адрес которой хранится в указателе, можно при помощи *оператора*

обращения по адресу или, другими словами, *оператора разыменования*, который имеет следующий вид:

```
*операнд
```

Здесь `операндом` должен быть указатель. Возвращает оператор обращения по адресу L-value. Например:

```
int n = 2, m;
int *p = &n;
m = *p;     /* m = 2 */
*p = 3;     /* n = 3 */
```

Особенно отметим, что работа оператора обращения по адресу не определена, если указатель содержит ошибочный адрес. В этом случае может произойти как прерывание программы, так и успешное завершение операции разыменования, но полученные данные будут ошибочными. Ошибку, возникшую во втором случае, особенно трудно обнаружить.

Два последовательных оператора получение адреса и обращение по адресу аннулируют друг друга независимо от порядка их следования.

Приоритет операторов определения адреса и обращения по адресу совпадает с приоритетом унарных арифметических и побитовых операторов. Эти операторы ассоциативны справа налево.

5.4. Указатели на константы и константные указатели

При работе с указателями следует различать использование модификатора доступа `const` перед типом переменной, на которую ссылается указатель, и перед самим указателем. В первом случае указатель ссылается на типизированную константу, а во втором — является константным указателем. Например:

```
const int n = 2;
const int *p = NULL;  /* p – указатель на константу */
p = &n;  /* правильно, т. к. p – указатель на константу */
```

Но:

```
const int n = 2;
int *p = NULL;    /* p – указатель на переменную */
p = &n;  /* ошибка, так как p – указатель на переменную */
```

Указатель на константу запрещает косвенное изменение этой константы. Например:

```
const int n = 2;
const int *p = &n;  /* p – указатель на константу */
(*p)++;  /* ошибка, так как p – указатель на константу */
```

Отметим, что указателю на константу может быть присвоен адрес переменной. Например:

```
int n = 2;
const int *p = &n;  /* p – указатель на константу */
p = &n;  /* правильно */
```

Значение константного указателя изменять нельзя. Например:

```
int n = 2;
int* const p = &n;  /* p – константный указатель */
(*p)++;  /* правильно, n = 3 */
p++;     /* ошибка, так как p – константа */
```

Теперь объявим константный указатель на константу. В этом случае нельзя изменить как значение такого указателя, так и значение константы, на которую он ссылается. Например:

```
const int n = 2;
const int* const p =&n; /* конст. указатель на константу */
(*p)++;  /* ошибка, так как n – константа */
p++;     /* ошибка, так как p – константа */
```

5.5. Одномерные массивы

Массив — это последовательность данных одного типа, которые хранятся в непрерывной области памяти друг за другом. Единица данных массива называется *элементом* массива. Доступ к элементам массива производится по их номерам в последовательно-

сти, которые называются *индексами*. Одномерный массив объявляется следующим образом:

```
спецификатор_типа идентификатор[диапазон];
```

Здесь `спецификатор_типа` определяет тип элементов массива, `идентификатор` задает имя массива, а `диапазон` — количество элементов в массиве. При этом имя массива является константным указателем на его первый элемент, а `диапазон` должен быть задан положительным целочисленным литералом или именованной константой. Например:

```
int  a[10];  /* массив из десяти целых чисел */
```

При объявлении массива его можно явно инициализировать, перечислив значения элементов массива в фигурных скобках в порядке возрастания индекса. Например:

```
char s[3] = {'a', 'b', 'c'};
int a[3] = {0, 1, 2};
```

При инициализации массива возможны три ситуации:

- размер массива равен количеству значений в списке инициализации;
- размер массива больше количества значений в списке инициализации;
- размер массива меньше количества значений в списке инициализации.

В первом случае происходит обычная инициализация элементов массива. Во втором случае оставшиеся неинициализированными элементы массива инициализируются нулями. В третьем случае компилятор выдаст сообщение об ошибке.

Если значения элементов массива перечисляются явно, то размерность массива можно не указывать. В этом случае компилятор сам определит размерность по количеству элементов. Например:

```
int a[] = {0, 1, 2};  /* массив из трех элементов */
```

Доступ к элементам массива выполняется при помощи *оператора индексирования*, который имеет следующий вид:

```
имя_массива[индекс]
```

Возвращает оператор индексирования L-value. При этом выход индекса за пределы допустимого диапазона не проверяется. Например:

```
int a[3] = {0, 1, 2};
int n;
n = a[0];
a[0] = 10;
```

Особенно отметим, что нумерация элементов массива начинается с нуля. Приоритет оператора индексирования превышает приоритет унарных арифметических и побитовых операторов и совпадает с приоритетом постфиксных операций инкремента и декремента.

Так как верхний предел индекса массива может использоваться в программе многократно, то его лучше определить как именованную константу. В этом случае легче вносить изменения в программу при изменении размерности массива. Например:

```
const int n = 10;
int a[n];  /* массив из десяти целых чисел */
```

В связи с этим возникает вопрос, какой модификатор знака выбрать для типа индекса массива `unsigned` или `signed`. Так как индексы это неотрицательные целые числа, то можно предположить, что лучше использовать модификатор знака `unsigned`. Но на практике для индекса обычно выбирают модификатор знака `signed`, т. к. в этом случае легче находить ошибки, связанные с неправильным изменением значения индекса.

Так как имя массива фактически представляет собой указатель, то оператор индексирования может применяться к любым указателям. Отсюда следует, что имя массива может также быть операндом оператора разыменования.

В языке программирования C++ имя массива имеет следующий тип:

```
спецификатор_типа[диапазон]
```

Однако этот тип неявно приводится к типу указателя на `спецификатор_типа`.

5.6. Строки

Строкой называется массив символов, который заканчивается пустым символом \0. Для инициализации строки используются строковые литералы. Если длина массива больше чем длина строкового литерала, то оставшиеся свободными байты инициализируются нулями. Например:

```
char s[10] = "string";
```

Если длина строки должна быть равна длине строкового литерала, то размерность массива лучше не указывать, ее подсчитает сам компилятор. Например:

```
char s[] = "string";
```

Пустую строку можно объявить следующим образом:

```
char s[10] = "";
```

В языке программирования C для работы со строками существует большое количество функций, прототипы которых описаны в заголовочном файле и string.h. Работа с этими функциями будет рассмотрена в *гл. 29*.

5.7. Арифметические действия с указателями

Над указателями одного типа можно выполнять арифметическую операцию вычитания -. Результатом этой операции будет целое число, которое указывает, на сколько один адрес памяти смещен относительно другого в единицах, равных длине базового типа данных указателей. Например:

```
int a[5];
int n, m;
n = &a[5] - &a[3];  /* n = 2  */
m = &a[3] - &a[5];  /* m = -2 */
```

К указателю можно прибавить или вычесть целое число. В этом случае значение указателя соответственно увеличится или

уменьшится на это число, умноженное на длину базового типа данных указателя. Например:

```
int a[5];
int *p = a+2;  /* p = &a[2] */
```

К указателю можно применять операции ++ и --. В этом случае значение указателя соответственно увеличится или уменьшится на длину базового типа данных указателя. Например:

```
double* p;
++p;
```

5.8. Многомерные массивы

Массив, элементы которого нумеруются несколькими индексами, называется *многомерным*. Компилятор должен поддерживать многомерные массивы, элементы которых нумеруются, по крайней мере, 12 индексами. Объявляется многомерный массив следующим образом:

```
спецификатор_типа идентификатор[диапазон_1]...[диапазон_n];
```

Тип многомерного массива определяется следующим образом:

```
спецификатор_типа(*)[диапазон_2]...[диапазон_n]
```

Причина такой типизации заключается в том, что при объявлении многомерного массива память резервируется под все элементы массива сразу, а сами элементы массива располагаются по строкам. Такое решение было принято для увеличения производительности программы при работе с многомерными массивами. Например, двумерный целочисленный массив может быть объявлен как:

```
int arr[2][2];
```

В этом случае `arr` является именованной константой, тип которой равен `int(*)[2]`.

При инициализации многомерного массива его элементы заключаются в фигурные скобки. Например:

```
int a[2][2] = {1, 1, 2, 2};
```

Можно использовать вложенные фигурные скобки, которые разбивают список значений элементов массива на строки. Например:

```
int a[2][2] = {{1, 1}, {2, 2}};
```

Если многомерный массив инициализируется при объявлении, то диапазон значений первого индекса можно опустить. В этом случае компилятор сам вычислит диапазон первого индекса в зависимости от количества констант, используемых для инициализации. Например:

```
int a[][2] = {{1, 1}, {2, 2}};
```

При инициализации элементов многомерных массивов действуют те же правила, что и при инициализации элементов одномерных массивов. Но при этом следует отметить, что отсутствующие элементы многомерного массива инициализируются нулями в соответствии с вложенностью фигурных скобок. То есть:

```
int a[2][2] = {1, 1};       /* a[1][0] = 0, a[1][1] = 0 */
```

не эквивалентно:

```
int a[2][2] = {{1}, {2}};  /* a[0][1] = 0, a[1][1] = 0 */
```

Для доступа к элементам многомерного массива используется оператор индексирования `[]`. Причем этот оператор должен применяться столько раз, какова размерность многомерного массива. Например:

```
int a[2][2] = {{1, 2}, {3, 4}};
int n = a[1][1];    /* n = 4 */
```

Если оператор индексирования применяется меньшее число раз, то это в результате дает массив меньшей размерности. Например:

```
int a[2][2][2] = {{{1, 1}, {2, 2}}, {{3, 3}, {4, 4}}};
int *b;
int (*c)[2];
b = a[1][1];  /* b = {4, 4} */
c = a[1];     /* c = {{3, 3}, {4, 4}} */
```

Но операторы присваивания в следующем примере вызовут ошибку, т. к. имя массива является именованной константой и не может принимать нового значения.

```
int a[2][2][2] = {{{1, 1}, {2, 2}}, {{3, 3}, {4, 4}}};
int b[2];
int c[2][2];
b = a[1][1];  /* ошибка, так как b - константа */
c = a[1];     /* ошибка, так как c - константа */
```

В этом случае необходимо поэлементное копирование элементов массива `a` в массивы `b` и `c`.

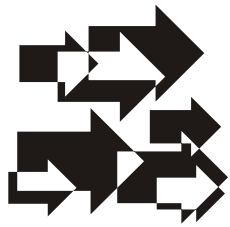
В языке программирования C++ многомерные массивы имеют следующий тип:

```
спецификатор_типа[диапазон_1]...[диапазон_n]
```

Однако такой тип неявно приводится компилятором к типу массива, который используется в языке программирования C, при условии совпадения типа элементов и диапазонов массивов.

Массивы, которые объявляются в программе явно, также называются *встроенными* или *статическими массивами*.

ГЛАВА 6



Функции

6.1. Объявление функции

Функция — это последовательность инструкций, предназначенных для решения определенной задачи. Прежде чем функция может использоваться в программе, она должна быть объявлена. То есть до инструкции с вызовом функции в тексте программы должно находиться *объявление функции*, которое имеет следующий вид:

```
спецификатор_типа имя_функции(список_параметров);
```

и описывает соответственно тип возвращаемого функцией значения, имя функции и список параметров. Тип возвращаемого функцией значения не может быть типом статического массива или именем функции. Параметры функции перечисляются в скобках через запятую. Например, объявление функции, которая вычисляет сумму двух целых чисел, может выглядеть следующим образом:

```
int add(int x, int y);
```

Если функция не возвращает значения, то предполагается, что эта функция возвращает пустое значение. В этом случае для типа возвращаемого значения используется тип `void`. Например:

```
void print(int a);
```

Если тип возвращаемого значения пропущен, то считается, что функция возвращает значение типа `int` (но это очень старое пра-

вило, касающееся языка C и сейчас, как правило, оно не используется). Например, следующая функция:

```
print(int x, int y);
```

возвращает значение типа `int`.

Замечание

В языке программирования C++ определение функции должно содержать спецификатор типа возвращаемого значения. Однако некоторые компиляторы могут не поддерживать это требование.

Если в объявлении функции указаны только типы параметров, то такое объявление функции называется *прототипом функции*. Например, прототип функции `add` имеет следующий вид:

```
int add(int, int);
```

Если функция не имеет аргументов, то в списке параметров этой функции нужно указать слово `void`. Пустой список параметров обозначает, что функция имеет неопределенное количество параметров. То есть в этом случае функцию можно вызвать с любым списком аргументов, которые функция будет извлекать из стека.

Замечание

В языке программирования C++ как пустой список параметров, так и список параметров, содержащий только слово `void`, обозначают, что функция не имеет параметров.

Учитывая замечание, в языке программирования C++ следующая функция:

```
void hello();
```

не имеет параметров и не возвращает никакого значения.

В языке программирования C допускается использование функций с переменным количеством параметров. В этом случае в конце списка параметров ставится многоточие. При этом в списке параметров должен присутствовать хотя бы один параметр. Например, можно объявить следующую функцию:

```
int add_num(int n, ...);  /* n - количество слагаемых */
```

которая, как предполагается, находит сумму произвольного количества целых чисел.

Замечание

В языке программирования C++ допускается опускать символ , (запятая) перед многоточием при объявлении функции с неопределенным количеством параметров. Также в C++ допускается использовать в списке параметров просто многоточие, которое в этом случае указывает на то, что функция имеет неопределенное количество параметров.

Подробно работа с функциями, имеющими неопределенное количество параметров, рассмотрена в *гл. 23*.

6.2. Определение функции

Определение функции имеет следующий вид:

```
спецификатор_типа имя функции(список_параметров)
{
  /* тело функции */
}
```

и состоит из заголовка, который совпадает с объявлением функции, и тела функции, которое заключено в блок. Тело функции содержит определения и инструкции, которые выполняются при вызове функции. В отличие от объявления функции, которое может располагаться в любом месте программы, определение функции должно находиться вне блока. Поэтому обычно функции определяются в начале программы. Если определение функции встречается до ее использования, то оно также является и объявлением этой функции.

Для возврата из функции значения используется инструкция `return`, которая имеет следующий синтаксис:

```
return выражение;
```

После выполнения инструкции `return` исполнение функции прекращается и функция возвращает значение `выражения`. Перед возвратом из функции все побочные эффекты, возникающие при вычислении `выражения`, завершаются. Тип `выражения` должен соответствовать типу возвращаемого функцией значения. В противном случае выполняется неявное приведение типов.

В качестве примера определим функцию, которая возвращает сумму двух чисел.

```
int add(int x, int y) { return x + y; }
```

Если функция не возвращает никакого значения, то для выхода из функции используется инструкция `return` без выражения. Если выход из функции, не возвращающей значения, происходит как выход из блока, содержащего тело функции, то в этом случае инструкцию `return` можно опустить. Например, следующая функция напечатает сумму двух чисел:

```
void print(int x, int y)
{
  printf("x + y = %d\n", x + y);
}
```

Параметры функции можно рассматривать как локальные переменные, которые видны только в теле функции. Поэтому параметр функции не может иметь тип `void`. Однако в качестве параметра допускается указатель на тип `void`. Значения параметров можно изменять в теле функции, если они не имеют квалификатора `const`. Например, можно написать следующую функцию:

```
int inc(int n) { return ++n; }
```

которая возвращает значение `n+1`. Заметим, что функция:

```
int inc(int n) { return n++; }
```

вернет значение `n`.

6.3. Стандартные функции

Язык программирования C содержит библиотеку стандартных функций, которую можно подключить к программе на этапе создания исполняемого файла. Как это делается, зависит от среды разработки. Как правило, библиотека стандартных функций языка программирования C, так же как и стандартная библиотека языка программирования C++, подключается к проекту по умолчанию.

Для того чтобы в программе можно было использовать стандартные функции, нужно объявить их прототипы. Прототипы стандартных функций содержатся в специальных файлах, которые имеют расширение h и называются *заголовочными файлами*. Каждый заголовочный файл содержит объявления только тех функций и переменных, которые предназначены для решения одной группы задач.

Для вставки прототипов функций в текст программы используется директива препроцессора `#include`, которая описана в *разд. 9.5*. Например, вставить в программу прототипы стандартных функций ввода/вывода можно следующим образом:

```
#include <stdio.h>
```

Подробно использование стандартных функций языка программирования C рассмотрено в *части III*. А в *части IV* рассмотрена стандартная библиотека языка программирования C++.

Отметим, что при разработке программ программист может создавать свои заголовочные файлы, куда он помещает объявления своих функций и переменных, используемые в разных исходных файлах своей программы.

6.4. Вызов функции

Функция должна быть объявлена перед своим вызовом. Вызов функции имеет следующий вид:

```
имя_функции(список_аргументов)
```

и называется *оператором вызова функции*. Этот оператор возвращает R-value. `Список_аргументов` содержит операнды оператора вызова функции, которые перечисляются через запятую, могут быть произвольными выражениями и называются *аргументами функции*. Приоритет оператора вызова функции совпадает с приоритетом оператора индексирования и постфиксных операторов инкремента и декремента. Ассоциативность этого оператора — слева направо.

Аргументы функции используются для инициализации параметров функции, во время которой может выполняться неявное приведение типов. Порядок вычисления аргументов функции не оп-

ределен. Однако до передачи управления в тело функции все побочные эффекты, возникающие при вычислении аргументов функции, завершаются. Особенно отметим, что аргументы передаются в функцию по значению. То есть значения аргументов переписываются в параметры функции. Так как значение аргумента копируется в тело функции, то сам аргумент изменить в теле функции невозможно. После инициализации параметров функции происходит выполнение тела функции. В листинге 6.1 приведена программа, в которой вызывается функция сложения двух чисел.

Листинг 6.1. Примеры вызова функции

```
#include <stdio.h>
int add(int x, int  y) { return x + y; }
int main(void)
{
  int z;

  z = add(1, 2);            /* вызываем функцию   */
  printf("z = %d\n", z);    /* печатаем результат */

  /* можно вызвать функцию и так */
  printf("1 + 2 = %d\n", add(1, 2));
  /* можно также и так, но в этом случае непонятно зачем */
  add(1, 2);

  return 0;
}
```

Для того чтобы функция могла изменить значения переменных в вызывающей ее функции, она должна получить в качестве аргументов указатели на эти переменные. В листинге 6.2 приведена функция `swap`, выполняющая обмен значений, на которые указывают аргументы.

Листинг 6.2. Передача указателей в качестве аргументов функции

```
#include <stdio.h>
void swap(int* x, int* y)
```

```
{
  int t;
  t = *x;
  *x = *y;
  *y = t;
}
int main(void)
{
  int x = 1, y = 2;

  swap(&x, &y);
  printf("x = %d y = %d\n", x, y);  /* x = 2, y = 1 */

  return 0;
}
```

6.5. Передача массивов в функции

Имя статического массива это константный указатель на его первый элемент. Поэтому массивы передаются в функции через указатели. Например, следующая функция увеличивает значение элементов массива на единицу.

```
void  f(int a[3])
{
  int i;
  for (i = 0; i < 3; ++i)
    ++a[i];
}
```

Здесь диапазон индекса массива в параметре функции не имеет значения, т. к. все равно в функцию передается только указатель. Поэтому возможны также следующие способы объявления массива как параметра функции:

```
  int f(int a[]);
```

или:

```
  int f(int* const a);
```

Но в этом случае диапазон индекса лучше указать, т. к. он явно указывает границы диапазона. Дело обстоит сложнее, если функ-

ция должна работать с массивами, диапазон индекса которых не фиксирован. Так как указатель на массив не содержит информации о диапазоне индекса, то его нужно передавать в качестве второго параметра такой функции. Например, следующая функция увеличивает на единицу значение каждого элемента произвольного одномерного целочисленного массива:

```
void f(int a[], int n)
{
  int i;
  for (i = 0; i < n; ++i)
    ++a[i];
}
```

Вызов этой функции будет выглядеть следующим образом:

```
  int a[5] = {1, 2, 3, 4, 5};
  f(a, 5);
```

При передаче многомерного массива как параметра функции нужно указать его полный тип, при этом разрешается опустить диапазон первого индекса. Значения диапазонов остальных индексов многомерного массива необходимо знать компилятору для того, чтобы он мог правильно вычислить смещение к элементам массива. Например, следующая функция увеличивает на единицу значение каждого элемента двумерного массива:

```
void f(int a[2][2])
{
  int i, j;
  for (i = 0; i < 2; ++i)
    for (j = 0; j < 2; ++j)
      ++a[i][j];
}
```

Для передачи произвольной многомерной статической матрицы в функцию необходимо в качестве параметра передать адрес ее первого элемента. Например, следующая функция увеличивает на единицу значение каждого элемента произвольной матрицы второго порядка.

```
void  f(int* const a, int n, int m)
{
  int i;
  for (i = 0; i < n * m; ++i)
    ++a[i];
}
```

В этом случае используется тот факт, что элементы статических массивов хранятся в памяти последовательно. Вызов этой функции будет выглядеть следующим образом:

```
int a[2][2] = {1, 2, 3, 4};
f(&a[0][0], 2, 2);
```

6.6. Указатели на функции (функторы)

Язык программирования C позволяет определять указатели на функции, которые имеют следующий вид:

```
спецификатор_типа (*имя_указателя)(список_параметров)
```

Например, указатель на функцию, которая возвращает значение типа `int` и имеет два параметра типа `int`, может быть объявлен как:

```
int (*fp)(int x, int y);
```

Указателю на функцию можно присвоить адреса различных функций, которые имеют такие же типы возвращаемого значения и параметров. Например, если определена функция:

```
int add(int x, int y) { return x + y; }
```

То ее адрес можно присвоить указателю `fp` следующим образом:

```
fp = &add;
```

Учитывая, что имя функции является адресом этой функции в памяти, указателю на функцию можно просто присвоить имя функции. То есть:

```
fp = add;
```

Так как указателю на функцию могут присваиваться адреса различных функций при условии совпадения типа параметров и типа возвращаемого значения, то указатель на функцию часто называют *функтором*.

Вызвать функцию через указатель можно двумя способами:

- (*имя_указателя)(список_аргументов);
- имя_указателя(список_аргументов).

Представляется, что первый способ более наглядный, т. к. явно указывает, что функция вызывается через указатель.

В листинге 6.3 приведена программа, в которой функции сложения и вычитания двух чисел вызываются через указатель.

Листинг 6.3. Вызов функций через указатель

```
#include <stdio.h>
int add(int x, int y) { return x + y; }
int sub(int x, int y) { return x - y; }
int main(void)
{
  int (*p)(int, int);
  p = add;
  printf("1 + 2 = %d\n", (*p)(1, 2));
  p = sub;
  printf("1 - 2 = %d\n", p(1, 2));
  return 0;
}
```

Тип указателя на функцию может также использоваться как тип возвращаемого функцией значения и тип параметра функции. Например, в листинге 6.4 приведена функция `foo`, которая получает в качестве параметра указатель на функцию и просто возвращает его.

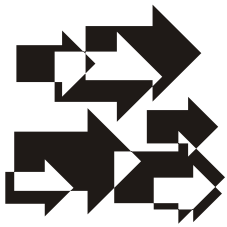
Листинг 6.4. Использование типа указателя на функцию как типа возвращаемого значения и типа параметра функции

```
#include <stdio.h>
int (*foo(int (*p)(int)))(int) { return p; }
```

```
int inc(int x) { return ++x; }
int dec(int x) { return --x; }
int main(void)
{
  int (*f) (int);     /* указатель на функцию */

  f = foo(inc);       /* устанавливаем указатель на inc */
  printf("2 + 1 = %d\n", (*f)(2));

  f = foo(dec);       /* устанавливаем указатель на dec */
  printf("2 - 1 = %d\n", (*f)(2));

  return 0;
}
```

В заключение этого раздела отметим, что над указателями на функции нельзя выполнять никаких арифметических операторов.

ГЛАВА 7



Структура программы на языке С

7.1. Область видимости идентификатора

Если переменная объявлена вне какого-либо блока, то такая переменная называется *глобальной*. В противном случае переменная называется *локальной*.

Областью видимости идентификатора называется наибольшая область программы, в которой можно сослаться на этот идентификатор. Существуют четыре типа областей видимости:

- блок;
- функция;
- прототип функции;
- исходный файл.

Глобальные переменные видимы внутри исходного файла, в котором они определены.

Локальные переменные видимы только внутри блока или функции, в которой они определены. Локальная переменная скрывает любую переменную с тем же именем, объявленную вне этого блока.

В листинге 7.1 приведен пример, демонстрирующий сокрытие имен переменных.

Листинг 7.1. Сокрытие имен переменных

```
#include  <stdio.h>
int n = 1;
int main(void)
{
  printf("n = %d\n", n);       /* n = 1 */
  {
    int n = 2;
    printf("n = %d\n", n);     /* n = 2 */
    {
      n = 3;
      printf("n = %d\n", n);   /* n = 3 */
    }
  }
  return 0;
}
```

Параметры функции видны только внутри функции или объявления этой функции.

Функции всегда определяются вне какого-либо блока. Областью видимости функции является исходный файл, в котором эта функция определена.

7.2. Время существования переменных и функций

Время существования переменной или *функции* определяется как время, в течение которого эта переменная или функция хранится в памяти компьютера.

Глобальные переменные существуют в течение всего времени выполнения программы и хранятся в фиксированной области памяти программы, которая задается во время ее компиляции. Если глобальная переменная не проинициализирована, то компилятор устанавливает ее значение в ноль.

Память для локальных переменных выделяется динамически во время исполнения программы. При вызове функции или входе в

блок, в котором объявлены локальные переменные, память под эти переменные распределяется, а при возврате из функции или выходе из блока эта память освобождается. Поэтому данные, содержащиеся в локальных переменных, недоступны при последующих входах в блок или вызовах функции. Если локальная переменная не проинициализирована при ее объявлении, то ее начальное значение не определено.

Функции существуют в течение всего времени исполнения программы.

7.3. Связывание идентификаторов

Программа на языке программирования С состоит из одного или нескольких исходных файлов, которые содержат код программы. Эти файлы также называются *единицами компиляции*, т. к. каждый из них является исходным данным для компилятора. Результатом компиляции исходного файла является объектный файл, который представляет собой программу в кодах микропроцессора, эквивалентную по смыслу исходной программе на языке программирования С. Если программа на языке программирования С состоит из нескольких исходных файлов, то в результате ее компиляции получается такое же количество объектных файлов. Все эти объектные файлы нужно собрать в один исполняемый модуль. Эту задачу решает редактор связей (linker), который также называется компоновщиком.

При разработке программы, состоящей из нескольких исходных файлов, может возникнуть следующая проблема. В одном исходном файле нужно сослаться на идентификатор переменной или функции, которая определена в другом исходном файле. Для решения этой проблемы между одинаковыми идентификаторами в разных исходных файлах программы устанавливается связь, которая называется *связыванием* или *сцеплением* (linkage) *переменных*. Связывание реализуется при помощи назначения идентификатору типа связывания, который определяет видимость идентификатора в других исходных файлах. Определены три типа связывания идентификаторов:

- внутреннее связывание, т. е. на идентификатор можно ссылаться только внутри исходного файла, в котором он определен;
- внешнее связывание, т. е. на идентификатор можно ссылаться как внутри исходного файла, в котором он определен, так и в других исходных файлах;
- отсутствие связывания, т. е. на идентификатор можно ссылаться только внутри блока, в котором он определен.

Глобальные переменные и функции имеют внешнее связывание, т. е. на них можно ссылаться в других исходных файлах. Локальные переменные не имеют связывания.

Так как глобальная переменная имеет внешнее связывание, то глобальная переменная с одним и тем же именем может быть объявлена в программе только один раз. Если глобальные переменные с одним именем будут объявлены в разных исходных файлах программы, то компилятор эти объявления пропустит, а компоновщик выдаст ошибку, указывающую на повторение имени переменной.

В связи с введением понятия связывания идентификаторов нужно различать объявление и определение переменной. Если объявление переменной вызывает распределение памяти для этой переменной, то такое объявление называется *определением переменной*.

7.4. Спецификаторы классов памяти

В языке программирования C для управления временем существования переменных и связыванием идентификаторов используются специальные ключевые слова, которые называются *спецификаторами классов памяти*. Отметим, что спецификаторы классов памяти не изменяют области видимости идентификатора. Существуют четыре спецификатора классов памяти: `extern`, `auto`, `static` и `register`. Используются спецификаторы классов памяти при объявлении переменных и функций и записываются перед идентификатором или спецификатором типа переменной или функции.

Спецификатор *extern*

Спецификатор `extern` управляет связыванием идентификаторов и указывает компилятору, что переменная или функция уже определена вне области видимости своего объявления в том же или другом исходном файле.

Если спецификатор `extern` используется при объявлении локальной переменной, то в этом случае он указывает компилятору, что эта переменная ссылается на глобальную переменную с тем же именем, определенную в том же или другом исходном файле. Этот случай рассмотрен в листинге 7.2.

Листинг 7.2. Использование спецификатора `extern` с локальной переменной

```
#include <stdio.h>
int n = 1;
int main(void)
{
  int n = 2;
  {
    extern int n;
    printf("n = %d\n", n); /* печатает n = 1 */
  }
  return 0;
}
```

Если спецификатор `extern` используется с глобальной переменной, то это указывает компилятору, что эта переменная ссылается на глобальную переменную с тем же именем, но которая определена в другом исходном файле. Этот случай рассмотрен в листингах 7.3 и 7.4, которые описывают исходные файлы, принадлежащие одной программе.

Листинг 7.3. Исходный файл с определением глобальной переменной и прототипом функции

```
#include <stdio.h>
int n;
```

```
void foo(void);
int main(void)
{
  foo();
  printf("n = %d\n", n); /* печатает n = 1 */

  return 0;
}
```

Листинг 7.4. Исходный файл с объявлением глобальной переменной и определением функции

```
extern int n;
void foo(void) { ++n; }
```

Спецификатор `extern` также может использоваться при объявлении функции. Это указывает компилятору, что функция определена в другом исходном файле. Однако объявление функции имеет этот спецификатор по умолчанию и поэтому он редко используется в этом случае. В листинге 7.4 определена функция `foo`, которая вызывается в листинге 7.3. Поэтому в листинге 7.3 эта функция только объявляется перед своим использованием.

Спецификатор *auto*

Спецификатор `auto` управляет временем существования локальной переменной и указывает компилятору, что переменная, определенная внутри блока, существует только во время исполнения этого блока. Локальные переменные имеют этот спецификатор по умолчанию, поэтому он редко используется при объявлении таких переменных.

Спецификатор *static*

Спецификатор `static` управляет временем существования локальных переменных и связыванием идентификаторов глобальных переменных и функций.

Если локальная переменная определена со спецификатором `static`, то она существует в течение всего времени исполнения программы. Если такая переменная не инициализируется при

своем определении, то компилятор устанавливает начальное значение этой переменной в ноль.

В листинге 7.5 приведена программа, в которой показано как изменяется значение статической локальной переменной.

Листинг 7.5. Изменение значения статической локальной переменной

```
#include  <stdio.h>
int count(void)
{
  static int c;

  return ++c;
}
int main(void)
{
  count();
  printf("count = %d\n", count());  /* печатает count = 2 */

  return 0;
}
```

Для глобальной переменной или функции спецификатор `static` ограничивает ее связывание только исходным файлом, в котором эта переменная или функция определена. То есть к этой переменной или функции нельзя обратиться из другого исходного файла.

Например, если в листингах 7.3 и 7.4 добавить спецификатор `static` в определения глобальной переменной `n` и функции `foo`, то на них нельзя будет сослаться в других исходных файлах. В этом случае программы успешно пройдут компиляцию, однако редактирование связей вызовет ошибку.

Спецификатор *register*

Спецификатор `register` указывает компилятору, что значение переменной нужно хранить в регистре микропроцессора. Этот спецификатор может применяться только к локальным переменным и параметрам функций. Компилятор не обязан исполнять требования спецификатора `register` и может хранить значения

переменной в оперативной памяти. Поэтому спецификатор `register` редко используется на практике.

В листинге 7.6 приведен пример использования спецификатора `register`.

Листинг 7.6. Пример использования спецификатора `register`

```
#include <stdio.h>
int n;
int main(void)
{
  register int i;

  for (i = 0; i < 5; ++i)
    ++n;
  printf("n = %d\n", n); /* печатает n = 5 */

  return 0;
}
```

7.5. Структура программы

Программа на языке программирования C состоит из определений переменных и функций, которые размещаются в одном или нескольких текстовых файлах. Как уже говорилось в *разд. 7.3*, такие файлы называются *исходными файлами* или *единицами компиляции*. Если в исходном файле используются функции или переменные из других файлов, то они должны быть объявлены перед своим использованием. Для объявления функций используются их прототипы, а для объявления переменных — ключевое слово `extern`.

Подготовка исполняемого файла программы осуществляется в два этапа. Сначала компилируются все исходные файлы программы. Результатом компиляции исходных файлов являются *объектные файлы*, которые содержат исполняемый код программы, но в которых не разрешены ссылки на переменные и функции, определенные в других объектных файлах. Если компиляция

прошла успешно, то можно создать исполняемый модуль. Для этого вызывается программа, которая называется *компоновщик* или *редактор связей* (linker). Эта программа разрешает внешние ссылки и создает исполняемый файл.

Исполняемая программа на языке С имеет структуру, которая показана на рис. 7.1. То есть исполняемая программа представляет собой непрерывный блок памяти, который состоит из четырех частей:

- программный код;
- глобальные переменные;
- куча;
- стек.

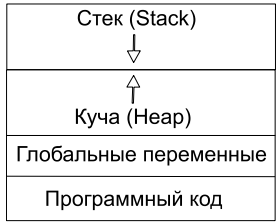


Рис. 7.1. Структура программы

Стек предназначен для хранения адресов возврата из функций, значения аргументов функций и локальных переменных. Куча предназначена для динамического выделения памяти программе. Размеры стека и кучи задаются при вызове программы. Значения этих величин, заданные по умолчанию, зависят от операционной системы. Также отметим, что стек и куча растут навстречу друг другу.

7.6. Функция *main*

Каждая программа на языке С должна содержать одну функцию с именем `main`. При вызове программы средой исполнения или операционной системой управление передается этой функции. Функция `main` имеет один из следующих прототипов:

```
int main(void);
int main(int argc, char **argv, char **envp);
int main();
```

Во втором случае операционная система передает функции `main` три параметра `argc`, `argv` и `envp`. Первые два параметра являются обязательными, а третий параметр может отсутствовать. Параметры функции `main` имеют следующее назначение:

- `argc` — определяет количество элементов в массиве `argv`;
- `argv` — указатель на массив указателей на строки, которые передаются программе при ее вызове;
- `envp` — указатель на массив указателей на строки, которые описывают среду исполнения программы, последняя строка в массиве имеет нулевую длину.

Если в среде исполнения доступно имя программы, то указатель на строку с именем программы хранится первым в массиве указателей, на который указывает параметр `argv`. Если имя программы недоступно, то первый элемент этого массива указывает на пустую строку.

Параметры исполняемой программе передаются при ее запуске. Способ запуска программы зависит от операционной системы. Например, при запуске исполняемого файла из командной строки аргументы функции `main` записываются после имени программы.

В листинге 7.7 приведен текст функции `main`, которая распечатывает свои параметры.

Листинг 7.7. Печать параметров функции `main`

```
#include <stdio.h>
int main(int argc, char *argv[], char *argp[])
{
  int i;

  printf("Program name: %s\n", argv[0]);
  for (i = 1; i < argc; ++i)
    printf("argument %d: %s\n", i, argv[i]);
```

```
  printf("Environment:\n");
  while(*argp)
    puts(*argp++);

  return 0;
}
```

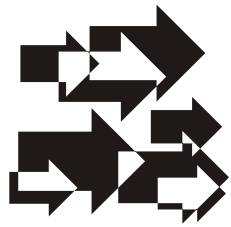
В языке программирования C спецификатор класса памяти `static` при определении функции `main` игнорируется.

В языке программирования C++ на функцию `main` накладываются следующие ограничения:

- нельзя получить адрес функции `main`;
- нельзя вызвать функцию `main` внутри программы;
- нельзя объявить `main`, как статическую функцию.
- нельзя объявить функцию `main` как встроенную;
- функцию `main` нельзя перегрузить.

По соглашению в случае успешного завершения функция `main` должна вернуть нулевое значение, а в противном случае — код ошибки.

ГЛАВА 8



Типы данных, определяемые программистом

8.1. Введение

При разработке программ могут потребоваться типы данных, отличные от типов данных, встроенных в язык программирования C. Введение новых типов данных преследует три цели:

- ограничить множество значений существующего интегрального типа данных;
- определить тип, который описывает данные, имеющие сложную структуру;
- переименовать тип данных, чтобы придать ему более осмысленное имя.

В первом случае используются перечисления, которые содержат только необходимые элементы из базового интегрального типа.

Во втором случае используются структурные или составные типы данных. В языке программирования C программист может конструировать структурные типы данных, используя такие конструкции, как *структуры* и *объединения*.

В третьем случае используется определение `typedef`.

Все эти подходы к конструированию и определению новых типов данных будут рассмотрены в этой главе.

Объявляются новые типы данных так же, как и переменные вне блока или в начале блока.

Замечание

В языке программирования C++ новые типы данных могут объявляться в любом месте программы.

Область видимости типов данных, определяемых программистом, подчиняется тем же правилам, что и имена переменных.

В заключение этого введения скажем, что так новые типы данных определяются программистом, то они часто называются *пользовательскими* типами данных, где под пользователем подразумевается программист.

8.2. Перечисления

Перечислением называется тип данных, который включает набор именованных целочисленных констант. Элементы перечисления называются *членами перечисления* или *перечислимыми* константами. Объявляются перечисления следующим образом:

```
enum [имя_перечисления] {список_значений};
```

Здесь `enum` — ключевое слово, `имя_перечисления` служит для именования типа перечисления и называется *тегом перечисления*, а `список_значений` перечисляет именованные целочисленные константы, принадлежащие перечислению. При объявлении перечисления его элементы могут инициализироваться. Например, перечисление может быть объявлено как:

```
enum color {r = 2, g = 4, b = 6};  /* объявление типа */
```

По умолчанию перечислимым константам присваиваются последовательные значения, начиная с нуля. Например:

```
enum color {r, g, b};  /* r = 0, g = 1, b = 2 */
```

Если, начиная с некоторой перечислимой константы, инициализация прекращается, то следующей перечислимой константе присваивается значение на единицу большее, чем значение предыдущей перечислимой константы. Например:

```
enum color {r, g = 3, b};  /* r = 0, g = 3, b = 4 */
```

Допускается объявление перечислений, не имеющих имени. Такие перечисления называются *анонимными*. Например:

```
enum {off, on};  /* анонимное перечисление */
```

Анонимные перечисления используются просто для определения именованных целочисленных констант.

Тип перечисления определяется ключевым словом `enum` и тегом перечисления. Поэтому переменная типа "перечисление" объявляется следующим образом:

```
enum имя_перечисления  имя_переменной;
```

Например:

```
enum color c;  /* c – переменная типа enum color */
```

Замечание

В языке программирования C++ тип перечисления может определяться только именем перечисления.

Учитывая это замечание, в языке программирования C++ переменную типа "перечисление" можно объявить следующим образом:

```
color d;  // d – переменная типа color
```

Объявление типа перечисления и переменной, которая имеет этот тип, может быть объединено в одну инструкцию. Например:

```
enum light {off, on} t;  /* t – имя переменной */
```

Допускается объявление переменных, которые имеют тип анонимного перечисления. Например:

```
enum {off, on} t;
```

Переменной типа перечисления можно присваивать только значения, принадлежащие ее типу. Эти значения могут присваиваться переменной, используя как имена перечислимых констант, так и целочисленные значения. Например:

```
color c = g;
color c = 2;
```

Замечание

В языке программирования C++ переменным типа перечисления можно присваивать значения, используя только имена перечислимых констант.

Учитывая это замечание, следующая инструкция:

```
color c = 0;
```

вызовет ошибку в языке программирования C++.

Целочисленным переменным можно присваивать значения перечислимых констант. Например:

```
int  e = r;
```

8.3. Структуры

Структурой называется упорядоченное множество данных, которые называются *полями* или *членами структуры*. Поля структуры могут иметь любой тип, исключая типы `void` и функцию. При этом следует учитывать, что структура может содержать только такие поля, длина которых известна компилятору в момент определения структуры. *Структурным типом данных* или просто *структурой* называется тип, описывающий структуру. Объявляется структурный тип следующим образом:

```
struct [имя_структуры]
{
  /* объявления членов структуры */
};
```

Здесь `struct` — ключевое слово, а `имя_структуры` служит для именования структурного типа и называется *тегом структуры*, а члены структуры перечисляются в блоке. Особенно следует обратить внимание на то, что объявление структуры должно заканчиваться символом `;`. Например:

```
struct emp
{
  int empno;
```

```
  char name[20];
  double salary;
};
```

Допускается объявление структур, не имеющих имени. Такие структуры называются *анонимными*. Например:

```
struct {double x, y;};  /* анонимная структура */
```

Тип структуры определяется ключевым словом `struct` и тегом структуры. Поэтому переменная структурного типа объявляется следующим образом:

```
struct имя_структуры  имя_переменной;
```

Например:

```
struct emp e;  /* e – переменная типа struct emp */
```

Замечание

В языке программирования C++ тип структуры также определяется только именем структуры.

Учитывая это замечание, в языке программирования C++ переменную структурного типа можно также объявить следующим образом:

```
emp d;  // d – переменная типа emp
```

Объявление структурного типа и переменной, которая имеет этот тип, может быть объединено в одну инструкцию. Например:

```
struct emp
{
  int empno;
  char name[20];
  double salary;
} director;  /* director – имя переменной */
```

Допускается объявление переменных, которые имеют тип анонимной структуры. Например:

```
struct {double x, y;} vect;
```

Обычно переменные структурного типа также называются *структурами*.

Инициализируются структуры так же, как и массивы. Например:

```
struct emp a = {10, "Paul", 2000};
```

Так как члены структуры описываются в блоке, то их имена принадлежат локальной области видимости внутри этого блока. Для доступа к элементу структуры используется *оператор . (точка)*, который имеет следующий вид:

```
переменная.имя_поля
```

где переменная должна иметь структурный тип. Например:

```
director.empno = 20;
strcpy(director.name, "John");  /* копирование строки */
director.salary = 3000;
```

Здесь `strcpy` стандартная функция для копирования строк, которая описана в *разд. 29.2*.

Оператор . (точка) возвращает L-value, имеет ассоциативность слева направо и его приоритет совпадает с приоритетом постфиксных операций инкремента и декремента.

Элементами структуры могут быть массивы и другие структуры. Отсюда следует, что допускается вложенность структур. Для доступа к членам вложенной структуры оператор . (точка) используется столько раз, какова вложенность структуры. Область видимости имени вложенной структуры совпадает с областью видимости внешней структуры.

Замечание

В языке программирования C++ область видимости вложенной структуры ограничивается блоком, в котором эта структура объявлена.

В листинге 8.1 показан пример обращения к членам вложенной структуры.

Листинг 8.1. Пример вложенных структур

```
#include <stdio.h>
struct demo
```

```
{
  int a;
  struct inside
  {
    int b;
    int c;
  } b;
};
int main(void)
{
  struct demo c = {1, {2, 3}};
  printf("%d %d %d\n", c.a, c.b.b, c.b.c);  /* 1 2 3 */
  return 0;
}
```

Так как компилятору должна быть известна длина поля структуры, то в структуре нельзя объявлять член, тип которого совпадает с типом объявляемой структуры. Однако возможно объявление члена структуры, тип которого является указателем на тип объявляемой структуры. Например:

```
  struct node    /* вершина двоичного дерева */
{
  struct node *left;   /* левый потомок */
  struct node *right;  /* правый потомок */
};
```

В языке программирования C допускается в качестве последнего члена структуры объявлять массив неопределенной длины, если этот массив не является единственным членом структуры. Такие структуры не могут быть вложены в другую структуру, а также быть элементами массива. Например:

```
  struct demo
  {
    double d;
    int a[];
  };
```

При объявлении переменной такого типа память под последний элемент не резервируется. Используются такие структуры для

обработки структур, последними членами которых являются массивы различной длины.

Структуры одного типа можно присваивать друг другу. В этом случае оператор присваивания выполняет почленное копирование структур. Например:

```
struct emp boss;
boss = director;
```

Поля структуры располагаются в памяти последовательно друг за другом. При этом начальный адрес каждого поля структуры удовлетворяет условию:

```
адрес_поля_структуры % длина_поля_структуры == 0
```

Если поле структуры является массивом, то в этом выражении в качестве длины поля структуры берут длину элемента массива. В этом случае говорят, что адреса полей структуры выровнены по границам соответствующего типа данных. Начальный адрес самой структуры выровнен на границу, кратную наибольшей из длин полей структуры. Такое выравнивание необходимо для правильной работы микропроцессора. Отсюда следует, что в структуре могут быть пропущенные байты с неопределенным содержанием. Поэтому работать с полями структуры нужно, используя их имена.

Отметим, что выравнивание адресов структуры зависит от микропроцессора, на котором будет работать программа, и поэтому может отличаться от рассмотренного выше подхода. Обычно каждый компилятор содержит директиву для управления выравниванием адресов данных в зависимости от их типа.

8.4. Объединения

Объединением называется область памяти, используемая для хранения данных разных типов. Причем одновременно в объединении могут храниться данные только одного определенного типа. Данные, входящие в объединение, называются *членами объединения*. Как и в случае структуры, члены объединения могут иметь любой тип, исключая типы `void` и функцию, при условии, что из-

вестна длина этого типа данных. Тип, описывающий объединение, также называется *объединением*. Объявляются объединения следующим образом:

```
union имя_объединения
{
  /* объявления членов объединения */
};
```

Здесь `union` — ключевое слово, а `имя_объединения` служит для именования типа объединения и называется *тегом объединения*, а члены объединения перечисляются в блоке. Объявление объединения должно заканчиваться символом `;`. Например:

```
union num
{
  int n;
  double f;
};
```

Не допускается объявление *анонимных* объединений.

Тип объединения определяется ключевым словом `union` и тегом объединения. Поэтому переменная типа объединения объявляется как:

```
union имя_объединения имя_переменной;
```

Например:

```
union num d;  /* d – переменная типа union num */
```

Замечание

В языке программирования C++ тип объединения также определяется только именем объединения.

Учитывая это замечание, в языке программирования C++ переменную типа объединения можно также объявить следующим образом:

```
num d;  /* d – переменная типа num */
```

Объявление типа объединения и переменной, которая имеет этот тип, может быть объединено в одну инструкцию.

Например:

```
union num
{
  int n;
  double f;
} d;  /* d – переменная типа union num */
```

Допускается объявление переменных, которые имеют тип анонимного объединения. Например:

```
union
{
  int n;
  double f;
} d;  /* d – переменная типа анонимного объединения */
```

При объявлении переменной типа объединения ее можно инициализировать значением, которое имеет тип первого члена объединения. Например:

```
union num d = {10};
```

Обычно переменные, типом которых является объединение, также называются *объединениями*.

Как и в случае со структурами, для доступа к члену объединения используется оператор . (точка). Например:

```
union num e;
e.n = 1;
e.f = 1.1;
```

Объединения одного типа можно присваивать друг другу. В этом случае оператор присваивания выполняет почленное копирование объединений. Например:

```
union num g;
g = e;
```

Длина памяти, распределяемой компилятором под объединения, равна наибольшей из длин членов объединения. Адрес этой памяти выравнивается на границу, кратную наибольшей из длин членов объединения.

8.5. Битовые поля

Битовым полем называется элемент структуры или объединения, который определяет последовательность бит и описывается следующим образом:

```
тип [имя_поля] : длина_в_битах;
```

где `тип` может принимать одно из следующих значений: `int`, `unsigned` или `signed`, а `длина_в_битах` должна быть неотрицательным целым числом. Причем эта длина не должна превышать длины базового типа данных битового поля. Например:

```
struct demo
{
  int count;
  unsigned b1 : 1;
  unsigned b2 : 1;
  double d;
};
```

Если длина битового поля равна нулю, то адрес следующего поля структуры будет выровнен на границу типа `int`.

Запрещается использовать спецификаторы доступа `const` и `volatile` при определении битового поля.

Инициализируются битовые поля, как и обычные члены структуры, например:

```
struct demo d = {10, 1, 0, 10.5};
```

Используются битовые поля для установки различных флагов. Если в программе нужно обрабатывать только некоторые биты из поля, то остальные биты можно не именовать. В этом случае содержимое неименованных битов не определено. Например:

```
struct  demo
{
  unsigned b1: 1;
  unsigned   : 2;
  unsigned b2: 1;
};
```

Доступ к элементам битового поля осуществляется так же, как и доступ к обычным членам структуры при помощи оператора . (точка).

Битовые поля могут использоваться в выражениях точно так же, как и переменные базового типа битового поля. Однако к битовому полю не может применяться оператор определения адреса.

В листинге 8.2 приведен пример работы с битовыми полями.

Листинг 8.2. Работа с битовыми полями

```
#include <stdio.h>
struct demo
{
  int count;
  unsigned b1 : 1;
  unsigned b2 : 1;
  double d;
};
int main(void)
{
  struct demo d;
  unsigned n = 10;
  d.b1 = 1;
  d.b2 = 0;
  printf("d.b1 = %u\n", d.b1); /* d.b1 = 1 */
  printf("d.b2 = %u\n", d.b2); /* d.b2 = 0 */
  n += d.b1;
  printf("n = %u\n", n); /* n = 11 */
  return 0;
}
```

8.6. Передача структур в функции

Когда структура используется как параметр функции, то она передается в функцию по значению, как и принято в языке программирования C. Например, в листинге 8.3 приведена функция, которая распечатывает поля структуры.

Листинг 8.3. Передача структуры в функцию по значению

```
#include <stdio.h>
struct emp
{
  int empno;
  char name[20];
  double salary;
};
void print(struct emp s)
{
  printf("%d %s %f\n", s.empno, s.name, s.salary);
}
int main(void)
{
  struct emp e = {10, "Tim", 1000};

  print(e);  /* печать значений членов структуры */

  return 0;
}
```

Если необходимо, чтобы функция изменяла значение членов структуры, то в функцию передается указатель на структуру. В этом случае для доступа к членам структуры используется оператор ->, который имеет следующий вид:

```
указатель->имя_поля
```

Здесь `указатель` должен указывать на структуру. Этот оператор возвращает L-value, имеет ассоциативность слева направо и его приоритет совпадает с приоритетом постфиксных операций инкремента и декремента.

В листинге 8.4 приведен пример функции, которая инициализирует члены структуры через указатель.

Листинг 8.4. Доступ к членам структуры через указатель

```
#include <stdio.h>
#include <string.h>
struct emp
```

```
{
  int empno;
  char name[20];
  double  salary;
};
void init_emp(struct emp *ps, int en, char *nm, double sal)
{
  ps->empno = en;
  strcpy(ps->name, nm);
  ps->salary = sal;
}
int main(void)
{
  struct emp e;

  init_emp(&e, 10, "Tim", 1000);
  printf("%d %s %f\n", e.empno, e.name, e.salary);

  return 0;
}
```

Аналогично передаются в функции и объединения. Например, в листинге 8.5 приведена функция, которая выводит на консоль содержимое объединения, получая указатель на это объединение.

Листинг 8.5. Доступ к членам объединения через указатель

```
#include <stdio.h>
union num
{
  int n;
  double f;
};
void print(char c, union num *n)
{
  switch(c)
  {
  case 'i':
    printf("n = %d\n", n->n);
    break;
```

```
  case'f':
    printf("f = %f\n", n->f);
    break;
  default:
    printf("Unknown type.\n");
  }
}
int main(void)
{
  union num n;

  n.n = 10;
  print('i', &n);

  n.f = 10.5;
  print('f', &n);

  return 0;
}
```

8.7. Объявление *typedef*

Для определения другого имени для существующего типа данных используется объявление `typedef`, которое имеет следующий вид:

```
typedef имя_типа список_идентификаторов;
```

Здесь `typedef` — ключевое слово, `имя_типа` является именем существующего типа данных, а `список_идентификаторов` содержит новые имена существующего типа данных, которые перечисляются через запятую и называются *синонимами* имени существующего типа данных. Как уже говорилось в *разд. 8.1*, синонимы используются для придания именам типов более осмысленных названий или для их сокращения.

Можно определять синонимы как для встроенных типов данных, так и для типов данных, объявляемых программистом. Например:

```
typedef double salary;
typedef unsigned int order;
```

Теперь эти типы можно использовать для объявления переменных:

```
salary s = 1000;
order ord = 5;
```

Можно определять синонимы для массивов. Например:

```
typedef int arr[10];
arr a;
```

Здесь `a` массив из десяти элементов типа `int`.

Можно определять синонимы для анонимных структур. Например:

```
typedef struct
{
  int  empno;
  char name[20];
  double salary;
} emp;
```

Теперь можно объявлять переменные типа `emp` и работать с этими переменными как с обычными структурами. Например:

```
emp e = {10, "Tim", 1000};
e.salary = 2000;
```

При определении синонима для типа можно также объявить и указатель на этот тип. Например:

```
typedef struct s
{
  int n;
} ti, *tip;
```

Теперь можно объявлять указатели на тип `s`, используя синоним `tip`. Например:

```
ti n = {10};
tip pn = &n;
```

В заключение этого раздела в листинге 8.6 покажем, как объявлять синонимы для типов функций и указателей на функции.

Листинг 8.6. Объявление функции при помощи синонима типа

```
#include <stdio.h>

int foo(int n) { return n; }
typedef int f(int), (*pf)(int);

f foo;
pf pfoo = &foo;

int main(void)
{
  int n;

  n = foo(10);
  printf("%d\n", n);  /* 10 */
  n = pfoo(20);
  printf("%d\n", n);  /* 20 */

  return 0;
}
```

8.8. Оператор *sizeof*

Для определения длины, как встроенных типов данных, так и типов данных, определенных программистом, используется оператор `sizeof`, который имеет следующий синтаксис:

```
sizeof выражение
```

или:

```
sizeof(тип)
```

Работает этот оператор на этапе компиляции программы. Оператор `sizeof` возвращает целочисленную константу без знака, которая равна длине операнда в байтах. Эта константа имеет тип `size_t`, который описан в заголовочном файле stddef.h. Оператор `sizeof` применяется для того, чтобы избежать использования аппаратно-зависимых длин данных в программе. Не допускаются операнды, которые имеют тип битового поля, функции или не-

полного типа. К *неполным* (incomplete) типам данных относятся следующие типы:

- тип `void`;
- массив неопределенного размера;
- массив, элементы которого имеют неполный тип;
- перечисление, структура или объединение с неопределенными членами.

Например, если объявлен указатель на структуру:

```
struct demo *p;
```

а сама структура не объявлена, то тип структуры будет неполным типом.

Если операндом оператора `sizeof` является массив или структура, то в результате получаем длину этого массива или структуры. Если последним элементом структуры является массив неопределенной длины, то оператор `sizeof` возвращает размер структуры без этого массива.

В листинге 8.7 приведены примеры работы оператора `sizeof`, операндом которого являются выражения.

Листинг 8.7. Определение длины переменных посредством оператора `sizeof`

```
#include <stdio.h>
int main(void)
{
  int  n = 0;
  double d = 0;
  char c = 'c';
  char a[10] = "array";

  printf("sizeof n = %d\n", sizeof n);  /* sizeof n = 4 */
  printf("sizeof d = %d\n", sizeof d);  /* sizeof d = 8 */
  printf("sizeof c = %d\n", sizeof c);  /* sizeof c = 1 */
  printf("sizeof a = %d\n", sizeof a);  /* sizeof a = 10 */

  return 0;
}
```

В листинге 8.8 приведены примеры работы оператора `sizeof`, операндом которого является тип данных.

Листинг 8.8. Определение длины типов посредством оператора `sizeof`

```
#include <stdio.h>
struct emp
{
  int empno;
  char name[20];
  double salary;
};
int main()
{
  /* sizeof(int) = 4 */
  printf("sizeof(int) = %d\n", sizeof(int));
  /* sizeof(char) = 1 */
  printf("sizeof(char) = %d\n", sizeof(char));
  /* sizeof(double) = 8 */
  printf("sizeof(double) = %d\n\n", sizeof(double));
  /* sizeof(struct emp) = 32 */
  printf("sizeof(emp) = %d\n", sizeof(struct emp));

  return 0;
}
```

Замечание

В языке программирования C++ при использовании оператора `sizeof` допускается любой тип операнда брать в скобки.

Учитывая это замечание, в языке программирования C++ можно написать следующие выражения:

```
int a[] = {0, 1, 2};
size_t s = sizeof(a);  // s = 12
```

8.9. Сравнимые типы

Если рассматривать типы как множества значений и операций, допустимых над этими значениями, то можно определить отношение сравнения между типами. Тип A сравним с типом B, если тип A содержит все значения типа B и операции, допустимые над этими значениями. Например, тип `int` сравним с типом `char`. Два типа называются *сравнимыми*, если каждый из них сравним с другим. Можно также сказать, что два типа сравнимы, если один из них может использоваться вместо другого. Очевидно, что идентичные типы сравнимы. Сравнимы также следующие встроенные типы данных:

- `char` и `signed char`;
- `short int`, `short`, `signed short int` и `signed short`;
- `unsigned short int` и `unsigned short`;
- `signed int`, `signed` и `int`;
- `unsigned int` и `unsigned`;
- `long int`, `long`, `signed long int` и `signed long`;
- `unsigned long int` и `unsigned long`.

Собственно говоря, понятие сравнимых типов и было введено в язык программирования C по причине существования разных спецификаторов встроенных типов, по сути описывающих один тип данных.

Сравнимые типы можно объединять, в результате получается тип, который называется *композиционным* (composite). Композиционный тип данных не теряет множество значений ни одного из сравнимых типов и допускает одинаковый набор операций над этими значениями. Отсюда следует, что сравнимые типы допускают связывание идентификаторов. В этом случае связанные идентификаторы будут ссылаться на переменную композиционного типа.

Новые типы данных могут объявляться в разных исходных файлах программы. Вопрос состоит в том, каким требованиям должны удовлетворять объявления одного и того же типа в разных

исходных файлах, чтобы программа работала правильно. Компилятор языка программирования C требует, чтобы такие типы были сравнимыми.

Обычно, для того чтобы избежать ошибок кодирования, объявления типов помещают в заголовочный файл, который затем включают в тексты разных исходных файлов командой препроцессора `#include`. Работа с заголовочными файлами рассмотрена в *разд. 9.5*.

Сейчас же перечислим требования, которым должны удовлетворять сравнимые типы, объявляемые в разных исходных файлах программы:

- сравнимые перечисления, структуры и объединения должны иметь одинаковые имена тегов, если они присутствуют, а также одинаковые имена своих членов, типы которых должны быть сравнимы;
- члены сравнимых перечислений должны иметь одинаковые значения;
- члены сравнимых структур должны быть расположены в одинаковом порядке.

Кроме того, в языке программирования C перечисления сравнимы с целочисленными типами, которые включают их значения.

Замечание

В языке программирования C++ перечисления не сравнимы с целочисленными типами.

Теперь рассмотрим сравнимость других сложных типов данных.

Два указателя сравнимы по типу, если они указывают на сравнимые типы.

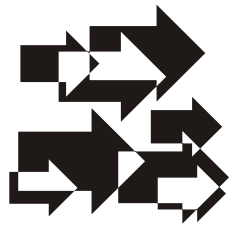
Два массива сравнимы, если сравнимы типы их элементов, и если оба массива имеют размерности, то эти размерности должны совпадать. Отсюда следует, что массив неопределенной длины сравним с массивом любой длины, если сравнимы типы элементов этих массивов.

Две функции считаются сравнимыми, если сравнимы типы возвращаемых ими значений. Если для функций объявлены прото-

типы, то для сравнения функций также необходимо, чтобы они имели одинаковое количество параметров и типы этих параметров были сравнимы.

В заключение этого раздела отметим, что одинаковые квалификаторы типов `const` или `volatile` не нарушают сравнимость типов. Разные квалификаторы типов или отсутствие квалификатора у одного из типов делает эти типы не сравнимыми.

ГЛАВА 9



Директивы препроцессора

9.1. Препроцессор

Препроцессором называется программа, которая вызывается компилятором и обрабатывает исходную программу перед ее компиляцией. Эта программа различает специальные команды, которые называются *директивами*. Строка с директивой отмечается символом `#`, который должен быть первым не пробельным символом строки. Этот символ не является составной частью директивы, а только указывает, что данная строка содержит директиву препроцессора. Поэтому после символа `#` могут располагаться пробелы. Заканчивается директива символом перехода на новую строку `\n`. Если перед символом перехода на новую строку встретился символ `\`, то препроцессор считает, что директива продолжается на следующей строке.

После обработки директив препроцессором исходный текст программы передается для обработки компилятору. Отметим также, что препроцессор различает все лексемы языка программирования C.

Препроцессор языка C обрабатывает следующие директивы:

```
#if      #else     #elif   #endif
#define  #undef    #ifdef  #ifndef
#error   #include  #line   #pragma
```

а также операторы `#` и `##`, которые и будут рассмотрены в этой главе.

9.2. Директивы *#define* и *#undef*

Директива `#define` определяет символическое имя для констант и выражений языка программирования C. Такое определение называется *макроопределением*, а символическое имя — *макросом*.

Если определяется символическое имя константы, то директива `#define` имеет следующий вид:

```
#define имя константа
```

Здесь `имя` определяет символическое имя для `константы` языка программирования C. В этом случае символическое имя также называется *символической константой*. Например, директива:

```
#define MAX_VAL 255
```

определяет символическое имя `MAX_VAL` для целочисленной константы `255`.

Если определяется символическое имя выражения, то директива `#define` имеет следующий вид:

```
#define имя(список_параметров) выражение
```

Здесь `имя` является символическим именем для `выражения` языка программирования C, а `список_параметров` содержит параметры, которые разделены запятыми и входят в `выражение`. В этом случае символическое имя также называется *макрокомандой*. Например, следующая директива:

```
#define MUL(a, b) ((a)*(b))
```

определяет макрокоманду для умножения двух чисел.

При определении макрокоманды ее параметры следует брать в круглые скобки, чтобы избежать возможных ошибок. Например, в следующем примере:

```
#define MUL(a, b) (a*b)
int n = MUL(1+2, 2+1);  /* c = 1+2*2+1 = 6 */
```

Переменная `n` инициализируется числом `6`, что явно противоречит намерениям программиста.

При помощи директивы `#define` можно также определить макрокоманду для вызова функции. Например, следующая директива:

```
#define print_int(val) printf("%d\n", val)
```

определяет символическое имя для функции, которая печатает значение целочисленного выражения. Теперь напечатать целочисленное значение можно так:

```
print_int(10);
```

Используются символические имена только после их определения. Если препроцессор встретит в исходном файле программы определенное символическое имя, то он заменит его соответствующим текстовым значением. По соглашению символические имена определяются прописными буквами.

Директива `#undef` служит для отмены определения символического имени и имеет следующий синтаксис:

```
#undef имя
```

После исполнения этой директивы символическое `имя` нельзя использовать в программе. Например:

```
#undef MAX_VAL
#undef MUL
```

9.3. Директивы *#if*, *#elsif*, *#else* и *#endif*

Директивы `#if`, `#elsif`, `#else` и `#endif` используются для условного включения и исключения фрагментов исходного кода в текст программы.

Директива `#if` служит для вставки в текст программы фрагмента исходного кода в зависимости от истинностного значения выражения. Значение выражения считается истинным, если оно не равно нулю. Завершение фрагмента исходного кода отмечается директивой `#endif`.

Синтаксис директив `#if` и `#endif` имеет следующий вид:

```
#if выражение
  исходный_код
#endif
```

Здесь выражение должно быть целочисленным константным выражением, а исходный_код — это исходный код на языке программирования C. Этот код вставляется в текст программы, если значение выражения истинно. В противном случае текст программы не изменяется.

Пример использования директив `#if` и `#endif` приведен в листинге 9.1.

Листинг 9.1. Пример использования директив `#if` и `#endif`

```
#include <stdio.h>
#define MAX_VAL 255
int main(void)
{
  int n = 10;

#if MAX_VAL > 20
  n = 20;
#endif

  printf("n = %d\n", n);  /* n = 20 */

  return 0;
}
```

Директива `#else` используется только с директивой `#if` и служит для вставки в текст программы фрагмента исходного кода, если истинностное значение выражения в директиве `#if` ложно. Комбинация директив `#if` и `#else` имеет следующий вид:

```
#if выражение
  исходный_код_1
#else
  исходный_код_2
#endif
```

Если значение выражения ложно, то в текст программы вставляется исходный_код_2. Например:

```
#define VERSION 2
#if VERSION > 2
  printf("Version = %d\n", 3);
```

```
#else
  printf("Version = %d\n", 2);  /* Version = 2 */
#endif
```

Директива `#elif` служит для вставки в текст программы фрагмента исходного кода в зависимости от истинностного значения выражения, альтернативного выражению директивы `#if`. Комбинация директив `#if` и `#elif` имеет следующий вид:

```
#if выражение_1
  исходный_код_1
#elif выражение_2
  исходный_код_2
...
#elif выражение_n
  исходный_код_n
#endif
```

В этом случае в текст программы вставляется тот фрагмент кода, который соответствует первому истинному выражению. При этом значения выражений вычисляются последовательно.

Совместно с директивами `#if` и `#elif` возможно использование директивы `#else`. При этом нужно учитывать, что директива `#else` всегда должна быть одна и находиться перед директивой `#endif`.

Так как вставка фрагмента кода в исходный текст программы вызывает компиляцию этого кода, то рассмотренные в этом разделе директивы часто называют *директивами условной компиляции*.

9.4. Директивы *#ifdef* и *#ifndef*

Директива `#ifdef` служит для включения фрагмента исходного кода в текст программы при условии, что определено некоторое символическое имя. Эта директива имеет следующий синтаксис:

```
#ifdef символическое_имя
  исходный_код
```

Здесь `исходный_код` включается в текст программы при условии, что определено `символическое_имя`. Заканчивается фрагмент ис-

ходного кода директивой #endif. Вместе с директивой #ifdef может использоваться директива #else. Например:

```
#define DEBUG

#ifdef DEBUG
  printf("DEBUG is defined.\n");  /* DEBUG is defined */
#else
  printf("DEBUG is not defined.\n");
#endif
```

Директива #ifndef служит для условного включения фрагмента исходного кода в текст программы при условии, что не определено некоторое символическое имя. Эта директива имеет следующий синтаксис:

```
#ifndef символическое_имя
  исходный_код
```

Здесь исходный_код включается в текст программы при условии, что символическое_имя не определено. Остальные правила использования директивы #ifndef совпадают с правилами использования директивы #ifdef.

В заключение этого раздела отметим, что директива #ifdef может заменяться директивой #if defined, а директива #ifndef — директивой #if !defined.

9.5. Директива *#include*

Директива #include предписывает препроцессору поместить в программу текст из файла на место этой директивы. Порядок поиска файла предопределен в компиляторе. Эта директива имеет один из следующих вариантов использования:

```
#include "имя_файла"
#include <имя_файла>
#include символическое_имя
```

Здесь имя_файла должно быть символьной строкой, определяющей путь к файлу.

Обычно в первом случае файл с заданным именем ищется сначала в текущем каталоге, а затем в каталогах, заданных в среде разработки. Например, директива:

```
#include "my_file.h"
```

вставляет в программу текст из файла `my_file.h`, который находится в текущем каталоге проекта.

Во втором случае файл с заданным именем обычно ищется только в каталогах, заданных в среде разработки. Например, директива:

```
#include <stdio.h>
```

вставляет в программу текст из файла `stdio.h`, который находится в стандартных каталогах (указанных компилятору).

Если в кавычках задан полный путь к файлу, то препроцессор ищет файл только по этому пути, игнорируя стандартные каталоги. Например, директива:

```
#include "C:\my_file.h"
```

подключает в программу текст из файла `my_file.h`, который находится на диске C:.

В третьем случае препроцессор подставляет в текст программы значение символической переменной, которое должно представлять символьную строку с именем файла, заключенном в кавычки или угловые скобки.

Отметим, что директива `#include` обрабатывается препроцессором рекурсивно. То есть, если подставленный фрагмент текста содержит другую директиву `#include`, то она также обрабатывается препроцессором.

В файлах, которые подключаются к исходному тексту программы директивой `#include`, обычно хранятся объявления внешних глобальных переменных и функций. Поэтому такие файлы называются *заголовочными файлами* (header files) и имеют расширение h.

Сложные программы могут включать несколько заголовочных файлов. Поэтому возможно многократное включение в текст программы содержимого одного и того же заголовочного файла.

Чтобы предотвратить такую ситуацию, обычно в заголовочный файл ставят "часового". Например, заголовочный файл может начинаться с такого текста:

```
#if !defined MY_NAME
#define MY_NAME
  /* здесь содержимое заголовочного файла */
#endif    /* MY_NAME */
```

В этом случае содержимое заголовочного файла будет вставляться в текст программы только в том случае, если в ней не определено символическое имя `MY_NAME`.

9.6. Директивы *#error*

Директива `#error` предписывает препроцессору прекратить обработку текста программы и выдать сообщение об ошибке. Эта директива имеет следующий синтаксис:

```
#error сообщение
```

Здесь `сообщение` представляет собой произвольный текст. Например:

```
#ifndef VERSION
  #error Vesrsion is not defined
#endif
```

9.7. Директива *#line*

Директива `#line` предписывает препроцессору изменить номер текущей строки и имя компилируемого файла. Эта директива имеет следующий синтаксис:

```
#line номер_строки "имя_файла"
```

где `номер_строки` — константа, которая устанавливает новый номер для текущей строки в обрабатываемом исходном файле, а `имя_файла` задает новое имя файла для компилируемого файла. Например:

```
#line 50 "user.c"
```

9.8. Директива *#pragma*

Директива `#pragma` предписывает компилятору выполнить некоторое действие. Можно также сказать, что эта директива предназначена для определения других директив препроцессора, которые зависят от реализации и версии компилятора. Директива `#pragma` имеет следующий синтаксис:

```
#pragma директива
```

Здесь `директива` представляет собой предопределенную инструкцию для компилятора, которая является произвольным текстом.

Например, следующие директивы соответственно выключают и включают оптимизацию кода в компиляторе Visual C++ фирмы Microsoft:

```
#pragma optimize("", off)
#pragma optimize("", on)
```

9.9. Пустая директива

Если строка в программе содержит единственный символ `#`, то это не вызовет никаких действий препроцессора. В этом случае за символом `#` следует пустая директива. Например:

```
#ifdef MAX
  #
#else
  #define MAX 255
#endif
```

9.10. Оператор

Оператор `#` используется в макрокомандах с одним параметром, для преобразования этого параметра в символьную строку языка программирования C. Такая макрокоманда имеет следующий вид:

```
#define имя(параметр) шаблон
```

Здесь шаблон — это последовательность лексем, содержащая параметр, которому предшествует символ #. При обработке макрокоманды пробелы, следующие до и после шаблона, удаляются. А при преобразовании шаблона в строку несколько пробелов между лексемами заменяются одним пробелом. Кроме того, при использовании оператора # нужно учитывать следующие ограничения:

- порядок вычисления нескольких операторов # и ## в шаблоне не определен;
- если шаблон содержит лексемы, которые являются строками языка программирования С, и эти лексемы следуют после параметра, то они соединяются с ним как строки.

Например, определим макрокоманду:

```
#define print_str(x) printf("%s", #x "\n")
```

Используя эту макрокоманду, вывести на печать строку можно следующим образом:

```
print_str(This is a string);
```

Оператор # используется для упрощения вывода на печать строк в формате языка программирования С. Например, при помощи макрокоманды print_str можно вывести на печать следующую строку:

```
print_str("\This is a string/");
```

9.11. Оператор

Оператор "##" служит для соединения двух параметров макрокоманды в одну последовательность символов. Такая макрокоманда должна иметь следующий вид:

```
#define имя(список_параметров) шаблон
```

Здесь шаблон — это последовательность лексем, содержащая два параметра макрокоманды, которые соединены символами ##. Например, определим макрокоманду:

```
#define var(type, s, var) type##s var
```

Используя эту макрокоманду, можно определить указатель на целочисленную переменную:

```
var(int, *, n);
```

При использовании оператора ## нужно учитывать, что порядок вычисления нескольких таких операторов в шаблоне макрокоманды не определен.

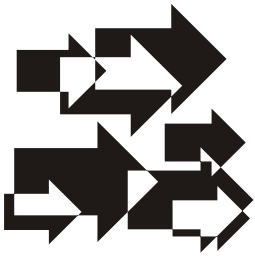
9.12. Предопределенные макрокоманды

Следующие макрокоманды предопределены в препроцессоре и не могут быть изменены программистом:

- __DATE__ — строка с датой;
- __TIME__ — строка со временем;
- __FILE__ — строка с именем исходного файла;
- __LINE__ — номер текущей строки;
- __STDC__ — константа, которая равна 1, если компилятор поддерживает стандартный язык C, в противном случае — равна 0.

Например, следующие инструкции распечатывают значения, которые дают эти макрокоманды:

```
printf("Date = %s\n", __DATE__);
printf("Time = %s\n", __TIME__);
printf("File = %s\n", __FILE__);
printf("Line = %d\n", __LINE__);
```



ЧАСТЬ II

Язык программирования C++

Глава 10. Дополнения к типам данных языка C

Глава 11. Дополнение к функциям языка C

Глава 12. Пространства имен

Глава 13. Обработка исключений

Глава 14. Классы

Глава 15. Конструкторы и деструкторы

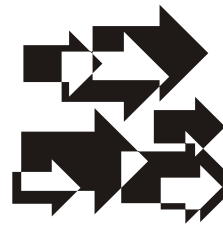
Глава 16. Перегрузка операторов

Глава 17. Наследование классов

Глава 18. Шаблоны функций

Глава 19. Шаблоны классов

ГЛАВА 10



Дополнения к типам данных языка C

10.1. Тип данных *bool*

В языке программирования C++ введен новый встроенный тип данных `bool`. Переменная типа `bool` может принимать одно из двух значений: `true` или `false`. Например:

```
bool  b = true;
```

Объекты типа `bool` неявно преобразуются в объекты типа `int`. А именно `false` в `0` и `true` в `1`. Обратно, любое целое значение может быть явно преобразовано в значение типа `bool`. А именно `0` в `false` и не `0` в `true`. Например:

```
bool  b;
b = 0;
b = (bool)10;
```

10.2. Тип данных *wchar_t*

Тип данных `wchar_t` является целочисленным типом данных, для хранения значений которого используются 2 байта. Следовательно, диапазон типа `wchar_t` включает значения от 0 до 65 535.

Тип данных `wchar_t` предназначен для хранения широких символов. Символ называется *широким* (wide), если для хранения кода этого символа требуется 2 байта. Для кодировки символов раз-

личных алфавитов при помощи широких символов предназначена спецификация Unicode, которая включает коды символов национальных алфавитов и любых печатных символов.

Широкие символьные константы определяются следующим образом:

```
L'символ'
```

Таким же образом определяется любой символ из кодовой таблицы ASCII. Например, объявление:

```
wchar_t w = L'S';
```

определяет переменную типа `wchar_t` и инициализирует ее широкой символьной константой `S`.

Управляющие символы языка программирования C++ кодируются обычным образом, например, `L'\n'`, `L'\0'`. Ноль всегда соответствует пустому широкому символу.

Строка, состоящая из широких символов, называется *широкой* (wide) строкой. Широкий строковый литерал определяется следующим образом:

```
L"строка_символов"
```

Например, объявление:

```
wchar_t ws[] = L"This is a string.";
```

определяет массив типа `wchar_t` и инициализирует его широкой строкой.

В листинге 10.1 приведен пример объявления и вывода на консоль широких символов и строк. В этом листинге `wcout` — это стандартный поток вывода широких символов на консоль. Для вывода в этот поток данных разнообразных типов используется оператор `<<`, а манипулятор `endl` предназначен для перехода на начало следующей строки. Подробно работа с потоками ввода/вывода языка программирования C++ рассмотрена в *гл. 40*.

Листинг 10.1. Объявление широких символов и строк

```
#include <iostream>
using namespace std;
```

```
int main()
{
  wchar_t w = L'S';
  wchar_t ws[] = L"This is a string.";

  wcout << w << endl;  // S
  wcout << ws << endl; // This is a string.

  return 0;
}
```

10.3. Анонимные объединения

В языке программирования C++ допускается объявление *анонимных объединений*. Анонимные объединения являются не типами, а объектами. Доступ к членам анонимного объединения осуществляется по их именам, которые должны быть уникальными внутри области видимости, в которой объявлено это объединение.

В листинге 10.2 приведен пример доступа к членам анонимного объединения. В этом листинге `cout` — это стандартный поток вывода на консоль, который работает с символами в коде ASCII. Для вывода в этот поток данных разнообразных типов используется оператор `<<`, а манипулятор `endl` предназначен для перехода на начало следующей строки. Подробно работа с потоками ввода/вывода языка программирования C++ рассмотрена в *гл. 40*.

Листинг 10.2. Доступ к членам локального анонимного объединения

```
#include <iostream>
using namespace std;

int main()
{
  union {
    int  n;
    double d;
  };
```

```
  n = 10;
  cout << n << endl;  // печатает 10
  d = 10.5;
  cout << d << endl;  // печатает 10.5
  return 0;
}
```

Глобальные анонимные определения должны определяться как статические. В листинге 10.3 показан пример объявления глобального анонимного объединения и доступа к его членам.

Листинг 10.3. Доступ к членам глобального анонимного объединения

```
#include <iostream>
using namespace std;

static union
{
  int  n;
  double d;
};
int main()
{
  n = 10;
  cout << n << endl;  // печатает 10
  d = 10.5;
  cout << d << endl;  // печатает 10.5

  return 0;
}
```

10.4. Ссылки

Ссылкой называется указатель на переменную или константу, который всегда *разыменовывается* при его использовании. Поэтому ссылка всегда содержит значение переменной или константы, на которую она ссылается. Для объявления переменной типа "ссылка" используется символ &, который записывается перед именем переменной. Если ссылка используется как самостоя-

тельный объект, то она должна быть инициализирована константой или переменной при своем определении. В первом случае ссылка называется *константной ссылкой*. Во втором случае — *псевдонимом переменной*. После инициализации ссылка не может быть изменена. Со ссылкой на объект можно производить такие же действия, что и с объектом, на который она ссылается.

В листинге 10.4 приведен пример объявления и инициализации ссылки.

Листинг 10.4. Объявление и инициализация ссылки

```
#include <iostream>
using namespace std;

int main()
{
  const int  n = 10;
  const int  &rn = n;  // ссылка на константу n

  int  m = 2;
  int  &rm = m;  // ссылка на переменную m
  rm = 20;  // изменяем значение переменной m через ссылку

  cout << rn << endl;  // печатает 10
  cout << m << endl;   // печатает 20

  return 0;
}
```

Ссылки в язык программирования C++ введены для двух целей:

- чтобы обеспечить возможность передачи аргументов в функции без их копирования в параметры функции;
- чтобы обеспечить возможность возврата из функции L-value.

Отметим, что оператор `sizeof` от ссылки дает размер объекта, на который ссылается ссылка. Например:

```
short  n;
short&  r = n;
size_t  s;

s = sizeof(r);     // s = 2
```

10.5. Новые операторы явного преобразования типов

В язык программирования C++ для явного преобразования типов данных введены операторы: `const_cast`, `static_cast`, `dynamic_cast` и `reinterpret_cast`, которые имеют следующий синтаксис:

```
оператор<тип>(выражение)
```

Здесь `оператор` — это один из операторов преобразования типа, а `тип` — является типом данных, к которому приводится `выражение`. Если `тип` является ссылкой, то `оператор` вернет L-value, в противном случае `оператор` вернет R-value. Все новые операторы преобразования типа имеют ассоциативность справа налево, а их приоритет совпадает с приоритетом постфиксных операций инкремента и декремента. Далее новые операторы преобразования типов будут рассмотрены подробнее.

Оператор *static_cast*

Оператор `static_cast` выполняет преобразования типов, которые не могут быть сделаны компилятором неявно. Этот оператор не отменяет квалификаторы `const` или `volatile`. Работает оператор `static_cast` на этапе компиляции программы.

В листинге 10.5 приведен пример использования оператора `static_cast`.

Листинг 10.5. Использование оператора `static_cast`

```
#include <iostream>
using namespace std;

int main()
{
  double d = 97.0;
  char c = static_cast<char>(d);
  cout << c << endl;  // печатает символ 'a'

  return 0;
}
```

Оператор *const_cast*

Оператор const_cast отменяет для переменных действие квалификаторов const и volatile. При этом тип переменной не должен изменяться. Работает оператор const_cast на этапе компиляции программы.

Оператор const_cast используется главным образом для преобразования типов указателей на константы, отменяя действие модификаторов доступа const или volatile.

В листинге 10.6 приведен пример использования оператора const_cast.

Листинг 10.6. Использование оператора const_cast

```
#include <iostream>
using namespace std;

int foo(int* p) { return *p; }
int main()
{
  const int n = 10;
  cout << foo(const_cast<int*>(&n)) << endl;

  return 0;
}
```

Отметим, что при попытке записи в константный объект через указатель, тип которого изменен оператором const_cast, поведение программы не определено. Чаще всего это вызовет ошибку, так константные объекты обычно размещаются в памяти, в которую запрещена запись.

Оператор *reinterpret_cast*

Оператор reinterpret_cast используется для преобразования типа выражения, но сохранения его битового представления. Работает оператор reinterpret_cust на этапе компиляции программы.

В листинге 10.7 приведен пример использования оператора reinterpret_cust.

Листинг 10.7. Использование оператора `reinterpret_cast`

```
#include <iostream>
using namespace std;

int main()
{
  int  n = (int)'a' + (int)'b' * 256;
  char*  p = reinterpret_cast<char*>(&n);

  cout << p[0] << p[1] << endl;  // печатает ab

  return 0;
}
```

Оператор *dynamic_cast*

Оператор `dynamic_cast` используется для преобразования указателя или ссылки на производный класс соответственно в указатель или ссылку на базовый класс при условии, что эти классы полиморфны. Работает оператор `dynamic_cast` на этапе исполнения программы, причем при компиляции программы должна быть включена опция поддержки проверки типов объектов.

В листинге 10.8 приведен пример использования оператора `dynamic_cast`.

Листинг 10.8. Использование оператора `dynamic_cast`

```
#include <iostream>
using namespace std;

struct Base
{
  virtual void foo() { cout << "Base" << endl; }
};
struct Derived: Base
{
  virtual void foo() { cout << "Derived" << endl; }
};
```

```
void foo(Base* pb)
{
  Derived* pd = dynamic_cast<Derived*>(pb);
  pd->foo();
}
int main()
{
  Derived d;
  Derived* pd = &d;
  foo(pd);

  return 0;
}
```

10.6. Типизированные операторы распределения памяти

В языке программирования C++ для динамического размещения объекта в куче используется оператор `new`, который имеет следующий синтаксис:

```
new тип
```

Здесь `тип` является именем встроенного или определенного программистом типа данных. Например:

```
int* p;
p = new int;  // значение *p не определено
*p = 10;      // определяем значение *p
```

Объект можно инициализировать при его динамическом создании. Для этого используется оператор `new` следующего вида:

```
new тип(выражение)
```

Значение `выражения` задает начальное значение объекта. Например:

```
int* p;
p = new int(10);  // *p = 10
```

Для динамического удаления объектов из кучи используется оператор delete, который имеет следующий синтаксис:

```
delete указатель
```

Здесь указатель должен указывать на объект, предварительно распределенный оператором new. Например:

```
delete p;
```

Для динамического размещения массива объектов в куче оператор new имеет следующий синтаксис:

```
new тип[диапазон_1] ... [диапазон_n]
```

Все диапазоны, за исключением диапазона_1, должны быть константами. Первый диапазон может быть переменной. Например:

```
int* p = new int[10];  // динамический массив
```

Элементы динамически создаваемого массива нельзя проинициализировать.

Для динамического удаления массива из кучи используется оператор delete, который имеет следующий синтаксис:

```
delete[] указатель
```

Указатель должен указывать на массив, предварительно распределенный оператором new[]. Например:

```
delete[] p;
```

Для динамического конструирования объекта в заданной области памяти используется оператор new, который имеет следующий синтаксис:

```
new(указатель) тип(значение)
```

Этот оператор строит объект заданного типа в области памяти, адрес которой содержит указатель. Такая форма оператора new называется *оператором размещения* new.

В листинге 10.9 приведен пример конструирования объектов в заданной области памяти. Естественно в этом примере можно было бы просто присвоить новые значения переменной n. Но этот подход не годится, когда для конструирования объекта нужно вызвать конструктор класса.

Листинг 10.9. Использование оператора размещения new

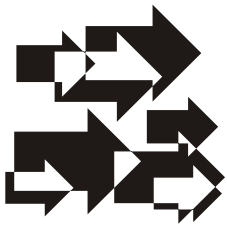
```
#include <iostream>
using namespace std;

int main()
{
  int n;

  int *p = new(&n) int(10);
  cout << *p << ' ' << n << endl;  // 10 10

  int *q = new(&n) int(20);
  cout << *q << ' ' << n << endl;  // 20 20

  return 0;
}
```

По стандарту в случае ошибки при размещении объекта в куче оператор `new` генерирует исключение `bad_alloc`. Но в некоторых компиляторах возможны такие реализации этого оператора, которые в случае невозможности размещения объекта в куче возвращают значение `NULL`.

При использовании оператора `delete` нужно учитывать следующие моменты. Во-первых, если операндом этого оператора является пустой указатель, то оператор не выполняет никаких действий. Во-вторых, если операндом этого оператора является неправильный указатель, то поведение оператора непредсказуемо.

Новые операторы распределения памяти имеют ассоциативность справа налево, а их приоритет совпадает с приоритетом унарных арифметических операторов.

ГЛАВА 11



Дополнение к функциям языка С

11.1. Передача аргументов по умолчанию

В языке программирования С++ в прототипе функции или в определении функции, если оно используется как прототип, можно указать значения, которые задаются параметрам по умолчанию. В этом случае, если при вызове функции некоторые или все аргументы пропущены, то компилятор подставит вместо них значения, заданные по умолчанию. Если некоторый параметр имеет значение по умолчанию, то значения по умолчанию должны также иметь все параметры, которые расположены после него. Соответственно, если при вызове функции аргумент опускается, то должны быть опущены и все аргументы после него.

В листинге 11.1 приведен пример передачи в функцию аргументов по умолчанию.

Листинг 11.1 Передача аргументов по умолчанию

```
#include  <iostream>
using namespace std;
void print(int i, int j = 1, int k = 2)
{
  cout << i << ' ' << j << ' ' << k << endl;
}
```

```
int main()
{
  print(0);     // печатает 0 1 2
  print(1, 3);  // печатает 1 3 2

  return 0;
}
```

11.2. Встроенные функции

Функция называется *встроенной*, если компилятор генерирует не вызов этой функции, а встраивает ее код в тело программы. Для определения встроенной функции используется ключевое слово `inline`, которое записывается перед именем функции или типом возвращаемого функцией значения.

В листинге 11.2 приведен пример определения встроенной функции.

Листинг 11.2. Определение встроенной функции

```
#include <iostream>
using namespace std;
inline double cube(double x) { return (x * x * x); }
int  main( )
{
  double x = 1.1;

  cout << "x^3 = " << cube(x) << endl;  // x^3 = 1.331

  return 0;
}
```

Ключевое слово `inline` рассматривается компилятором только как пожелание встроить код функции в программу. Компилятор может и не выполнить это пожелание. Это обычно происходит в следующих двух случаях:

- функция является рекурсивной;
- функция вызывается только через указатель в данном исходном файле.

Встроенная функция должна быть видна компилятору в месте ее вызова, т. е. компилятор выполняет внутреннее связывание встроенных функций. Поэтому, в отличие от обычной функции, встроенная функция должна быть определена в каждом исходном файле, где есть обращение к ней. Причем эти определения должны совпадать. Обычно для упрощения реализации определение встроенной функции помещают в заголовочный файл, а затем включают этот файл во все исходные файлы, где есть обращение к встроенной функции.

11.3. Передача аргументов и возврат значения через ссылки

В языке программирования C++ значения параметров могут передаваться функции через ссылки. Так как ссылка является псевдонимом переменной, то вызов такой функции синтаксически не отличается от вызова функции, аргументы которой передаются по значению.

В листинге 11.3 приведен пример передачи аргументов через ссылки.

Листинг 11.3. Передача аргументов через ссылки

```
#include <iostream>
using namespace std;
void swap(int& x, int& y)
{
  int t;

  t = x;
  x = y;
  y = t;
}
int main()
{
  int a = 1, b = 2;
```

```
  cout << a << ' ' << b << endl;  // печатает 1 2
  swap(a, b);
  cout << a << ' ' << b << endl;  // печатает 2 1

  return 0;
}
```

Ссылку можно использовать как тип возвращаемого функцией значения. Это позволяет присваивать функции значения. Так как ссылка является *псевдопеременной*, то возврат ссылки синтаксически не отличается от возврата по значению.

В листинге 11.4 приведен пример возвращения значения функции через ссылку.

Листинг 11.4. Возврат значения через ссылку

```
#include <iostream>
using namespace std;
int& item(int* a, int i) { return a[i]; }
int main()
{
  int a[] = {1, 2, 3};

  cout << a[1] << endl;  // печатает 2
  item(a, 1) = 15;
  cout << a[1] << endl;  // печатает 15

  return 0;
}
```

Однако в этом случае нельзя допустить ошибку, когда ссылка возвращается на объект, определенный в теле функции. Этот объект является локальным и после выхода из функции он уничтожается. Поэтому такая ссылка будет ссылаться на неопределенное значение.

11.4. Перегрузка функций

Перегрузкой функций называется объявление в одной области видимости нескольких функций с одинаковым именем, но раз-

ными сигнатурами. *Сигнатурой функции* называется список типов ее параметров. На перегрузку функций накладываются следующие ограничения. Нельзя перегружать функции:

- которые отличаются только возвращаемым значением, т. к. в этом случае компилятор не знает, какую из них использовать при вызове, если для возвращаемого значения требуется преобразование типов;
- параметры которых отличаются только использованием квалификаторов `const` и `volatile`;
- параметры которых отличаются только тем, что переопределены при помощи объявления `typedef`;
- параметры которых отличаются только тем, что один из них массив, а второй указатель на тип, который соответствует типу элементов массива;
- параметры которых отличаются только значениями, заданными по умолчанию.

Например, в языке программирования C++ в одной программе можно определить следующие функции:

```
int neg(int n) { return -n; }
float neg(float f) { return -f; }
```

Возможность перегрузки функций упрощает выбор имен для функций, которые имеют схожую функциональность для разных типов параметров.

В случае вызова одной из перегруженных функций, компилятор выбирает нужную функцию по лучшему соответствию типов аргументов типам параметров. Выбор такой функции разбивается на три этапа, которые и описаны ниже.

На первом этапе строится множество *функций кандидатов* (candidate functions), т. е. таких функций, имя которых совпадает с именем вызываемой функции.

На втором этапе из множества функций кандидатов выбираются *осуществимые функции* (viable functions), т. е. такие функции, у которых количество параметров равно количеству аргументов в вызываемой функции.

На третьем этапе из множества осуществимых функций выбирается наилучшая из осуществимых функций, которая и вызывается. Для выбора наилучшей из осуществимых функций вводится понятие *последовательности неявных преобразований* (implicit conversion sequence) аргумента. Из самого названия этой последовательности видно, что она описывает последовательные преобразования типов, которые позволяют привести тип аргумента в вызове функции к типу соответствующего параметра в объявлении функции. Последовательности неявных преобразований ранжируются, что позволяет компилятору выбрать наилучшую из осуществимых функций. Компилятор выбирает эту функцию таким образом, что для каждого параметра этой функции последовательность неявных преобразований имеет ранг не ниже по сравнению с соответствующими рангами последовательностей неявных преобразований других осуществимых функций. Если наилучшая из осуществимых функций не единственна, то компилятор выдает сообщение об ошибке.

Приведем ранжирование последовательностей неявных преобразований аргументов.

Сначала последовательности неявных преобразований делятся на следующие три группы:

- последовательности стандартных преобразований;
- последовательности определенных пользователем преобразований;
- последовательности преобразования в неопределенные параметры.

Ранги этих групп понижаются от первой к последней группе. Причем в каждой группе последовательности неявных преобразований также ранжируются.

Группа последовательностей стандартных преобразований разбивается на следующие подгруппы:

- точное совпадение:
  - не требуется преобразования типов;
  - преобразование L-value в R-value;

- преобразование массива в указатель;
- преобразование квалификаторов `const` и `volatile`;

❒ продвижение по типам:

- интегральное продвижение;
- плавающее продвижение;

❒ стандартные преобразования типов:

- целочисленные преобразования без продвижения;
- преобразования типов с плавающей точкой без продвижения;
- преобразования из целочисленного типа в плавающий тип;
- преобразования указателей;
- преобразования в тип данных `bool`;
- преобразования указателей на члены классов.

Ранги этих подгрупп понижаются от первой к последней подгруппе. То есть подгруппа точное совпадение имеет наивысший ранг. Компилятор ранжирует последовательность стандартных преобразований по наихудшему преобразованию типов в этой последовательности.

Ниже рассмотрены различные случаи вызова перегруженных функций `neg`, определенных выше.

```
float f = neg(5.5f);  // вызывается функция neg(float)
char c = abs('a');    // вызывается neg(int)
double d = neg(5.5);  // ошибка
```

В первом случае преобразование типов не требуется, т. к. тип аргумента точно совпадает с типом параметра функции `neg(float)`. Во втором случае точного соответствия типов нет, но наилучшим преобразованием типа аргумента является интегральное продвижение, поэтому вызывается функция `neg(int)`. В третьем случае компилятор выдаст сообщение об ошибке, т. к. существует два стандартных преобразования, одно к типу `double`, а второе к типу `int`, и оба этих преобразования имеют одинаковый ранг.

Группа последовательностей определенных пользователем преобразований содержит такие последовательности, которые вклю-

чают преобразование типов данных, определенное пользователем. Эти последовательности имеют следующую структуру. Сначала следует последовательность стандартных преобразований, за которой следует определенное пользователем преобразование. Завершающей является опять последовательность стандартных преобразований. Очевидно, что первый и третий этапы такого преобразования могут отсутствовать.

В качестве преобразования типов, определенного пользователем, может выступать конструктор класса или конвертер, т. е. функция класса, которая используется для преобразования типа объекта.

Компилятор сравнивает последовательности определенных пользователем преобразований только в том случае, если они имеют одинаковые преобразования, определенные пользователем. В противном случае последовательности не сравнимы и компилятор выдает сообщение об ошибке. При сравнении таких последовательностей компилятор считает, что одна последовательность определенных пользователем преобразований лучше другой, если вторая последовательность стандартных преобразований в этой последовательности имеет высший ранг по сравнению с рангом второй последовательности стандартных преобразований в другой последовательности.

Группа последовательностей преобразований в неопределенные параметры содержит последовательности, которые включают преобразования аргументов для функций с неопределенным количеством параметров. Если существует несколько таких последовательностей, то компилятор считает наилучшей ту из них, которая содержит меньше неопределенных типов аргументов.

Возможность функций с разными сигнатурами иметь одно и то же имя называется *перегружающим* (overloading) *полиморфизмом*.

11.5. Искажение имен функций

Для реализации перегрузки функций компилятор добавляет к внутреннему имени функции дополнительные символы, которые определяют тип и порядок параметров функции. Такое именова-

ние функций называется *декорированием* (decorating) или *искажением* (mangling) *имен функций*. К сожалению, декорирование имен функций не стандартизировано и поэтому в каждом компиляторе определяется свой порядок декорирования имен.

Из определения декорирования следует, что из программы, написанной на языке программирования C++, нельзя непосредственно вызвать функцию из объектного модуля, полученную при помощи компилятора языка программирования C. Чтобы решить эту проблему, в языке программирования C++ введена *спецификация связывания*, которая имеет следующий синтаксис:

```
extern "C" имя_функции
```

или:

```
extern "C"
{
  // блок объявления имен
}
```

Заметим, что блок объявления имен может содержать как имена функций, так и имена переменных.

Например, предположим, что в программе на языке программирования C++ используется функция `fun` из объектной библиотеки, разработанной на языке программирования C, и эта функция имеет следующий прототип:

```
int fun(int);
```

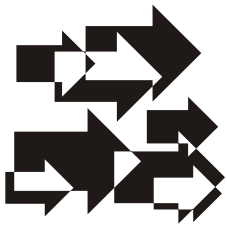
Тогда в этой программе для разрешения связей между объектными модулями нужно использовать спецификацию сцепления:

```
extern "C" int fun(int);
```

Теперь предположим, что в программе на языке программирования C++ используются функции из библиотеки, разработанной на языке программирования C, а прототипы этих функций описаны в заголовочном файле import.h. Тогда в этой программе для разрешения связей между объектными модулями нужно использовать спецификацию сцепления

```
extern "C" { #include import.h }
```

ГЛАВА 12



Пространства имен

12.1. Определение пространства имен

Пространство имен определяет область видимости имен объектов, которые не входят ни в один блок. Поэтому пространство имен не может быть определено внутри блока. Объявление пространства имен имеет следующий синтаксис:

```
namespace идентификатор
{
  // тело пространства имен
}
```

Здесь `идентификатор` определяет имя пространства имен, а тело пространства имен содержит описания и определения имен. Например, определим пространство имен `MyNames`, которое содержит определение одной переменной и одной функции.

```
namespace MyNames
{
  int m;
  int add(int a, int b) { return a + b; }
}
```

Пространства имен *открыты*, т. е. одно и то же пространство имен можно расширять в разных исходных файлах. Например,

пространство имен MyNames можно расширить в другом исходном файле:

```
namespace  MyNames
{
  int n;
  int sub(int a, int b);
}
```

Пространства имен могут быть вложенными друг в друга. Например:

```
namespace PublicNames
{
  int n;
  namespace MyNames
  {
    int add(int a, int b) { return a + b; }
  }
}
```

12.2. Анонимные пространства имен

По умолчанию существует анонимное пространство имен, которое доступно во всех исходных файлах и которое не нужно объявлять. Это пространство имен называется *глобальным*. Можно сказать, что все пространства имен вложены в глобальное пространство имен. Все имена, которые не входят ни в одно пространство имен, по умолчанию принадлежат глобальному пространству имен.

Имена из заголовочных файлов, которые имеют расширение h, принадлежат глобальному пространству имен.

В каждом исходном файле также можно объявить *локальное анонимное пространство имен*, которое видимо только в этом файле. Это пространство имен объявляется следующим образом:

```
namespace
{
  // тело пространства имен
}
```

Локальное анонимное пространство имен может быть вложено в другое именованное пространство имен. Например:

```
namespace MyNames
{
  namespace
  {
    int add(int a, int b) { return a + b; }
  }
}
```

12.3. Стандартное пространство имен

По умолчанию в языке программирования C++ существует пространство имен `std`, которое называется *стандартным*. Имена из заголовочных файлов, которые не имеют расширения, принадлежат стандартному пространству имен.

В листинге 12.1 приведен пример использования имен из стандартного пространства `std`. Как видно, в этом случае нужно использовать оператор разрешения области видимости, который будет рассмотрен в следующем разделе.

Листинг 12.1. Использование имен из стандартного пространства `std`

```
#include <iostream>
int main()
{
  std::cout << "Hello word" << std::endl;
  return 0;
}
```

12.4. Оператор разрешения области видимости

Для доступа к именам пространства имен используется оператор разрешения области видимости `::`, который имеет следующий синтаксис:

```
имя_пространства_имен::имя_объекта
```

В этом случае имя пространства имен называется *квалификатором имени объекта*. При этом отметим, что оператор разрешения области видимости выполняется на этапе компиляции программы и не влияет на ее производительность. Этот оператор ассоциативен справа налево и имеет наивысший приоритет из всех операторов.

В листинге 12.2 приведен пример доступа к именам при помощи оператора разрешения области видимости.

Листинг 12.2. Доступ к именам при помощи оператора разрешения области видимости

```
#include <iostream>
namespace MyNames
{
  int n, m;
  int add(int a, int b) { return a + b; }
}
int main()
{
  int sum;
  MyNames::n = 2;
  MyNames::m = 3;

  sum = MyNames::add(MyNames::n, MyNames::m);
  std::cout << sum << std::endl;  // печатает 5

  return 0;
}
```

Для доступа к именам из глобального пространства имен используется оператор разрешения области видимости без квалификатора. Это необходимо в случае, если глобальные имена скрыты локальными именами из не глобальной области видимости.

В листинге 12.3 приведен пример доступа к именам из глобального пространства имен.

Листинг 12.3. Доступ к именам из глобального пространства имен

```
#include <iostream>
int n = 10;
```

```
namespace MyNames
{
  int add(int n) { return n + ::n; }
}
int main()
{
  int n = 5;
  int m = MyNames::add(n);

  std::cout << m << std::endl;  // печатает 15

  return 0;
}
```

Для доступа к именам из локального анонимного пространства оператор разрешения области видимости не используется.

В листинге 12.4 приведен пример доступа к именам из локального анонимного пространства имен.

Листинг 12.4. Доступ к именам из локального анонимного пространства имен

```
#include <iostream>
int n = 10;
namespace
{
  int add(int n) { return n + ::n; }
}
int main()
{
  int n = 5;
  int m = add(n);

  std::cout << m << std::endl;  // печатает 15

  return 0;
}
```

Имя переменной, принадлежащее локальному анонимному пространству имен, может быть недоступно в другом пространстве

имен. Это происходит в том случае, если это имя скрыто именем глобальной или локальной переменной.

В листинге 12.5 приведен пример неоднозначного определения имени переменной в локальном анонимном пространстве имен.

Листинг 12.5. Неоднозначное определение переменной

```
#include <iostream>
int n = 10;
namespace
{
  int n = 20;
}
int main()
{
  int m = n;  // ошибка компиляции: n - неоднозначный символ

  std::cout << m << std::endl;

  return 0;
}
```

12.5. Объявление *using*

Для объявления имени из некоторого пространства имен используется объявление `using`, которое может находиться в любой области видимости, включая область видимости любого пространства имен. Объявление `using` имеет следующий синтаксис:

```
using имя_пространства_имен имя_объекта;
```

В листинге 12.6 приведен пример использования объявления `using`.

Листинг 12.6. Использование объявления `using`

```
#include <iostream>
using std::cout;
int main()
```

```
{
  int n = 10;
  cout << n;  // печатает 10
  return 0;
}
```

Объявление `using` может также использоваться для вложенных пространств имен.

В листинге 12.7 приведен пример использования объявления `using` для вложенных пространств имен.

Листинг 12.7. Использование объявления `using` с вложенными пространствами имен

```
#include <iostream>
using std::cout;
namespace PublicNames
{
        namespace MyNames
        {
                int add(int a, int b) { return a + b; }
        }
}
int main()
{
        using PublicNames::MyNames::add;
        cout << add(2,3);  // печатает 5
        return 0;
}
```

12.6. Директива *using*

Для объявления сразу всех имен из некоторого пространства имен используется директива `using`, которая имеет следующий синтаксис:

```
using namespace имя_пространства_имен;
```

В листинге 12.8 приведен пример использования директивы `using`.

Листинг 12.8. Использование директивы `using`

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello word" << endl;
  return 0;
}
```

Правила использования директивы `using` совпадают с правилами использования объявления `using`. Но при этом следует учитывать, что директива `using` для вложенного пространства имен не объявляет имена из объемлющего пространства имен.

12.7. Псевдонимы

Чтобы избежать дублирования имен, для пространства имен может быть выбрано длинное имя. В этом случае для удобства записи этого длинного имени также определяют короткий *псевдоним*, который обычно является сокращением от полного имени пространства имен. Псевдоним используется в тех исходных файлах, которые заведомо не содержат имен, совпадающих с псевдонимом. Объявление псевдонима имеет следующий синтаксис:

```
namespace псевдоним = имя_пространства_имен;
```

Псевдоним используется точно так же, как и само имя пространства имен.

В листинге 12.9 приведен пример использования псевдонима для имени пространства имен.

Листинг 12.9. Использование псевдонима для имени пространства имен

```
#include <iostream>
using namespace std;
namespace Very_Important_Namespace
```

```
{
  int n = 1;
}
namespace VIN = Very_Important_Namespace;
int main()
{
  cout << VIN::n << endl;  // печатает 1
  return 0;
}
```

Псевдоним можно определять и для имен вложенных пространств имен. Например:

```
namespace Names = PublicNames::MyNames;
```

Кроме того, одно пространство имен может иметь несколько псевдонимов, например:

```
namespace  N = Names;
namespace  M = Names;
namespace  L = M;
```

12.8. Искажение имен переменных

Для реализации оператора разрешения области видимости компилятор добавляет к внутреннему имени переменной дополнительные символы, которые определяют пространство имен переменной. Такое именование переменных называется *декорированием* (decorating) или *искажением* (mangling) *имен переменных*. Так же как и декорирование имен функций, декорирование имен переменных не стандартизировано. Поэтому в каждом компиляторе определяется свой порядок декорирования имени переменной.

Как и в случае с функциями, для обращения в программе, написанной на языке программирования C++, к имени переменной, определенной в объектном модуле, полученном при помощи компилятора языка программирования C, нужно использовать *спецификацию связывания*, которая имеет следующий синтаксис:

```
extern "C" имя_переменной
```

или:

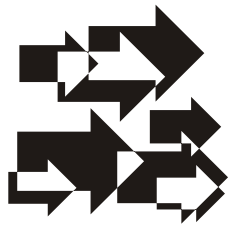
```
extern "C"
{
  // блок объявления имен
}
```

Например, предположим, что в программе на языке программирования C++ используется переменная `n` типа `int` из объектной библиотеки, разработанной на языке программирования C.

Тогда, в этой программе, для разрешения связей между объектными модулями нужно использовать спецификацию связывания:

```
extern "C" int n;
```

ГЛАВА 13



Обработка исключений

13.1. Механизм обработки исключений

Механизм обработки исключений (исключительных ситуаций) предназначен для передачи управления при возникновении таких ситуаций во время исполнения программы, которые не могут быть разрешены на месте их возникновения. Механизм обработки исключений построен на базе следующих четырех компонентов:

- исключение;
- выражение `throw`;
- блок `try`;
- блок `catch`.

Синтаксически *исключение* (exception) представляет собой объект произвольного типа. Этот объект используется для идентификации ситуации, которая требует обработки.

Исключение *генерируется* или, как еще говорят, *выбрасывается* выражением `throw`, которое имеет следующий вид:

```
trow выражение
```

Результатом вычисления `выражения` является объект, который представляет исключение.

Для работы механизма обработки исключений в программе должны быть определены блоки `try` и `catch`, которые организованы следующим образом:

```
try
{
  // тело блока try
}
catch(тип_исключения)
{
  // тело блока catch
}
```

Тело блока `try` содержит код, при исполнении которого может произойти выброс исключения, а тело блока `catch` содержит код, который обрабатывает исключение и поэтому также называется *обработчиком исключения*. Отметим, что блок `catch` должен следовать непосредственно за блоком `try`.

Работает механизм обработки исключений следующим образом. Если внутри блока `try` произошел выброс исключения, посредством исполнения инструкции с выражением `throw` внутри этого блока или внутри любой функции, которая вызывается в этом блоке, то исполнение блока `try` немедленно прекращается независимо от уровня вложенности, на котором произошел выброс исключения, а управление передается блоку `catch`. После завершения работы блока `catch` управление передается на первую инструкцию, следующую за этим блоком. При этом блок `catch` обрабатывает только то исключение, тип которого соответствует его `типу_исключения`. Для упрощения терминологии `тип_исключения` часто также называют *типом обработчика исключения*.

В листинге 13.1 приведен пример обработки исключения, которое генерируется при попытке целочисленного деления на ноль.

Листинг 13.1. Обработка исключения

```
#include <iostream>
using namespace std;
```

```
int divide(int a, int b)
{
  if (!b)
    throw "Zero divide";
  return a / b;
}
int main()
{
  int n, m;
  try
  {
    // введите два целых числа через пробел
    cout << "Input two integers: ";
    cin >> n >> m;
    cout << "n/m = " << divide(n,m) << endl;
  }
  catch(char* str)
  {
    cout << str << endl;  // печатает: Zero divide
  }
  cout << "OK" << endl;

  return 0;
}
```

Кратко поясним работу этой программы. Пользователь вводит два целых числа. Если делитель не равен нулю, то программа печатает частное от деления этих чисел, которое вычисляется функцией `divide`. Тонкость в том, что функция `divide` проверяет делитель на равенство нулю. Если он равен нулю, то функция не вычисляет частное, а выбрасывает исключение "`Zero divide`" типа `char*`. Если исключение выброшено, то как исполнение функции `divide`, так и исполнение блока `try` немедленно завершается, а управление передается в блок `catch`. Для этого тип параметра блока `catch` выбран в соответствии с типом выбрасываемого исключения.

Часто при обработке исключения нужно знать только тип этого исключения. В этом случае для типа исключения используют

пустые структуры. Например, в нашем случае можно было бы определить исключение типа

```
struct ZeroDivide{};
```

Программа обработки такого исключения приведена в листинге 13.2. Заметим, что в этой программе инструкция с выражением `throw` выбрасывает объект типа `ZeroDivide`, а не сам тип. Для этого используется конструктор по умолчанию структуры `ZeroDivide`.

Листинг 13.2. Обработка исключения (используя пустой тип)

```
#include <iostream>
using namespace std;
struct ZeroDivide {};
int divide(int a, int b)
{
  if (!b)
    throw ZeroDivide();
  return a / b;
}
int main()
{
  int n, m;
  try
  {
    // введите два целых числа через пробел
    cout << "Input two integers: ";
    cin >> n >> m;
    cout << "n/m = " << divide(n,m) << endl;
  }
  catch(ZeroDivide)
  {
    cout << "Zero divide" << endl;  // печатает: Zero divide
  }
  cout << "OK" << endl;

  return 0;
}
```

В заключение этого раздела скажем, что для выхода из блоков `try` и `catch` можно использовать инструкции передачи управления `return`, `break`, `continue` и `goto`. Невозможно передать управление внутрь блоков `try` или `catch`, используя инструкцию `goto`.

13.2. Обработка нескольких исключений

Для обработки исключений разных типов допускается использование нескольких последовательных блоков `catch`. Если внутри блока `try` произошел выброс исключения, то управление передается первому обработчику исключения, тип которого соответствует типу выброшенного исключения. Если такого обработчика нет, то вызывается функция `terminate`, которая в свою очередь вызывает функцию `abort`, которая и завершает исполнение программы. Подробно работа этими функциями рассмотрена в *гл. 36*.

После обработки исключения управление передается на первую инструкцию, которая следует за последним обработчиком исключения.

В листинге 13.3 приведена программа, в которой обрабатываются два типа исключений.

Листинг 13.3. Обработка исключений двух типов

```
#include <iostream>
using namespace std;
struct ZeroDivide {};
int divide(int a, int b)
{
  int r;  // остаток от деления
  if (!b)
    throw ZeroDivide();
  r = a % b;
  if (r)
    throw r;
```

```
  return a / b;
}
int main()
{
  int n, m;
  try
  {
    // введите два целых числа через пробел
    cout << "Input two integers: ";
    cin >> n >> m;
    cout << "n/m = " << divide(n,m) << endl;
  }
  catch(ZeroDivide)
  {
    cout << "Zero divide" << endl;  // печатает: Zero divide
  }
  catch(int &r)
  {
    cout << "Remainder = " << r << endl;  // печатает остаток
  }
  cout << "OK" << endl;

  return 0;
}
```

13.3. Перехват всех исключений

Для перехвата исключений любого типа используется обработчик исключения, который имеет следующий вид:

```
  catch(...)
  {
    // тело блока catch
  }
```

Очевидно, что если такой обработчик исключений используется, то он должен быть последним в последовательности обработчиков исключений. Иначе он будет перехватывать все исключения.

В листинге 13.4 приведена программа, в которой используется обработчик исключений любого типа.

Листинг 13.4. Использование обработчика исключений любого типа

```
#include <iostream>
using namespace std;
struct Zero {};
struct NonZero {};
int main()
{
  int n;

  try
  {
    cout << "Input integer: ";  // Введите целое число
    cin >> n ;

    if (n)
      throw NonZero();
    throw Zero();
  }
  catch(...)
  {
    cout << "Zero or NonZero exception" << endl;
  }
  cout << "OK" << endl;

  return 0;
}
```

13.4. Вложенные исключения

В блок `try` могут быть вложены другие блоки `try` со своими обработчиками исключений. В этом случае обработчик исключения ищется последовательно от самого внутреннего блока `try` к самому внешнему блоку `try`.

В листинге 13.5 приведен пример программирования вложенных блоков `try`.

Листинг 13.5. Программирование вложенных блоков `try`

```
#include <iostream>
using namespace std;

struct UnderZero {};
struct Zero {};
struct AboveZero {};

int main()
{
  int n;

  cout << "Input integer: ";
  cin >> n;
  try
  {
    if (n < 0)
      throw UnderZero();
    try
    {
      if (n > 0)
        throw AboveZero();
      throw Zero();
    }
    catch(AboveZero)
    {
      cout << "AboveZero exception." << endl;
    }
  }
  catch(UnderZero)
  {
    cout << "UnderZero exception." << endl;
  }
  catch(Zero)
  {
    cout << "Zero exception." << endl;
  }
```

```
  cout << "OK" << endl;

  return 0;
}
```

Блок `try` с обработчиками исключений может быть вложен в блок `catch`. В этом случае подходящий обработчик исключения ищется только внутри блока `catch`, в который вложен блок `try`.

Для передачи управления из обработчика исключений другому обработчику исключений, который находится выше текущего обработчика исключений в иерархии обработчиков, используется инструкция с выражением `throw` без объекта. В этом случае новое исключение не создается, а выбрасывается обрабатываемое исключение. Этот прием часто используется для освобождения программой захваченных ранее ресурсов.

В листинге 13.6 приведен пример передачи управления по иерархии обработчиков исключений.

Листинг 13.6. Передача управления по иерархии обработчиков исключений

```
#include <iostream>
using namespace std;
struct FreeMemory {};

int main()
{
  int* n;
  int* m;

  try
  {
    n = new int;
    cout << "Input n: ";  // введите целое число
    cin >> *n ;

    try
    {
      m = new int;
```

```
      cout << "Input m: ";  // введите целое число
      cin >> *m ;

      throw FreeMemory();   // нужно освободить память
    }
    catch(FreeMemory)
    {
      cout << "m = " << *m << endl;
      delete m;
      throw;  // нужно еще раз освободить память
    }
  }
  catch(FreeMemory)
  {
    cout << "n = " << *n << endl;
    delete n;
  }

  return 0;
}
```

Отметим, что использование инструкции `throw` без выражения в блоке `try` приводит к завершению программы.

13.5. Реализация исключений

Когда функция выбрасывает исключение, то управление передается обработчику исключений в функцию, которая еще не завершила свою работу. Это происходит потому, что поиск нужного обработчика исключения происходит по вложенным блокам `try`, которые могут находиться в разных функциях. В стеке хранятся адреса возврата и локальные переменные вызванных, но еще не завершивших свою работу, функций. Механизм обработки исключений реализован таким образом, что если обработчик выброшенного исключения не найден в текущей функции, то все локальные объекты, принадлежащие этой функции, уничтожаются, т. е. удаляются из стека, а затем поиск обработчика исключения продолжается в следующей функции по иерархии вызовов.

При удалении локальных объектов из стека для каждого из них вызывается деструктор. Такой процесс поиска нужного обработчика исключения называется *раскруткой стека* (stack unwinding).

Так как механизм обработки исключений работает во время исполнения программы, а не ее компиляции, то он замедляет работу программы. Поэтому обработкой исключений пользуются только в исключительных ситуациях для обработки серьезных ошибок в программе. Как правило, это такие ошибки, обработка которых невозможна в месте их появления, а решение о дальнейшей работе программы может приниматься на несколько функциональных уровней выше, чем блок, в котором произошла ошибка.

В заключение этого раздела отметим, что, для того чтобы уменьшить издержки на обработку исключения, в обработчике исключения нужно использовать ссылки.

13.6. Спецификация исключений

Спецификацией исключений называется перечисление в объявлении функции всех исключений, которые она может выбрасывать. Спецификации исключений должны совпадать в определении и во всех объявлениях одной и той же функции. Спецификация исключений имеет следующий синтаксис:

```
throw(список_типов)
```

Здесь `список_типов` включает, через запятую, типы всех исключений, которые может выбросить функция при своем исполнении. Например, предполагается, что функция:

```
void foo(void) throw(char*, int);
```

может выбросить только исключения типов `char*` и `int`.

Если список типов исключений пуст, то это означает, что функция не выбрасывает никаких исключений. Например:

```
void foo(void) throw();
```

Если же в объявлении функции спецификация исключений отсутствует, то это значит, что функция может выбросить исключение любого типа.

Если функция выбрасывает исключение, типа которого нет в спецификации, то вызывается функция:

```
void unexpected();
```

из стандартной библиотеки языка программирования C++, которая в свою очередь вызывает функцию `terminate`. Такие исключения называются *неожидаемыми* или *непредусмотренными*. Для обработки таких исключений можно установить свой обработчик с помощью функции `set_unexpected`, работа с которой описана в *разд. 36.5*.

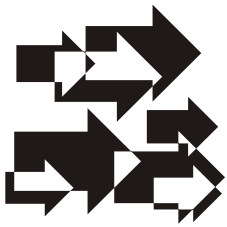
Спецификаторы исключений можно также задавать и при объявлении указателя на функцию. Например:

```
void (*pf)(void) throw(int, double);
```

Присваиваемое такому указателю значение должно указывать на функцию, которая может выбрасывать только такие исключения, которые присутствуют в спецификации исключений указателя на функцию. Например:

```
void  f() throw (int);
pf = f;
```

ГЛАВА 14



Классы

14.1. Введение

Классом называется множество объектов, имеющих общие свойства, и набор операций, допустимых над этими объектами. В языке программирования C++ объектами являются переменные. Поэтому классы являются типами, которые отличаются от встроенных типов данных тем, что позволяют определять операции, которые допускается выполнять над переменными, принадлежащими классу. Новые типы в языке программирования C определяются при помощи структур и объединений. Такой же подход к определению классов принят и в языке программирования C++, но, дополнительно, в блоке структуры или объединения можно определить функции, которые разрешается выполнять над переменными типа структуры или объединения.

Однако такой подход к определению класса имеет один существенный недостаток, а именно, т. к. доступ к полям структуры или объединения открыт, то над переменными, принадлежащими классу, можно выполнять также и операции, не принадлежащие этому классу. Для решения этой проблемы в языке программирования C++ для определения классов также используется ключевое слово `class`, которое служит для определения структуры, а точнее класса, доступ к членам которого закрыт. Возникает вопрос, если доступ к членам класса закрыт, то как их использовать? Для этого в языке программирования C++ определены спе-

цификаторы доступа `public`, `protected` и `private`, которые разрешают или ограничивают доступ к отдельным членам класса.

Теперь после этого краткого введения перейдем к техническим вопросам, касающимся работы с классами в языке программирования C++.

14.2. Определение класса

Синтаксически класс определяется следующим образом:

```
class|struct|union имя_класса
{
  // члены класса
};
```

При определении класса должно использоваться только одно из ключевых слов `class`, `struct` или `union`. Каждое из этих ключевых слов устанавливает различные режимы доступа к членам класса. При использовании ключевого слова `class` доступ ко всем членам класса закрыт, а при использовании ключевых слов `struct` или `union` — открыт. Чтобы открыть или закрыть доступ к определенным членам класса, используются спецификаторы доступа, которые будут рассмотрены в *разд. 14.4*.

Например, в листинге 14.1 определен контейнерный класс `IntBox`, который предназначен для хранения одного целого числа. Предположим, что это определение класса хранится в заголовочном файле IntBox.h. Отметим, что этот заголовочный файл содержит "часового", который исключает возможность повторного определения класса `IntBox` в одном исходном файле. В качестве этого "часового" выступает символическое имя `INT_BOX`.

Листинг 14.1. Определение класса `IntBox`

```
#if !defined INT_BOX
#define INT_BOX
struct IntBox
{
  // данные-члены класса
  bool empty;  // состояние контейнера
```

```
  int  room;   // место для хранения целого числа
  // функции-члены класса
  void put(int n);  // положить целое число в контейнер
  int  get();       // извлечь целое число из контейнера
  bool isEmpty();   // проверить, свободен ли контейнер
};
#endif
```

Допускается объявлять только имя класса, не определяя его членов. Такое объявление класса называется *предварительным*. В этом случае можно определять только указатели или ссылки на такой класс, которые не требуют инициализации.

Определение класса вводит область видимости, которой принадлежат имена членов класса. Определение функции-члена класса также принадлежит области видимости класса. Отсюда следует, что в теле функции-члена класса можно обращаться к другим членам этого класса просто по имени.

Функции-члены класса определяются как обычные функции. Но если такая функция определяется вне класса, то, для того чтобы показать, что она принадлежат классу, перед ее именем нужно записать имя класса и оператор разрешения области видимости (`::`).

Заметим, что определение функции, члена класса, также называется *методом* класса, а определение данных, членов класса — *атрибутами* класса.

Например, в листинге 14.2 определены функции-члены класса `IntBox`. Предположим, что эти определения хранятся в исходном файле IntBox.cpp.

Листинг 14.2. Реализация функций-членов класса `IntBox`

```
#include "IntBox.h"
void IntBox::put(int n) { room = n; empty = false; }
int  IntBox::get() { empty = true; return room; }
bool IntBox::isEmpty() { return empty; }
```

Отметим построение проекта, содержащего классы. Определение класса записывается в отдельный заголовочный файл, имя кото-

рого обычно совпадает с именем класса. Затем этот заголовочный файл вставляется, при помощи директивы `#include`, во все исходные файлы, которые используют этот класс. Реализация функций-членов класса обычно также записывается в отдельный исходный файл, имя которого совпадает с именем класса. Такой подход упрощает разработку больших программ.

14.3. Объекты, доступ к членам класса

После определения класса можно объявлять переменные, которые имеют тип этого класса, или, другими словами, принадлежат этому классу, или являются экземплярами этого класса. В языке программирования C++ экземпляры класса называются *объектами*. Каждый объект содержит свой набор атрибутов класса. После создания объектов класса над ними можно выполнять методы этого класса.

Для доступа к членам класса вне области видимости этого класса используются операторы `.` и `->`, как и в случае доступа к членам структуры.

Пример использования класса `IntBox` приведен в листинге 14.3, который содержит текст исходного файла main.cpp. Таким образом, весь проект содержит три файла: IntBox.h, IntBox.cpp, main.cpp. Кроме того, заметим, что перед использованием объекта типа `IntBox` нужно установить значение атрибута `empty` этого объекта в `true`.

Листинг 14.3. Пример использования класса `IntBox`

```
#include <iostream>
#include "IntBox.h"
using namespace std;
int main()
{
  IntBox box = {true};  // объявление объекта типа IntBox
  int n;
```

```
  if (box.isEmpty())
    box.put(10);
  if (!box.isEmpty())
    n = box.get();

  cout << n << endl;   // печатает 10

  return 0;
}
```

14.4. Указатель *this*

По умолчанию каждая не статическая функция-член класса имеет доступ к указателю с именем `this`, который указывает на объект, для которого эта функция вызвана. Здесь `this` является ключевым словом языка программирования C++. Можно сказать, что указатель `this` является неявным параметром каждой функции-члена класса. Указатель `this` имеет следующий тип:

```
имя_класса* const
```

В принципе к любому члену класса, принадлежащему текущему объекту, можно обратиться через указатель `this`. Например, в функции-члене класса `IntBox` можно написать так:

```
this -> empty = false;
```

Но, т. к. функции-члены класса имеют доступ ко всем членам класса по умолчанию, то указатель `this` используется в основном для возврата указателя или ссылки на объект, к которому применяется функция-член класса.

14.5. Спецификаторы доступа к членам класса

Для ограничения доступа к членам класса вне его области видимости используются спецификаторы доступа: `public`, `protected` и `private`, которые также называются *метками* и могут использоваться внутри класса произвольное число раз и в любом порядке.

Все члены класса, объявленные после метки `public` и расположенные до следующей метки, доступны как в классе, так извне класса.

Все члены класса, объявленные после метки `protected` и расположенные до следующей метки, доступны только внутри класса и внутри потомков класса.

Все члены класса, объявленные после метки `private` и расположенные до следующей метки, доступны только внутри класса.

По умолчанию все члены класса, объявленные после ключевого слова `class` до первой метки, имеют спецификацию доступа `private`, а объявленные после ключевых слов `struct` или `union` до первой метки — спецификацию доступа `public`.

Чтобы запретить пользователям доступ к атрибутам класса `IntBox`, этот класс должен быть определен следующим образом:

```
class IntBox
{
  bool empty;  // состояние контейнера
  int  room;   // место для хранения целого числа
public:
  IntBox();          // конструктор
  void put(int n);   // положить целое число в контейнер
  int  get();        // взять целое число из контейнера
  bool isEmpty();    // проверить, свободно ли место
};
```

Как видим, в этом случае клиенты класса имеют доступ к атрибутам класса только через методы этого класса или, как еще говорят, через интерфейс класса. Такой подход к проектированию структуры класса называется *инкапсуляцией*. Но в этом случае возникает проблема инициализации атрибута `empty` при определении объекта класса. Для этого используются специальные функции-члены класса, которые называются *конструкторами*. В нашем случае это функция `IntBox`, которая имеет следующее определение:

```
IntBox::IntBox() { empty = true; }
```

Более подробно конструкторы классов будут рассмотрены в *гл. 15*.

14.6. Друзья класса

При работе с классами может возникнуть следующая проблема. Допустим, что класс с закрытыми атрибутами определяет тип параметра функции. Возможна такая ситуация, что эта функция не может быть реализована, т. к. она не имеет доступа к атрибутам экземпляра класса. Чтобы разрешить эту проблему, функция должна быть объявлена как друг класса.

Второй причиной при определении друзей класса является желание реализовать более быстрые функции, которые не являются членами класса, но работают с атрибутами этого класса. Очевидно, что прямой доступ к атрибутам класса будет работать быстрее, чем доступ к этим же атрибутам через функции-члены класса.

Замечание

Отношение дружбы нарушает принцип инкапсуляции в объектно-ориентированном программировании. Поэтому друзей класса нужно выбирать осторожно.

Доступ к закрытым членам класса из другой функции допускается только в том случае, если она объявлена как друг класса. Объявление друга класса начинается с ключевого слова `friend` и может встречаться только внутри определения класса. Так как друзья класса не являются членами класса, то не имеет значения после какой из меток они объявлены. Друзьями класса могут быть объявлены как функции из пространства имен, так и функции-члены другого класса.

Определяются функции-друзья класса как обычные функции внутри или вне класса. Если функция определяется внутри класса, то это определение одновременно является и объявлением функции. В этом случае имя функции принадлежит тому же пространству имен, что и имя класса. Если функция друг класса определяется вне класса, то при этом ключевое слово `friend` не используется.

Например, объявим для класса `IntBox` две дружественные функции: `in` и `out`, которые соответственно предназначены для ввода целого числа в контейнер с консоли и для вывода целого числа из

контейнера на консоль. Тогда определение класса `IntBox` будет выглядеть следующим образом:

```
class IntBox
{
  bool empty;  // состояние контейнера
  int  room;   // место для хранения целого числа
public:
  IntBox();          // конструктор
  void put(int n);  // положить целое число в контейнер
  int  get();        // взять целое число из контейнера
  bool isEmpty();   // проверить, свободно ли место
  friend void in(IntBox& box);   // ввод числа с консоли
  friend void out(IntBox& box);  // вывод числа на консоль
};
```

Реализацию этих функций можно поместить как в исходный файл IntBox.cpp, так и в новый исходный файл. Выберем второй вариант и создадим исходный файл InOutBox.cpp. В листинге 14.4 приведен текст этого файла.

Листинг 14.4. Реализация функций-друзей класса

```
#include "IntBox.h"
#include <iostream>
using namespace std;

void in(IntBox& box)
{
  cin >> box.room;
  box.empty = false;
}
void out(IntBox& box)
{
  cout << box.room;
  box.empty = true;
}
```

В листинге 14.5 показано, как использовать эти функции для ввода целого числа с консоли в контейнер и вывода этого же целого числа на консоль из контейнера.

Листинг 14.5. Использование функций-друзей класса

```
#include "IntBox.h"
#include <iostream>
using namespace std;
int main()
{
  IntBox box;

  cout << "Input integer: ";
  in(box);
  out(box);
  cout << endl;

  return 0;
}
```

Друзьями класса могут быть также объявлены другие классы. В этом случае доступ к закрытым членам класса допускается для всех функций друга этого класса. Например, объявим для класса `IntBox` дружественный класс `PrintBox`, который содержит функцию для вывода на консоль целого числа, хранящегося в контейнере, не удаляя его оттуда. Тогда определение класса `IntBox` будет выглядеть следующим образом:

```
  class IntBox
  {
    bool empty;  // состояние контейнера
    int  room;   // место для хранения целого числа
  public:
    IntBox();          // конструктор
    void put(int n);  // положить целое число в контейнер
    int  get();        // взять целое число из контейнера
    bool isEmpty();   // проверить, свободно ли место

    friend void in(IntBox& box);   // ввод числа с консоли
    friend void out(IntBox& box);  // вывод числа на консоль

    friend struct PrintBox;  // печать содержимого контейнера
  };
```

Определение класса `PrintBox` поместим в заголовочный файл PrintBox.h, текст которого приведен в листинге 14.6.

Листинг 14.6. Определение дружественного класса

```
#include "IntBox.h"
#include <iostream>
using namespace std;

#if !defined PRINT_BOX
#define PRINT_BOX
struct PrintBox
{
  void print(IntBox& box);
};
#endif
```

Реализацию функции `print` из класса `PrintBox` поместим в исходный файл PrintBox.cpp, текст которого приведен в листинге 14.7.

Листинг 14.7. Реализация функции дружественного класса

```
#include "PrintBox.h"
void PrintBox::print(IntBox& box) { cout << box.room; }
```

В листинге 14.8 показано, как распечатывать содержимое контейнера при помощи функции `print`.

Листинг 14.8. Использование функции дружественного класса

```
#include "IntBox.h"
#include "PrintBox.h"
#include <iostream>
using namespace std;
int main()
{
  IntBox box;
  PrintBox prn;
```

```
  box.put(10);
  prn.print(box);  // печатает 10
  cout << endl;
  return 0;
}
```

При использовании друзей класса следует учитывать следующие правила:

- отношение дружбы не является симметричным, т. е. если A есть друг B, то из этого не следует, что B есть друг A;
- отношение дружбы не является транзитивным, т. е. если A друг B и B друг C, то из этого не следует, что A является другом C;
- отношение дружбы не наследуется, т. е. если класс A является другом класса C и предком класса B, то из этого не следует, что класс B является другом класса C.

14.7. Встроенные функции-члены класса

Если функция-член класса определена прямо в теле класса, то она считается *встроенной*. В этом случае ключевое слово `inline` можно не писать. Например, определим все функции класса `IntBox` как встроенные.

```
class IntBox
{
  bool empty;
  int  room;
public:
  IntBox() { empty = true; }
  void put(int n) { room = n; empty = false; }
  int  get() { empty = true; return room; }
  bool isEmpty() { return empty; }
};
```

В этом случае исходный файл IntBox.cpp нужно исключить из проекта.

Если при объявлении функции-члена класса используется ключевое слово `inline`, то определять функцию можно и вне тела класса, при этом слово `inline` можно опустить. Так как встроенные функции-члены класса должны быть определены в каждом исходном файле, где они вызываются, то определение этих функций следует поместить в тот же заголовочный файл, где описан класс.

14.8. Функции-члены класса с квалификаторами *const* и *volatile*

Функции-члены класса могут иметь квалификатор `const`, или `volatile`, или два этих квалификатора вместе. Эти квалификаторы записываются после имени функции и имеют следующее назначение.

Для константного объекта класса могут быть вызваны только те функции-члены класса, которые объявлены с квалификатором `const`. Для неконстантного объекта класса могут быть вызваны функции как с квалификатором `const`, так и без него. Поэтому функции-члены класса, которые не изменяют значения его атрибутов, нужно объявлять с квалификатором `const`.

Запрещено объявлять константную функцию-член класса, которая модифицирует члены класса прямо или косвенно через неконстантные функции-члены класса. Константную функцию-член класса можно перегружать неконстантной функцией с тем же списком параметров.

Учитывая это замечание, переопределим класс `IntBox` следующим образом:

```
class IntBox
{
  bool empty;
  int  room;
public:
  IntBox();       // инициализация пустого контейнера
  IntBox(int n);  // инициализация полного контейнера
```

```
  void put(int n);
  int  get();
  int  get() const;
  bool isEmpty() const;
};
```

Обратим внимание на следующих два момента. Во-первых, появился новый конструктор, который позволяет определять контейнер и сразу заносить в него число. Также появился новый метод `get` с квалификатором `const`, который позволит просматривать содержимое константного контейнера. Реализация этих новых функций выглядит следующим образом:

```
IntBox::IntBox(int n) { empty = false; room = n; }
int  IntBox::get() const { return room; }
```

А реализация функции `isEmpty()` не изменится. Теперь можно определять константные контейнеры для хранения целых чисел и определять их содержимое. Ниже приведен пример использования константного контейнера.

```
int main()
{
  const IntBox box(10);
  int n;

  if (!box.isEmpty())
    n = box.get();
  cout << n << endl;  // печатает 10
  return 0;
}
```

Для объекта класса, объявленного со спецификатором доступа `volatile`, могут быть вызваны только те функции-члены класса, которые также объявлены со спецификатором доступа `volatile`. Для объекта класса, объявленного без квалификатора `volatile`, могут быть вызваны функции как с квалификатором `volatile`, так и без него. Функцию-член класса с квалификатором `volatile` можно перегружать функцией-членом класса с тем же списком параметров, но без этого квалификатора.

Если в классе перегружены функции-члены с квалификаторами `const` и `volatile`, то при вызове такой функции для объекта класса, объявленного без квалификаторов `const` и `volatile`, компилятор выдаст ошибку неоднозначного выбора функции.

14.9. Данные-члены класса с квалификатором *mutable*

Чтобы разрешить модификацию члена класса, принадлежащего константному объекту, этот член класса должен быть объявлен с квалификатором `mutable`. Пример использования этого квалификатора приведен в листинге 14.9.

Листинг 14.9. Использование квалификатора `mutable`

```
#include <iostream>
using namespace std;
class counter
{
  mutable int n;  // счетчик обращений
public:
  counter() { n = 0; };
  int count() const { return ++n; }
};
int main()
{
  const counter n;
  cout << n.count() << endl;  // печатает 1

  return 0;
}
```

14.10. Статические члены класса

Статические члены класса имеют класс памяти `static` и являются глобальными объектами, которые принадлежат всему классу и существуют в единственном экземпляре.

Статические данные-члены класса только объявляются внутри класса. Они должны быть определены, а в случае констант и инициализированы вне класса. Пример работы со статическими данными-членами класса приведен в листинге 14.10.

Листинг 14.10. Работа со статическими данными-членами класса

```
#include <iostream>
using namespace std;

struct Demo
{
  static int n;
  static const int m;
};
int Demo::n;
const int Demo::m = 10;

int main()
{
  Demo::n = 20;
  cout << Demo::n << ' ' << Demo::m << endl;  // 20 10

  return 0;
}
```

Можно определять указатели на статические данные-члены класса, если они расположены в разделе `public`. Например:

```
  int* p = &Demo::n;
```

Статические функции-члены класса имеют доступ только к статическим данным-членам класса. Для того чтобы эти функции имели доступ к нестатическим членам класса, они должны получить адрес объекта класса как параметр. Пример работы со статическими функциями-членами класса приведен в листинге 14.11.

Листинг 14.11. Работа со статическими функциями-членами класса

```
#include <iostream>
using namespace std;
```

```
class Demo
{
  int n;
  static int m;
public:
  Demo() { n = 10; ++m; }
  static int getN(Demo& d) { return d.n; }
  static int getM() { return m; }
  int getSum() { return n + m; }
};
int Demo::m;
int main()
{
  Demo d;

  cout << d.getN(d) << endl;   // 10
  cout << d.getM() << endl;    // 1
  cout << d.getSum() << endl;  // 11

  return 0;
}
```

Статические функции-члены класса ведут себя аналогично глобальным функциям. Например, указателю на функцию можно присвоить адрес статической функции-члена класса.

```
int (*f)(void) = Demo::getM;
```

14.11. Указатели на нестатические члены класса

При определении указателей на нестатические члены класса следует учитывать как тип члена класса, так и тип самого класса.

Определение указателя на нестатический атрибут класса имеет следующий вид:

```
тип_атрибута имя_класса::*  имя_указателя
```

В свою очередь определение указателя на нестатический метод класса имеет следующий вид:

```
тип (имя_класса::* имя_указателя)(список_параметров)
```

Для доступа к нестатическим членам класса через указатели используются операторы:

- `.*` — для объектов или ссылок на объекты;
- `->*` — для указателей на объекты.

Эти операторы имеют ассоциативность слева направо, а их приоритет ниже приоритета унарных арифметических операторов, но выше приоритета бинарных арифметических операторов.

В листинге 14.12 приведен пример использования операторов доступа к нестатическим членам класса через указатели.

Листинг 14.12. Доступ к нестатическим членам класса через указатели

```
#include <iostream>
using namespace std;
struct Demo
{
  int n;
  int Count() { return n; };
};
int main()
{
  Demo d;
  Demo* dp = &d;

  int Demo::*ip;       // указатель на атрибут
  int (Demo::*fp)();   // указатель на метод

  // инициализация указателей
  ip = &Demo::n;
  fp = Demo::Count;

  // доступ к атрибуту через объект
  d.*ip = 10;
  cout << d.*ip << endl;     // печатает 10

  // доступ к атрибуту через указатель на объект
  dp->*ip = 20;
  cout << dp->*ip << endl;  // печатает 20
```

```
  d.*ip += 10;
  // доступ к методу через объект
  cout << (d.*fp)() << endl;    // печатает 30
  // доступ к методу через указатель на объект
  cout << (dp->*fp)() << endl;  // печатает 30

  return 0;
}
```

14.12. Вложенные классы

Класс, объявленный внутри другого класса, называется *вложенным* классом. Класс, в котором объявлен вложенный класс, будем называть *объемлющим* классом.

Функции-члены объемлющего класса не имеют прав доступа к закрытым членам своего вложенного класса. Также и функции-члены вложенного класса не имеют прав доступа к закрытым членам класса, внутри которого он объявлен. Чтобы предоставить такие права вложенному или объемлющему классу, нужно объявить такой класс другом соответственно объемлющего или вложенного класса.

В листинге 14.13 приведен пример объявления вложенного класса.

Листинг 14.13. Пример объявления вложенного класса

```
#include <iostream>
using namespace std;
class Outer
{
  class Inner
  {
    int n;
  public:
    Inner() { n = 0; }
    int count() { return ++n; }
  };
  Inner n;
```

```
public:
  int count() { return n.count(); }
};
int main()
{
  Outer n;
  cout << n.count() << endl;  // печатает 1

  return 0;
}
```

Для того чтобы объект, который имеет тип вложенного класса, можно было объявить вне объемлющего класса, вложенный класс должен быть объявлен в разделе открытых членов объемлющего класса. При этом, т. к. определение объемлющего класса вводит область видимости, при ссылке на имя вложенного класса вне этой области видимости оно должно квалифицироваться именем объемлющего класса.

В листинге 14.14 приведен пример объявления объекта, который имеет тип вложенного класса.

Листинг 14.14. Объявление объекта типа вложенного класса

```
#include <iostream>
using namespace std;

class Outer
{
public:
  class Inner
  {
    int n;
  public:
    Inner() { n = 0; }
    int count() { return ++n; }
  };
private:
  Inner n;
```

```
public:
  int count() { return n.count(); }
};

int main()
{
  Outer::Inner m;
  cout << m.count() << endl;  // печатает 1

  return 0;
}
```

14.13. Локальные классы

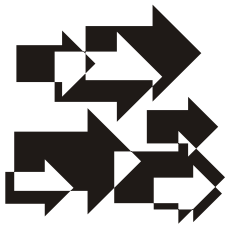
Класс, определенный внутри тела функции, называется *локальным*. Такой класс виден только в теле функции. Функции-члены локального класса должны определяться внутри самого класса.

В листинге 14.15 приведен пример определения локального класса.

Листинг 14.15. Определение локального класса

```
#include <iostream>
using namespace std;

int inc(int n)
{
  struct Inc
  {
    int inc(int n) { return ++n; }
  };
  Inc i;
  return i.inc(n);
}

int main()
{
  cout << inc(10) << endl;  // печатает 11
  return 0;
}
```

ГЛАВА 15



Конструкторы и деструкторы

15.1. Конструктор класса

Конструктором называется функция-член класса, имя которой совпадает с именем этого класса. Вызов конструктора указывает компилятору, что создается новый объект класса. Вызов конструктора может выполняться как явно, так и неявно. При определении объекта вызов конструктора происходит неявно. В конструкторе можно использовать данные и функции-члены класса, а также указатель `this`. Тело конструктора используется для инициализации данных-членов класса. Конструкторы должны удовлетворять следующим требованиям:

- конструктор не имеет типа возвращаемого значения и, следовательно, не должен возвращать значения;
- конструктор не может быть объявлен с квалификаторами `const` или `volatile`;
- конструктор не может быть объявлен со спецификатором `static`;
- конструктор не может быть объявлен со спецификатором `virtual`.

Из первого требования следует, что для обработки ошибок в конструкторе должны использоваться исключения. Хотя следует заметить, что возвращать значения из конструктора можно через параметры-ссылки.

Кроме того, отметим, что классы, содержащие конструкторы, не могут быть членами объединений.

15.2. Список инициализации

Если членом класса является константа, ссылка или объект другого класса, то инициализация такого члена класса невозможна в теле конструктора. В этих случаях используется список инициализации данных-членов класса, который записывается после списка параметров конструктора и отделяется от него символом :. Список инициализации должен содержать вызовы конструкторов инициализируемых объектов. В списке инициализации также могут инициализироваться и другие данные-члены класса.

В листинге 15.1 приведен пример инициализации данных-членов класса, используя список инициализации.

Листинг 15.1. Использование списка инициализации

```
#include <iostream>
using namespace std;
class Demo
{
  const int a;
  int b, c;
public:
  Demo(int _b, int _c): a(0), b(_b) { c = _c; }
  int getSum() { return a + b + c; }
};
int main()
{
  Demo d(1, 2);
  cout << d.getSum() << endl;  // печатает 3

  return 0;
}
```

В принципе программа работает быстрее, если данные-члены класса инициализируются в списке инициализации, а не в теле

конструктора. Так как в последнем случае перед передачей управления телу конструктора для каждого атрибута класса вызывается конструктор по умолчанию этого атрибута.

В заключение этого раздела заметим, что конструкторы данных-членов класса вызываются в порядке объявления этих данных в классе, а не в порядке их появления в списке инициализации.

15.3. Конструктор по умолчанию

Конструктором по умолчанию называется конструктор, который можно вызвать без аргументов. Конструктор по умолчанию можно определить двумя способами:

- без параметров;
- со списком параметров, в котором все параметры имеют значения по умолчанию.

Класс может иметь только один конструктор по умолчанию. Если класс совсем не имеет конструкторов и не содержит ссылок и констант в качестве членов класса, то конструктор по умолчанию генерируется компилятором. Этот конструктор имеет следующий прототип:

```
public: inline имя_класса();
```

Такой конструктор вызывает сначала конструкторы по умолчанию базовых классов, а затем конструкторы по умолчанию для данных-членов класса.

В листинге 15.2 приведен пример вызова конструкторов по умолчанию.

Листинг 15.2. Вызов конструкторов по умолчанию

```
#include <iostream>
using namespace std;
struct A
{
  int a;
  A(int _a = 20): a(_a) { cout << "a = " << a << endl; }
};
```

```
struct B
{
  int b;
  B(int _b = 10): b(_b) { cout << "b = " << b << endl; }
};
struct C: public B  // C наследуется от B
{
  A a;              // C содержит A
};
int main()
{
  C c;    // печатает: b = 10, a = 20
  return 0;
}
```

Если класс имеет хотя бы один конструктор, то конструктор по умолчанию не генерируется компилятором и его должен определить разработчик класса.

Чтобы запретить использование конструктора по умолчанию, его нужно поместить в `private` раздел класса. Это предотвращает возможность создания неинициализированных объектов.

15.4. Конструктор копирования

Конструктором копирования называется конструктор, первый параметр которого является ссылкой на объект класса, которому принадлежит конструктор, а остальные параметры либо отсутствуют, либо имеют значения, заданные по умолчанию. Конструктор копирования используется компилятором для инициализации объекта другим объектом этого же класса. Если конструктор копирования не определен, то компилятор сам генерирует конструктор копирования по умолчанию, который имеет следующий прототип:

```
public: inline имя_класса(const имя_класса&);
```

Выполняет копирование данных из объекта аргумента конструктора в конструируемый объект.

Отсюда следует, что если в конструкторе копирования выполняется тривиальное копирование данных-членов класса, то такой конструктор копирования определять не нужно.

Ниже приведена реализация конструктора копирования для демонстрационного класса, который требует определения конструктора копирования.

```
class Demo
{
   unsigned size;
public:
   int* p;
   // конструкторы
   Demo(unsigned n): size(n), p(new int[size]) {}
   Demo(const Demo& d);
   // деструктор
   ~Demo() { delete[] p; }
};
// реализация конструктора копирования
Demo::Demo(const Demo& d): size(d.size), p(new int[size])
{
  for (unsigned i = 0; i < size; ++i)
    p[i] = d.p[i];
}
```

Конструктор копирования вызывается явно в случае явной инициализации объекта другим объектом этого же класса. Например:

```
  Demo d(3);
  d.p[0] = 0;
  d.p[1] = 1;
  Demo c(d);  // вызывается конструктор копирования
```

Конструктор копирования вызывается неявно в трех случаях:

- при передаче объекта в тело функции по значению;
- при возврате объекта из функции по значению;
- при инициализации объекта другим объектом этого же класса при помощи оператора присваивания.

Например, в следующем примере вызывается конструктор копирования, а не оператор присваивания:

```
Demo d(3);
d.p[0] = 0;
d.p[1] = 1;
Demo c = d;  // вызывается конструктор копирования
```

Отсюда следует, что нетривиальный конструктор копирования нужно обязательно определить, иначе возможны ошибки во время исполнения программы.

Инициализацию через копирование можно запретить, если поместить конструктор копирования в раздел `private`.

15.5. Явный вызов конструкторов

Конструктор может вызываться явно в двух случаях:

- в списке инициализации;
- в выражении.

Во втором случае конструктор рассматривается как оператор, возвращающий анонимный константный объект соответствующего типа. Например:

```
int n = 1 + int(1.7);  // n = 2
```

Отсюда следует, что конструкторы могут использоваться в выражения для явного преобразования типов данных.

Конструктор с одним параметром рассматривается компилятором как определенный пользователем оператор преобразования типов. То есть компилятор может использовать этот конструктор для неявного преобразования типов в выражении.

В листинге 15.3 приведен пример использования конструктора с одним параметром как оператора преобразования типа.

Листинг 15.3. Использование конструктора в качестве оператора преобразования типа

```
#include <iostream>
using namespace std;
```

```
struct Demo
{
  int n;
public:
  Demo(int _n = 0): n(_n) {}
  operator+=(const Demo& d) { n += d.n; }
};

int main()
{
  Demo d;
  d += 10;
  cout << d.n << endl;

  return 0;
}
```

Чтобы запретить такое использование конструктора с одним параметром, этот конструктор объявляется как *явный* при помощи спецификатора `explicit`. Например:

```
struct Demo
{
  int n;
public:
  explicit Demo(int _n = 0): n(_n) {}
  operator+=(const Demo& d) { n += d.n; }
};
```

Тогда приведенный ниже оператор присваивания вызовет ошибку компиляции.

```
  Demo d;
  d += 10;  // ошибка компиляции
```

15.6. Деструкторы

Деструктор — это функция-член класса, имя которой начинается с символа ~, за которым без пробелов следует имя класса. Деструктор вызывается компилятором *неявно при уничтожении*

объекта. Деструктор предназначен для освобождения ресурсов, захваченных во время жизни объекта.

Например, в классе `Demo` из *разд. 15.4* деструктор был определен следующим образом:

```
~Demo() { delete[] p; }  // деструктор
```

Этот деструктор освобождает память, распределенную объекту в конструкторе.

Если ресурсы освобождать не нужно, то деструктор не пишется. В этом случае компилятор генерирует деструктор по умолчанию, который выполняет следующие действия: сначала вызывает деструкторы данных-членов класса, а затем — деструкторы базовых классов.

Деструктор должен удовлетворять следующим требованиям:

- деструктор не может иметь параметров;
- деструктор не может возвращать значения;
- деструктор не может быть объявлен с квалификаторами `const` или `volatile`;
- деструктор не может быть объявлен со спецификатором `static`.

Так как деструктор не может иметь параметров, то его нельзя перегрузить. Следовательно, деструктор должен быть всегда один.

Деструкторы вызываются неявно в следующих случаях:

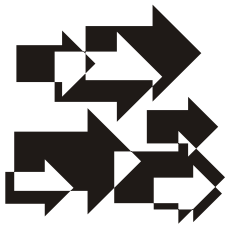
- для `static` объектов при завершении программы;
- для локальных объектов при выходе из блока, в котором эти объекты определены;
- при применении оператора `delete` к указателю на объект;
- при раскрутке стека во время обработки исключения.

Деструктор вызывается явно при уничтожении объекта, который был создан размещающим оператором `new`. В других случаях вызывать деструктор явно не нужно, т. к. в этом случае объект не уничтожается и при выходе из блока деструктор будет вызван еще раз. Чтобы удалить объект до выхода из блока, нужно поместить этот объект в искусственный блок.

15.7. Исключения в деструкторах

Если во время раскрутки стека *(см. разд. 13.5)* выброшено новое исключение, то механизм обработки исключений вызывает функцию `terminate`, которая завершает программу посредством вызова функции `abort`. Поэтому выбрасывать исключения из деструкторов не рекомендуется, т. к. они могут вызываться во время раскрутки стека. Чтобы определить, обрабатывается ли в данный момент исключение, можно использовать функцию `uncaught_exception`, которая определена в заголовочном файле exception. Использование этой функции рассмотрено в *разд. 36.6*.

ГЛАВА 16



Перегрузка операторов

16.1. Общие правила

В языке программирования C++ *операторы* рассматриваются как функции, которые имеют следующий прототип:

```
тип operator имя_оператора(список_параметров);
```

Здесь `operator` — ключевое слово. Поэтому операторы, так же как и функции, можно перегружать. Перегружаются почти все операторы, исключая нижеприведенные случаи:

- нельзя перегружать следующие операторы:
  - `.` — точка, оператор доступа к члену класса;
  - `.*` — оператор доступа к члену класса через указатель;
  - `::` — оператор разрешения области видимости;
  - `:?` — условный оператор;
- нельзя перегружать препроцессорные операторы:
  - `#` — преобразование в строку;
  - `##` — сцепление строк;
- нельзя перегружать операторы `sizeof` и `typeof`;
- нельзя перегружать операторы преобразования типов данных: `static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast`.

Перегрузка операторов должна удовлетворять следующим требованиям:

- нельзя вводить новых операторов;
- не допускается перегрузка операторов со встроенными типами данных в качестве параметров, тип, по крайней мере, одного параметра перегруженного оператора должен быть классом;
- нельзя изменять количество параметров оператора;
- ассоциативность перегруженных операторов не изменяется;
- приоритеты перегруженных операторов не изменяются;
- оператор не может иметь аргументов по умолчанию, за исключением оператора вызова функции `()`;
- оператор не может иметь неопределенного количества параметров, за исключением оператора вызова функции `()`.

Перегруженные операторы разрешается определять как члены класса и как функции не члены класса. Один оператор может быть перегружен несколько раз. Если оператор перегружен как член класса, то он не должен иметь спецификатор `static` и его первым операндом по умолчанию является объект класса, для которого вызывается этот оператор. В данном случае преобразование типа первого аргумента такого оператора к типу первого параметра не производится в силу невозможности его эффективной реализации. При выборе способа перегрузки оператора нужно учитывать его семантику, т. е. смысл. Хорошо бы при перегрузке операторов принимать во внимание то, что они в основном выполняют алгебраические операции над объектами и поэтому сохранять алгебраическую природу этих операторов.

Перегрузку унарных и бинарных операторов будем рассматривать на примере класса `Bool`, определение которого приведено в листинге 16.1.

Листинг 16.1. Определение класса четных чисел

```
#include <iostream>

#if !defined BOOL
#define BOOL
```

```
class Bool
{
  int n;
public:
  // конструкторы
  Bool() { n ? n = 1 : n = 0; }
  Bool(int _n) : n(_n ? 1 : 0) {}
  // унарные операторы логического сложения и умножения
  Bool& operator +=(const Bool& b);
  Bool& operator *=(const Bool& b);
  // оператор преобразования типа
  operator bool(void) const;
  // унарный логический оператор отрицания
  friend Bool operator !(const Bool& b);
  // бинарные операторы логического сложения и умножения
  friend Bool operator +(const Bool& b1, const Bool& b2);
  friend Bool operator *(const Bool& b1, const Bool& b2);
  // бинарные операторы сравнения
  friend Bool operator ==(const Bool& b1, const Bool& b2);
  friend Bool operator !=(const Bool& b1, const Bool& b2);
  // операторы ввода/вывода
  friend std::istream& operator >>(std::istream& in, Bool& b);
  friend std::ostream& operator <<(std::ostream& out,
                                   const Bool& b);
};

#endif
```

Реализация перегруженных операторов будет приведена в следующих разделах. При этом заметим, что большинство из рассматриваемых операторов имеют простую реализацию. Поэтому их можно определять прямо в теле класса как встроенные функции.

Отметим, что перегруженные операторы могут вызываться двумя способами: явно, как функции, или неявно, используя операторную запись. Например, над объектами типа `Bool` можно проводить следующие операции:

```
  Bool a(0), b(1), c(0);
  c.operator +=(a);      // явный вызов оператора +=
```

```
c += b;                 // неявный вызов оператора +=
c = operator +(a, b);   // явный вызов оператора +
c = a + b;              // неявный вызов оператора +
```

16.2. Унарные операторы

Префиксные унарные операторы могут быть перегружены как нестатические члены класса без параметров:

```
тип operator @();
```

или как операторы (не члены класса) с одним параметром:

```
тип operator @(параметр);
```

Здесь `@` обозначает один из следующих унарных операторов:

```
&  *  +  -  ~  !
```

При перегрузке выбирают тот вариант, который соответствует смыслу оператора.

Например, возможны следующие два варианта перегрузки унарного оператора отрицания как члена класса `Bool`:

```
Bool& operator !() { n ? n = 0 : n = 1; return *this; }
Bool operator !() const { return Bool(n ? 0 : 1); }
```

Выбираем второй вариант, так как он не изменяет логическое значение объекта, к которому применяется оператор `!`. Это согласуется с семантикой этого оператора при работе с логическими значениями.

Унарный оператор отрицания можно также перегрузить как дружественную функцию класса:

```
Bool operator !(const Bool& b) { return Bool(b.n ? 0 : 1); }
```

Такая реализация более предпочтительна, чем реализация унарного отрицания как члена класса, так как она позволяет преобразование типа параметра `b`. Эту реализацию и оставляем для класса `Bool`. Такой же подход используем и для остальных префиксных унарных операторов.

После перегрузки унарного отрицания над логическими значениями можно выполнять следующие операции:

```
Bool a(0), b(0);
a = !b;  // a = 1
```

Можно оставить два определения префиксного унарного оператора: как члена и как друга класса. В этом случае при вызове унарного оператора компилятор выбирает тот вариант, для которого преобразования типа считается наилучшим.

16.3. Оператор присваивания

Оператор присваивания может быть перегружен только как нестатический член класса. Перегруженный оператор присваивания должен иметь следующий прототип:

```
имя_класса& operator =(const имя_класса&);
```

При реализации оператора присваивания должна проверяться возможность присваивания объекта самому себе. Например, перегрузим оператор присваивания для класса `Bool`:

```
Bool& operator =(const Bool& b)
{
  if (&b != this)
    n = b.n;
  return *this;
}
```

Теперь можно выполнять присваивание логических значений, например:

```
Bool a(0), b(1);
a = b;  // a = 1
```

Если оператор присваивания не определен в классе, то компилятор генерирует оператор присваивания по умолчанию, который выполняет по членное копирование атрибутов класса. Если при присваивании объектов класса дополнительно к копированию атрибутов требуется выполнить еще какие-то действия, то оператор присваивания нужно обязательно перегрузить. Так как для класса `Bool` оператор присваивания выполняет только почленное копирование атрибутов, то его определение можно опустить.

Оператор присваивания и конструктор копирования обычно содержат много общего кода. Поэтому в этом случае часто пишут отдельную, закрытую функцию, которая реализует этот код и вызывается как в операторе присваивания, так и конструкторе копирования.

В листинге 16.2 приведен пример класса, для которого нужно обязательно перегрузить оператор присваивания. Этот класс является расширением класса `Demo`, приведенного в *разд. 15.4*.

Листинг 16.2. Определение класса с перегруженным оператором присваивания

```
#if !defined DEMO
#define DEMO
class Demo
{
  unsigned size;
  void copy(const Demo& d);  // функция копирования
public:
  int* p;
  // конструкторы
  Demo(unsigned n): size(n), p(new int[size]) {}
  Demo(const Demo& d);
  // оператор присваивания
  Demo& operator =(const Demo& d);
  // деструктор
  ~Demo() { delete[] p; }
};
#endif
```

В листинге 16.3 приведена реализация конструктора копирования и оператора присваивания для класса `Demo`.

Листинг 16.3. Реализация оператора присваивания

```
#include "Demo.h"
// реализация функции копирования объектов
void Demo::copy(const Demo& d)
```

```
{
  for (unsigned i = 0; i < size; ++i)
    p[i] = d.p[i];
}
// реализация конструктора копирования
Demo::Demo(const Demo& d): size(d.size), p(new int[size])
{
  copy(d);
}
// реализация оператора присваивания
Demo& Demo::operator =(const Demo& d)
{
  if (&d != this)
  {
    delete[] p;  // удаляем старый массив
    size = d.size;
    p = new int[size];
    copy(d);     // копируем
  }
  return *this;
}
```

Если посредством оператора присваивания инициализируется новый объект, то вместо вызова конструктора по умолчанию и оператора присваивания сразу вызывается конструктор копирования.

16.4. Составные операторы присваивания

Составные операторы присваивания также могут быть перегружены только как нестатические члены класса. Перегруженные составные операторы присваивания имеют следующий прототип:

```
имя_класса& operator @(const имя_класса&);
```

Символ `@` обозначает составной оператор присваивания. Как видим, этот прототип аналогичен прототипу простого оператора

присваивания. Очевидно, что при реализации составных операторов присваивания не нужно проверять на присваивание объекта самому себе.

Например, приведем реализацию одного из составных операторов присваивания, перегруженных в классе `Bool`:

```
Bool& Bool::operator +=(const Bool& b)
{
  n ? n : n = b.n;
  return *this;
}
```

16.5. Оператор индексирования *[ ]*

Оператор индексирования `[]` перегружается только как нестатический член класса с одним параметром. Перегружается этот оператор в основном для контейнерных классов, чтобы обеспечить доступ к отдельным элементам класса по их индексу. При перегрузке оператора присваивания следует учитывать, что он может использоваться как для константного, так и для не константного объекта. Например:

```
  struct BadIndex {};
  class Coord
  {
    int x, y;
  public:
    Coord() {}
    Coord(int _x, int _y): x(_x), y(_y) {}
    int& operator [](const int& i)
    {
      switch (i)
      {
      case 0: return x;
      case 1: return y;
      default: throw BadIndex();
      }
    }
```

```
    const int& operator [](const int& i) const
    {
      switch (i)
      {
      case 0: return x;
      case 1: return y;
      default: throw BadIndex();
      }
    }
  };
```

16.6. Оператор вызова функции

Оператор вызова функции может быть перегружен только как нестатический член класса. Перегруженный оператор вызова функции `()` должен иметь следующий прототип:

```
тип operator ()(список_параметров);
```

где количество параметров может быть неопределенным, а также допускается определять значения параметров по умолчанию. Вызывается оператор вызова функции путем применения списка аргументов к объекту класса, в котором он определен. Так как в этом случае объект может использоваться как функция, то он называется *объектом-функцией* или *функциональным объектом*.

В листинге 16.4 приведен пример перегрузки оператора вызова функции.

Листинг 16.4. Перегрузка оператора вызова функции

```
#include <iostream>
using namespace std;
struct Inc
{
  int& operator()(int& n) { return ++n; }
};
int main()
{
  Inc inc;
  int n = 1;
```

```
  inc(n);
  cout << inc(n) << endl;     // печатает 3

  return 0;
}
```

16.7. Оператор доступа к членам класса

Оператор доступа к членам класса `->` должен быть перегружен как нестатический член класса без параметров, а также обеспечивать семантику указателя потому, что для объекта `x` выражение `x->m` интерпретируется компилятором как:

```
(x.operator->()) -> m
```

Перегрузка оператора `->` используется для создания указателей на объекты с дополнительными возможностями.

Например, в листинге 16.5 реализован класс `ptr`, который не позволяет создавать неинициализированные указатели на объекты типа `int`. Кроме того, в этой программе приведены примеры перегрузки операторов определения адреса и разыменования.

Листинг 16.5. Перегрузка операторов доступа к членам класса, определения адреса и разыменования

```
#include <iostream>
using namespace std;
struct Demo
{
  int n;
  int what() { return n; }
};
class ptr
{
  Demo* p;
public:
  ptr(Demo& d): p(&d) {}
```

```
  Demo* operator ->() const { return p; }
  Demo* operator &() const { return p; }
  Demo& operator *() const { return *p; }
};
int main()
{
  Demo d = {0};
  ptr p(d);

  (*p).n += 10;
  cout << (&p)->n << endl;     // печатает 10
  cout << p->what() << endl;   // печатает 10

  return 0;
}
```

16.8. Операторы инкремента и декремента

Операторы инкремента и декремента могут быть перегружены как члены класса без аргументов или как не члены класса с одним аргументом. Для того чтобы префиксные операторы ++ и -- отличать от соответствующих постфиксных операторов, в объявлении последних вводят дополнительный фиктивный параметр типа `int`.

В листинге 16.6 приведен пример перегрузки префиксного и постфиксного операторов инкремента.

Листинг 16.6. Перегрузка префиксного и постфиксного операторов инкремента

```
#include <iostream>
using namespace std;
class Count
{
  int n;
public:
  Count(int _n): n(_n) {}
```

```
  Count& operator ++()     // префиксный ++
  {
    ++n;
    return *this;
  }
  Count operator ++(int)  // постфиксный ++
  {
    ++n;
    return Count(n - 1);
  }
  int get() { return n; }
};

int main()
{
  Count c(0);
  cout << ((++c)++).get() << endl;  // печатает 1
  cout << c.get() << endl;          // печатает 2

  return 0;
}
```

16.9. Бинарные операторы

Бинарные операторы могут быть перегружены как нестатические члены класса с одним параметром:

```
тип operator @(параметр);
```

Или как операторы не члены класса с двумя параметрами.

```
тип operator @(параметр_1, парметр_2);
```

Символ `@` обозначает один из следующих бинарных операторов:

```
+  -  *  .  %  ==  <=  >=  !=  &&  ||  <<  >>
```

При перегрузке бинарных операторов следует учитывать симметричность их параметров. Если аргументами бинарного оператора являются объекты одного класса, то этот оператор лучше перегружать как дружественный оператор с двумя параметрами.

Например, для класса `Bool` оператор сравнения на равенство следует перегрузить следующим образом:

```
Bool operator ==(const Bool& b1, const Bool& b2)
{
  return Bool(b1.n ? (b2.n ? 1 : 0) : (b2.n ? 0 : 1));
}
```

Если же оператор сравнения на равенство будет определен как член класса, то первый параметр этого оператора будет иметь тип `const Bool* const`, а второй параметр — тип `const Bool&`, что нарушает симметричность параметров и требует дополнительного преобразования типов.

16.10. Перегрузка операторов >> и << для ввода и вывода

Перегруженные операторы ввода/вывода >> и << должны определяться как дружественные операторы класса, которые имеют следующий прототип:

```
  friend istream& operator >>(istream&, имя_класса&);
  friend ostream& operator <<(ostream&, const имя_класса&);
```

Это позволяет избежать путаницы с общепринятым написанием этих операторов. Например, перегрузим операторы >> и << соответственно для ввода и вывода данных типа `Bool`:

```
std::istream& operator >>(std::istream& in, Bool& b)
{
  in >> b.n;
  b.n ? b.n = 1 : b.n = 0;
  return in;
}
std::ostream& operator <<(std::ostream& out, const Bool& b)
{
  return out << b.n;
}
```

Теперь эти операторы можно использовать следующим образом:

```
Bool a;

cin >> a;
cout << a << endl;
```

16.11. Операторы преобразования типов (конвертеры)

Конвертер — это функция-член класса, которая преобразует тип объекта класса в некоторый другой тип. Конвертер имеет следующий прототип:

```
operator тип();
```

Здесь `тип` задает тип данных, к которому приводится объект. Этот тип может быть как встроенным типом данных, так и типом данных, определенным пользователем. Конвертер может вызываться как явно, так и неявно — при преобразованиях типов данных.

Например, определим для класса `Bool` оператор преобразования в тип `bool`:

```
Bool::operator bool(void) const { return n ? true : false; }
```

После определения этого оператора значения типа `Bool` можно использовать в выражениях со значениями типа `bool`. Например:

```
bool a;
Bool b = 0;

a = true && b;  // a = false
```

16.12. Операторы *new* и *delete*

Перегрузка операторов `new` и `delete` не подчиняется общим правилам перегрузки операторов. Эти операторы могут быть перегружены только как статические члены класса, даже если спецификатор `static` не используется. Перегруженные операторы `new` и

delete не могут быть виртуальными. Эти же замечания справедливы и для перегруженных операторов new[] и delete[].

Перегруженный оператор new должен иметь следующий прототип:

```
[static] void* operator new(size_t[, параметры]);
```

То есть перегруженный оператор new должен возвращать значение типа void* и его первый параметр должен иметь тип size_t. Значение первого параметра передается перегруженному оператору new средой исполнения неявно и содержит размер памяти, необходимой для размещения объекта класса. После первого параметра могут следовать другие параметры, значения которых должны быть переданы перегруженному оператору new явно.

В прототипе перегруженного оператора new можно также указать спецификацию выбрасываемых этим оператором исключений.

Перегруженный оператор delete должен иметь следующий прототип:

```
[static] void operator delete(void*[, параметры]);
```

То есть перегруженный оператор delete не должен возвращать значения и его первый параметр должен иметь тип void*. Значение первого параметра передается перегруженному оператору delete явно и должно содержать адрес памяти, предварительно распределенной для объекта перегруженным оператором new. После первого параметра могут следовать другие параметры.

В прототипе перегруженного оператора delete можно также указать спецификацию выбрасываемых этим оператором исключений.

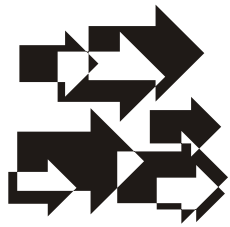
В листинге 16.7 приведен пример перегрузки операторов new и delete как членов класса.

Листинг 16.7. Перегрузка операторов new и delete

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
class Demo
{
public:
  static void* p;

  void *operator new(size_t n);
  void operator delete(void*);
};
void *Demo::operator new(size_t n)
{
  if (p)
    return p;
  else
  {
    p = malloc(n);
    return p;
  }
}
void Demo::operator delete(void*)
{
  if (p)
  {
    free(p);
    p = NULL;
  }
}

void* Demo::p = NULL;

int main()
{
  Demo *d;

  cout << !Demo::p << endl;  // 1
  d = new Demo();
  cout << !Demo::p << endl;  // 0
  d = new Demo();
  cout << !Demo::p << endl;  // 0
  delete d;
  cout << !Demo::p << endl;  // 1

  return 0;

}
```

ГЛАВА 17



Наследование классов

17.1. Определение наследования

Наследование — это отношение между классами, которое позволяет классу потомку использовать члены одного или нескольких классов предков. В языке программирования C++ класс потомок называется *производным* (derived) *классом* от наследуемых классов, а классы предки называются *базовыми классами*. Из определения наследования видно, что оно решает задачу повторного использования кода. В производном классе нет необходимости заново определять атрибуты и методы, которые наследуются им от базового класса. Синтаксически наследование между двумя классами определяется следующим образом:

```
class имя_производного_класса: public имя_базового_класса
{
  // тело производного класса
};
```

При таком наследовании в производном классе возможен доступ к членам базового класса, которые объявлены в разделах `public` и `protected` базового класса. В производном классе запрещен доступ к членам базового класса, которые объявлены в разделе `private` базового класса.

В листинге 17.1 приведен пример определения наследования классов и доступа к членам базового класса через объект производного класса.

Листинг 17.1. Наследование классов

```
#include <iostream>
using namespace std;

class Base
{
  int n;
public:
  Base(): n(0) {}
  int inc() { return ++n; }
};
class Derived: public Base
{
public:
  int _inc() { return inc(); }
};

int main()
{
  Derived d;

  cout << d.inc() << endl;   // печатает: 1
  cout << d._inc() << endl;  // печатает: 2

  return 0;
}
```

Так как производный класс содержит члены базового класса, то естественно допустить использование объектов производного класса там, где требуются объекты базового класса. В языке программирования C++ это обеспечивается тем, что указатель и ссылка на базовый класс совместимы по типам соответственно с указателем и ссылкой на производный класс.

В листинге 17.2 приведен пример использования объектов производного класса вместо объектов базового класса через указатели и ссылки на базовый класс.

Листинг 17.2. Использование объектов производного класса через указатели и ссылки на объекты базового класса

```
#include <iostream>
using namespace std;

class Base
{
public:
  int one() { return 1; }
};
class Derived: public Base
{
public:
  int two() { return 2; }
};

int foo(Base* b) { return b->one(); }
int foo(Base& b) { return b.one(); }

int main()
{
  Derived d;
  cout << foo(&d) << ' ' << foo(d) << endl;  // печатает: 1 1

  return 0;
}
```

Если атрибут или метод производного класса имеет такое же имя, как соответственно атрибут или метод базового класса, то этот член производного класса *скрывает* (hides) соответствующий член базового класса. В этом случае для доступа к скрытому члену базового класса нужно использовать оператор разрешения области видимости с именем базового класса.

В листинге 17.3 приведен пример доступа к скрытым членам базового класса.

Листинг 17.3. Доступ к скрытым членам базового класса

```
#include <iostream>
using namespace std;
```

```
class Base
{
public:
  int what() { return 10; }
};
class Derived: public Base
{
public:
  int what() { return 20; }
};

int main()
{
  Derived d;

  cout << d.Base::what() << endl;  // 10
  cout << d.what() << endl;        // 20

  return 0;
}
```

17.2. Доступ к членам, наследуемым от базового класса

При определении производного класса можно также определить права доступа к членам, наследуемым от базового класса. Для этого в определении производного класса перед именем базового класса записывается один из спецификаторов доступа `public`, `protected` или `private`.

Если класс наследуется как `public`, то права доступа к `public` и `protected` членам, наследуемым от базового класса, в производном классе не изменяются.

Если класс наследуется как `protected`, то все `public` и `protected` члены базового класса становятся `protected` членами производного класса.

Если класс наследуется как `private`, то все `public` и `protected` члены, наследуемые от базового класса, становятся `private` членами производного класса.

В любом случае `private` члены базового класса становятся `private` членами производного класса, причем доступ к этим членам закрыт в производном классе. Чтобы открыть этот доступ, производный класс должен быть объявлен другом базового класса.

Если спецификатор доступа перед базовым классом опущен, то по умолчанию для классов он устанавливается в `private`, а для структур — в `public`.

В табл. 17.1 суммированы правила действия спецификатора доступа для членов базового класса.

Таблица 17.1. *Спецификаторы доступа для членов базового класса*

Спецификатор доступа	Доступ в базовом классе	Доступ в производном классе
`public`	`public`	`public`
	`protected`	`protected`
	`private`	`private`
`protected`	`public`	`protected`
	`protected`	`protected`
	`private`	`private`
`private`	`public`	`private`
	`protected`	`private`
	`private`	`private`

17.3. Конструкторы, деструкторы и наследование

Конструкторы не наследуются. Если бы конструкторы наследовались, даже при условии их переименования, то можно было бы создать объект производного класса путем вызова конструктора только базового класса. А это привело бы к ошибке. Хотя конструкторы и не наследуются, но при создании нового объекта производного класса компилятор всегда вызывает конструктор по

умолчанию базового класса. Чтобы изменить такую инициализацию объекта производного класса, нужно в списке инициализации конструктора производного класса явно вызвать требуемый конструктор базового класса.

В листинге 17.4 приведен явный и неявный вызов конструктора базового класса.

Листинг 17.4. Явный вызов конструктора базового класса

```
#include <iostream>
using namespace std;
struct Base
{
  int n;
  Base(const int _n = 10): n(_n) {}
};

struct Derived: Base
{
  int m;
  Derived(const int _m = 10): m(_m) {}
  Derived(const int _n, const int _m): Base(_n), m(_m) {}
};
int main()
{
  Derived c;
  cout << c.n << ' ' << c.m << endl;  // 10 10

  Derived d(5, 15);
  cout << d.n << ' ' << d.m << endl;  // 5 15

  return 0;
}
```

При инициализации любого объекта конструкторы вызываются в следующем порядке:

1. Конструкторы базовых классов. Если таких классов несколько, то их конструкторы вызываются в порядке следования этих классов в списке базовых классов, а не в порядке их появления в списке инициализации.

2. Конструкторы объектов-членов класса. Если таких объектов несколько, то их конструкторы вызываются в порядке объявления этих объектов в классе, а не в порядке их появления в списке инициализации.
3. Конструктор производного класса.

Деструкторы не наследуются по той же причине, что и конструкторы. Однако при уничтожении объекта всегда вызываются деструкторы базовых классов. Причем вызываются деструкторы в порядке, обратном вызову конструкторов.

В листинге 17.5 приведен пример неявного вызова деструктора базового класса.

Листинг 17.5. Неявный вызов деструктора базового класса

```
#include <iostream>
using namespace std;

class Base
{
public:
  ~Base() { cout << "Base class" << endl; }
};
class Derived: Base
{
public:
  ~Derived() { cout << "Derived class" << endl; }
};

int main()
{
  {
    Derived  d;
  }
  // печатает: Derived class
  // печатает: Base class

  return 0;
}
```

17.4. Наследование и присваивание

В языке программирования C++ базовый класс совместим по типу со всеми производными от него классами. Поэтому объекты производных классов можно присваивать объекту базового класса, но не наоборот.

В листинге 17.6 приведен пример присваивания объекту базового класса объекта производного класса.

Листинг 17.6. Присваивание объекту базового класса объекта производного класса

```
#include <iostream>
using namespace std;

class Base
{
  int n;
public:
  Base(int _n): n(_n) {}
  int what() { return n; }
};
class Derived: public Base
{
  int m;
public:
  Derived(int _n, int _m): Base(_n), m(_m) {}
  int what() { return m; }
};

int main()
{
  Base b(10);
  Derived d(20, 20);

  cout << b.what() << endl;  // печатает: 10
  b = d;
  cout << b.what() << endl;  // печатает: 20

  return 0;
}
```

Операторы присваивания не наследуются, т. к. при отсутствии оператора присваивания в производном классе компилятор генерирует для этого класса оператор присваивания по умолчанию, который вызывает операторы присваивания для базовых классов и выполняет тривиальное копирование членов-данных производного класса.

17.5. Виртуальные функции

Язык программирования C++ обеспечивает механизм для вызова функции производного класса в случае, если доступ к объекту производного класса осуществляется через указатель или ссылку на базовый класс. Для этого в производном классе должна быть определена функция, имя и сигнатура которой совпадают с именем и сигнатурой функции базового класса, причем функция базового класса должна быть объявлена со спецификатором `virtual`. В этом случае говорят, что функция производного класса *замещает* (`overrides`) функцию базового класса.

В листинге 17.7 приведен пример вызова функции производного класса через указатель на базовый класс.

Листинг 17.7. Вызов функции производного класса через указатель на базовый класс

```
#include <iostream>
using namespace std;

class Base
{
public:
  virtual int what() { return 10; }
};
class Derived: public Base
{
public:
  virtual int what() { return 20; }
};
```

```
int main()
{
  Derived d;
  Base *b = &d;
  cout << b->what() << endl;  // печатает: 20
  Base &c = d;
  cout << c.what() << endl;   // печатает: 20

  return 0;
}
```

Повторять ключевое слово `virtual` в производном классе при замещении функции базового класса не обязательно. Но лучше повторить, чтобы не просматривать иерархию интерфейсов.

Функция, объявленная в базовом классе со спецификатором `virtual`, а также все функции, замещающие ее в производных классах, называются *виртуальными*. Виртуальная функция не может быть статическим членом класса.

Если виртуальные функции базового и производного классов отличаются только типом возвращаемого значения, то компилятор выдаст ошибку. Например:

```
struct  Base
{
  virtual int what() { return 1; }
};
struct  Derived: Base
{
  virtual double what() { return 2.2; } // ошибка компиляции
};
```

Но при этом следует учитывать, что указатель и ссылка на базовый класс совместимы по типам с указателем и ссылкой на производный класс соответственно. Поэтому виртуальная функция производного класса может возвращать соответственно указатель или ссылку на класс, который является производным от класса, указатель или ссылку на который возвращает замещаемая виртуальная функция базового класса.

Если виртуальная функция производного класса имеет такое же имя, как и виртуальная функция базового класса, но другую сиг-

натуру, то виртуальная функция в производном классе просто скрывает соответствующую виртуальную функцию в базовом классе.

В листинге 17.8 приведен пример сокрытия виртуальной функции базового класса.

Листинг 17.8. Сокрытие виртуальной функции базового класса

```
#include <iostream>
using namespace std;

class Base
{
public:
  virtual int what(int n) { return n + 10; }
};
class Derived: public Base
{
public:
  virtual double what(double d) { return d + 20; }
};

int main()
{
  Derived d;
  Base *b = &d;

  cout << b->what(5) << endl;   // 15
  cout << d.what(5.5) << endl;  // 25.5

  return 0;
}
```

Механизм замещения виртуальных функций можно обойти, если при вызове виртуальной функции указать имя класса, которому она принадлежит.

В листинге 17.9 приведен пример вызова виртуальной функции базового класса через указатель на этот класс.

Листинг 17.9. Обход механизма виртуальных функций

```
#include <iostream>
using namespace std;

class Base
{
public:
  virtual int what() { return 10; }
};
class Derived: public Base
{
public:
  virtual int what() { return 20; }
};

int main()
{
  Base *b = new Derived;
  cout << b->Base::what() << endl;  // 10

  return 0;
}
```

17.6. Полиморфизм и позднее связывание

Возможность вызова виртуальных функций производных классов через указатель или ссылку на базовый класс называется *полиморфизмом подтипов* (subtyping polymorphism), или *включающим* (inclusion) *полиморфизмом* , или просто *полиморфизмом*. Класс, который содержит или наследует виртуальную функцию, называется *полиморфным классом*.

Термин *позднее*, или *отложенное*, или *динамическое связывание* (dynamic binding) относится к тому обстоятельству, что компилятор не может определить, какая виртуальная функция должна вызываться, если к ней обращаются через указатель или ссылку на базовый класс. Эта особенность виртуальных функций приводит

к тому, что адрес нужной виртуальной функции может быть определен только во время исполнения программы. Позднее связывание реализуется следующим образом:

- для каждого полиморфного класса компилятор строит таблицу адресов виртуальных функций (vtable);
- каждый объект полиморфного класса содержит скрытый указатель (vptr) на таблицу адресов виртуальных функций;
- компилятор автоматически вставляет в начало конструктора полиморфного класса фрагмент кода, который инициализирует vptr;
- при вызове виртуальной функции ее адрес извлекается из таблицы vtable, на которую указывает vptr объекта.

17.7. Передача аргументов по умолчанию в виртуальные функции

Механизм виртуальных функций не поддерживает передачу аргументов по умолчанию в такие функции. То есть, если функция производного класса вызывается через указатель или ссылку на базовый класс, то ей передаются аргументы по умолчанию, указанные в базовом классе. Это обусловлено тем, что аргументы по умолчанию определяются на этапе компиляции.

В листинге 17.10 приведен пример передачи аргументов по умолчанию в виртуальную функцию.

Листинг 17.10. Передача аргументов по умолчанию в виртуальную функцию

```
#include <iostream>
using namespace std;

class Base
{
public:
```

```
  virtual int what(int n = 10) { return n; }
};
class Derived: public Base
{
public:
  virtual int what(int n = 20) { return n; }
};

int main()
{
  Base* b = new Derived;
  cout << b->what() << endl;  // печатает: 10
  return 0;
}
```

17.8. Виртуальные деструкторы

Так как объекты производных классов могут уничтожаться через указатели на базовые классы при помощи оператора `delete`, то деструкторы можно объявлять виртуальными в отличие от конструкторов. Общее правило таково: если класс является полиморфным, то его деструктор должен быть виртуальным. Причем этот деструктор всегда должен иметь реализацию, т. к. он может быть вызван из деструктора производного класса.

В листинге 17.11 приведен пример вызова виртуальных деструкторов.

Листинг 17.11. Вызов виртуальных деструкторов

```
#include <iostream>
using namespace std;

class Base
{
public:
  virtual ~Base() { cout << "~Base" << endl; }
};
```

```
class Derived: public Base
{
public:
  virtual ~Derived() { cout << "~Derived" << endl; }
};

int main()
{
  Base* b = new Derived;
  delete b;  // ~Derived ~Base

  return 0;
}
```

17.9. Абстрактные классы

Виртуальная функция называется *чистой* (pure), если она определена только как спецификатор интерфейса, т. е. *не имеет реализации*. Чисто виртуальные функции имеют следующий прототип:

```
virtual тип имя_функции(параметры) = 0;
```

Предполагается, что чисто виртуальная функция определяется в производных классах.

Класс, который содержит хотя бы одну чисто виртуальную функцию, называется *абстрактным* (abstract) *классом*. Такой класс может использоваться только в качестве базового класса для других классов.

Так как у чисто виртуальных функций отсутствует реализация, то невозможно создать объект, принадлежащий абстрактному классу. Отсюда следует, что абстрактные классы также подчиняются следующим требованиям:

- абстрактный класс не может использоваться в качестве аргумента или типа возвращаемого функцией значения;
- абстрактный класс нельзя использовать в явном преобразовании типов.

Можно определить указатель или ссылку на абстрактный класс — это позволит использовать абстрактные классы для опи-

сания интерфейсов базовых классов и работы с производными классами через указатели или ссылки на базовый класс.

Если класс, производный от абстрактного класса, не определяет все чисто виртуальные функции, то он также является абстрактным классом.

Так как деструктор абстрактного класса может быть вызван компилятором при уничтожении объекта производного класса, то деструктор абстрактного класса должен быть определен как виртуальная функция с пустым телом, а не как чисто виртуальная функция.

В листинге 17.12 приведен пример использования ссылки на абстрактный класс в качестве параметра функции.

Листинг 17.12. Ссылка на абстрактный класс

```
#include <iostream>
using namespace std;

class Abstract
{
public:
  virtual int what() = 0;
  virtual ~Abstract() {}
};
class Concrete: public Abstract
{
public:
  virtual int what() { return 10; }
  virtual ~Concrete() {}
};
int foo(Abstract& a)
{
  return a.what();
}

int main()
{
```

```
  Concrete c;
  cout << foo(c) << endl;  // 10

  return 0;
}
```

17.10. Множественное наследование

В языке программирования C++ разрешается наследовать производный класс от нескольких базовых классов. Такое наследование называется *множественным* (multiple). В иерархии с множественным наследованием производный класс может косвенно наследовать несколько экземпляров базового класса. В этом случае для доступа к повторяющимся членам базового класса используется оператор разрешения области видимости `::` с именами базовых классов в качестве квалификаторов.

В листинге 17.13 приведен пример множественного наследования классов.

Листинг 17.13. Множественное наследование классов

```
#include <iostream>
using namespace std;

struct Base
{
  int n;
  int what() { return n; }
};
struct Derived_1: Base {};
struct Derived_2: Base {};
struct Derived: Derived_1, Derived_2 {};

int main()
{
  Derived d;

  d.Derived_1::n = 10;
  d.Derived_2::n = 20;
```

```
  cout << d.Derived_1::what() << endl;  // печатает: 10
  cout << d.Derived_2::what() << endl;  // печатает: 20

  return 0;
}
```

17.11. Виртуальное наследование

Для того чтобы исключить передачу нескольких экземпляров базового класса по иерархии при множественном наследовании, непосредственные потомки этого базового класса должны объявить его виртуальным при помощи ключевого слова `virtual` в списке базовых классов. Такое наследование базового класса называется *виртуальным*, а сам базовый класс называется *виртуальным базовым классом*.

В листинге 17.14 приведен пример наследования членов виртуального базового класса.

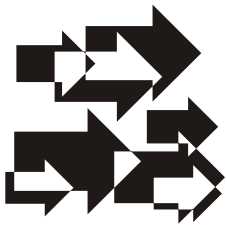
Листинг 17.14. Наследование членов виртуального базового класса

```
#include <iostream>
using namespace std;

struct Base
{
  int n;
  int what() { return n; }
};
struct Derived_1: virtual Base {};
struct Derived_2: virtual Base {};
struct Derived: Derived_1, Derived_2 {};

int main()
{
  Derived d;

  d.n = 10;
  cout << d.what() << endl;  // печатает: 10
```

```
  return 0;
}
```

При виртуальном наследовании необходимо, чтобы конструктор виртуального базового класса вызывался только один раз. Поэтому при создании объекта производного класса сначала вызываются конструкторы всех базовых виртуальных классов, а затем конструкторы остальных базовых классов в порядке их следования в иерархии.

ГЛАВА 18



Шаблоны функций

18.1. Определение шаблона функции

Функция, которая абстрагируется от типа хотя бы одного из своих параметров, называется *обобщенной* или *родовой* (generic) *функцией*. В языке программирования C++ родовые функции определяются *шаблонами функций*.

Определение шаблона функции имеет следующий вид:

```
template <список_параметров>
определение_функции
```

В угловых скобках указан список параметров шаблона функции, который не может быть пустым. Параметрами шаблона функции могут быть:

- типы;
- не типы;
- шаблоны класса.

Объявление параметра-типа шаблона имеет следующий вид:

```
class|typename имя_параметра
```

Ключевые слова `class` и `typename` являются взаимозаменяемыми. В качестве аргумента для параметра-типа запрещается использование:

- локального типа;
- типа без связывания (linkage);
- анонимного типа.

Параметр-тип представляет собой тип данных в определении функции.

Например, в листинге 18.1 определен шаблон функции `xchange`, обменивающей значения переменных. Тип переменных определяется параметром `T` шаблона.

Листинг 18.1. Определение шаблона функции

```
template <class T> inline void xchange(T &a, T &b)
{
  T tmp = a;
  a = b;
  b = tmp;
}
```

Параметр-тип шаблона можно использовать для задания типа возвращаемого функцией значения.

Например, в листинге 18.2 определен шаблон функции `min`, находящей минимальный элемент массива. При этом тип массива является параметром `T` шаблона.

Листинг 18.2. Параметр шаблона как тип возвращаемого функцией значения

```
template <class T> T min(T *a, const int size)
{
  T m = a[0];
  for (int i = 1; i < size; ++i)
    if (a[i] < m)
      m = a[i];
  return m;
}
```

Параметр-не-тип (non-type template parameter) шаблона функции объявляется как переменная, которая используется в определе-

нии функции. Эта переменная может иметь один из следующих типов:

- целочисленный тип;
- тип перечисления;
- указатель на объект, функцию или член класса;
- ссылка на объект или функцию.

Параметр-не-тип может быть определен с квалификаторами `const` и `volatile`.

Например, в листинге 18.3 определен шаблон функции `min` для нахождения минимального элемента массива, параметр `size` которого не является типом и определяет размерность массива.

Листинг 18.3. Определение шаблона функции с параметром, который не является типом

```
template <int size, class T> T min(T *a)
{
  const int n = size;
  T m = a[0];
  for (int i = 1; i < n; ++i)
    if (a[i] < m)
      m = a[i];
  return m;
}
```

Параметром шаблона функции может быть также шаблон класса. Использование шаблонов класса в качестве параметров шаблонов функций рассмотрено в *разд. 19.10*.

18.2. Конкретизация шаблона функции

Процесс создания из шаблона функции определения конкретной функции называется *конкретизацией* или *инстанциированием* (instantiation) шаблона. Определения функции, созданные из шаблона функции, называются *специализациями шаблона функции*

или *шаблонными функциями*. Конкретизация шаблона функции может быть явной или неявной.

При явной конкретизации шаблона функции после имени функции нужно сначала указать параметры шаблона в угловых скобках, а затем — параметры функции в круглых скобках.

Например, следующий вызов функции `xchange` инициирует явную конкретизацию этой функции:

```
int n = 10, m = 20;
xchange<int>(n, m);
```

В этом случае компилятор создаст из шаблона функции определение следующей функции:

```
inline void xchange<int>(int &a, int &b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
```

Отметим, что после имени шаблонной функции в угловых скобках указаны аргументы шаблона функции, что позволяет компилятору отличать шаблонные функции от обычных нешаблонных функций.

Шаблон функции конкретизируется неявно, если опущен хотя бы один его параметр. Это может быть как при вызове функции, так и при определении адреса функции. Если какой-то параметр шаблона опущен, то также должны быть опущены и все последующие параметры.

Например, следующий вызов функции `xchange` инициирует неявную конкретизацию шаблона этой функции:

```
double a = 1.5, b = 2.5;
  xchange(a, b);
```

В листинге 18.4 приведены примеры явной и неявной конкретизации шаблона функции `xchange`. Предполагается, что определе-

ние шаблона функции `xchange` хранится в заголовочном файле xchange.h, текст которого был приведен в листинге 18.1.

Листинг 18.4. Конкретизация шаблона функции

```
#include "xchange.h"
#include <iostream>
using namespace std;
int main()
{
  int n = 10, m = 20;
  xchange<int>(n, m);  // явная конкретизация шаблона функции
  cout << n << ' ' << m << endl;  // печатает 20 10
  double a = 1.5, b = 2.5;
  xchange(a, b);       // неявная конкретизация функции
  cout << a << ' ' << b << endl;  // печатает 2.5 1.5
  return 0;
}
```

Когда компилятор встречает обращение к функции, то адрес нужной функции ищется в следующей последовательности:

- выполняется поиск нешаблонной функции с точным соответствием параметров;
- выполняется поиск шаблонной функции с точным соответствием параметров;
- выполняется поиск нешаблонной функции, которая может быть приведена к нужной функции, используя преобразования типов аргументов.

Отсюда видно, что преобразования типов аргументов шаблона при неявной конкретизации шаблона не используются. Однако при явной конкретизации шаблона производный класс может быть приведен к базовому классу.

В листинге 18.5 приведен пример явной конкретизации шаблона функции с приведением производного класса к базовому классу. Предполагается, что определение шаблона функции `xchange` хранится в заголовочном файле xchange.h, текст которого был приведен в листинге 18.1.

Листинг 18.5. Приведение производного класса к базовому классу при явной конкретизации шаблона функции

```
#include "xchange.h"
#include <iostream>
using namespace std;

struct Base
{
  int n;
};
struct Derived: Base {};

int main()
{
  Base b;
  Derived d;

  b.n = 10;
  d.n = 20;

  xchange<Base>(b, d);
  cout << b.n << ' ' << d.n << endl;  // печатает: 20 10

  return 0;
}
```

Способность компилятора генерировать из шаблона функции определения различных функций в соответствии с аргументами-типами шаблона называется *параметрическим* (parametric) *полиморфизмом*.

18.3. Вывод аргументов шаблона функции

Определение типов аргументов шаблона при его неявной конкретизации называется *выводом* или *дедукцией* (deduction) аргументов шаблона. Вывод аргументов шаблона заключается в сравнении типов аргументов функции с параметрами-типами шаблона.

При выводе аргументов шаблона тип значения, возвращаемого шаблонной функцией, не принимается во внимание. При выводе аргументов шаблона могут выполняться только следующие преобразования аргументов функции:

- преобразования L-value:
  - из L-value в R-value;
  - из типа массива в указатель на тип элементов массива;
  - из типа функции в указатель на функцию;
- добавление квалификаторов доступа `const` и `volatile`;
- преобразования производного класса к базовому классу, если параметром шаблона является шаблон класса.

Например, в листинге 18.6 при вызове шаблонной функции `min` тип массива `int[3]` приводится к типу указателя `int*`. Предполагается, что определение шаблона функции `min` хранится в заголовочном файле min.h, текст которого представлен в листинге 18.2.

Листинг 18.6. Приведение типа массива к типу указателя

```
#include "min.h"
#include <iostream>
using namespace std;

int main()
{
  int a[3] = {3, 1, 2};
  int n = min(a, 3);  // приведение типа массива к указателю
  cout << n << endl;  // печатает 1

  return 0;
}
```

Заметим, что ограничения на допустимые преобразования относятся только к тем аргументам функции, которые участвуют в выводе аргументов шаблона. Для остальных аргументов функции возможно обычное приведение типов, которое используется при вызове функций.

Если аргумент шаблона можно вывести более чем из одного аргумента функции, то в результате все эти выводы должны дать один и тот же тип аргумента шаблона. Например, следующий вызов шаблонной функции `xchange` вызовет ошибку компиляции:

```
int n = 1;
short m = 2;
xchange(n, m);  // ошибка: неоднозначный параметр шаблона
```

Очевидно, что если параметром шаблона функции является не тип, то этот параметр должен быть указан явно при вызове шаблонной функции. Однако можно опустить параметры-типы шаблона, которые выводятся из вызова функции, при условии, что все эти параметры следуют после параметров, которые определены явно.

Например, в листинге 18.7 приведен пример неявной конкретизации шаблона функции, который содержит в качестве параметра не тип. Предполагается, что определение шаблона функции `min` хранится в заголовочном файле min.h, текст которого приведен в листинге 18.3.

Листинг 18.7. Неявная конкретизация шаблона функции с параметром не типом

```
#include "min.h"
#include <iostream>
using namespace std;

int main()
{
  int  a[3] = {3, 1, 2};
  int m = min<3>(a);  // неявная конкретизация шаблона

  cout << m << endl;  // печатает 1

  return 0;
}
```

18.4. Явная специализация шаблона функции

Явной специализацией шаблона функции называется определение шаблона функции для конкретных значений параметров этого шаблона. Явная специализация шаблона функции имеет следующий синтаксис:

```
template<>
имя_функции<список_аргументов_шаблона>(параметры_функции)
блок_функции
```

Здесь `имя_функции` задает имя шаблона функции, а `список_аргументов_шаблона` содержит аргументы, для которых выполняется явная специализация этого шаблона функции.

Явная специализация шаблона функции используется в двух случаях:

- какой-то тип данных не может быть параметром шаблона функции в силу невозможности такой реализации шаблона, которая была бы совместима с другими типами данных;
- специальная реализация шаблона функции для какого-то типа данных более эффективна, чем общая реализация шаблона.

Например, рассмотрим шаблон функции:

```
template <class T> T max(T t1, T t2)
{
  return  t1 > t2 ? t1 : t2;
}
```

Так как общая реализация этого шаблона некорректна для строк, то для них пишут специальную реализацию шаблона. В этом случае параметры шаблона не указываются, а после имени функции в угловых скобках записываются аргументы шаблона для его явной специализации.

```
template <> char* max<char*>(char* s1, char* s2)
{
  return strcmp(s1, s2) > 0 ? s1 : s2;
}
```

Так как в нашем случае аргументы шаблона могут быть неявно выведены из аргументов функции, то можно специализацию шаблона написать так:

```
template <> char* max<>(char* s1, char* s2)
{
  return strcmp(s1, s2) > 0 ? s1 : s2;
}
```

или так:

```
template <> char* max(char* s1, char* s2)
{
  return strcmp(s1, s2) > 0 ? s1 : s2;
}
```

А т. к. в нашем случае шаблон не содержит дополнительных параметров, то явную специализацию шаблона можно заменить нешаблонной функцией, которая никак не связана с шаблоном и не является его явной специализацией:

```
char* max(char* s1, char* s2)
{
  return strcmp(s1, s2) > 0 ? s1 : s2;
}
```

18.5. Модели компиляции шаблонов

В языке программирования C++ поддерживаются две модели компиляции шаблонов:

- модель компиляции с включением;
- модель компиляции с разделением.

В модели компиляции с включением определение шаблона нужно включать в каждый файл, в котором этот шаблон конкретизируется. Обычно, в этом случае определение шаблона помещается в заголовочный файл. Затем этот заголовочный файл включается директивой `#include` в каждый исходный файл, в котором используется шаблон Такая компиляция шаблонов напоминает компиляцию `inline` функций. Шаблон функции конкретизирует-

ся только один раз, но где и когда это происходит — решает компилятор.

В модели компиляции с включением программист может и *явно* определить момент *конкретизации* шаблона функции. Для этого используется ключевое слово `template`, после которого идет объявление шаблона функции, в котором явно указаны аргументы шаблона и параметры функции. Естественно, что явная конкретизация шаблона функции выполняется в исходном файле.

В листингах 18.8 и 18.9 приведен пример явной конкретизации шаблонной функции `xchange`. Заметим, что в этих листингах приведены два исходных файла, принадлежащих одному проекту.

Листинг 18.8. Исходный файл с явной конкретизацией шаблона функции

```
template <class T> inline void xchange(T &a, T &b)
{
  T tmp = a;
  a = b;
  b = tmp;
}
// явные конкретизации шаблона функции xchange
template void xchange<int>(int &a, int &b);
template void xchange<double>(double &a, double &b);
```

Листинг 18.9. Исходный файл, использующий конкретизированные шаблонные функции

```
#include <iostream>
using namespace std;

void xchange(int &a, int &b);
void xchange(double &a, double &b);

int main()
{
  int n = 10, m = 20;
  xchange(n, m);  // вызов шаблонной функции
  cout << n << ' ' << m << endl;  // печатает 20 10
```

```
  double a = 1.5, b = 2.5;
  xchange(a, b);  // вызов шаблонной функции
  cout << a << ' ' << b << endl;  // печатает 2.5 1.5

  return 0;
}
```

Заметим, что, хотя шаблонные функции и содержат в угловых скобках аргументы шаблона, при объявлении прототипов функций нужно использовать их обычные имена.

В модели компиляции с разделением объявление шаблона функции помещаются в заголовочный файл, а определение — в исходный файл. Причем определению шаблона должно предшествовать ключевое слово `export`. В этом случае шаблон называется *экспортируемым*. Шаблон функции должен быть определен как экспортируемый только один раз во всей программе. После этого заголовочный файл с объявлением шаблона включается во все исходные файлы, в которых требуется конкретизация шаблона. Такая компиляция шаблонов функций напоминает компиляцию не `inline` функций.

Отметим, что эта модель компиляции с разделением редко поддерживается компиляторами по причине сложности ее реализации.

18.6. Перегрузка шаблонов функций

Шаблон функции может быть перегружен, т. е. допускается определение нескольких шаблонов функций с одинаковыми именами функций, но разными сигнатурами.

Из того, что перегруженные шаблоны функций успешно объявлены, не следует, что они могут быть успешно конкретизированы. Это происходит потому, что из параметров функции могут быть выведены аргументы нескольких перегруженных шаблонов функций. В этом случае компилятор может и не выбрать шаблон для конкретизации функции.

В листинге 18.10 приведен пример неоднозначного выбора перегруженных шаблонов функции.

Листинг 18.10. Перегрузка шаблонов функций

```
#include <iostream>
using namespace std;

template <class S> S sum(S a, S b) { return a + b; }
template <class S, class T> S sum(S a, T b) { return a + b; }

int main()
{
  cout << sum(1.1, 2) << endl; // конкретизация второго
                               // шаблона функции
//  cout << sum(1, 2) << endl; // ошибка, неоднозначный
                               // выбор шаблона функции
  return 0;
}
```

В случае, рассмотренном в листинге 18.10, лучше оставить второй шаблон функции `sum` с двумя параметрами разных типов, т. к. он перекрывает по специализациям первый шаблон.

Если компилятор из параметров функции выводит аргументы нескольких перегруженных шаблонов функций, то он выбирает тот шаблон функции, который более специализирован по сравнению с остальными шаблонами функций. Один из перегруженных шаблонов функции называется более *специализированным*, чем другой, если он допускает менее широкое множество списков аргументов по сравнению с другим шаблоном. Другими словами, один перегруженный шаблон функции является более специализированным, чем другой перегруженный шаблон функции, если любой список аргументов подходящий для первого перегруженного шаблона функции также подходит и для второго перегруженного шаблона функции, но не наоборот.

Например, следующие шаблоны одноименных функций упорядочены в порядке возрастания их специализации:

```
template<class T> void f(T);
template<class T> void f(const T);
```

и:

```
template<class T> void g(T);
template<class T> void g(T*);
template<class T> void g(const T*);
```

Специализация перегруженных шаблонов функций вводит на этих шаблонах частичный порядок. Поэтому процесс выбора перегруженного шаблона функции называется *частичным упорядочиванием* перегруженных шаблонов.

В листинге 18.11 приведен пример выбора компилятором более специализированного из перегруженных шаблонов функций для конкретизации.

Листинг 18.11. Конкретизация более специализированного шаблона функции

```
#include <iostream>
using namespace std;
// шаблон функции для вычисления суммы двух чисел
template <class T> T sum(T m, int n) { return m + n; }
// шаблон функции для вычисления суммы элементов массива
template <class T> T sum(T* a, int size)
{
  T s = 0;
  for (int i = 0; i < size; ++i)
    s += a[i];
  return s;
}

int main()
{
  cout << sum(1, 2) << endl;  // печатает 3
  int a[] = {1, 2, 3};
  cout << sum(a, 3) << endl;  // печатает 6

  return 0;
}
```

В случае `sum(a, 3)` конкретизируется шаблон функции для вычисления суммы элементов массива, т. к. он более специализиро-

ван, т. е. первым аргументом этого шаблона может быть только указатель. Шаблон же функции для вычисления суммы двух чисел может принимать в качестве первого аргумента как указатель на тип, так и сам тип.

Заметим, что частичное упорядочивание шаблонов функций поддерживают не все компиляторы.

18.7. Шаблоны функций как члены класса

Шаблон функции может быть членом класса. Объявление шаблона функции членом класса обозначает, что класс может содержать бесконечно много различных функций-членов, которые являются специализациями этого шаблона. Для шаблонов функций-членов класса действуют те же правила доступа, что и для других членов класса.

В листинге 18.12 приведен пример определения шаблона функции как члена класса.

Листинг 18.12. Шаблон функции как член класса

```
#include <iostream>
using namespace std;

class Demo
{
public:
  template <class T> T foo(T &t) { return t; }
};

int main()
{
  Demo d;
  cout << d.foo(10) << endl;  // печатает 10

  return 0;
}
```

Не разрешается объявлять шаблонами виртуальные функции-члены класса. Это обусловлено тем, что возникают проблемы с построением таблицы виртуальных функций. Так как в этом случае при обработке определения класса компилятор не может определить количество виртуальных функций класса, которые будут конкретизированы из шаблона функции.

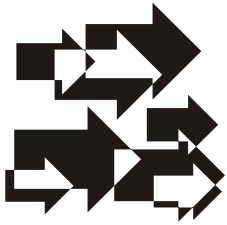
Кроме того, следует отметить, что специализация шаблона не виртуальной функции-члена класса в производном классе не замещает соответствующую виртуальную функцию-член базового класса. Это происходит потому, что компилятор добавляет к имени шаблонной функции типы аргументов, для которых эта функция специализирована.

В листинге 18.13 показано, что шаблонная функция производного класса не замещает виртуальную функцию базового класса.

Листинг 18.13. Определение шаблона функции в производном классе

```
#include <iostream>
using namespace std;

class Base
{
public:
  virtual int foo(const int &n) { return n; }
};
class Derived: public Base
{
public:
  template <class T> T foo(const T &t) { return 2*t; }
};

int main()
{
  Derived d;
  Base* b = &d;

  cout << b->foo(10) << endl;  // печатает: 10
  cout << d.foo(10) << endl;   // печатает: 20

  return 0;
}
```

ГЛАВА 19



Шаблоны классов

19.1. Определение шаблона класса

Класс, который абстрагируется от типа хотя бы одного своего члена, называется *родовым* или *обобщенным* (generic) *классом*. В языке программирования C++ родовые классы определяются *шаблонами классов*. Определение шаблона класса имеет следующий вид:

```
template <список_параметров>
определение_класса
```

В угловых скобках указан список параметров шаблона класса, который не может быть пустым. Параметры шаблона класса имеют тот же смысл и подчиняются тем же требованиям, что и параметры шаблона функции.

Функции-члены шаблона класса могут быть определены как в шаблоне класса, так и вне его. В последнем случае они должны быть явно объявлены как члены шаблона класса.

В листинге 19.1 приведен пример шаблона контейнерного класса `Box`, предназначенного для хранения значения определенного типа `T`, который является параметром шаблона класса. Используя этот шаблон класса, можно создавать контейнерные объекты для хранения значений любого типа.

Листинг 19.1. Определение шаблона класса Box

```
template <class T> class Box
{
  bool empty;  // состояние контейнера
  T    room;   // место для хранения значения
public:
  Box(): empty(true) {}
  void put(T a);   // положить значение в контейнер
  T    get();      // извлечь значение из контейнера
  bool isEmpty() const;  // проверить, свободен ли контейнер
};

template <class T> void Box<T>::put(T a)
{
  room = a;
  empty = false;
}
template <class T> T Box<T>::get()
{
  empty = true;
  return room;
}
template <class T> bool Box<T>::isEmpty() const
{
  return empty;
}
```

В листинге 19.2 приведен пример шаблона контейнерного класса `Stack`, предназначенного для хранения значений определенного типа `T`, который является параметром шаблона класса. Размерность стека задается целочисленным параметром `n` шаблона класса. Конечно, параметр для задания размерности стека можно было бы определить и в конструкторе класса, а не в списке параметров шаблона.

Листинг 19.2. Шаблон класса Stack

```
template <class T, int n> class Stack
{
  int size;  // размер стека
```

```
  int top;   // верхушка стека
  T* a;      // указатель на массив
public:
  Stack(): size(n), top(0), a(new int[size]) {}
  ~Stack() { delete[] a; }

  void push(const T& n) { a[top++] = n; }
  void pop(T& n) { n = a[--top]; }

  bool isEmpty() const { return !top; }
  bool isFull() const { return top == size; }
};
```

Параметром шаблона класса может быть также шаблон класса. Использование шаблонов класса в качестве параметров шаблонов классов будет рассмотрено в *разд. 19.12*.

Для шаблонов классов поддерживаются те же модели компиляции, что и для шаблонов функций, а именно модели компиляции с включением и разделением.

19.2. Конкретизация шаблона класса

Процесс создания определения класса из шаблона класса называется *конкретизацией* или *инстанциированием* (instantiation) *шаблона класса*. Класс, созданный из шаблона, называется *специализацией шаблона класса* или *шаблонным классом*. Шаблон класса конкретизируется только явно.

В листинге 19.3 приведены примеры конкретизации шаблона класса `Box`, определенного в листинге 19.1. Предполагается, что определение этого шаблона хранится в заголовочном файле Box.h.

Листинг 19.3. Конкретизация шаблона класса `Box`

```
#include "Box.h"
#include <iostream>
using namespace std;
```

```
int main()
{
  // объект для хранения целого числа
  Box<int> n;
  n.put(10);
  cout << n.isEmpty() << endl;
  cout << n.get() << endl;
  cout << n.isEmpty() << endl;

  // объект для хранения числа с плавающей точкой
  Box<double> m;
  m.put(5.5);
  cout << m.isEmpty() << endl;
  cout << m.get() << endl;
  cout << m.isEmpty() << endl;

  return 0;
}
```

В листинге 19.4 приведен пример конкретизации шаблона класса `Stack`, определенного в листинге 19.2. Предполагается, что определение этого шаблона хранится в заголовочном файле Stack.h.

Листинг 19.4. Конкретизация шаблона класса `Stack`

```
#include "Stack.h"
#include <iostream>
using namespace std;

int main()
{
  int n;
  Stack<int, 2> s;  // стек для хранения двух целых чисел
  cout << s.isEmpty() << ' ' << s.isFull() << endl;
  s.push(10);
  s.push(20);
  cout << s.isEmpty() << ' ' << s.isFull() << endl;
  s.pop(n);
  cout << n << endl;
  s.pop(n);
```

```
  cout << n << endl;
  cout << s.isEmpty() << ' ' << s.isFull() << endl;

  return 0;
}
```

В отличие от шаблонов функций, шаблоны классов могут иметь значения параметров, заданные по умолчанию. Это относится как к параметрам типам, так и к параметрам не типам. Если шаблон класса имеет значения параметров, заданные по умолчанию, то при конкретизации этого шаблона класса такие параметры можно опустить. Если для некоторого параметра шаблона класса задано значение по умолчанию, то значения по умолчанию должны быть заданы и для всех последующих параметров шаблона.

Например, шаблон класса `Box` можно объявить следующим образом:

```
template <class T = int> class Box;
```

Тогда, если при конкретизации шаблона класса `Box` не указать его параметр-тип, то по умолчанию будет конкретизироваться класс для хранения целого числа.

В листинге 19.5 показан пример конкретизации шаблона класса `Box`, параметр `T` которого по умолчанию имеет значение `int`.

Листинг 19.5. Конкретизация шаблона класса `Box` со значением параметра, заданным по умолчанию

```
#include "Box.h"
#include <iostream>
using namespace std;

int main()
{
  Box<> n;
  n.put(10);
  cout << n.isEmpty() << endl;
  cout << n.get() << endl;
  cout << n.isEmpty() << endl;
```

```
  return 0;
}
```

В отличие от шаблона функции шаблон класса конкретизируется только при определении объекта класса. При объявлении ссылок и указателей на класс шаблон класса не конкретизируется, т. к. в этом случае объект класса может и не существовать.

По определению считается, что функция-член шаблона класса сама является шаблоном. Стандарт языка программирования C++ требует, чтобы она конкретизировалась только при вызове или при получении ее адреса. Поэтому при конкретизации шаблона класса компилятор генерирует код только тех функций-членов класса, которые явно или неявно вызываются в программе.

Способность компилятора генерировать из шаблона класса определения различных классов в соответствии с аргументами-типами шаблона называется *параметрическим* (parametric) *полиморфизмом*.

19.3. Использование ключевого слова *typename*

Ключевое слово `typename` может использоваться в списке параметров шаблона для задания параметра типа вместо ключевого слова `class`. Кроме этого, ключевое слово `typename` может также использоваться в теле шаблона, чтобы указать компилятору на то, что идентификатор является типом. То есть в теле шаблона ключевое слово `typename` нужно использовать всегда, когда идентификатор является именем типа и квалифицируется именем параметра-типа шаблона или именем шаблона класса. В этом случае имена типов должны иметь следующий вид:

```
typename имя_параметра_типа::имя_типа
```

или

```
typename имя_шаблона_класса::имя_типа
```

В листинге 19.6 приведен пример использования ключевого слова `typename` в определении шаблона класса `Demo`. Если в этом слу-

чае ключевое слово typename опустить, то компилятор не сможет определить смысл идентификатора T::Inner.

Листинг 19.6. Использование ключевого слова typename в определении класса

```
#include <iostream>
using namespace std;
template <class T> struct Demo
{
  typename T::Inner* s;  // уточняем, что T::Inner – это тип
  int what() { return s->n; }
};
struct Outer
{
  struct Inner
  {
    int n;
  };
};

int main()
{
  Outer::Inner s;
  s.n = 10;

  Demo<Outer> d;
  d.s = &s;
  cout << d.what() << endl;  // печатает 10

   return 0;
}
```

В заключение этого раздела отметим, что в принципе компилятор может определить смысл любого идентификатора в шаблоне при конкретизации этого шаблона. Поэтому часто при отсутствии ключевого слова typename компилятор все равно верно определяет смысл идентификатора, которому по стандарту должно предшествовать это ключевое слово. Однако такое поведение компилятора не предусмотрено стандартом языка программирования C++.

19.4. Явная специализация шаблона класса

Явной специализацией шаблона класса называется определение шаблона класса для конкретных значений всех параметров этого шаблона. Явная специализация шаблона класса имеет следующий синтаксис:

```
template<>
class|struct|union имя_класса<список_аргументов_шаблона>
блок_класса
```

где `имя_класса` задает имя шаблона класса, а в угловых скобках указан список аргументов шаблона, для которых выполняется явная специализация этого шаблона класса.

Явная специализация шаблона класса используется в тех же случаях, что и явная специализация шаблонов функций, а именно если:

- какой-то тип данных не может быть параметром шаблона класса в силу невозможности такой реализации этого шаблона, которая была бы совместима с другими типами данных;
- специальная реализация шаблона класса для какого-то типа данных более эффективна, чем общая реализация шаблона.

В листинге 19.7 приведен пример явной специализации шаблона класса.

Листинг 19.7. Явная специализация шаблона класса

```
#include <iostream.h>
#include <string.h>

template <class T> class Demo
{
  T t;
public:
  Demo(T _t): t(_t) {}
  T max(T _t) { return t > _t ? t : _t; }
};
```

```
// явная специализация шаблона для строк
template<> class Demo<char*>
{
  char* t;
public:
  Demo(char* _t) { t = new char[strlen(_t) + 1]; strcpy(t,
_t); }
  char* max(char* _t) { return strcmp(t, _t) > 0 ? t : _t; }
};

int main()
{
  Demo<int> d(10);
  cout << d.max(20) << endl;     // печатает: 20

  Demo<char*> s("aaa");
  cout << s.max("bbb") << endl;  // печатает: bbb

  return 0;
}
```

19.5. Частичная специализация шаблона класса

Шаблон класса, как он был определен в начале этой главы, называется *первичным* шаблоном класса. Шаблон класса можно специализировать для одного или нескольких параметров, оставляя другие параметры неспециализированными. Такая специализация шаблона класса называется *частичной*. Определение частичной специализации шаблона класса имеет следующий синтаксис:

```
template<список_параметров_шаблона>
class|struct|union имя_класса<список_аргументов_шаблона>
блок_класса
```

где `имя_класса` должно совпадать с именем первичного шаблона класса, `список_параметров_шаблона` должен быть подсписком списка параметров первичного шаблона класса, `список_аргументов_`

шаблона должен содержать аргументы частичной специализации шаблона класса.

Например, если объявление первичного шаблона класса имеет вид:

```
template <class S, class T> class Demo;
```

то объявления частичных специализаций этого шаблона могут иметь следующий вид:

```
template <class S, class T> class Demo<*S, T>;
template<class S> class Demo<S, *S>;
template<class S> class Demo<S, int>;
```

Определение частичной специализации никак не связано с определением первичного шаблона класса, для которого может быть совершенно другой набор членов, а также собственные определения функций-членов. Содержащиеся в первичном шаблоне класса определения никогда не употребляются для конкретизации членов его частичной специализации.

Отметим, что компилятор может и не конкретизировать шаблон класса по причине неоднозначности его специализаций, которые получены из определения первичного шаблона класса и его частичных специализаций. Для решения этой задачи специализации шаблона класса частично упорядочивают. Частичный порядок на специализациях шаблона класса строится по аналогии с частичным порядком специализаций перегруженных шаблонных функций. То есть параметры шаблона класса рассматриваются как параметры шаблонной функции и дальше применяются правила упорядочивания специализаций шаблонных функций.

При конкретизации шаблона класса компилятор *частично упорядочивает* специализации шаблона класса, полученные из первичного шаблона и частичных специализаций этого шаблона класса. После чего выбирается наилучшая специализация. Если такая отсутствует, то компилятор выдает ошибку.

В заключение этого раздела заметим, что частичную специализацию шаблонов классов поддерживают не все компиляторы.

19.6. Статические члены шаблона класса

В шаблоне класса могут быть объявлены статические данные-члены класса. Каждый конкретизированный экземпляр шаблона класса имеет собственный набор таких членов.

Если тип статического атрибута класса является параметром шаблона класса, то и сам такой атрибут является шаблоном. Так как статические атрибуты класса только объявляются внутри класса, то шаблон статического атрибута класса должен быть определен вне шаблона класса. Любой статический член шаблона класса конкретизируется только в том случае, если он реально используется в программе.

В листинге 19.8 приведен пример работы со статическими членами шаблона класса.

Листинг 19.8. Статические члены шаблона класса

```
#include <iostream>
using namespace std;

template <class T> struct Demo
{
  static T d;
  static void set(T _d) { d = _d; }
};
template<class T> T Demo<T>::d = 10;

int main()
{
  Demo<int>::d += 10;
  cout << Demo<int>::d << endl;  // печатает 20
  Demo<int>::set(5);
  cout << Demo<int>::d << endl;  // печатает 5

  return 0;
}
```

19.7. Шаблоны как члены шаблона класса

Шаблоны функции или класса могут быть членами шаблона класса. Объявление шаблона-члена в шаблоне класса имеет собственные параметры. Для шаблона-члена шаблона класса действуют те же правила доступа `public`, `protected` и `private`, что и для других членов класса.

Например, в листинге 19.9 определен шаблон класса, членами которого являются шаблон класса и шаблон функции.

Листинг 19.9. Шаблон класса с шаблонами-членами

```
#include <iostream>
using namespace std;

template <class T1> class Demo
{
  T1 d;
public:
  Demo(const T1& _d): d(_d) {}
  template <class T2> struct Inner
  {
    T1 n;
    T2 m;
  };
  template<class T3> T3 add(T3 const &p1, T3 const &p2)
  {
    T3 t;
    t = p1 + p2;

    return t;
  };
};

int main()
{
  Demo<double> d(10.5);
```

```
  Demo<int>::Inner<int> in;
  in.n = 10;
  in.m = 20;

  int a = d.add(10,20);
  cout << a << endl;    // 30

  return 0;
}
```

19.8. Объявление друзей в шаблоне класса

В шаблоне класса можно объявить друзей шаблона класса. После конкретизации шаблона класса эти друзья станут друзьями полученной специализации шаблона класса. Друзей шаблона класса можно разбить на три группы:

- друзья шаблона класса, которые не являются шаблонами;
- друзья шаблона класса, которые сами являются шаблонами, и их параметры связаны с параметрами шаблона класса;
- друзья шаблона класса, которые сами являются шаблонами, и их параметры не связаны с параметрами шаблона класса.

К первой группе друзей шаблона класса принадлежат обычные функции и классы, которые объявлены в шаблоне класса с ключевым словом `friend`.

В листинге 19.10 приведен пример функции `what`, объявленной другом шаблона класса `Demo`.

Листинг 19.10. Функция-друг шаблона класса

```
#include <iostream>
using namespace std;

template <class T> class Demo
{
  static int n;
```

```
public:
  const T& foo(const T& t) { return t; }
  friend int what() { return 10; }
};
template<class T> int Demo<T>::n = 10;

int main()
{
  Demo<int> d;
  cout << what() << endl;      // печатает 10
  cout << d.foo(20) << endl;   // печатает 20

  return 0;
}
```

Ко второй группе друзей шаблона класса принадлежат шаблоны функций и классов, которые объявлены в шаблоне класса с ключевым словом `friend`, и параметры этих шаблонов являются также параметрами шаблона рассматриваемого класса. В этом случае говорят, что параметры друзей шаблона класса связаны с параметрами шаблона класса.

Например, в листинге 19.11 приведен пример шаблона функции, параметром которой является параметр-тип шаблона класса.

Листинг 19.11. Шаблон функции-друга шаблона класса

```
#include <iostream>
using namespace std;

template <class T> class Demo
{
  T t;
public:
  Demo(T _t): t(_t) {};
  T get() { return t; }
  template <class T> friend T add(Demo<T> &d, T t);
};
template <class T> T add(Demo<T> &d, T t) { return d.t + t; }
```

```
int main()
{
  Demo<int> d(10);
  cout << add(d, d.get()) << endl;    // 20

  return 0;
}
```

Так как члены шаблона класса также являются шаблонами, то можно определить дружественную шаблону класса функцию, параметром которой будут различные специализации этого шаблона. Пример определения такой функции `what` приведен в листинге 19.12.

В связи с этим примером отметим следующие моменты. Если параметром дружественной функции являются специализации шаблона класса, то для каждой специализации компилятор генерирует свою дружественную функцию, хотя для этой функции и не существует шаблона. Поэтому определять такую дружественную функцию нужно внутри класса.

Листинг 19.12. Функция друг шаблона класса

```
#include <iostream>
using namespace std;

template <class T> class Demo
{
  T t;
public:
  Demo(T _t): t(_t) {}
  friend T what(const Demo<T> &d) { return d.t; }
};

int main()
{
  Demo<int> a(10);
  cout << what(a) << endl;  // печатает 10
```

```
  Demo<double> b(5.5);
  cout << what(b) << endl;  // печатает 5.5

  return 0;
}
```

К третьей группе друзей шаблона класса принадлежат шаблоны функций и классов, которые объявлены в шаблоне класса с ключевым словом `friend`, но параметры этих шаблонов не являются параметрами шаблона рассматриваемого класса. В этом случае говорят, что параметры друзей шаблона класса не связаны с параметрами шаблона класса.

Например, в листинге 19.13 приведен пример шаблона функции, параметром которой является тип, не являющийся параметром шаблона класса.

Листинг 19.13. Шаблон функции друг шаблона класса

```
#include <iostream>
using namespace std;

template <class T> class Demo
{
  int n;
  T t;
public:
  Demo(T _t, int _n): t(_t), n(_n) {};
  T get() { return n; }
  template <class S> friend S sum(Demo<S> *d, S s);
};
template <class S> S sum(Demo<S> *d, S s) { return d->n + s; }

int main()
{
  Demo<int> d(10, 10);
  cout << sum<double>((Demo<double>*)&d, d.get())
       << endl;  // 20
  return 0;
}
```

19.9. Шаблоны класса и наследование

Шаблоны классов можно использовать в построении иерархии классов. Но в этом случае следует принимать во внимание, что имя каждой специализации шаблона класса включает также аргументы этого шаблона, заключенные в угловые скобки.

Например, в листинге 19.14 приведена иерархия классов, которая включает шаблонные классы.

Листинг 19.14. Иерархия классов, включающая шаблонные классы

```
#include <iostream>
using namespace std;

template <class S, class T> struct Base
{
  S s;
  T t;
};
template <class S, class T, class R>
struct Derived: Base<S, T>
{
  R r;
};
struct Demo: Base<int, double>
{
  char* p;
};
int main()
{
  Derived<int, double, char*> a;
  a.s = 10;
  a.t = 1.5;
  a.r = "aaa";

  Demo d;
  d.s = 20;
```

```
  d.t = 2.5;
  d.p = "bbb";

  Base<int, double> &b = d;
  cout << b.s << ' ' << b.t << endl;  // 20 2.5

  return 0;
}
```

19.10. Шаблоны класса как параметры шаблонов функций

Шаблон класса может быть параметром шаблона функции. Например, в листинге 19.15 приведен пример определения шаблона функции `foo`, параметром которой является шаблон класса. Заметим, что параметр (тип) `S` шаблона класса, который является параметром шаблона функции, используется только в списке параметров шаблона и не может использоваться в заголовке или теле функции.

Листинг 19.15. Шаблон функции, параметром которого является шаблон класса

```
#include <iostream>
using namespace std;

template <class T> class Demo
{
  T t;
public:
  Demo(const T& _t): t(_t) {}
  T foo() { return t; }
};
template <class R, template<class S> class T> R foo(Demo<T> d)
{
  return d.foo();
};
```

```
int main()
{
  Demo<int> d(10);
  cout << foo<int>(d) << endl;  // 10

  return 0;
}
```

Для параметров шаблона класса, который является параметром шаблона функции, могут быть установлены значения по умолчанию. Например, шаблон функции `foo` из листинга 19.15 может быть определен следующим образом:

```
template <class R, template<class S = int> class T> R
foo(Demo<T> d)
{
    return d.foo();
};
```

19.11. Шаблоны класса как параметры шаблона класса

Шаблон класса может быть также параметром шаблона класса. Но в этом случае при определении шаблона класса запрещается использовать ключевые слова `struct` и `union`.

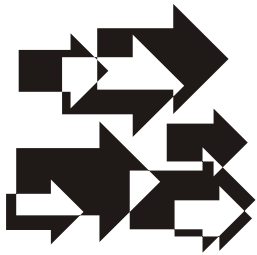
Например, в листинге 19.16 приведен пример определения шаблона класса `Demo`, параметром которого является шаблон класса `Box`. Как и в случае с шаблоном функции заметим, что параметр (тип) `S` шаблона класса, который является параметром шаблона класса, используется только в списке параметров шаблона и не может использоваться в теле класса при определении его членов.

Листинг 19.16. Шаблон класса, параметром которого является шаблон класса

```
#include <iostream>
using namespace std;
```

```
template <class B> struct Box
{
  B b;
};
template <template<class R> class S, class T> class Demo
{
  S<T> *p;
public:
  Demo(S<T> &b): p(&b) {}
  T get() { return p->b; }
};

int main()
{
  Box<int> b = {10};
  Demo<Box, int> d(b);
  cout << d.get() << endl; // 10

  return 0;
}
```

Как и в случае с шаблонами функций, для параметров шаблона класса, который является параметром другого шаблона класса, могут быть установлены значения по умолчанию.



ЧАСТЬ III

Стандартная библиотека языка программирования С

Глава 20. Стандартные определения <stddef.h>

Глава 21. Стандартные функции <stdlib.h>

Глава 22. Диагностика ошибок <assert.h>

Глава 23. Функции с переменным количеством параметров <stdarg.h>

Глава 24. Диапазоны целочисленных данных <limits.h>

Глава 25. Диапазоны чисел с плавающей точкой <float.h>

Глава 26. Обработка ошибок <errno.h>

Глава 27. Математические функции <math.h>

Глава 28. Функции классификации символов <ctype.h>

Глава 29. Функции для работы со строками <string.h>

Глава 30. Функции для работы с файлами <stdio.h>

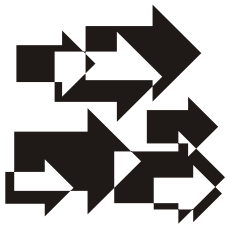
Глава 31. Организация нелокальных переходов <setjmp.h>

Глава 32. Обработка исключительных ситуаций <signal.h>

Глава 33. Поддержка локализации <locale.h>

Глава 34. Работа с датами и временем <time.h>

ГЛАВА 20



Стандартные определения <stddef.h>

В заголовочном файле stddef.h определена символическая константа `NULL`, макрокоманда `offsetof` и типы данных `ptrdiff_t`, `size_t` и `wchar_t`.

20.1. Типы данных

Тип `ptrdiff_t` используется для определения типа значения, полученного при вычитании одного указателя из другого. Этот тип является синонимом целочисленного типа со знаком и может быть определен следующим образом:

```
typedef int ptrdiff_t;
```

Тип `size_t` используется для определения типа значения, которое возвращает оператор `sizeof`. Этот тип является синонимом целочисленного типа без знака и может быть определен следующим образом:

```
typedef unsigned int size_t;
```

Тип `wchar_t` используется при работе с символами, для хранения которых требуется два байта. Этот тип является синонимом целочисленного типа без знака и может быть определен следующим образом:

```
typedef unsigned short wchar_t;
```

20.2. Макрос *NULL*

Макрос `NULL` обычно используется для инициализации указателей, начальное значение которых неизвестно. Определен этот макрос следующим образом:

```
#define NULL 0
```

В языке программирования C++ макрос `NULL` может быть также определен как:

```
#define NULL ((void *)0)
```

То есть в обоих случаях компилятор подставляет `0` вместо этой константы.

20.3. Макрос *offsetof*

Макрос `offsetof` предназначен для определения смещения члена структуры относительно начала структуры. Смещение определяется в байтах. Этот макрос может быть определен следующим образом:

```
#define offsetof(s, m) (size_t)&(((s*)0)->m)
```

где параметр `s` задает тип структуры, а параметр `m` — имя члена этой структуры. Если `m` задает битовое поле структуры, то поведение макрокоманды `offsetof` в общем случае не определено.

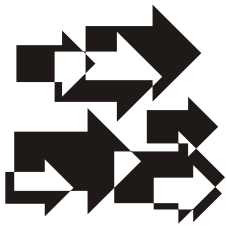
В листинге 20.1 приведен пример использования макроса `offsetof`.

Листинг 20.1. Использование макроса `offsetof`

```
#include<stddef.h>
#include<stdio.h>

struct emp
{
  int empno;
  char name[20];
  double salary;
};
```

```
int main(void)
{
  printf("shift of 'salary' = %d bytes\n",
          offsetof(struct emp, salary));  // печатает 24

  return 0;
}
```

ГЛАВА 21



Стандартные функции <stdlib.h>

21.1. Динамическое распределение памяти

Под динамической памятью понимают память, которая выделяется программе из кучи. Динамическое распределение памяти заключается в выделении программе по ее запросу памяти из кучи. Если программа завершила работу с распределенной ей памятью, то она должна освободить эту память, т. е. вернуть ее обратно в кучу.

Для динамического распределения памяти в языке программирования C используются функции `malloc`, `calloc` и `realloc`. Ниже приведены прототипы этих функций:

```
void* malloc(size_t size);
void* calloc(size_t n, size_t size);
void* realloc(void* ptr, size_t size);
```

Работают эти функции следующим образом.

Функция `malloc` выделяет программе память размером в `size` байт.

Функция `calloc` выделяет программе память размером в `(n*size)` байт, при этом выделенная память обнуляется.

Функция `realloc` перераспределяет память, на которую указывает параметр `ptr`, до размера в `size` байтов. То есть программе вместо

старого распределяется новый блок памяти размером в `size` байт. При этом содержимое старого блока, но не более чем `size` байт, копируется в новый блок. Функция `realloc` используется главным образом для увеличения размера выделенной памяти.

Все эти функции в случае успеха возвращают указатель на выделенную память, в случае неудачи — `NULL`.

Символическая константа `NULL` также определена в заголовочном файле stdlib.h. Подробно эта константа была описана в *разд. 20.2*.

Параметры `size` и `n` этих функций имеют тип `size_t`, который обычно является переопределением типа `unsigned int`. Определение этого типа было дано в *разд. 20.1*. Тип `size_t` также определен и в заголовочном файле stdlib.h.

Тип `void*` называется *родовым указателем*, т. е. указателем на "сырую" или "пустую" память. Впоследствии этот указатель приводится к указателю на нужный тип.

Для освобождения ранее выделенной программе памяти используется функция `free`, которая имеет следующий прототип:

```
void free(void* ptr);
```

Параметр `ptr` этой функции является указателем на освобождаемую память.

Сделаем два важных замечания. Во-первых, динамическую память нужно обязательно освобождать после ее использования. Иначе остаются неиспользованные блоки памяти, которые называются "мусором" или *утечками памяти* (memory leaks). Во-вторых, динамическое выделение памяти используется только в том случае, если программе заранее неизвестно, какой размер памяти нужно отвести под данные. В противном случае нужно использовать статическую память.

В листинге 21.1 приведена программа, которая динамически распределяет память под целое число.

Листинг 21.1. Динамическое распределение памяти

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
  char c;
  int* p = NULL;
  /* Нажмите 'y', чтобы ввести целое число */
  printf("Press 'y' to input integer: ");
  scanf("%c", &c);
  if (c == 'y')
  {
    /* выделяем память под целое число */
    p = (int*)malloc(sizeof(int));
    if (!p)  /* если ошибка, то выходим из программы */
    {
      printf("Memory allocation failed.\n");
      return 0;
    }
    printf("Input integer: ");  /* Вводим целое число */
    scanf("%d", p);

    printf("You input integer: %d\n", *p);
    free(p); /* освобождаем память */
  }
  else
    printf("You don't want to input integer.\n");

  return 0;
}
```

21.2. Динамические массивы

Массив, память под который выделяется во время исполнения программы, называется *динамическим массивом*. Для того чтобы динамически создать одномерный массив, нужно, во-первых, объявить в программе указатель на тип, соответствующий типу элементов этого массива, а во-вторых, выделить память под массив, используя одну из функций `malloc` или `calloc`.

В листинге 21.2 приведена программа, которая динамически создает одномерный целочисленный массив.

Листинг 21.2. Создание одномерного динамического массива

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *a, n;  /* указатель на массив и размерность массива */
  int i;

  /* введите размерность массива */
  printf("Input a size of an array: ");
  scanf("%d", &n);
  /* выделяем память под массив */
  a = (int*)malloc(n * sizeof(int));
  if (!a)  /* если ошибка, то выходим из программы */
  {
    printf("Memory allocation failed.\n");
    return 0;
  }
  /* вводим через пробел элементы массива */
  printf("Input elements of the array: ");
  for (i = 0; i < n; ++i)
    scanf("%d", &a[i]);
  /* что-то делаем с массивом */
  printf("You input the array: ");
  for (i = 0; i < n; ++i)
    printf("%d ", a[i]);
  printf("\n");

  free(a); /* освобождаем выделенную память */

  return 0;
}
```

В связи с этой программой заметим, что элементы одномерного динамического массива можно ввести и следующим образом:

```
  for (i = 0; i < n; ++i)
    scanf("%d", a + i);  /* a – адрес массива */
```

Память под многомерные динамические массивы выделяется по строкам, начиная с первого индекса. Это делается для того, чтобы обеспечить применение оператора индексирования `[]` столько раз, какова размерность многомерного массива. В этом случае тип динамического массива объявляется как указатель, который содержит оператор обращения по адресу `*` столько раз, какова размерность массива. Например, указатель на двумерный целочисленный массив объявляется следующим образом:

```
int  **a;   /* указатель на двумерный массив */
```

В качестве примера в листингах 21.3 и 21.4 приведены тексты функций `alloc_matr` и `free_matr`, которые соответственно динамически выделяют и освобождают память для двумерной целочисленной матрицы.

Листинг 21.3. Динамическое выделение памяти под двумерную матрицу

```
#include <stdlib.h>
int** alloc_matr(int n, int m) /* n, m - диапазоны индексов */
{
  int **a;  /* указатель на матрицу */
  int i, j;
  /* захватываем память для указателей на строки */
  a =  (int**)malloc(n * sizeof(int*));
  if (!a)  /* если ошибка, то выходим из функции */
    return NULL;
  for (i = 0; i < n; ++i) /* захватываем память для строк */
  {
    a[i] = (int*)malloc(m * sizeof(int));
    if (!a[i])  /* если ошибка, то освобождаем память */
    {
      for (j = 0; j < i; ++j)
        free(a[j]);
      free(a);
      return NULL;
    }
  }
  return a;
}
```

Листинг 21.4. Освобождение памяти, занятой двумерной матрицей

```
#include <stdlib.h>
/* a - адрес строк, n - количество строк */
void free_matr(int **a, int n)
{
  int i;
  for (i = 0; i < n; ++i)
    free(a[i]);
  free(a);
}
```

21.3. Стандартные функции сортировки и поиска

На практике при работе с данными часто встречаются задачи сортировки массива и поиска элементов в отсортированном массиве. Причем эти задачи нужно решать для массивов разных типов. Для решения таких задач в стандартной библиотеке языка программирования C предназначены специальные функции `qsort` и `bsearch`.

Функция `qsort` выполняет сортировку массива, элементы которого имеют произвольный тип. Эта функция реализует "быстрый алгоритм" сортировки массивов и имеет следующий прототип:

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *e1, const void *e2));
```

Параметры этой функции имеют следующее назначение:

- `base` — адрес массива;
- `n` — количество элементов в массиве;
- `size` — длина элемента массива;
- `cmp` — указатель на функцию сравнения.

Функция сравнения пишется программистом и предназначена для сравнения элементов массива. Эту функцию стандартная функция сортировки вызывает через указатель, который задан пара-

метром cmp. Функция сравнения должна возвращать одно из следующих значений:

- отрицательное число, если элемент e1 меньше элемента e2;
- 0, если элемент e1 равен элементу e2;
- положительное число, если элемент e1 больше элемента e2.

Для поиска элемента в отсортированном массиве используется функция bsearch, которая выполняет бинарный поиск элемента. Эта функция имеет следующий прототип:

```
void* bsearch(const void *key, const void *base,
              size_t n, size_t size,
              int (*cmp)(const void *e1, const void *e2);
```

Первый параметр key этой функции является указателем на элемент, который нужно найти. Остальные параметры повторяют параметры функции qsort. В случае успешного завершения поиска функция bsearch возвращает адрес найденного элемента, а в случае неудачи NULL.

В листинге 21.5 приведена программа, в которой функции qsort и bsearch используются для сортировки целочисленного массива и дальнейшего поиска элементов в этом отсортированном массиве.

Листинг 21.5. Сортировка целочисленного массива и поиск элемента в этом массиве

```
#include <stdio.h>
#include <stdlib.h>
/* функция для сравнения элементов массива */
int comp_int(const int* e1, const int* e2)
{
  return *e1 - *e2;
}
int main()
{
  int  n;  /* размер массива */
  int* a;  /* массив */
  int  i;  /* индекс */
```

```
  int  k;  /* число для поиска */
  int* s;  /* индекс найденного числа */

  /* введите размерность массива: */
  printf("Input an array size: ");
  scanf("%d", &n);
  a = (int*)malloc(n*sizeof(int));
  /* проверяем на нормальное завершение */
  if (!a)
  {
    printf("Memory allocation failed.");
    return 0;
  }

  /* вводим массив */
  printf("Input elements: ");
  for (i = 0; i < n; ++i)
    scanf("%d", &a[i]);

  /* сортируем массив */
  qsort(a, n, sizeof(int),
    (int (*)(const void*, const void*))comp_int);

  /* выводим отсортированный массив */
  printf("The sorted array: ");
  for (i = 0; i < n; ++i)
    printf("%d ", a[i]);
  printf("\n");

  /* вводим число для поиска */
  printf("Input a number to search.\n>");
  scanf("%d", &k);

  /* поиск числа в отсортированном массиве */
  if(!(s = (int*) bsearch(&k, a, n, sizeof(int),
      (int (*)(const void*, const void*))comp_int)))
    printf("There is no such an integer.\n");
  else
    printf("The integer index = %d.\n", s-a);
  free(a);

  return 0;
}
```

21.4. Взаимодействие с системой

Для установки функций, которые вызываются при нормальном завершении программы, используется функция `atexit`. Под нормальным завершением программы здесь понимается выход из функции `main` посредством выполнения инструкции `return` или выход из программы посредством вызова функции `exit`. Функция `atexit` имеет следующий прототип:

```
int atexit(void (*f)(void));
```

Единственный параметр `f` этой функции является указателем на функцию, которая не имеет параметров и не возвращает значение. В случае успешного завершения функция `atexit` возвращает `0`. Возможна установка до 32-х функций. Установленные функции вызываются системой в порядке, обратном их установке.

В листинге 21.6 приведен пример установки функции `fun` при помощи функции `atexit`.

Листинг 21.6. Установка функции, исполняемой перед завершением программы

```
#include <stdio.h>
#include <stdlib.h>
void fun(void)
{
  printf("fun\n");
}
int main()
{
  if (atexit(&fun))  /* устанавливаем функцию fun */
  {
    printf("atexit failed.\n");
    return 1;
  }
  return 0;  /* печать fun */
}
```

Для нормального завершения программы используется функция `exit`, которая имеет следующий прототип:

```
void exit(int status);
```

Единственный параметр `status` этой функции является кодом завершения программы. В заголовочном файле stdlib.h предопределены следующие два макроса: `EXIT_SUCCESS` и `EXIT_FAILURE`, которые могут использоваться как код завершения и соответственно отмечают успешное или неудачное завершение программы. Эти символические константы обычно определяются следующим образом:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Остальные коды завершения определяются разработчиком.

Используется функция `exit` в основном для выхода из программы, который выполняется из вложенной функции.

Работает функция `exit` следующим образом. Сначала вызываются все функции, установленные функцией `atexit`. Затем освобождаются все буферы ввода/вывода, закрываются все потоки ввода/вывода и удаляются все временные файлы, созданные функцией `tmpfile`. После этого управление передается системе.

В листинге 21.7 приведен пример использования функции `exit`.

Листинг 21.7. Выход из программы с кодом завершения

```
#include <stdio.h>
#include <stdlib.h>
void fun(void)
{
  printf("Exit from the program.\n");
  exit(EXIT_SUCCESS);
}
int main()
{
  fun();
  printf("After fun.\n");
  return 0;
}
```

Для аварийного завершения программы используется функция `abort`, которая имеет следующий прототип:

```
void abort(void);
```

Функция abort используется для аварийного завершения программы в случае возникновения ошибки, которую невозможно исправить.

Работает эта функция следующим образом. Сначала она генерирует сигнал SIGABRT. Если программой не установлен обработчик этого сигнала, то система завершает программу с выдачей сообщения об аварийном завершении. Если для обработки сигнала SIGABRT установлен обработчик, то управление передается ему. Если этот обработчик возвращает управление функции abort, то эта функция завершает исполнение программы вызовом функции exit с кодом завершения EXIT_FAILURE.

В листинге 21.8 приведен пример использования функции abort.

Листинг 21.8. Аварийное завершение программы

```
#include <stdio.h>
#include <stdlib.h>
void fun(void)
{
  printf("Abort the program.\n");
  abort();
}
int main()
{
  fun();
  printf("After fun.\n");
  return 0;
}
```

Для поиска строки, содержащей информацию о среде исполнения программы, используется функция getenv, которая имеет следующий прототип:

```
char* getenv(const char *name);
```

Единственный параметр name этой функции должен указывать на строку с именем строки, которую необходимо найти. Имена строк зависят от среды исполнения. В случае успешного завершения функция возвращает адрес строки с информацией о среде

исполнения. В противном случае функция возвращает значение NULL.

Пример получения информации о типе операционной системы при помощи функции getenv приведен в листинге 21.9.

Листинг 21.9. Получение информации о среде исполнения

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  char* os = "OS";
  printf("%s\n", getenv(os));

  return 0;
}
```

Для интерпретации строки командным процессором используется функция system, которая имеет следующий прототип:

```
int system(const char *str);
```

Единственный параметр str этой функции должен указывать на командную строку, которая передается для интерпретации командному процессору. Поведение программы после передачи управления командному процессору зависит от системы.

Если в параметре str функции установлено значение NULL, то функция определяет, присутствует ли в среде исполнения командный процессор. Если командный процессор присутствует, то функция возвращает ненулевое значение, в противном случае — ноль.

В листинге 21.10 приведен пример запуска программы Notepad.exe посредством вызова командного процессора.

Листинг 21.10. Передача строки для интерпретации командным процессором

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
  if(!system(NULL))
  {
    printf("There is no command processor.\n");
    return 0;
  }
  /* запускаем программу Notepad.exe */
  printf("ret code = %d\n", system("Notepad.exe"));

  return 0;
}
```

21.5. Преобразования строк в числа

Для преобразования строк в числовые значения предназначены функции `atoi`, `atol`, `atof`, `strtol`, `strtoul` и `strtod`.

Для преобразования строки в целое число используется функция `atoi`, которая имеет следующий прототип:

```
int atoi(const char *str);
```

В случае успешного завершения эта функция возвращает целое число, в которое преобразована строка `str`, а в случае неудачи — `0`. Если число, записанное в строке `str`, выходит за диапазон типа `int`, то результат работы функции непредсказуем. Например:

```
int n;
char *str = "-123";
n = atoi(str);    /* n = -123 */
```

Для преобразования строки в длинное целое число используется функция `atol`, которая имеет следующий прототип:

```
long int atol(const char *str);
```

В случае успешного завершения эта функция возвращает целое число, в которое преобразована строка `str`, а в случае неудачи — `0`. Если число, записанное в строке `str`, выходит за диапазон типа `long`, то результат работы функции непредсказуем.

Например:

```
long int  n;
char *str = "-123";
n = atol(str);     /* n = -123 */
```

Для преобразования строки в число типа `double` используется функция `atof`, которая имеет следующий прототип:

```
double atof(const char *str);
```

В случае успешного завершения эта функция возвращает число типа `double`, в которое преобразована строка `str`, а в случае неудачи — `0`. Если число, записанное в строке `str`, выходит за диапазон типа `double`, то результат работы функции непредсказуем. Например:

```
double n;
char *str = "-123.321";
n = atof(str);     /* n = -123.321 */
```

Функции `strtol`, `strtoul` и `strtod` выполняют действия, аналогичные функциям `atoi`, `atol`, `atof`, но предоставляют более широкие возможности.

Для преобразования строки в длинное целое число со знаком используется функция `strtol`, которая имеет следующий прототип:

```
long int strtol(const char *str, char **endptr, int base);
```

Параметр `str` указывает на строку, которая преобразуется в число типа `long int`. Остальные параметры этой функции имеют следующее назначение.

Значение параметра `endptr` устанавливается функцией `strtol`. Это значение содержит указатель на символ, который остановил преобразование строки `str`. В случае успешного завершения функция `strtol` возвращает число типа `long int`, в которое преобразуется строка, а в случае неудачи — `0`.

Параметр `base` указывает на систему счисления, которая использовалась для записи числа в строке. Если параметр `base` равен `0`, то преобразование зависит от первых двух символов строки `str`:

- если первый символ — цифра от 1 до 9, то предполагается, что число представлено в десятичной системе счисления;
- если первый символ — цифра 0, а второй — цифра от 1 до 7, то предполагается, что число представлено в восьмеричной системе счисления;
- если первый символ — цифра 0, а второй — символ X или x, то предполагается, что число представлено в шестнадцатеричной системе счисления.

Если параметр base равен числу от 2 до 36, то это значение принимается за основание системы счисления и любой символ, выходящий за рамки этой системы, прекращает преобразование. В системах счисления с основанием от 11 до 36 для обозначения цифр используются символы от A до Z (или от a до z).

В листинге 21.11 приведен пример использования функции strtol.

Листинг 21.11. Преобразование строки в длинное целое число

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  long int n;
  char *p;

  n = strtol("12a", &p, 0);  /* n = 12, *p = a */
  printf("n = %ld, stop = %c\n", n, *p);

  n = strtol("012b", &p, 0);  /* n = 10, *p = b */
  printf("n = %ld, stop = %c\n", n, *p);

  n = strtol("0x12z", &p, 0);  /* n = 18, *p = z */
  printf("n = %ld, stop = %c\n", n, *p);

  n = strtol("01119", &p, 0);  /* n = 73, *p = 9 */
  printf("n = %ld, stop = %c\n", n, *p);

  return 0;
}
```

Для преобразования строки в длинное целое число без знака используется функция `strtoul`. Эта функция имеет следующий прототип:

```
unsigned long int strtoul(const char *str,
                          char **endptr, int base);
```

Параметры этой функции имеют то же назначение, что и аналогичные параметры функции `strtol`. В случае успешного завершения функция `strtoul` возвращает число типа `unsigned long`, в которое преобразуется строка, а в случае неудачи — `0`.

Для преобразования строки в число типа `double` используется функция `strtod`, которая имеет следующий прототип:

```
double strtod(const char *str, char **endptr);
```

Параметры этой функции имеют то же назначение, что и аналогичные параметры функции `strtol`. В случае успешного завершения функция `strtod` возвращает число типа `double`, в которое преобразуется строка, а в случае неудачи — `0`.

Все функции, рассмотренные в этом разделе, прекращают свою работу при встрече первого символа, который не подходит под формат рассматриваемого числа.

Кроме того, в случае если символьное значение числа превосходит диапазон допустимых значений для соответствующего типа данных, то функции `strtol`, `strtoul`, `strtod` устанавливают в переменной `errno` значение `ERANGE`. Переменная `errno` и макрос `ERANGE` определены в заголовочном файле math.h. При этом функция `strtod` возвращает значение `HUGE_VAL`, функция `strtol` возвращает значение `LONG_MAX` или `LONG_MIN`, а функция `strtoul` — значение `ULONG_MAX`. Макрос `HUGE_VAL` определен в заголовочном файле math.h, а остальные макросы определены в заголовочном файле limits.h.

Для преобразования числовых данных в символьные строки могут использоваться нестандартные функции `itoa`, `ltoa`, `utoa`, `ecvt`, `fcvt` и `gcvt`. Но лучше для этих целей использовать стандартную функцию `sprintf`.

21.6. Арифметические функции

Для получения абсолютной величины целого числа предназначена функция `abs`, которая имеет следующий прототип:

```
int abs(int x);
```

Эта функция возвращает абсолютное значение целого числа `x`. Например:

```
int n = abs(-10);  /* n = 10 */
```

Для получения абсолютной величины длинного целого числа предназначена функция `labs`, которая имеет следующий прототип:

```
long labs(long x);
```

Эта функция возвращает абсолютное значение целого числа `x`. Например:

```
long n = abs(-10);  /* n = -10 */
```

Заметим, что если параметр `x` функций `abs` и `labs` содержит отрицательное целое число, максимальное для заданного диапазона, то результат работы этих функций не определен.

Для вычисления частного и остатка от деления целых чисел используется функция `div`, которая имеет следующий прототип:

```
div_t div(int n, int d);
```

Здесь `n` обозначает делимое, а `d` — делитель. Функция возвращает структуру типа `div_t`, которая определена следующим образом:

```
typedef struct _div_t
{
  int quot;  /* частное */
  int rem;   /* остаток */
} div_t;
```

Например:

```
div_t d = div(5, 2);
printf("%d %d\n", d.quot, d.rem);  /* печатает 2 1 */
```

Для вычисления частного и остатка от деления длинных целых чисел используется функция ldiv, которая имеет следующий прототип:

```
ldiv_t ldiv(long n, long d);
```

Здесь n обозначает делимое, а d — делитель. Функция возвращает структуру типа ldiv_t, которая определена следующим образом:

```
typedef struct _ldiv_t
{
  long quot;  /* частное */
  long rem;   /* остаток */
} ldiv_t;
```

Например:

```
ldiv_t d = ldiv(5, 2);
printf("%d %d\n", d.quot, d.rem);  /* печатает 2 1 */
```

21.7. Генерация случайных чисел

Для генерации последовательности псевдослучайных чисел предназначена функция rand, которая имеет следующий прототип:

```
int rand(void);
```

При каждом последующем вызове эта функция возвращает следующее псевдослучайное число. Псевдослучайные числа находятся в интервале от 0 до RAND_MAX. По стандарту макрос RAND_MAX определяется как целое число, которое не меньше чем 32 767.

В листинге 21.12 приведен пример генерации пяти случайных чисел.

Листинг 21.12. Генерация псевдослучайных чисел

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i;
```

```
  for (i = 0; i < 5; ++i)
    printf("%d ", rand());  /* печать псевдослучайных чисел */
  printf("\n");

  return 0;
}
```

Если программу из листинга 21.12 запустить несколько раз подряд, то можно увидеть, что каждый раз будет печататься одна и та же последовательность псевдослучайных целых чисел.

Для изменения последовательности псевдослучайных чисел, которые генерирует функция `rand`, используется функция `srand`, которая имеет следующий прототип:

```
void srand(unsigned int seed);
```

Значение параметра `seed` используется для определения начальной точки последовательности псевдослучайных чисел. Каждый вызов функции `srand` вызывает установку начальной точки для последовательности псевдослучайных чисел. Использование функции `rand` без предварительного вызова функции `srand` эквивалентно вызову функции `srand(1)`.

В листинге 21.13 приведен пример генерации последовательности случайных чисел, начальная точка которой задается числом `2`.

Листинг 21.13. Генерация псевдослучайных чисел с выбором последовательности

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int i;
  srand(2);
  for (i = 0; i < 5; ++i)
    printf("%d ", rand());
  printf("\n");

  return 0;
}
```

21.8. Обработка длинных символов

Длинным (multibyte) называется символ, для хранения которого может потребоваться более одного байта. Такие символы используются в кодовых таблицах для некоторых восточных алфавитов.

Максимальное количество байт, которые могут использоваться для хранения одного символа алфавита, задается макросом `MB_CUR_MAX`. Значение этого макроса зависит от локальной категории, которая определяется значением макроса `LC_CTYPE`, определенного в заголовочном файле locale.h.

Кодировка длинных символов может быть *зависимой от состояния* (state dependent encoding). Это значит, что длинные символы в данной кодировке могут занимать различное количество байт.

Для определения количества байт, которые используются для хранения длинного символа, предназначена функция `mblen`, которая имеет следующий прототип:

```
int mblen(const char *str, size_t n);
```

Если значение параметра `str` не равно `NULL`, то функция возвращает длину первого длинного символа, который хранится в строке `str`. Причем рассматриваются только первые `n` байтов строки `str`. Если строка `str` не содержит такого символа, то функция `mblen` возвращает значение `-1`.

Если значение параметра `str` равно `NULL`, то функция `mblen` возвращает ненулевое значение, если кодировка длинных символов зависит от состояния. В противном случае функция возвращает нулевое значение.

Для преобразования длинного (multibyte) символа в *широкий* (wide) символ предназначена функция `mbtowc`, которая имеет следующий прототип:

```
int mbtowc(wchar_t *pwc, const char *str, size_t n);
```

Параметр `str` указывает на исходную строку с длинным символом, параметр `pwc` — на строку, в которую функция поместит соответствующий широкий символ, а параметр `n` определяет максимальное количество байт, которые рассматривает функция.

Если значение параметра str не равно NULL, то функция mbtowc возвращает длину преобразованного символа или значение -1, в случае если параметр str не указывает на длинный символ.

Если значение параметра str равно NULL, то функция mbtowc возвращает ненулевое значение, если кодировка длинных символов зависит от состояния. В противном случае функция возвращает нулевое значение.

Для обратного преобразования из широкого символа в длинный символ предназначена функция wctomb, которая имеет следующий прототип:

```
int wctomb(char *str, wchar_t wchar);
```

Параметр wchar задает широкий символ, а параметр str указывает на строку, в которую будет записан соответствующий длинный символ.

Если значение параметра str не равно NULL, то функция wctomb возвращает количество байтов, в которые записан длинный символ или значение -1, в случае если для широкого символа, заданного параметром wchar, не существует соответствующего длинного символа.

Если значение параметра str равно NULL, то функция wctomb возвращает ненулевое значение, если кодировка длинных символов зависит от состояния. В противном случае функция возвращает нулевое значение.

В листинге 21.14 приведен пример использования функций mblen, mbtowc и wctomb.

Листинг 21.14. Преобразования между длинными и широкими символами

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
int main()
{
  wchar_t a = L'a';
```

```
  wchar_t w;
  char str[10];

  printf("%d\n", wctomb(str, a));  /* печатает 1 */
  printf("%d\n", mblen(str, 10));  /* печатает 1 */
  printf("%d\n", mbtowc(&w, str, 10));  /* печатает 1 */
  wprintf(L"wide symbol: %c\n", w);     /* печатает a */

  return 0;
}
```

Для преобразования строки длинных символов в строку широких символов предназначена функция `mbstowcs`, которая имеет следующий прототип:

```
size_t mbstowcs(wchar_t *pwcs, const char *str, size_t n);
```

Параметр `str` указывает на исходную строку длинных символов, а параметр `pwcs` — на строку, в которую функция поместит соответствующую строку широких символов. Преобразование прекращается при достижении конца строки, который задается пустым символом, или после обработки `n` длинных символов, или при встрече последовательности символов, которые не образуют длинный символ.

Функция `mbstowcs` возвращает количество преобразованных длинных символов без завершающего пустого символа или значение `-1`, если обработка была прервана при встрече "недлинного" символа.

Для обратного преобразования строки широких символов в строку длинных символов предназначена функция `wcstombs`, которая имеет следующий прототип:

```
size_t wcstombs(char *str, const wchar_t *pwcs, size_t n);
```

Параметр `pwcs` должен указывать на исходную строку широких символов, а параметр `str` на строку, в которую функция поместит соответствующую строку длинных символов. Преобразование прекращается при достижении конца строки, который задается пустым символом, или после обработки `n` широких символов, или при встрече широкого символа, для которого нет соответствующего длинного символа.

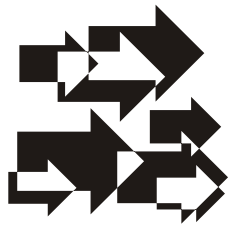
Функция `wcstombs` возвращает количество преобразованных широких символов без завершающего пустого символа или значение `-1`, если обработка была прервана при встрече широкого символа, для которого не нашлось соответствия.

В листинге 21.15 приведен пример использования функций `mbstowcs` и `wcstombs`.

Листинг 21.15. Преобразование строки длинных символов в строку широких символов и наоборот

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main()
{
  wchar_t a[] = L"abc";
  wchar_t w[4];
  char str[10];

  printf("%d\n", wcstombs(str, a, 4));  /* печатает 3 */
  printf("%d\n", mbstowcs(w, str, 10)); /* печатает 3 */
  wprintf(L"wide string: %s\n", w);     /* печатает: abc */

  return 0;
}
```

ГЛАВА 22



Диагностика ошибок <assert.h>

Для записи отладочной информации в стандартный поток вывода `stderr` предназначен макрос `assert`, который определен в заголовочном файле assert.h. Определение этого макроса имеет следующий вид:

```
#undef assert
#if defined NDEBUG
#define assert(тест) (void)0
#else
#define assert(тест) void_выражение
#endif
```

Здесь `тест` — это целочисленное выражение, а `void_выражение` — это выражение на языке программирования C, которое собственно и определяет реализацию макрокоманды `assert`. Работает макрос `assert` следующим образом. Если значение выражения `тест` равно `0`, то в стандартный поток `stderr` выводится сообщение, которое содержит следующую информацию:

- текст выражения `тест`;
- имя исходного файла (предопределенная макрокоманда `__FILE__`);
- номер строки (предопределенная макрокоманда `__LINE__`).

После этого макрокоманда завершает программу посредством вызова стандартной функции `abort`. Если значение выражения `тест` не равно `0`, то исполнение программы продолжается.

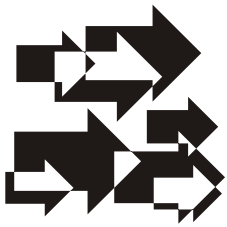
В листинге 22.1 приведен пример использования макрокоманды assert. При запуске этой программы в поток stderr запишется подобное сообщение:

```
Assertion failed: n < 0, file C:\test\main.cpp, line 7
```

После отладки программы макрокоманду assert можно не удалять из исходного текста, а просто отменить ее определение. Для того нужно определить символическое имя NDEBUG. Причем это имя должно быть объявлено перед директивой #include <assert.h>.

Листинг 22.1. Использование макрокоманды assert

```
#include <assert.h>
int main()
{
  int n = 10;

  assert(n > 0);
  assert(n < 0);

  return 0;
}
```



ГЛАВА 23

Функции с переменным количеством параметров <stdarg.h>

В языке программирования C допускается использование функций с переменным количеством параметров. В этом случае в конце списка параметров ставится многоточие ... При этом предполагается, что в списке параметров присутствует хотя бы один параметр.

Например, можно определить функцию для сложения произвольного количества целых чисел следующим образом:

```
int add_num(int n, ...);  /* n - количество слагаемых */
```

Также, в качестве примеров функций с переменным количеством параметров, можно назвать стандартные функции `scanf` и `printf`.

Для обработки функций с переменным количеством параметров в заголовочном файле `stdarg.h` определен тип данных `va_list` и три макроса `va_start`, `va_arg` и `va_end`.

Тип данных `va_list` обычно определяется следующим образом:

```
typedef char* va_list;
```

Этот тип служит для объявления указателя, в который макросы `va_start` и `va_arg` записывают адреса аргументов из вызова функции с переменным количеством параметров.

Макрос `va_start` выполняет инициализацию переменной типа `va_list`, необходимую для доступа к аргументам, для которых не

объявлены параметры в функции с переменным количеством параметров. Этот макрос определен следующим образом:

```
#define va_start(ap, v) void_выражение
```

Параметр `ap` обозначает имя переменной типа `va_list`, в которую макрокоманда помещает адрес последнего заданного аргумента в вызове функции; параметр `v` обозначает имя этого последнего заданного аргумента в вызове функции; а `void_выражение` представляет реализацию макрокоманды и зависит от системы.

На тип последнего заданного аргумента в функции с переменным количеством параметров накладываются следующие ограничения:

- не допускается спецификатор класса памяти `register`;
- не допускается тип "функция";
- не допускается тип "массив";
- не допускается тип, который не сравним с типом, полученным после интегрального продвижения по типам.

Если эти ограничения не выполняются, то результат работы макрокоманды `va_start` не определен.

Макрос `va_arg` используется для последовательного доступа к аргументам функции с переменным количеством параметров. Этот макрос определен следующим образом:

```
#define va_arg(ap, t) t_выражение
```

Параметр `ap` обозначает имя переменной, которая была инициализирована макрокомандой `va_start`; параметр `t` обозначает тип аргумента в вызове функции, значение которого нужно получить; а `t_выражение` представляет реализацию макрокоманды. Результатом вычисления `t_выражения` является R-value типа `t`, значение которого равно значению следующего аргумента. Если следующего аргумента нет, то результат работы макрокоманды непредсказуем.

Макрос `va_end` используется после завершения работы с аргументами функции с переменным количеством параметров. Назначение этой макрокоманды обеспечить правильную работу инструк-

ции return в функции с переменным количеством параметров. Определена макрокоманда va_end следующим образом:

```
#define va_end(ap) void_выражение
```

Параметр ap обозначает имя переменной, которая была инициализирована макрокомандой va_start, а void_выражение представляет реализацию макрокоманды.

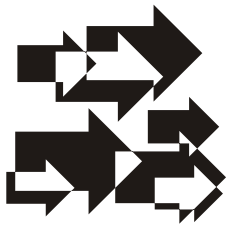
В листинге 23.1 приведен пример реализации функции с переменным количеством параметров print. Эта функция просто распечатывает свои аргументы, которые являются целыми числами.

Листинг 23.1. Функция с переменным количеством параметров

```
#include <stdio.h>
#include <stdarg.h>
int print(short n, ...)
{
  va_list a;
  if (n < 1)  /* проверяем, есть ли параметры */
  {
    printf("There are no numbers.\n");
    return 0;
  }
  printf("Number of parameters = %d\n", n);
  va_start(a, n);  /* инициализируем a */
  while (n--)      /* распечатываем аргументы */
  {
    int x;
    x = va_arg(a, int);  /* получаем следующий аргумент */
    printf("%d\n", x);   /* распечатываем аргумент */
  }
  va_end(a);  /* завершение продвижения по аргументам */
  return  0;
}
/* проверяем работу функции print */
int main()
{
  print(0);
```

```
  print(1, 1);
  print(2, 10, 20);
  print(3, 100, 200, 300);

  return 0;
}
```

ГЛАВА 24



Диапазоны целочисленных данных <limits.h>

В заголовочном файле limits.h определены символические константы, которые описывают диапазоны целочисленных типов данных. Ниже перечислены эти константы и приведены их значения. В общем случае значения этих символических констант зависят от реализации, но они не могут сужать диапазон типа данных, который задан значениями, приведенными ниже.

Для символьных типов данных определены следующие символические константы:

```
#define CHAR_BIT 8 /* количество бит, требуемых для */
                   /* представления типа char */
#define SCHAR_MIN (-127)  /* минимум типа signed char */
#define SCHAR_MAX 127     /* максимум типа signed char */
#define UCHAR_MAX 255     /* максимум типа unsigned char */
```

Диапазон типа `char` может совпадать с диапазоном типа `signed char` или `unsigned char`, что определяется реализацией компилятора. Обычно нужный диапазон можно задать посредством установки соответствующей опции компилятора. Поэтому для типа `char` определены два варианта диапазонов:

```
#define CHAR_MIN SCHAR_MIN  /* минимум типа signed char */
#define CHAR_MAX SCHAR_MAX  /* максимум типа signed char */
```

или

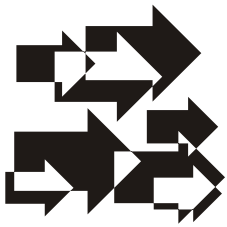
```
#define CHAR_MIN 0          /* минимум типа unsigned char */
#define CHAR_MAX UCHAR_MAX /* максимум типа unsigned char */
```

Для длинных (multibyte) символов также определена символическая константа MB_LEN_MAX, которая задает максимальное количество байтов, используемых для представления длинного символа.

Для целочисленных типов данных определены следующие символические константы:

```
#define SHRT_MIN (-32767)  /* минимум типа short */
#define SHRT_MAX 32767     /* максимум типа short */

#define USHRT_MAX 65535  /* максимум типа unsigned short */

#define INT_MIN (-32767) /* минимум типа int */
#define INT_MAX 32767    /* максимум типа int */

#define UINT_MAX 65535  /* максимум типа unsigned int */

#define LONG_MIN (-2147483647L) /* минимум типа long */
#define LONG_MAX 2147483647L    /* максимум типа long */

#define ULONG_MAX 4294967295UL /* максимум типа */
                               /* unsigned long */
```

ГЛАВА 25



Диапазоны чисел с плавающей точкой <float.h>

В заголовочном файле float.h определены символические константы, задающие диапазоны чисел с плавающей точкой. Ниже перечислены эти константы и приведены их значения. В общем случае значения этих символических констант зависят от реализации, но они не могут сужать диапазон типа данных, который задан значениями, приведенными ниже.

Для типа данных `float` определены следующие символические константы:

```
#define FLT_DIG 6          /* точность представления */
                           /* в знаках после точки */
#define FLT_EPSILON 1E-5F  /* такое значение, что */
                           /* 1.0 + FLT_EPSILON != 1.0 */
#define FLT_MANT_DIG 24    /* количество битов для */
                           /* представления мантиссы */
#define FLT_MAX 1E+37F     /* максимальное значение */
#define FLT_MAX_10_EXP 37  /* максимальная экспонента */
                           /* для числа 10 */
#define FLT_MAX_EXP 128    /* максимальная экспонента */
                           /* для числа 2 */
#define FLT_MIN 1E-37F     /* минимальное положительное */
                           /* значение */
#define FLT_MIN_10_EXP (-37)  /* минимальная экспонента */
                              /* для числа 10 */
```

```
#define FLT_MIN_EXP (-125) /* минимальная экспонента */
                           /* для числа 2 */
#define FLT_RADIX 2  /* система счисления экспоненты */
```

Для типа данных `double` определены следующие символические константы:

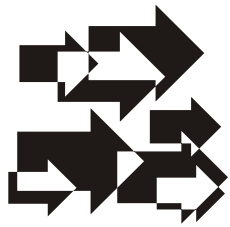
```
#define DBL_DIG 10          /* точность представления в */
                            /* знаках после точки */
#define DBL_EPSILON 1E-9    /* такое значение, что */
                            /* 1.0 + DBL_EPSILON != 1.0 */
#define DBL_MANT_DIG 24     /* количество битов для */
                            /* представления мантиссы */
#define DBL_MAX 1E+37       /* максимальное значение */
#define DBL_MAX_10_EXP 37   /* максимальная экспонента */
                            /* для числа 10 */
#define DBL_MAX_EXP 128     /* максимальная экспонента */
                            /* для числа 2 */
#define DBL_MIN 1E-37       /* минимальное положительное */
                            /* значение */
#define DBL_MIN_10_EXP (-37)  /* минимальная экспонента */
                              /* для числа 10 */
#define DBL_MIN_EXP (-125)  /* минимальная экспонента */
                            /* для числа 2 */
```

Для типа данных `long double` определены следующие символические константы:

```
#define LDBL_DIG 10          /* точность представления в */
                             /* знаках после точки */
#define LDBL_EPSILON 1E-9    /* такое значение, что */
                             /* 1.0 + DBL_EPSILON != 1.0 */
#define LDBL_MANT_DIG 24     /* количество битов для */
                             /* представления мантиссы */
#define LDBL_MAX 1E+37       /* максимальное значение */
#define LDBL_MAX_10_EXP 37   /* максимальная экспонента */
                             /* для числа 10 */
#define LDBL_MAX_EXP 128     /* максимальная экспонента */
                             /* для числа 2 */
```

```
#define LDBL_MIN 1E-37      /* минимальное положительное */
                            /* значение */
#define LDBL_MIN_10_EXP (-37) /* минимальная экспонента */
                              /* для числа 10 */
#define LDBL_MIN_EXP (-125) /* минимальная экспонента */
                            /* для числа 2 */
```

ГЛАВА 26



Обработка ошибок <errno.h>

В заголовочном файле errno.h определены макрокоманда `errno`, а также символические константы `EDOM`, `ERANGE` и `EILSEQ`, которые используются для обработки ошибок, возникающих при работе стандартных функций.

Определение макроса `errno` имеет следующий вид:

```
#define errno int_модифицируемое_L_value
```

Этот макрос определяет модифицируемое L-value типа `int`, в которое стандартная функция может записать код ошибки, возникшей во время ее исполнения. С точки зрения исполняемой программы `errno` можно рассматривать как внешнюю глобальную переменную.

После завершения исполнения стандартной функции вызвавшая ее функция может проверить значение переменной `errno`, чтобы определить, произошла ли ошибка во время исполнения стандартной функции. Для этого перед вызовом стандартной функции в переменную `errno` нужно записать нулевое значение. Если во время исполнения стандартной функции могут произойти ошибки, то для этой функции определены коды, которые она записывает в переменную `errno`. Все коды ошибок, устанавливаемые стандартными функциями, имеют положительное значение.

Макрос `EDOM` определяет код ошибки, который устанавливает стандартная функция, если значение параметра этой функции не

входит в область определения этой функции. Этот код ошибки устанавливают в переменной `errno` некоторые математические функции.

Макрос `ERANGE` определяет код ошибки, который устанавливает стандартная функция, если результат вычисления этой функции определен, но его значение выходит за диапазон типа возвращаемого функцией значения. Этот код ошибки устанавливают в переменной `errno` некоторые математические функции.

Макрос `EILSEQ` определяет код ошибки, который устанавливает стандартная функция, если при обработке встретилась неправильная последовательность длинных (multibyte) символов.

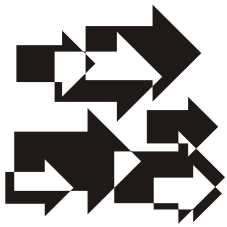
Кроме того, в зависимости от реализации могут быть определены и другие коды ошибок, возникающих во время исполнения стандартных функций.

В листинге 26.1 приведен пример обнаружения ошибки, возникающей при исполнении математической функции `acos`.

Листинг 26.1. Обнаружение ошибки при исполнении стандартной функции

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
int main()
{
  double y;

  errno = 0;
  y = acos(5.5);  /* аргумент вне области определения acos */
  if (errno == EDOM)
    printf("Domain error\n"); /* печатает: Domain error */
  else
    printf("acos = %f\n", y);

  return 0;
}
```

ГЛАВА 27



Математические функции <math.h>

Все математические функции, определенные в стандартной библиотеке, имеют параметры и возвращают значение типа `double`. Однако заметим, что в языке программирования C++ эти функции также перегружены для типов `float` и `long double`.

27.1. Обработка ошибок

При работе математических функций могут возникать ошибки двух типов:

- значение параметра не входит в область определения математической функции;
- результат вычисления определен, но его величина выходит за диапазон типа данных, значение которого возвращает математическая функция.

Математические функции обрабатывают эти ошибки следующим образом.

В первом случае, если значение параметра не входит в область определения математической функции, то эта функция устанавливает в переменную `errno` значение макроса `EDOM` и возвращает значение, которое зависит от реализации.

Во втором случае математическая функция устанавливает в переменную `errno` значение макроса `ERANGE` и возвращает одно из следующих значений:

- HUGE_VAL, если значение функции, возвращающей значение типа double, существует и положительно, но слишком велико для представления типом double;
- 0, если значение функции, возвращающей значение типа double, существует, но слишком мало для представления типом double;
- -HUGE_VAL, если значение функции, возвращающей значение типа double, существует и отрицательно, но слишком велико по модулю для представления типом double.

Переменная errno, а также макросы EDOM и ERANGE определены в заголовочном файле errno.h. Макрос HUGE_VAL определен в заголовочном файле math.h.

Пример обработки ошибки, возникающей при вызове математической функции, приведен в *гл. 26*.

27.2. Тригонометрические функции

В стандартной библиотеке определены тригонометрические функции acos, asin, atan, atan2, cos, sin, tan, которые имеют следующие прототипы:

```
double acos(double x);  /* возвращает arccos x */
double asin(double x);  /* возвращает arcsin x */
double atan(double x);  /* возвращает arctg x */
double atan2(double y, double x); /* возвращает arctg y/x */
double cos(double x);   /* возвращает cos x */
double sin(double x);   /* возвращает sin x */
double tan(double x);   /* возвращает  tg x */
```

Заметим, что тригонометрическая функция atan2 возвращает угол, который находится в интервале от $-\pi$ до π.

В листинге 27.1 приведен пример вычисления значения sin x.

Листинг 27.1. Вычисление значения sin x

```
#include <stdio.h>
#include <math.h>
```

```
int main()
{
  double x;
  /* Введите действительное число */
  printf("Input a real number: ");
  scanf("%lf", &x);
  printf("sin x = %f\n", sin(x));

  return 0;
}
```

27.3. Экспоненциальная и логарифмическая функции

В стандартной библиотеке определена экспоненциальная функция `exp`, которая имеет следующий прототип:

```
double exp(double x);  /* возвращает exp x */
```

Также в стандартной библиотеке определены логарифмические функции `log` и `log10`, которые имеют следующие прототипы:

```
double log(double x);  /* возвращает ln x */
double log10(x);       /* возвращает десятичный log x */
```

В листинге 27.2 приведен пример вычисления значений функций `exp` и `log`.

Листинг 27.2. Вычисление exp x и ln x

```
#include <stdio.h>
#include <math.h>
int main()
{
  double x, y;
  /* Введите действительное число */
  printf("Input a real number: ");
  scanf("%lf", &x);
  y = exp(x);
  printf("exp x = %f\n", y);
```

```
  x = log(y);
  printf("ln y = %f\n", x);

  return 0;
}
```

27.4. Гиперболические функции

В стандартной библиотеке определены гиперболические функции `cosh`, `sinh` и `tanh`, которые имеют следующие прототипы:

```
double cosh(double x);  /* возвращает cosh x */
double sinh(double x);  /* возвращает sinh x */
double tanh(double x);  /* возвращает tanh x */
```

27.5. Степень и квадратный корень

В стандартной библиотеке определена степенная функция `pow`, которая имеет следующий прототип:

```
double pow(double x, double y);  /* возвращает x^y */
```

Также в стандартной библиотеке определена функция `sqrt` для нахождения квадратного корня из числа, которая имеет следующий прототип:

```
double sqrt(double x);  /* возвращает квадр. корень из x */
```

В листинге 27.3 приведен пример использования функций `pow` и `sqrt`.

Листинг 27.3. Вычисление степени и квадратного корня числа

```
#include <stdio.h>
#include <math.h>
int main()
{
  double d, r;

  d = pow(2.2, 2);
  printf("d = %lf\n", d);  /* d = 4.84 */
```

```
  r = sqrt(d);
  printf("r = %lf\n", r);  /* r = 2.2 */

  return 0;
}
```

27.6. Экспонента и мантисса

Для определения мантиссы и экспоненты числа с плавающей точкой предназначена функция `frexp`, которая имеет следующий прототип:

```
double frexp(double x, int *p);
```

Эта функция находит такие мантиссу `m` и экспоненту `e` числа `x`, что:

```
x = m * 2^e
```

и, кроме того, мантисса находится в диапазоне:

```
0.5 ≤ m < 1.0;
```

После этого функция `frexp` возвращает мантиссу `m`, а по адресу `p` записывает экспоненту `e`.

Для вычисления числа с плавающей точкой по его мантиссе и экспоненте предназначена функция `ldexp`, которая имеет следующий прототип:

```
double ldexp(double x, int e);   /* возвращает x*2^e */
```

В листинге 27.4 приведен пример использования функций `frexp` и `ldexp`.

Листинг 27.4. Работа с мантиссой и экспонентой

```
#include <stdio.h>
#include <math.h>
int main()
{
  int n;
  double d, e;
  d = ldexp(3, 2);
```

```
  printf("d = %lf\n", d);  /* d = 12 */
  e = frexp(d, &n);
  printf("m = %d, e = %lf\n", n, e);  /* m = 4, e = 0.75 */

  return 0;
}
```

27.7. Целая и дробная часть числа с плавающей точкой

Для определения целой и дробной части числа с плавающей точкой предназначена функция `modf`, которая имеет следующий прототип:

```
double modf(double x, int *a);
```

Функция возвращает дробную часть числа `x`, а по адресу `a` записывает целую часть этого числа. Например:

```
double n, d;
d = modf(2.5, &n);  /* n = 2.0, d = 0.5 */
```

27.8. Целые границы числа с плавающей точкой

Для определения наименьшего целого числа, которое больше заданного числа с плавающей точкой, используется функция `ceil`, которая имеет следующий прототип:

```
double ceil(double x);
```

Для нахождения наибольшего целого числа, которое меньше заданного числа с плавающей точкой, используется функция `floor`, которая имеет следующий прототип:

```
double floor(double x);
```

Например:

```
double c, f;
c = ceil(2.5);   /* c= 3.0 */
f = floor(2.5);  /* f = 2.0 */
```

27.9. Остаток от деления

Для нахождения остатка от деления двух чисел с плавающей точкой предназначена функция `fmod`, которая имеет следующий прототип:

```
double fmod(double x, double y);
```

Функция возвращает остаток от деления `x` на `y`. Например:

```
double r;
r = fmod(3.5, 1.1);  /* r = 0.2 */
```

27.10. Абсолютное значение числа

Для определения абсолютного значения числа с плавающей точкой предназначена функция `fabs`, которая имеет следующий прототип:

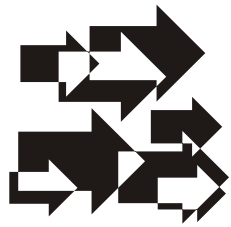
```
double fabs(double x);
```

Например:

```
double m;
m = fabs(-3.5);  /* m = 3.5 */
```

Заметим, что в стандартной библиотеке языка программирования C++ также перегружена функция `abs`, которая находит абсолютное значение числа с плавающей точкой для значения любого типа с плавающей точкой.

ГЛАВА 28



Функции классификации символов <ctype.h>

28.1. Определение типа символа

Множество символов, каждый из которых удовлетворяет некоторому условию, называется классом символов. Условие, которое определяет класс символов, называется правилом классификации символов. Определение класса символов, используя заданное правило классификации, называется классификацией символов. В стандартной библиотеке языка программирования C для классификации символов определены следующие функции:

```
int isalnum(int c);  /* буква или цифра? */
int isalpha(int c);  /* буква? */
int isupper(int c);  /* прописная буква? */
int islower(int c);  /* строчная буква? */
int iscntrl(int c);  /* управляющий символ? */
int isdigit(int c);  /* десятичная цифра? */
int isxdigit(int c); /* шестнадцатеричная цифра? */
int isspace(int c);  /* пробельный символ? */
int ispunct(int c);  /* знак препинания? */
int isprint(int c);  /* печатный символ? */
int isgraph(int c);  /* буква, цифра, знак препинания? */
```

Единственный параметр `c` каждой из этих функций должен содержать символ, тип которого проверяется. Все эти функции возвращают ненулевое значение — если условие проверки выполняется, а в противном случае они возвращают нулевое значение.

Например:

```
int n;
char s[] = "aB, 12\n";
n = isalnum(s[5]);  /* n > 0 */
n = isalpha(s[0]);  /* n > 0 */
n = isupper(s[1]);  /* n > 0 */
n = islower(s[0]);  /* n > 0 */
n = iscntrl(s[6]);  /* n > 0 */
n = isdigit(s[4]);  /* n > 0 */
n = isxdigit(s[1]); /* n > 0 */
n = isspace(s[3]);  /* n > 0 */
n = ispunct(s[2]);  /* n > 0 */
n = isprint(s[3]);  /* n > 0 */
n = isgraph(s[3]);  /* n = 0 */
```

Здесь все функции определения типа символа, за исключением функции `isgraph`, возвращают положительные значения, которые зависят от реализации. Функция `isgraph` возвращает нулевое значение.

28.2. Отображения прописных и строчных символов

Буква является прописной, если функция `isupper` возвратит ненулевое значение при передаче ей кода этой буквы как аргумента. Для отображения прописной буквы в соответствующую строчную букву предназначена функция `tolower`, которая имеет следующий прототип:

```
int tolower(int c);
```

Если параметр `c` является кодом прописной буквы, то функция `tolower` возвращает код соответствующей строчной буквы, в противном случае — значение параметра `c`.

Буква является строчной, если функция `islower` возвратит ненулевое значение при передаче ей кода этой буквы как аргумента. Для отображения строчной буквы в соответствующую пропис-

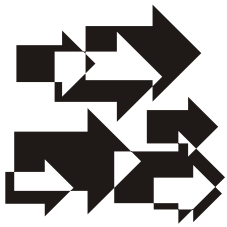
ную букву предназначена функция `toupper`, которая имеет следующий прототип:

```
int toupper(int c);
```

Если параметр `c` является кодом строчной буквы, то функция `tolower` возвращает код соответствующей прописной буквы, в противном случае — значение параметра `c`. Например:

```
char a = 'A';
char b = 'b';
printf("%c %c\n", tolower(a), toupper(b));  /* a B */
```

ГЛАВА 29



Функции для работы со строками <string.h>

Кроме функций для работы со строками в заголовочном файле string.h также объявлен тип `size_t` и макрос `NULL`. Их описание было приведено в *гл. 20*.

29.1. Сравнение строк

Для сравнения строк используются функции `strcmp` и `strncmp`.

Функция `strcmp` имеет следующий прототип:

```
int strcmp(const char *str1, const char *str2);
```

Эта функция лексикографически сравнивает строки `str1` и `str2` и возвращает отрицательное, нулевое или положительное значение, если строка `str1` соответственно меньше, равна или больше строки `str2`.

Функция `strncmp` имеет следующий прототип:

```
int strncmp(const char *str1, const char *str2, size_t n);
```

Эта функция лексикографически сравнивает не более чем `n` первых символов из строк `str1` и `str2`. Функция возвращает отрицательное, нулевое или положительное значение, если первые `n` символов из строки `str1` соответственно меньше, равны или больше первых `n` символов из строки `str2`.

В листинге 29.1 приведен пример сравнения строк при помощи функций `strcmp` и `strncmp`.

Листинг 29.1. Сравнение строк

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str1[] = "aa bb";
  char str2[] = "aa aa";
  char str3[] = "aa bb cc";

  printf("%d\n", strcmp(str1, str3));  /* печатает: -1 */
  printf("%d\n", strcmp(str1, str1));  /* печатает: 0 */
  printf("%d\n", strcmp(str1, str2));  /* печатает: 1 */
  printf("%d\n", strncmp(str1, str3, 5));  /* печатает: 0 */

  return 0;
}
```

29.2. Копирование строк

Для копирования строк используются функции `strcpy` и `strncpy`.

Функция `strcpy` имеет следующий прототип:

```
char *strcpy(char *strDst, const char *strSrc);
```

Эта функция копирует строку `strSrc` в строку `strDst`. Строка `strSrc` копируется полностью, включая завершающий нулевой байт. Функция возвращает указатель на строку `strDst`. Если строки перекрываются, то результат работы функции непредсказуем.

Функция `strncpy` имеет следующий прототип:

```
char *strncpy(char *strDst, const char *strSrc, size_t n );
```

Эта функция копирует `n` символов из строки `strSrc` в строку `strDst`. Если строка `strSrc` содержит меньше чем `n` символов, то последний нулевой байт копируется столько раз, сколько нужно для расширения строки `strSrc` до `n` символов. Функция возвращает указатель на строку `strDst`.

В листинге 29.2 приведен пример использования функций `strcpy` и `strncpy` для копирования строк.

Листинг 29.2. Копирование строк

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str1[80];
  char str2[] = "Copy string.";

  strcpy(str1, str2);
  printf(str1);  /* печатает: Copy string. */

  return 0;
}
```

29.3. Соединение строк

Для соединения двух строк в одну строку используются функции `strcat` и `strncat`.

Функция `strcat` имеет следующий прототип:

```
char* strcat(char *strDst, const char *strSrc);
```

Эта функция присоединяет строку `strSrc` к строке `strDst`, причем завершающий нулевой байт строки `strDst` стирается. Функция возвращает указатель на строку `strDst`.

Функция `strncat` имеет следующий прототип:

```
char* strncat(char *strDst, const char *strSrc, size_t n);
```

Эта функция присоединяет `n` символов из строки `strSrc` к строке `strDst`, причем завершающий нулевой байт строки `strDst` стирается. Если длина строки `strSrc` меньше `n`, то присоединяются только символы, входящие в строку `strSrc`. После соединения строк к строке `strDst` всегда добавляется нулевой байт. Функция возвращает указатель на строку `strDst`.

В листинге 29.3 приведен пример использования функций `strcat` и `strncat` для соединения строк.

Листинг 29.3. Соединение строк

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str1[80] = "String ";
  char str2[] = "catenation ";
  char str3[] = "Yes No";

  strcat(str1, str2);
  printf("%s\n", str1);  /* печатает: String catenation */
  strncat( str1, str3, 3);
  printf("%s\n", str1);  /* печатает: String catenation Yes */

  return 0;
}
```

29.4. Поиск символа в строке

Для поиска символа в строке используются функции `strchr`, `strrchr`, `strspn`, `strcspn` и `strpbrk`.

Функция `strchr` имеет следующий прототип:

```
char* strchr(const char *str, int c);
```

Эта функция ищет первое вхождение символа, заданного параметром `c`, в строку `str`. В случае успеха функция возвращает указатель на первый найденный символ, а в случае неудачи — `NULL`.

Функция `strrchr` имеет следующий прототип:

```
char* strrchr(const char *str, int c);
```

Эта функция ищет последнее вхождение символа, заданного параметром `c`, в строку `str`. В случае успеха функция возвращает указатель на последний найденный символ, а в случае неудачи — `NULL`.

В листинге 29.4 приведен пример поиска символа в строке при помощи функций `strchr` и `strrchr`.

Листинг 29.4. Поиск символа в строке

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str[80] = "Char search";
  printf("%s\n", strchr(str, 'r'));  /* печатает: r search */
  printf("%s\n", strrchr(str, 'r'));  /* печатает: rch */

  return 0;
}
```

Функция `strspn` имеет следующий прототип:

```
size_t strspn(const char *str1, const char *str2);
```

Эта функция возвращает индекс первого символа из строки `str1`, который не входит в строку `str2`.

Функция `strcspn` имеет следующий прототип:

```
size_t strcspn(const char *str1, const char *str2);
```

Эта функция возвращает индекс первого символа из строки `str1`, который входит в строку `str2`.

В листинге 29.5 приведен пример поиска символов в строке при помощи функций `strspn` и `strcspn`.

Листинг 29.5. Поиск символов в строке

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str[80] = "123 abc";
  printf("%d\n", strspn(str, "321"));  /* печатает: n = 3 */
  printf("%d\n", strcspn( str, "cba"));  /* печатает: n = 4 */

  return 0;
}
```

Функция strpbrk имеет следующий прототип:

```
char* strpbrk(const char *str1, const char *str2);
```

Эта функция находит в строке str1 первый символ, который равен одному из символов в строке str2. В случае успеха функция возвращает указатель на этот символ, а в случае неудачи — NULL.

В листинге 29.6 приведен пример использования функции strpbrk для поиска символа в строке.

Листинг 29.6. Поиск символа в строке

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str[80] = "123 abc";
  printf("%s\n", strpbrk(str, "bca"));  /* печатает: abc */

  return 0;
}
```

29.5. Поиск подстроки в строке

Для поиска подстроки в строке используется функция strstr, которая имеет следующий прототип:

```
char* strstr(const char *str1, const char *str2);
```

Эта функция находит первое вхождение строки str2 без конечного нулевого байта в строку str1. В случае успеха функция возвращает указатель на найденную подстроку, а в случае неудачи — NULL. Если указатель str1 указывает на строку нулевой длины, то функция возвращает указатель str1.

В листинге 29.7 приведен пример поиска подстроки в строке при помощи функции strstr.

Листинг 29.7. Поиск подстроки в строке

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str[80] = "123 abc 456";
  printf("%s\n", strstr(str, "abc"));  /* печатает: abc 456 */

  return 0;
}
```

29.6. Разбор строки на лексемы

Для разбора строки на лексемы используется функция `strtok`, которая имеет следующий прототип:

```
char* strtok(char *str1, const char *str2);
```

Эта функция возвращает указатель на следующую лексему в строке `str1`, в которой разделителями лексем являются символы из строки `str2`. В случае если лексемы закончились, то функция `strtok` возвращает значение `NULL`. При первом вызове функции `strtok` параметр `str1` должен указывать на строку, которая разбирается на лексемы, а при последующих вызовах этот параметр должен быть установлен в `NULL`. Функция `strtok` записывает в строку `str1` после найденной лексемы на место разделителя нулевой байт.

В листинге 29.8 приведен пример разбора строки на лексемы при помощи функции `strtok`.

Листинг 29.8. Разбор строки на лексемы

```
#include <stdio.h>
#include <string.h>

int main()
```

```
{
  char  str[ ] = "12 34 ab cd";
  char *p;
  p = strtok ( str, " " );
  while (p)
  {
    printf ( "%s\n", p );  /* печатает: 12 34 ab cd */
    p = strtok ( NULL, " " );
  }

  return 0;
}
```

29.7. Определение длины строки

Для определения длины строки используется функция `strlen`, которая имеет следующий прототип:

```
size_t strlen(const char *str);
```

Эта функция возвращает длину строки, не учитывая последний нулевой байт.

В листинге 29.9 приведен пример определения длины строки при помощи функции `strlen`.

Листинг 29.9. Определение длины строки

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str[] = "123";
  printf ("%d\n", strlen(str));  /* печатает 3 */

  return 0;
}
```

29.8. Функции для работы с памятью

Для работы с блоками памяти используются функции `memchr`, `memcmp`, `memcpy`, `memmove` и `memset`.

Функция `memchr` имеет следующий прототип:

```
void* memchr(const void *str, int c, size_t n);
```

Эта функция ищет первое вхождение символа, заданного параметром `c`, в `n` байтах массива `str`. При поиске параметр `c` приводится к типу `unsigned char`. В случае успеха функция возвращает адрес первого найденного символа, а в случае неудачи — `NULL`.

Функция `memcmp` имеет следующий прототип:

```
int memcmp(const void *str1, const void *str2, size_t n);
```

Эта функция сравнивает `n` байтов массивов `str1` и `str2`. Функция возвращает отрицательное, нулевое или положительное значение, если первые `n` байтов массива `str1` соответственно меньше, равны или больше первых `n` байтов массива `str2`. При сравнении элементы массивов приводятся к типу `unsigned char`.

Функция `memcpy` имеет следующий прототип:

```
void* memcpy(const void *dst, const void *src, size_t n);
```

Эта функция копирует первые `n` байтов из массива `src` в массив `dst`. Если массивы перекрываются, то результат работы функции непредсказуем. Функция возвращает адрес массива `dst`.

Функция `memmove` имеет следующий прототип:

```
void* memmove(const void *dst, const void *src, size_t n);
```

Эта функция копирует первые `n` байтов из массива `src` в массив `dst`, обеспечивая корректную обработку перекрывающихся массивов. Функция возвращает адрес массива `dst`.

Функция `memset` имеет следующий прототип:

```
void* memset(const void *str, int c, size_t n);
```

Эта функция копирует символ, заданный параметром `c`, в первые `n` байтов массива строки `str`. При копировании параметр `c`

приводится к типу `unsigned char`. Функция возвращает адрес массива `str`.

В листинге 29.10 приведены примеры использования функций для работы с памятью.

Листинг 29.10. Примеры использования функций для работы с памятью

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str[] = "Char 1";
  char stq[] = "Char 2";
  char move[] = "char1 char2";

  /* печатает: r 1 */
  printf("%s\n", memchr(str, 'r', sizeof(str)));
  /* печатает: -1 */
  printf("%d\n", memcmp(str, stq, sizeof(str)));
  /* печатает: Char 1 */
  printf("%s\n", memcpy(stq, str, sizeof(stq)));
  /* печатает: char2 char2 */
  printf("%s\n", memmove(move, move + 6, 5));
  /* печатает: xxxxx char2 */
  printf("%s\n", memset(move, 'x', 5));

  return 0;
}
```

29.9. Сравнение строк, содержащих локальные символы

Для сравнения строк, содержащих локальные символы, используются функции `strcoll` и `strxfrm`. Под локальными символами подразумеваются символы, входящие в кодовую таблицу, используемую на компьютере. Эти символы могут принадлежать

национальным алфавитам и, следовательно, отличаться от латинских букв.

Для сравнения строк, содержащих локальные символы, используется функция `strcoll`, которая имеет следующий прототип:

```
int strcoll(const char *str1, const char *str2);
```

Эта функция лексикографически сравнивает строки `str1` и `str2` и возвращает отрицательное, нулевое или положительное значение, если строка `str1` соответственно меньше, равна или больше строки `str2`.

Так как последовательность кодов локальных символов не обязательно соответствует по величине лексикографической последовательности этих символов, то для лексикографического сравнения строк, содержащих локальные символы, и используется специальная функция `strcoll`. При сравнении эта функция учитывает значение символической константы `LC_COLLATE`, которая определена в заголовочном файле locale.h.

В листинге 29.11 приведен пример сравнения строк, содержащих локальные символы, при помощи функции `strcoll`.

Листинг 29.11. Сравнение строк, содержащих локальные символы

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str1[] = "ээ ююю";
  char str2[] = "ээ ээ";

  printf("%d\n", strcoll(str1, str1));  /* печатает: 0 */
  printf("%d\n", strcoll(str1, str2));  /* печатает: 1 */

  return 0;
}
```

Функция `strxfrm` имеет следующий прототип:

```
size_t strxfrm(char *str1, const char *str2, size_t n);
```

Эта функция преобразует n символов из строки str2, содержащей локальные символы к виду, в котором эти символы могут участвовать в сравнении строк, используя функцию strcmp. Преобразованная строка записывается в строку str1. Результат сравнения преобразованных строк, используя функцию strcmp, такой же, как и результат сравнения не преобразованных строк при помощи функции strcoll. При преобразовании символов эта функция учитывает значение символической константы LC_COLLATE, которая определена в заголовочном файле locale.h.

В листинге 29.12 приведен пример преобразования строк, содержащих локальные символы, при помощи функции strxfrm.

Листинг 29.12. Преобразование строк, содержащих локальные символы

```
#include <stdio.h>
#include <string.h>

int main()
{
  char str1[] = "ээ юю";
  char s1[10];

  strxfrm(s1, str1, sizeof(str1));
  printf("%d\n", strcmp(s1, s1));  /* печатает: 0 */

  return 0;
}
```

29.10. Получение строки, описывающей код ошибки

Для получения строки, описывающей код ошибки, используется функция strerror, которая имеет следующий прототип:

```
char* strerror(int errcode);
```

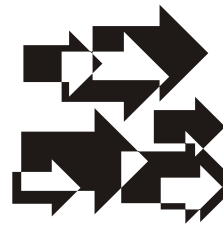
Эта функция возвращает адрес строки, которая описывает код ошибки, заданный параметром errcode. На практике этот пара-

метр обычно представляется значением символической константы `errno`, которая описана в заголовочном файле errno.h.

В листинге 29.13 приведен пример получения строки, описывающей код ошибки, при помощи функции `strerror`.

Листинг 29.13. Получение строки, описывающей код ошибки

```
#include <stdio.h>
#include <string.h>
#include <math.h>

extern int errno;

int main()
{
  double y = fmod(1,0);  /* остаток от деления 1 на 0 */
  printf("%s\n", strerror(errno));  /* печать: Domain error */

  return 0;
}
```



ГЛАВА 30

Функции для работы с файлами <stdio.h>

30.1. Файлы и потоки

Файлом называют последовательность однотипных данных, хранящихся на внешнем носителе информации. Под *доступом к файлу* понимают запись и чтение данных из файла. *Потоком* называется логический или, другими словами, программный интерфейс, который обеспечивает доступ к файлу. Прежде чем использовать поток для доступа к файлу, его необходимо соединить с этим файлом, т. е. обеспечить поток информацией о файле. Эта информация хранится в структуре типа `FILE`. Потому считается, что поток имеет тип `FILE*`, т. е. является указателем. Когда поток соединяют с файлом, то говорят, что *файл открывают*. Когда поток отсоединяют от файла, то говорят, что *файл закрывают*.

Каждый поток может работать в двух режимах: текстовом и бинарном. Режим работы потока задается при его соединении с файлом.

В текстовом режиме поток записывает и читает из файла текстовые строки, которые заканчиваются символом `\n` (переход на новую строку) и могут содержать символ `\t` (табуляция). По стандарту поток должен обеспечивать обработку строк длиной не менее 254 символа, включая символ `\n`. Стандартом допускается, что при чтении и записи данных текстовым потоком может происходить их преобразование.

В бинарном режиме поток записывает и читает данные из файла в том виде, в котором они хранятся в оперативной памяти.

Поясним подробнее разницу между текстовым и бинарным режимами работы потока. Как в текстовом, так и в бинарном режиме можно использовать все функции для доступа к файлу. При работе в бинарном режиме поток записывает на диск точные копии данных из оперативной памяти и считывает в оперативную память точные копии данных с диска. Работа потока в текстовом режиме отличается от работы потока в бинарном режиме тремя моментами.

Во-первых, в этом случае комбинация символов <Ctrl>+<Z> интерпретируется как конец файла.

Во-вторых, при записи в текстовый поток комбинации символов `\r` (возврат каретки) и `\n` в файл записывается только символ `\n`. А при чтении из текстового потока символ `\n` наоборот преобразуется в комбинацию символов `\r\n`.

В-третьих, т. к. при записи в текстовый поток может происходить преобразование количества и представления символов, то для получения позиции в файле, связанном с текстовым потоком, нужно использовать только функции `fgetpos` и `ftell`.

Из вышеизложенного следует, что файлы не являются бинарными или тестовыми, такие характеристики присущи потокам. Поэтому к каждому файлу можно получить доступ как с помощью бинарного, так и текстового потока. Но, для того чтобы избежать ошибок, связанных с неправильным доступом к файлу, следует придерживаться следующего правила: не нужно смешивать бинарный и текстовый потоки при доступе к одному файлу. Если данные в файл записывались бинарным потоком, то и читать эти данные из файла следует также бинарным потоком. Это же замечание справедливо и для текстовых потоков.

В следующих разделах будут описаны стандартные функции для работы с файлами, используемые в языке программирования С.

30.2. Открытие файла

Для соединения потока с файлом используется функция `fopen`, которая имеет следующий прототип:

```
FILE* fopen(const char* filename, const char* mode);
```

Эта функция открывает файл, имя которого задано параметром `filename`, в режиме, заданном параметром `mode`. В случае успешного завершения функция `fopen` возвращает указатель на поток, а в случае неудачи — `NULL`.

Параметр `mode` может принимать следующие значения:

- `"r"` — чтение в текстовом режиме;
- `"w"` — запись в текстовом режиме;
- `"a"` — присоединение в текстовом режиме;
- `"rb"` — чтение в бинарном режиме;
- `"wb"` — запись в бинарном режиме;
- `"ab"` — присоединение в бинарном режиме;
- `"r+"`, `"w+"` или `"a+"` — чтение и запись в текстовом режиме;
- `"r+b"`, `"w+b"` или `"a+b"` — чтение и запись в бинарном режиме;
- `"rb+"`, `"wb+"` или `"ab+"` — чтение и запись в бинарном режиме.

При открытии файла в режимах `"r"`, `"rb"`, `"r+"` или `"r+b"` его индикатор позиции устанавливается на начало файла. В случае если открывается несуществующий файл, то функция `fopen` заканчивается неудачей.

При открытии файла в режимах `"w"`, `"wb"`, `"w+"` или `"w+b"` создается новый файл. Если файл с заданным именем существует, то его содержимое стирается, а индикатор позиции устанавливается на начало файла.

При открытии файла в режимах `"a"`, `"ab"`, `"a+"` или `"a+b"` создается новый файл. Если файл с заданным именем существует, то он открывается, и индикатор позиции устанавливается на конец файла.

Следует учитывать, что если текстовый файл открывается в режиме чтения и записи, то базовая операционная система может

открыть его в бинарном режиме. Максимальное количество файлов, которые можно открыть одновременно задается макросом FOPEN_MAX. Максимальная длина имени файла задается макросом FILENAME_MAX.

30.3. Перенаправление потока

Для перенаправления потока используется функция freopen, которая имеет следующий прототип:

```
FILE* freopen(const char* filename,
              const char* mode, FILE* stream);
```

Эта функция закрывает файл, соединенный с потоком stream, и соединяет с этим потоком файл filename в режиме mode. В случае успеха функция возвращает указатель на поток, а в случае неудачи — NULL. Параметр mode может принимать те же значения, что и в функции fopen.

30.4. Закрытие файла

Для отсоединения потока от файла используется функция fclose, которая имеет следующий прототип:

```
int fclose(FILE* stream);
```

Эта функция закрывает файл, при этом освобождая все буферы потока. При успешном завершении функция возвращает 0, а в случае неудачи — значение EOF (End Of File, конец файла).

30.5. Работа с индикатором ошибки

С каждым потоком связан индикатор ошибки, который находится в установленном положении, если в потоке, связанном с файлом, произошла ошибка. В противном случае индикатор ошибки находится в сброшенном состоянии. Для работы с индикатором ошибки используются функции ferror и clearerr.

Функция ferror имеет следующий прототип:

```
int ferror(FILE* stream);
```

Эта функция возвращает ненулевое значение, если индикатор ошибки установлен, в противном случае возвращает 0.

Функция clearerr имеет следующий прототип:

```
void clearerr(FILE* stream);
```

Эта функция сбрасывает индикаторы ошибки и конца файла для потока stream.

30.6. Работа с индикатором конца файла

Структура FILE содержит индикатор конца файла, который устанавливается в ненулевое значение функцией чтения из файла при достижении этой функцией конца файла. Состояние конца файла читается функцией feof, которая имеет следующий прототип:

```
int feof(FILE* stream);
```

Эта функция возвращает ненулевое значение, если индикатор конца файла установлен, в противном случае функция возвращает 0.

30.7. Работа с индикатором позиции файла

Для каждого файла, после его открытия, определяется индикатор позиции, который указывает на смещение от начала файла в байтах. Тип индикатора позиции обычно определяется как

```
typedef long fpos_t;
```

Для работы с индикатором позиции предназначены следующие функции rewind, fseek, fsetpos и fgetpos.

Функция rewind имеет следующий прототип:

```
void rewind(FILE* stream);
```

Эта функция устанавливает индикатор позиции на начало файла, связанного с потоком stream. При этом сбрасываются индикаторы ошибки и конца файла.

Функция `fseek` имеет следующий прототип:

```
int fseek(FILE* stream, long offset, int mode);
```

Эта функция сдвигает индикатор позиции файла на `offset` байтов. В случае успешного завершения функция возвращает `0`, в противном случае — ненулевое значение. Параметр `mode` указывает на режим сдвига и может принимать значения, определенные следующими символическими константами:

- `SEEK_SET` — смещение от начала файла;
- `SEEK_CUR` — смещение от текущей позиции;
- `SEEK_END` — смещение от конца файла.

При работе с текстовым потоком должны использоваться только следующие комбинации значений параметров:

- `mode = SEEK_SET`, а `offset` равно 0 или значению, возвращенному функцией `ftell`;
- `mode = SEEK_CUR` и `offset = 0`;
- `mode = SEEK_END` и `offset = 0`.

Функция `fsetpos` имеет следующий прототип:

```
int fsetpos(FILE* stream, const fops_t *pos);
```

Эта функция устанавливает индикатор позиции файла `stream` в позицию, на которую указывает параметр `pos`. Индикатор конца файла сбрасывается. В случае успеха функция возвращает `0`, а в случае неудачи — ненулевое значение и устанавливает переменную `errno`.

Функция `ftell` имеет следующий прототип:

```
long ftell(FILE* stream);
```

Эта функция в случае успешного завершения возвращает текущую позицию файла `stream`, а в случае неудачи — возвращает значение `-1L` и устанавливает значение переменной `errno`. Для бинарного потока позиция равна смещению в байтах от начала файла, а в случае текстового потока — значению, которое может использоваться функцией `fseek`.

Функция fgetpos имеет следующий прототип:

```
int fgetpos(FILE* stream, fops_t* pos);
```

Эта функция записывает текущую позицию файла stream по адресу pos. В случае успеха функция возвращает 0, а в случае неудачи — ненулевое значение и устанавливает значение переменной errno.

30.8. Блочный ввод/вывод

Блоком называется область оперативной памяти, содержимое которой записывается в файл. Ввод/вывод блоками используется главным образом при работе с бинарными потоками.

Для записи блока в файл используется функция fwrite, которая имеет следующий прототип:

```
size_t fwrite(const void* ptr,
              size_t size, size_t nitems, FILE* stream);
```

Эта функция записывает содержимое блока памяти, на который указывает параметр ptr, в файл stream. Длина записываемого блока определяется как произведение size*nitems. Функция возвращает количество записанных единиц памяти. В случае удачи это число должно быть равно nitems. В случае неудачи функция fwrite устанавливает индикатор ошибки.

Для чтения блока из файла используется функция fread, которая имеет следующий прототип:

```
size_t fread(const void* ptr,
             size_t size, size_t nitems, FILE* stream);
```

Параметры этой функции имеют тот же смысл, что и параметры функции fwrite. Функция возвращает количество прочитанных единиц памяти. В случае удачи это число должно быть равно nitems. В случае неудачи функция fwrite устанавливает индикатор ошибки или индикатор конца файла.

В случае ошибки состояние индикатора позиции после работы функций fwrite и fread не определено. Иначе индикатор позиции сдвигается на количество записанных и прочитанных байтов.

В листинге 30.1 приведена программа, которая создает бинарный файл.

Листинг 30.1. Запись блоков в файл

```
#include <stdio.h>
struct emp
{
  int code;
  char name[20];
  double salary;
};

int main()
{
  int i, n;
  FILE* out;        /* выходной поток */
  struct emp s;     /* для записей файла */
  s.salary = 0.0;   /* для обработки плавающих чисел */

  /* открываем выходной поток в бинарном режиме */
  out = fopen("employee.bin", "wb");

  printf("Input a number of records to write: ");
  scanf("%d", &n);
  if (n < 0)
    return 0;
  printf("Input code, name and salary.\n");
  for (i = 0; i < n; ++i)
  {
    printf("%d> ", i + 1);
    /* вводим запись с консоли */
    scanf("%d%s%lf", &s.code, &s.name, &s.salary);
    /* пишем запись в файл */
    fwrite(&s, sizeof(struct emp), 1, out);
  }
  /* закрываем выходной поток */
  fclose(out);

  return 0;
}
```

В листинге 30.2 приведена программа, которая выполняет чтение записей из созданного бинарного файла.

Листинг 30.2. Чтение блоков из файла

```
#include <stdio.h>
struct emp
{
  int code;
  char name[20];
  double salary;
};

int main()
{
  FILE* in;         /* выходной поток */
  struct emp s;     /* для записей файла */
  s.salary = 0.0;   /* для обработки плавающих чисел */
  long i;           /* номер записи */

  /* открываем входной поток в бинарном режиме */
  if(!(in = fopen("employee.bin", "rb")))
  {
    printf("Open file failed.\n");
    return 0;
  }
  printf("Input negative index to exit.\n");

  for (;;)
  {
    /* читаем индекс */
    printf("Input an index: ");
    scanf("%d", &i);
    if (i < 0)
      break;
    /* устанавливает указатель на нужную запись */
    fseek(in, i*sizeof(struct emp), SEEK_SET);
    /* читаем запись из файла */
    if(!fread(&s, sizeof(struct emp), 1, in))
```

```
    {
      printf("The wrong index.\n");
      continue;
    }
    /* выводим запись на консоль */
    printf("\tcode = %d name = %s sal = %f\n",
      s.code, s.name, s.salary);
  }

  /* закрываем входной поток */
  fclose(in);

  return 0;
}
```

30.9. Ввод/вывод символов

Для записи и чтения символов (байтов) из файла используются функции `fputc`, `putc`, `fgetc` и `getc`.

Функция `fputc` имеет следующий прототип:

```
int fputc(int c, FILE* stream);
```

Эта функция записывает символ, заданный параметром `c`, в поток `stream` и продвигает индикатор позиции на следующий символ. В случае успеха функция возвращает символ `c`, а в случае неудачи — значение `EOF` и устанавливает индикатор ошибки.

Функция `putc` имеет следующий прототип:

```
int putc(int c, FILE* stream);
```

Эта функция работает так же, как и функция `fputc`, но может быть реализована как макрокоманда.

Пример использования функции `putc` приведен в листинге 30.3.

Листинг 30.3. Запись символов в файл

```
#include <stdio.h>
int main()
```

```
{
  FILE* out;
  char str[80];
  int n = 0;

  /* открываем поток для записи в текстовом режиме */
  out = fopen("text.txt", "w");

  printf("Input a string:\n");
  /* ввод строки с консоли */
  gets(str);
  /* пишем символы в файл */
  while (str[n])
    fputc(str[n++], out);
  /* закрываем выходной поток */
  fclose(out);

  return 0;
}
```

Функция `fgetc` имеет следующий прототип:

```
int fgetc(FILE* stream);
```

Эта функция читает символ из потока `stream` и продвигает индикатор позиции на следующий символ. В случае успешного завершения функция возвращает прочитанный символ. В случае достижения конца файла функция возвращает значение `EOF` и устанавливает индикатор конца файла. В случае ошибки функция возвращает значение `EOF` и устанавливает индикатор ошибки.

Функция `getc` имеет следующий прототип:

```
int getc(FILE* stream);
```

Эта функция работает так же, как и функция `fgetc`, но может быть реализована как макрокоманда.

Функция `ungetc` имеет следующий прототип:

```
int ungetc(int c, FILE* stream);
```

Эта функция возвращает символ `c` в поток `stream`. Функции `fseek`, `fsetpos` и `rewind` игнорируют такие символы. Доступ к записан-

ным символам выполняется по правилу FIFO. В случае успеха функция возвращает записанный символ, а в случае неудачи — EOF.

В листинге 30.4 приведен пример чтения символов из файла, созданного программой из листинга 30.3.

Листинг 30.4. Чтение символов из файла

```
#include <stdio.h>
int main()
{
  FILE* in;
  char c;

  /* открываем поток для чтения в текстовом режиме */
  if(!(in = fopen("text.txt", "r")))
  {
    printf("Open file failed.\n");
    return 0;
  }
  c = fgetc(in);  /* читаем первый символ из файла */
  while (!feof(in))
  {
    putc(c, stdout); /* выводим символ на консоль */
    c = fgetc(in);   /* читаем остальные символы из файла*/
  }
  putc('\n', stdout);
  /* закрываем входной поток */
  fclose(in);

  return 0;
}
```

30.10. Запись/чтение строк из файла

Для записи/чтения строки из файла в языке C предназначены функции fputs и fgets. Эти функции используются главным образом при работе с символьными потоками.

Функция `fputs` имеет следующий прототип:

```
int fputs(const char* str, FILE* stream);
```

Эта функция записывает строку `str` в файл `stream`, не включая завершающий нулевой байт. В случае успешного завершения функция возвращает ненулевое число, а в случае неудачи — `EOF`.

Пример использования функции `fputs` приведен в листинге 30.3.

Листинг 30.5. Запись строк в файл

```
#include <stdio.h>
int main()
{
  FILE* out;
  char str[80];

  /* открываем поток для записи в текстовом режиме */
  out = fopen("text.txt", "w");

  printf("Press Ctrl+z to exit.\n");
  printf("Input strings:\n");
  /* ввод строк с консоли */
  gets(str);
  while (!feof(stdin))
  {
    fputs(str, out);  /* пишем строку в файл */
    putc('\n', out);  /* переход на новую строку в файл */
    gets(str);
  }
  /* закрываем выходной поток */
  fclose(out);

  return 0;
}
```

Функция `fgets` имеет следующий прототип:

```
int fgets(char* str, int n, FILE* stream);
```

Эта функция читает строку из потока `stream` в строку `str`. Останавливается функция в случае, если прочитан `(n-1)` символ, или

встретился символ \n, или обнаружен конец файла. В любом из этих случаев в конец строки помещается символ \n, за которым записывается пустой символ \0. В случае успеха функция возвращает указатель str, а в случае неудачи — NULL. Если не прочитан ни один символ и обнаружен конец файла, то строка str не изменяется, а функция возвращает значение NULL.

В листинге 30.6 приведен пример чтения строк из файла, который был создан программой, приведенной в листинге 30.5.

Листинг 30.6. Чтение строк из файла

```
#include <stdio.h>
int main()
{
  FILE* in;
  char str[80];

  /* открываем поток для чтения в текстовом режиме */
  if(!(in = fopen("text.txt", "r")))
  {
    printf("Open file failed.\n");
    return 0;
  }
  while (fgets(str, 80, in))  /* чтение строк из файла */
    fputs(str, stdout);       /* выводим строку на консоль */
  /* закрываем выходной поток */
  fclose(in);

  return 0;
}
```

30.11. Форматированный вывод

Для форматированного вывода данных в файл предназначена функция fprintf, которая имеет следующий прототип:

```
int fprintf(FILE* stream, const char* format, ...);
```

Эта функция выполняет запись данных в файл stream в соответствии с форматной строкой format. Параметр format представляет

собой форматируемую строку, которая содержит спецификации форматов вывода. Количество и типы аргументов, которые следуют после строки форматирования, должны соответствовать количеству и типам спецификаций формата вывода, заданным в строке форматирования. Если эти условия не выполняются, то поведение функции не определено.

В случае успешного завершения функция возвращает количество отформатированных данных, а в случае неудачи — отрицательное значение и устанавливает индикатор ошибки.

В общем случае спецификация формата вывода имеет вид:

```
%[флаги][ширина][.точность][модификатор]тип
```

где:

- `флаги` — это символы, уточняющие формат вывода;
- `ширина` задает минимальное количество символов, выводимых по данной спецификации;
- `точность` задает максимальное или минимальное количество символов, выводимых по данной спецификации в зависимости от `типа`;
- `модификатор` уточняет `тип` аргумента;
- `тип` задает тип аргумента.

Для всех форматов вывода может быть установлен флаг –, который выравнивает поле вывода влево. Если этот флаг не установлен, то поле вывода выравнивается вправо.

Если поле `ширина` опущено в спецификации формата вывода, то по умолчанию она принимается равной `6` для типов данных с плавающей точкой и `1` для остальных типов данных. Если поле вывода аргумента больше, чем заданная ширина, то недостающие символы заполняются пробелами.

Перед десятичным числом, обозначающим `ширину`, может быть записан ноль. В этом случае ноль рассматривается как `флаг`, который предписывает заполнять поле вывода предшествующими нулями до достижения заданной ширины.

Если вместо десятичного числа, определяющего `ширину`, в спецификации указан символ `*`, то ширина поля вывода задается значе-

нием текущего аргумента функции `fprintf`. Если это число отрицательное, то знак минус рассматривается как флаг, который выравнивает вывод влево, а величина числа определяет ширину поля вывода.

Для вывода одного символа используется следующий формат:

```
%[-][0][ширина][h|l]c
```

где:

- `h` обозначает однобайтовый символ;
- `l` обозначает широкий (wide) символ;
- `c` обозначает данные типа `char`.

Для вывода строки используется следующий формат:

```
%[-][0][ширина][.точность][h|l]s
```

где:

- `точность` задает максимальное количество выводимых символов;
- `h` обозначает строку однобайтовых символов;
- `l` обозначает строку широких (wide) символов;
- `s` обозначает, что выводится строка.

Для вывода целого числа со знаком используется следующий формат:

```
%[-][+|пробел][0][ширина][.точность][h|l]{d|i}
```

где:

- `+` выводит знак `+` для положительных чисел, заметим, что для отрицательных чисел всегда выводится знак `-`;
- `пробел` выводит пробел в позицию знака, если число положительное;
- `точность` задает минимальное количество цифр, которые должны быть выведены, если цифр меньше, то пустые места заполняются пробелами или нулями в зависимости от установленного флага;
- `h` обозначает модификатор типа `short`;

- `l` обозначает модификатор типа `long`;
- `d` обозначает число типа `int`;
- `i` обозначает число типа `int`.

Для вывода целого числа без знака используется следующий формат:

```
%[-][#][0][ширина][.точность][h|l]{u|o|x|X}
```

где:

- `#` выводит начальный `0` для чисел в восьмеричной системе счисления или начальные `0x` или `0X` для чисел в шестнадцатеричной системе счисления;
- `точность` задает минимальное количество цифр, которые должны быть выведены, если цифр меньше, то пустые места заполняются пробелами или нулями в зависимости от установленного флага;
- `h` обозначает модификатор типа `short`;
- `l` обозначает модификатор типа данных `long`;
- `u` обозначает число типа `unsigned`, которое выводится в десятичной системе счисления;
- `o` обозначает число типа `unsigned`, которое выводится в восьмеричной системе;
- `x` обозначает число типа `unsigned`, которое выводится шестнадцатеричной системе;
- `X` обозначает данные типа `unsigned`, которое выводится в шестнадцатеричной системе.

Для вывода числа с плавающей точкой используется следующий формат:

```
%[-][#][+|пробел][0][ширина][.точность][L]{f|e|E|g|G}
```

где:

- `#` в любом случае выводит десятичную точку для типов `f`, `e` или `E`, а для типов `g` и `G`, кроме того, предотвращает подавление завершающих нулей;

- `+` выводит знак `+` для положительных чисел, заметим, что для отрицательных чисел всегда выводится знак `-`;
- `пробел` выводит пробел в позицию знака;
- `точность` задает количество цифр после десятичной точки для форматов `f`, `e` и `E` или количество значащих цифр для форматов `g` и `G`. Числа округляются отбрасыванием. По умолчанию принимается точность в шесть десятичных цифр;
- `L` обозначает модификатор типа данных `long`;
- `f` обозначает число с плавающей точкой типа `double`;
- `e` обозначает число с плавающей точкой типа `double`, которое выводится в экспоненциальной форме, экспонента обозначается буквой `'e'`;
- `E` обозначает число с плавающей точкой типа `double`, которое выводится в экспоненциальной форме, экспонента обозначается буквой `E`;
- `g` обозначает число с плавающей точкой типа `double`, для вывода которого используется наиболее короткий из форматов `f` или `e`;
- `G` обозначает число с плавающей точкой типа `double`, для вывода которого используется наиболее короткий из форматов `f` или `E`.

Для вывода указателя используется следующий формат:

```
%[-][+|пробел][ширина]p
```

где:

- `+` или `пробел` выравнивает поле вывода вправо;
- `p` обозначает данные типа `void*`.

Для определения количества символов, которые были успешно записаны в поток вывода, используется следующий формат:

```
%[l]n
```

где:

- `l` обозначает модификатор типа `long`;
- `n` указывает, что аргумент должен быть указателем на тип `int`.

В этом случае аргумент функции `fprintf` должен быть указателем на целую переменную. В эту переменную функция `fprintf` запишет количество успешно выведенных символов.

В листинге 30.7 приведены примеры использования функции `fprintf`.

Листинг 30.7. Форматированный вывод

```
#include <stdio.h>
int main()
{
  FILE *out;
  int n = 0;
  int *p = &n;

  out = fopen("text.txt", "w");

  fprintf(out, "c: %c\n", 'a');     /* a */
  fprintf(out, "lc: %lc\n", L'a');  /* a */

  fprintf(out, "s: %s\n", "string");     /* string */
  fprintf(out, "ls: %ls\n", L"string");  /* string */

  fprintf(out, "d: %d\n", -123);    /* -123 */
  fprintf(out, "d: %05d\n", -123);  /* -0123 */

  fprintf(out, "f: %f\n", 12.34);  /* 12.340000 */
  fprintf(out, "e: %e\n", 12.34);  /* 1.234000 */
  fprintf(out, "E: %E\n", 12.34);  /* 1.234000 */
  fprintf(out, "g: %g\n", 12.34);  /* 12.34 */
  fprintf(out, "G: %G\n", 12.34);  /* 12.34 */

  fprintf(out, "p: %p\n", p);
  fprintf(out, "%c%n\n", 'a', p);  /* a */
  fprintf(out, "n = %d\n", n);     /* n = 1 */

  fclose(out);

  return 0;
}
```

30.12. Форматированный ввод

Для форматированного ввода данных используется функция `fscanf`, которая имеет следующий прототип:

```
int fscanf(FILE* stream, const char* format, ...);
```

Эта функция выполняет чтение данных из файла `stream` в соответствии с форматной строкой `format`. Параметр `format` должен указывать на форматируемую строку, которая содержит спецификации форматов ввода. Количество и типы аргументов, которые следуют после строки форматирования, должны соответствовать количеству и типам форматов ввода, заданным в строке форматирования. Если это условие не выполняется, то поведение функции не определено. Кроме того, аргументы должны задавать адреса переменных, в которые будут вводиться данные.

В случае успешного завершения функция возвращает количество отформатированных данных. В случае ошибки функция устанавливает индикатор ошибки и возвращает значение `EOF`. В случае достижения конца файла функция устанавливает индикатор конца файла. Если при достижении конца файла не было выполнено ни одного преобразования формата, то функция возвращает значение `EOF`.

Пробел, символы `\t` или `\n` в форматной строке описывают один или более пробельных символов во входном потоке, к которым относятся символы: пробел, `\t`, `\n`, `\v` и `\f`. Функция `fscanf` пропускает пустые символы во входном потоке.

Литеральные символы в форматной строке, за исключением символа `%`, требуют, чтобы во входном потоке появились точно такие же символы. Если такого символа нет, то функция `fscanf` прекращает ввод. Функция `fscanf` пропускает литеральные символы.

В общем случае спецификация формата ввода имеет вид:

```
%[*][ширина][модификатор]тип
```

где:

- `*` обозначает пропуск при вводе поля, определенного данной спецификацией;

- ширина задает максимальное количество символов, вводимых по данной спецификации;
- модификатор уточняет тип аргумента;
- тип задает тип аргумента.

Для ввода массива символов используется следующий формат:

```
%[*][ширина][l]c
```

где:

- ширина задает количество символов, которые должны быть введены из потока; если ширина опущена, то вводится один символ;
- l указывает, что вводится массив широких (wide) символов;
- c указывает, что вводится массив символов.

Для ввода массива символов также используется следующий формат:

```
%[*][ширина][l]s
```

где:

- ширина задает количество символов, которые должны быть введены из потока; если ширина опущена, то вводится строка символов до первого пробельного символа;
- l указывает, что вводится массив широких (wide) символов;
- s указывает, что вводится массив символов.

В этом случае по окончании ввода символов функция `fscanf` записывает в массив пустой символ.

Для ввода целого числа со знаком используется следующий формат:

```
%[*][ширина][h|l]{d|i}
```

где:

- h обозначает модификатор типа `short`;
- l обозначает модификатор типа `long`;
- d обозначает число типа `int`, которое записано в десятичной системе счисления;

- `i` обозначает число типа `int`, которое записано в восьмеричной или шестнадцатеричной системе счисления.

Правила записи целых чисел, используя восьмеричную и шестнадцатеричную системы счисления, совпадают с правилами обозначения числовых констант в этих же системах счисления.

Для ввода целого числа без знака используется следующий формат ввода:

```
%[*][ширина][h|l]{u|o|x|X}
```

где:

- `h` обозначает модификатор типа `short`;
- `l` обозначает модификатор типа данных `long`;
- `u` обозначает число типа `unsigned`, которое записано в десятичной системе счисления;
- `o` обозначает число типа `unsigned`, которое записано в восьмеричной системе счисления;
- `x` и `X` обозначают число типа `unsigned`, которое записано в шестнадцатеричной системе счисления.

Для ввода числа с плавающей точкой используется следующий формат:

```
%[*][ширина][l|L]{f|e|E|g|G}
```

где:

- `l` обозначает, что вводится число с плавающей точкой в переменную типа `double`;
- `L` обозначает, что вводится число с плавающей точкой в переменную типа `long double`;
- `f`, `e`, `E`, `g` или `G` обозначает число с плавающей точкой, которое вводится в переменную типа `float`. Это число может содержать как точку, так и экспоненту, которая должна начинаться с буквы `e` или `E`.

Для ввода указателя используется следующий формат:

```
%[*][ширина]p
```

где `p` обозначает, что вводится указатель.

Для определения количества символов, которые были успешно прочитаны из потока ввода, используется следующий формат:

```
%[l]n
```

где:

- `l` обозначает модификатор типа `long`;
- `n` указывает, что аргумент должен быть адресом на указатель типа `int*` или `long int*`.

В этом случае в переменную, на которую указывает аргумент, функция `fscanf` запишет количество успешно введенных символов.

В листинге 30.8 приведены примеры использования функции `fscanf`.

Листинг 30.8. Форматированный ввод

```
#include <stdio.h>

int main()
{
  FILE *out, *in;
  char  c;
  char s[10];
  int  n;
  unsigned m;
  double  d = 0.0;

  char str[] = "a abcd -10 0x123 12.3E-1";

  /* создаем текстовый файл */
  out = fopen("text.txt", "w");
  fputs(str, out);  /* пишем строку в файл */
  fclose(out);

  /* открываем поток для чтения в текстовом режиме */
  in = fopen("text.txt", "r");
  /* читаем из входного потока */
  fscanf(in, "%c", &c);
```

```
  printf("%c\n", c);
  fscanf(in, "%s", s);
  printf("%s\n", s);

  fscanf(in, "%d", &n);
  printf("%d\n", n);
  fscanf(in, "%x", &m);
  printf("%#x\n", m);

  fscanf(in, "%lf", &d);
  printf("%f\n", d);

  fclose(in);

  return 0;
}
```

30.13. Форматирование и сканирование строк

Существуют варианты функций `fprintf` и `fscanf`, которые предназначены для форматирования строк и называются соответственно `sprintf` и `sscanf`.

Для форматирования строки предназначена функция `sprintf`, которая имеет следующий прототип:

```
int sprintf(char *buffer, const char *format, ...);
```

Эта функция форматирует строку в соответствии с форматом, который задан параметром `format`, и записывает полученный результат в символьный массив `buffer`. Возвращает функция количество символов, записанных в символьный массив `buffer`, исключая завершающий нулевой байт.

В листинге 30.9 приведен пример форматирования строки.

Листинг 30.9. Форматирование строки

```
#include <stdio.h>
int main()
```

```
{
  char buffer[80];

  char str[] = "c:%c d:%d g:%g s:%s";
  char c = 'c';
  int n = 10;
  double d = 1.2;
  char s[] = "The string";

  sprintf(buffer, str, c, n, d, s);
  printf("%s\n", buffer);  /* c:c n:10 d:1.2 s:The string */

  return 0;
}
```

Для сканирования строки предназначена функция `sscanf`, которая имеет следующий прототип:

```
int sscanf(const char *str, const char *format, ...);
```

Эта функция читает данные из строки, заданной параметром `str`, в соответствии с форматной строкой, заданной параметром `format`. В случае удачи функция возвращает количество отсканированных полей. В случае ошибки или в случае, если по достижении конца строки `str` не было отсканировано ни одного поля, функция возвращает значение `EOF`.

В листинге 30.10 приведен пример сканирования строки.

Листинг 30.10. Сканирование строки

```
#include <stdio.h>
int main()
{
  char str[] = "c 10 1.2 String No input";
  char c;
  int n;
  double d;
  char s[80];

  sscanf(str, "%c %d %lf %s", &c, &n, &d, s);
```

```
  printf("%c\n", c);     /* c */
  printf("%d\n", n);     /* 10 */
  printf("%g\n", d);     /* 1.2 */
  printf("%s\n", s);     /* String */
  return 0;
}
```

30.14. Работа с буферами

Буфером называется область оперативной памяти, используемая потоком для временного хранения данных из файла. Для работы с буферами используются функции `setvbuf`, `setbuf`, `fflush`.

Для создания буфера потока и установки режимов работы с этим буфером используется функция `setvbuf`, которая имеет следующий прототип:

```
int setvbuf(FILE* stream, char* buf, int mode, size_t size);
```

Вызывается эта функция после открытия файла, но перед доступом к нему. При успешном завершении функция возвращает значение `0`, а в случае неудачи — ненулевое значение.

- Параметр `stream` задает поток, для которого устанавливается буфер.
- Параметр `buf` указывает на блок памяти для буфера. Если этот параметр равен `NULL`, то функция `setvbuf` использует функцию `malloc` для распределения памяти под буфер.
- Параметр `mode` определяет режим работы с буфером и может принимать следующие значения:
  - `_IOFBF` — вывод данных из буфера во внешнюю память выполняется только при полной загрузке буфера или при закрытии файла;
  - `_IOLBF` — вывод данных из буфера во внешнюю память выполняется при записи в буфер символа `'\n'`;
  - `_IONBF` — нет буферизации, в этом случае параметры `size` и `buf` игнорируются.
- Параметр `size` определяет длину буфера в байтах.

Для создания буфера потока может также использоваться функция `setbuf`, которая имеет следующий прототип:

```
int setbuf(FILE* stream, char* buffer);
```

Эта функция вызывает функцию `setvbuf`. Причем если значение параметра `buffer` не равно `NULL`, то функция `setvbuf` вызывается следующим образом:

```
setvbuf(stream, buffer, _IOFBF, BUFSIZ);
```

Макрос `BUFSIZ` задает длину буфера по умолчанию. Этот макрос описан в заголовочном файле stdio.h. В противном случае функция `setvbuf` вызывается следующим образом:

```
setvbuf(stream, 0, _IOFBF, BUFSIZE);
```

То есть в этом случае буферизация не используется.

Для сброса данных из буфера в файл предназначена функция `fflush`, которая имеет следующий прототип:

```
int fflush(FILE* stream);
```

Эта функция записывает данные из буфера потока `stream` в соединенный с этим потоком файл. В случае успешного завершения функция возвращает значение `0`, а в случае неудачи — `EOF`. Если значение параметра `stream` равно `NULL`, то освобождаются буферы всех потоков, которые работают в режиме вывода.

В листинге 30.11 приведен пример создания буфера потока.

Листинг 30.11. Создание буфера потока

```
#include <stdio.h>
#define SIZE 1024  /* размер буфера */
int main()
{
  int code;
  char name[80];
  double salary = 0.0;

  FILE* out;          /* выходной поток */
  char buffer[SIZE];  /* буфер потока */
```

```
  /* открываем выходной поток в текстовом режиме */
  out = fopen("employee.txt", "w");

  /* создаем буфер для потока */
  if(setvbuf(out, buffer, _IOFBF, SIZE))
  {
    printf("Set buffer failed.\n");
    return 0;
  }

  printf("Input code, name and salary.\n");
  printf("Press Ctrl+z to exit.\n");
  printf(">");

  /* вводим первую запись с консоли */
  scanf("%d%s%lf", &code, &name, &salary);
  while (!feof(stdin))
  {
    /* пишем запись в файл */
    fprintf(out, "%d %s %f ", code, name, salary);
    printf(">");
    /* вводим следующие записи с консоли */
    scanf("%d%s%lf", &code, &name, &salary);
  }

  /* закрываем выходной поток */
  fclose(out);

  return 0;
}
```

30.15. Стандартные потоки

Каждой программе предоставляются три стандартных потока, которые по умолчанию соединены с консолью. Указатели на данные потоки возвращают макросы `stdout`, `stdin`, `stderr`. Поэтому эти потоки также называются `stdout`, `stdin`, `stderr`.

Поток stdout предназначен для вывода данных на консоль. Для работы с этим потоком используются функции putchar, puts и printf, которые имеют следующие прототипы:

```
int putchar(int c);          /* вывод символа */
int puts(const char* str);   /* вывод строки */
int printf(const char *format, ...);  ./* форматирование */
```

Работают эти функции так же, как и соответствующие функции для записи данных в файл.

Поток stdin предназначен для ввода данных с консоли. Для работы с этим потоком используются функции getchar, gets и scanf, которые имеют следующие прототипы:

```
int getchar(void);       /* ввод символа */
char* gets(char* str);   /* ввод строки */
int scanf(const char *format, ...);  /* форматирование */
```

Работают эти функции так же, как и соответствующие функции для чтения данных из файла.

Поток stderr предназначен для вывода сообщений об ошибках на консоль. Для работы с этим потоком используется функция perror, которая имеет следующий прототип:

```
void perror(const char* str);
```

Параметр str указывает на строку, содержащую сообщение об ошибке.

30.16. Удаление файла

Для удаления файла предназначена функция remove, которая имеет следующий прототип:

```
int remove(const char* filename);
```

Эта функция удаляет файл с именем filename. Если файл открыт, то работа функции зависит от реализации. В случае успешного завершения функция возвращает 0, а в случае неудачи — ненулевое значение.

В листинге 30.12 приведен пример удаления файла.

Листинг 30.12. Удаление файла

```
#include <stdio.h>
int main()
{
  if(remove("employee.bin"))
  {
    printf("There is no such a file.\n");
    return 0;
  }
  printf("The file was deleted.\n");

  return 0;
}
```

30.17. Переименование файла

Для переименования файла предназначена функция `rename`, которая имеет следующий прототип:

```
int rename(const char* old_name, const char* new_name);
```

Эта функция переименовывает файл с именем `old_name` в файл с именем `new_name`. Если файл с именем `new_name` уже существует, то работа функции зависит от реализации. В случае успешного завершения функция возвращает `0`, а в случае неудачи — ненулевое значение.

В листинге 30.13 приведен пример переименования файла.

Листинг 30.13. Переименование файла

```
#include <stdio.h>

int main()
{
  if(rename("employee.txt", "emp.txt"))
  {
    printf("There is no such a file.\n");
    return 0;
  }
```

```
  printf("The file was renamed.\n");

  return 0;
}
```

30.18. Работа с временными файлами

Для создания временного файла предназначена функция `tmpfile`, которая имеет следующий прототип:

```
FILE* tmpfile(void);
```

Эта функция создает временный файл и открывает его в режиме `"w+b"`. После закрытия потока временный файл удаляется. В случае успешного завершения функция возвращает указатель на файл, а в случае неудачи — `NULL`.

В листинге 30.14 приведен пример создания временного файла.

Листинг 30.14. Создание временного файла

```
#include <stdio.h>
int main()
{
  FILE* temp;  /* временный файл */
  char s[] = "This is a string.";
  char t[80];

  /* открываем временный файл */
  if(!(temp = tmpfile()))
  {
    printf("Create temp file failed.\n");
    return 1;
  }
  /* пишем строку во временный файл */
  fputs(s, temp);
  /* устанавливаем индикатор позиции на начало файла */
  rewind(temp);
```

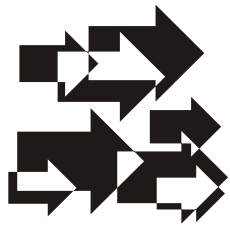
```
  /* читаем строку из временного файла */
  fgets(t, 80, temp);
  /* выводим строку на консоль */
  printf("%s\n", t);
  /* закрываем временный файл */
  fclose(temp);

  return 0;
}
```

Для получения имени, которое может быть использовано при создании временного файла, предназначена функция `tmpnam`, которая имеет следующий прототип:

```
char* tmpnam(char* str);
```

Эта функция возвращает имя для временного файла. Максимальное количество имен равно значению макроса `TMP_MAX`, а максимальная длина имени равна значению макроса `L_tmpnam`. Если значение параметра `str` равно `NULL`, то функция возвращает указатель на свою строку с именем файла. В противном случае функция записывает имя в строку `str` и возвращает указатель на эту строку.

ГЛАВА 31



Организация нелокальных переходов <setjmp.h>

В заголовочном файле setjmp.h определен тип данных `jmp_buf`, макрос `setjmp` и прототип функции `longjmp`, которые предназначены для организации нелокальных переходов в программе. Дело в том, что с помощью инструкции `goto` можно передавать управление только внутри одной функции. Функция `longjmp` позволяет передавать управление в произвольную точку программы сквозь вызовы функций.

31.1. Сохранение информации о состоянии стека

Тип данных `jmp_buf` обычно определен следующим образом:

```
typedef int jmp_buf[_JBLEN];
```

Макрос `_JBLEN` определяет длину буфера, который предназначен для хранения информации о состоянии стека.

Сохраняет информацию о состоянии стека в переменную типа `jmp_buf` макрос `setjmp`, определение которого имеет следующий вид:

```
#define setjmp(env) int_rvalue
```

Параметр `env` должен иметь тип `jmp_buf`, а `int_rvalue` представляет выражение, которое определяет реализацию этого макроса и результатом его вычисления является `rvalue` типа `int`. В случае

успешного сохранения информации о состоянии стека результатом вычисления выражения `int_rvalue` является `0`.

По стандарту макрос `setjmp` может использоваться только в выражениях:

- без операторов;
- с одним оператором `!`;
- с одним из операторов сравнения, вторым операндом которого является целочисленное константное выражение.

Выражения с макрокомандой `setjmp` можно использовать в управляющих инструкциях и в выражении с оператором присваивания, которое является инструкцией.

Часто вместо макроса `setjmp` используется просто функция `setjmp`, которая имеет следующий прототип:

```
int setjmp(jmp_buf env);
```

31.2. Нелокальный переход

Нелокальный переход выполняется при помощи функции `longjump`, которая имеет следующий прототип:

```
void longjmp(jmp_buf env, int val);
```

Параметр `env` должен содержать информацию о состоянии стека, которая была предварительно сохранена макрокомандой `setjmp`, а значение параметра `val` будет использовано как значение макроса `setjmp`.

Работает функция `longjump` следующим образом. Она возвращает управление на макрокоманду `setjmp`, которая сохранила состояние стека в переменной `env`. Повторное выполнение макрокоманды `setjmp` возвращает значение, которое задано в параметре `val` функции `longjump`. Если значение параметра `val` равно 0, то макрокоманда `setjmp` вернет значение `1`. После этого управление передается на следующую за макрокомандой `setjmp` инструкцию.

Другими словами можно сказать, что макрокоманда `setjmp` устанавливает контрольную точку в программе, в которой сохраняет-

ся информация о состоянии стека, а функция `longjump` выполняет откат на эту контрольную точку.

В листинге 31.1 приведен пример организации нелокального перехода.

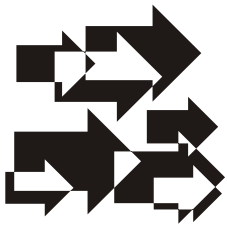
Листинг 31.1. Нелокальный переход

```
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>

int *p = NULL;
void fun(jmp_buf env)
{
  p = malloc(sizeof(int));
  printf("Function is working\n");
  longjmp(env, 10);
}

int main(void)
{
  int val;
  jmp_buf env;

  val = setjmp(env);
  if(val)
  {
    if (p)
      free(p);
    printf("Long jump value: %d\n", val);  /* 10 */

    return val;
  }
  printf("Long jump value: %d\n", val);    /* 0 */
  fun(env);

  return 0;
}
```

В результате работы этой программы на консоль будут выведены следующие сообщения:

```
Long jump value: 0
Function is working
Long jump value: 10
```

ГЛАВА 32



Обработка исключительных ситуаций <signal.h>

32.1. Сигналы

Сигналы служат для оповещения программы о следующих событиях:

- во время исполнения программы произошла ошибка;
- во время исполнения программы произошло асинхронное событие.

Например, ошибка может быть вызвана переполнением при операции над данными с плавающей точкой. В качестве примера асинхронного события можно привести нажатие клавиши клавиатуры.

Все сигналы перенумерованы. В заголовочном файле signal.h определены следующие символические константы для некоторых предопределенных сигналов:

```
/* аварийное завершение посредством вызова функции abort */
#define SIGABRT const_int_выражение
/* ошибка в операции с плавающей точкой */
#define SIGFPE  const_int_выражение
/* неправильная инструкция */
#define SIGILL const_int_выражение
/* прерывание */
#define SIGINT const_int_выражение
```

```
/* ошибка доступа к памяти */
#define SIGSEGV const_int_выражение
/* сигнал на завершение программы */
#define SIGTERM const_int_выражение
```

Здесь `const_int_выражение` представляет собой выражение, результатом вычисления которого является неотрицательное целочисленное значение.

Для каждой реализации в заголовочном файле signal.h могут быть определены и другие сигналы.

32.2. Установка функции обработки сигнала

Для установки функции обработки сигнала предназначена функция `signal`, которая имеет следующий прототип:

```
void (* signal(int sig, void (*f)(int)))(int);
```

где параметр `sig` задает номер сигнала, для которого устанавливается обработчик, заданный параметром `f`. Здесь `f` является указателем на функцию, которая имеет следующий прототип:

```
void f(int);
```

и часто называется *функцией обработки сигнала* или просто *обработчиком сигнала*. В случае успешного завершения функция `signal` возвращает указатель на предыдущую функцию, установленную для обработки этого сигнала, а в случае неудачи — значение, определенное макросом `SIG_ERR`.

Для максимальной переносимости программ рекомендуется, чтобы обработчики асинхронных сигналов использовали только следующие возможности:

- вызовы функции `signal`, которые успешно завершаются;
- присваивание значений переменным типа `sig_atomic_t`;
- возврат управления.

Тип `sig_atomic_t` определен в заголовочном файле signal.h и предназначен для определения переменных, значения которых

изменяются атомарными операциями, т. е. такими операциями, которые не прерываются во время своего исполнения. Определение типа `sig_atomic_t` обычно имеет следующий вид:

```
typedef int sig_atomic_t;
```

Переменные типа `sig_atomic_t` обычно имеют квалификатор `volatile` и используются для взаимодействия между функцией обработки сигнала и программой.

Обработчик сигнала может завершить свое исполнение вызовом функций `abort`, `exit` и `longjmp`.

В заголовочном файле signal.h определены макросы для следующих стандартных обработчиков сигналов:

```
#define SIG_DFL const_адрес  /* обработчик по умолчанию */
#define SIG_IGN const_адрес  /* игнорирование сигнала */
```

Выражение `const_адрес` задает адрес стандартного обработчика для сигнала. Чаще всего эти адреса являются просто целочисленными константами, проверяя значения которых среда исполнения выполняет некоторые действия при вызове соответствующего обработчика.

В заголовочном файле signal.h могут быть также определены макросы и для других обработчиков сигналов.

32.3. Возбуждение сигнала

Для возбуждения сигнала предназначена функция `raise`, которая имеет следующий прототип:

```
int raise(int sig);
```

Эта функция возбуждает сигнал, номер которого задан параметром `sig`. В случае успешного завершения функция возвращает значение `0`, а в случае неудачи — ненулевое значение.

Функция `raise` работает следующим образом. После возбуждения сигнала управление передается функции обработки этого сигнала. Функция `raise` не возвращает управление до тех пор, пока не завершит свое исполнение функция обработки сигнала. Однако

отметим, что после возврата управления из функции `raise` поведение программы не определено, если обрабатывались сигналы `SIGFPE`, `SIGILL` или `SIGSEGV`. В этом случае для выхода из функции обработки сигнала лучше использовать функцию `longjmp`.

При вызове функции обработки сигнала среда исполнения может выполнять следующие действия:

- блокировать повторное возбуждение сигнала с тем же номером, который обрабатывает функция обработки сигнала, до передачи управления из этого обработчика;
- для обработки повторного сигнала с тем же номером, который обрабатывает функция обработки сигнала, вызвать обработчик сигнала, заданный по умолчанию;
- для повторного сигнала `SIGILL` среда исполнения может продолжить обработку предыдущего такого сигнала.

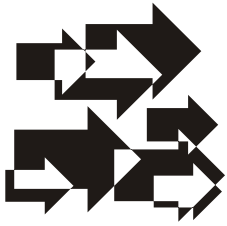
В листинге 32.1 приведен пример использования функций `signal` и `raise`.

Листинг 32.1. Обработка сигнала

```
#include<signal.h>
#include<stdio.h>
void handler(int sig)
{
  printf("Signal handler.\n");
}
int main()
{
  void (*f)(int);  /* указатель на обработчик */
  /* запоминаем предыдущий обработчик */
  f = signal(SIGINT, handler);
  if(f == SIG_ERR)
  {
    printf("Signal set failed.\n");  // неудача
    return 0;
  }
  printf("Raise signal.\n");
```

```
  if(raise(SIGINT))
  {
    printf("Raise failed.\n");
    return 2;
  }
  printf("OK.\n");
  /* восстанавливаем предыдущий обработчик */
  signal(SIGINT, f);
  return 0;
}
```

ГЛАВА 33



Поддержка локализации <locale.h>

Заголовочный файл locale.h предназначен для управления представлением и обработкой данных, специфических для различных регионов. Кроме специфических определений, в этом заголовочном файле также определен макрос `NULL`.

33.1. Локальность

Информация, описывающая представление и правила обработки данных для какого-то региона, называется *локальностью* (locale). Процесс преобразования данных в локальное представление называется *локализацией*.

Каждая локальность имеет имя, которое задается символьной строкой. В стандарте языка программирования C предопределены имена двух локальностей:

- `"C"` — локальность, используемая по умолчанию;
- `""` — местная (native) локальность.

В конкретной реализации могут быть также определены и другие локальности.

Перед передачей управления программе среда исполнения устанавливает локальность `"C"`.

33.2. Локальные категории

Локальность разбивается на разделы, каждый из которых содержит информацию, связанную по смыслу или назначению. Разделы локальности нумеруются. Номер раздела локальности называется *локальной категорией*. В заголовочном файле locale.h определены следующие макросы для локальных категорий:

- `LC_ALL` — все локальные категории;
- `LC_COLLATE` — локальная категория, связанная с работой функций `strcoll` и `strxfrm`;
- `LC_CTYPE` — локальная категория, связанная с работой функций классификации символов;
- `LC_MONETARY` — локальная категория, связанная с форматированием денежных значений;
- `LC_NUMERIC` — локальная категория, связанная с форматированием данных с плавающей точкой;
- `LC_TIME` — локальная категория, связанная с работой функции `strftime`.

Каждый из этих макросов определяет некоторое целочисленное константное выражение.

В заголовочном файле locale.h могут быть также определены и другие локальные категории.

33.3. Установка локальности

Для установки локальности, которая будет управлять обработкой и форматированием специфических для региона данных, предназначена функция `setlocale`, которая имеет следующий прототип:

```
char *setlocale(int category, const char *locale);
```

Параметр `category` должен содержать локальную категорию, а параметр `locale` указывать на имя локальности.

Функция `setlocale` устанавливает локальную категорию, заданную параметром `category`, из локальности на которую указывает

параметр `locale`. В случае успешного завершения функция возвращает указатель на строку, описывающую установленную локальность и категорию, а в случае неудачи — `NULL`.

Если в параметре `locale` установить значение `NULL`, то функция вернет указатель на строку, описывающую локальную категорию, заданную параметром `category` для текущей локальности.

В листинге 33.1 приведен пример использования функции `setlocale`.

Листинг 33.1. Установка местной локальности

```
#include<stdio.h>
#include<locale.h>

int main(void)
{
  char *locale;

  locale = setlocale(LC_ALL, "");
  printf("%s\n", locale);  /* Russian_Russia.1251 */
  printf("%g\n", 10.5);    /* 10,5 */

  return 0;
}
```

33.4. Форматы числовых и денежных данных

Информация о том, как форматировать числовые и денежные данные для локального представления, хранится в структуре типа `lconv`, поля которой:

```
  char *decimal_point;  /* символ десятичной точки в не
    денежных данных */
  char *thousands_sep;  /* разделитель для числовых групп
    слева от десятичной точки в не денежных данных */
  char *grouping;  /* строка, задающая количество цифр в
    каждой группе в не денежных данных;
```

```
    каждый символ представляет десятичное число, которое
    обозначает количество цифр в группе;
    ноль обозначает, что предыдущее значение будет
    использоваться для всех оставшихся групп;
    значение CHAR_MAX обозначает, что группировка цифр в числе
    прекращается */
  char *int_curr_symbol;  /* символ денег, определенный
    стандартом ISO 4217 */
  char *currency_symbol;  /* локальный символ денег */
  char *mon_decimal_point;  /* символ десятичной точки в
    денежных данных */
  char *mon_thousands_sep;  /* разделитель для числовых групп
    слева от десятичной точки в денежных данных */
  char *mon_grouping;  /* строка, задающая количество цифр в
    каждой группе в денежных данных; формат строки такой же,
    как и в поле grouping */
  char *positive_sign;  /* знак положительного значения для
    денежных величин */
  char *negative_sign;  /* знак отрицательного значения для
    денежных величин */
  char int_frac_digits;  /* количество цифр после десятичной
    точки для международного представления денежных данных */
  char frac_digits;  /* количество цифр после десятичной точки
    для представления денежных данных */
  char p_cs_precedes;  /* позиция символа денег для
    положительных денежных значений:
    0 — символ денег следует за денежным значением;
    1 — символ денег предшествует денежному значению */
  char n_cs_precedes;  /* позиция символа денег для
    отрицательных денежных значений:
    0 — символ денег следует за денежным значением;
    1 — символ денег предшествует денежному значению */
  char p_sep_by_space;  /* для положительного денежного
    значения:
    0 — символ денег не отделен от денежного значения;
    1 — символ денег отделен от денежного значения пробелом */
  char n_sep_by_space;  /* для отрицательного денежного
    значения:
    0 — символ денег не отделен от денежного значения;
    1 — символ денег отделен от денежного значения пробелом */
```

```
char p_sign_posn;  /* формат положительного денежного
  значения:
  0 — величина и символ денег заключаются в круглые скобки;
  1 — знак минус предшествует величине и символу денег;
  2 — знак минус следует после величины и символа денег;
  3 — знак минус непосредственно предшествует символу денег;
  4 — знак минус непосредственно следует за символом денег*/
char n_sign_posn;  /* формат положительного денежного
  значения:
  0 — величина и символ денег заключаются в круглые скобки;
  1 — знак плюс предшествует величине и символу денег;
  2 — знак плюс следует после величины и символа денег;
  3 — знак плюс непосредственно предшествует символу денег;
  4 — знак плюс непосредственно следует за символом денег */
```

В структуре эти поля могут следовать в произвольном порядке. Если в поле типа `char` структуры `lconv` установлено значение `CHAR_MAX`, то это значит, что для данной локализации это поле не содержит смыслового значения. Заметим, что символическая константа `CHAR_MAX` определена в заголовочном файле limits.h.

В табл. 33.1 приведены значения полей типа `char` и строки, на которые указывают поля типа `char*`, для локальности `"C"`, а также локальная категория каждого поля.

Таблица 33.1. *Форматы числовых данных в локальности :"C"*

Поле структуры `lconv`	Значение	Локальная категория
currency_symbol	""	LC_MONETARY
decimal_point	"."	LC_NUMERIC
grouping	""	LC_NUMERIC
int_curr_symbol	""	LC_MONETARY
mon_decimal_point	""	LC_MONETARY
mon_grouping	""	LC_MONETARY
mon_thousands_sep	""	LC_MONETARY
negative_sign	""	LC_MONETARY

Таблица 33.1 *(окончание)*

Поле структуры `lconv`	Значение	Локальная категория
positive_sign	""	LC_MONETARY
thousands_sep	""	LC_NUMERIC
frac_digits	CHAR_MAX	LC_MONETARY
int_frac_digits	CHAR_MAX	LC_MONETARY
n_cs_precedes	CHAR_MAX	LC_MONETARY
n_sep_by_space	CHAR_MAX	LC_MONETARY
n_sign_posn	CHAR_MAX	LC_MONETARY
p_cs_precedes	CHAR_MAX	LC_MONETARY
p_sep_by_space	CHAR_MAX	LC_MONETARY
p_sign_posn	CHAR_MAX	LC_MONETARY

33.5. Определение текущих форматов данных

Для определения форматов числовых и денежных данных, используемых в текущей локальности, предназначена функция `localeconv`, которая имеет следующий прототип:

```
struct lconv *localeconv(void);
```

Эта функция всегда возвращает указатель на структуру типа `lconv`, которая содержит требуемые форматы. Программа не должна менять поля этой структуры.

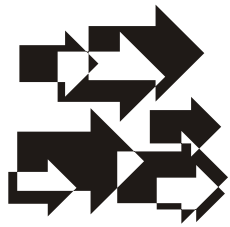
В листинге 33.2 приведен пример использования функции `lconv`.

Листинг 33.2. Определение форматов данных в текущей локальности

```
#include<stdio.h>
#include<locale.h>
```

```
int main(void)
{
  struct lconv *p;

  p = localeconv();
  printf("Decimal point: %s\n", p->decimal_point);

  return 0;
}
```

ГЛАВА 34



Работа с датами и временем <time.h>

Кроме типов данных и функций, используемых для обработки дат и времени, в заголовочном файле time.h также определена символическая константа `NULL` и тип данных `size_t`, которые описаны в *гл. 20*.

34.1. Арифметические типы для представления времени

Для числового представления количества тактовых или, другими словами, синхронизирующих импульсов процессора предназначен тип `clock_t`, который обычно определяется следующим образом:

```
typedef long clock_t;
```

Для числового представления календарного времени предназначен тип `time_t`, который обычно определяется следующим образом:

```
typedef long time_t;
```

34.2. Структура для представления календарного времени

Различные характеристики календарного времени хранятся в структуре типа `tm`, которая обычно имеет следующий формат:

```
struct tm
{
  int tm_sec;   /* секунда минуты - [0,59] */
  int tm_min;   /* минута часа - [0,59] */
  int tm_hour;  /* час дня - [0,23] */
  int tm_mday;  /* день месяца - [1,31] */
  int tm_mon;   /* месяц года - [0,11] */
  int tm_year;  /* год, начиная с 1900 года */
  int tm_wday;  /* день недели - [0,6] */
  int tm_yday;  /* день года - [0,365] */
  int tm_isdst; /* перевод часов летом на час вперед */
};
```

Значения поля `tm_isdst` имеют следующий смысл:

- положительное значение указывает на то, что летом часы переводятся на час вперед;
- нулевое значение указывает на то, что летом часы не переводятся;
- отрицательное значение указывает на то, что относительно перевода часов летом нет информации.

Заметим, что поля структуры типа `tm` могут располагаться в произвольном порядке. Кроме того, при конкретной реализации в структуре `tm` могут быть и другие поля.

34.3. Определение времени работы программы

Для определения количества тактовых импульсов процессора, сгенерированных с начала работы программы, предназначена функция `clock`, которая имеет следующий прототип:

```
clock_t clock(void);
```

В случае удачного завершения функция возвращает количество тактовых импульсов, сгенерированных процессором с начала работы программы. Если среда исполнения не подсчитывает количество сгенерированных процессором тактовых импульсов, то функция `clock` возвращает –1.

Количество тактовых импульсов, сгенерированных процессором в одну секунду, определяется макросом CLOCKS_PER_SEC.

В листинге 34.1 приведен пример использования функции clock.

Листинг 34.1. Пример использования функции clock

```
#include <stdio.h>
#include <time.h>
int main()
{
  int sec;

  printf("Press any key to count seconds.\n");
  getchar();
  sec = clock() / CLOCKS_PER_SEC;
  printf ("Number of seconds: %d\n", sec);

  return 0;
}
```

34.4. Определение текущего времени

Для определения текущего календарного времени в числовом формате предназначена функция time, которая имеет следующий прототип:

```
time_t time(time_t *tod);
```

В случае успешного завершения функция возвращает текущее календарное время, а также записывает его по адресу, который установлен в параметре tod. В противном случае функция возвращает значение –1. Если в параметре tod установлено значение NULL, то функция time просто возвращает текущее календарное время в числовом формате.

Пример использования функции time приведен в листинге 34.2.

34.5. Нахождение разности времен

Для нахождения разности между двумя календарными датами, представленными в числовом формате, предназначена функция `difftime`, которая имеет следующий прототип:

```
double difftime(time_t t1, time_t t0);
```

Функция возвращает разность между календарными датами `t1` и `t0` в секундах.

Пример использования функций `time` и `difftime` приведен в листинге 34.2.

Листинг 34.2. Пример использования функций `time` и `difftime`

```
#include <stdio.h>
#include <time.h>

int main()
{
  time_t t0, t1;
  double d;

  printf("Press any key to get the first time.\n");
  getchar();
  t0 = time(NULL);

  printf("Press any key to get the second time.\n");
  getchar();
  t1 = time(NULL);

  d = difftime(t1, t0);
  printf("The difference between times: %Lg\n", d);

  return 0;
}
```

34.6. Преобразование представления времени из числа в структуру

Для преобразования текущего календарного времени из числового формата в формат структуры типа `tm` предназначены функции `gmtime` и `localtime`. Различие между этими функциями заключается в том, что функция `gmtime` устанавливает в структуру календарное время в *универсальных временных координатах* (UTC, Universal Time Coordinated), которые раньше назывались "среднее время по Гринвичу", а функция `localtime` устанавливает в структуру локальное календарное время.

Функция `gmtime` имеет следующий прототип:

```
struct tm* gmtime(const time_t *tod);
```

Эта функция возвращает адрес статической структуры типа `tm`, в которую она преобразует числовое представление календарного времени, на которое указывает параметр `tod`. В структуре типа `tm` время задано в универсальных временных координатах.

В листинге 34.3 приведен пример использования функции `gmtime`.

Листинг 34.3. Пример использования функции `gmtime`

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t t;
  struct tm* p;

  t = time(NULL);
  p = gmtime(&t);

  printf("Second: %d\n", p->tm_sec);
  printf("Minute: %d\n", p->tm_min);
  printf("Houre : %d\n", p->tm_hour);
  printf("Day   : %d\n", p->tm_mday);
```

```
  printf("Month : %d\n", p->tm_mon);
  printf("Year  : %d\n", p->tm_year);

  return 0;
}
```

Функция `localtime` имеет следующий прототип:

```
struct tm* localtime(const time_t *tod);
```

Эта функция возвращает адрес статической структуры типа `tm`, в которую она преобразует числовое представление календарного времени, на которое указывает параметр `tod`. В структуре типа `tm` установлено локальное время.

В листинге 34.4 приведен пример использования функции `localtime`.

Листинг 34.4. Пример использования функции `localtime`

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t t;
  struct tm* p;

  t = time(NULL);

  p = localtime(&t);

  printf("Second: %d\n", p->tm_sec);
  printf("Minute: %d\n", p->tm_min);
  printf("Houre : %d\n", p->tm_hour);
  printf("Day   : %d\n", p->tm_mday);
  printf("Month : %d\n", p->tm_mon);
  printf("Year  : %d\n", p->tm_year);

  return 0;
}
```

34.7. Преобразование представления времени из структуры в число

Для преобразования календарного времени, представленного структурой типа `tm` в числовое представление, предназначена функция `mktime`, которая имеет следующий прототип:

```
time_t mktime(struct tm *tptr);
```

Параметр `tptr` должен указывать на структуру типа `tm`, в которой хранится календарное время. Это календарное время функция преобразует в числовое представление, которое и возвращает. В случае неудачного завершения функция возвращает значение `-1`. Функция завершается неудачей, если календарное время не может быть представлено в числовом формате.

34.8. Преобразование представления времени из числа в строку

Для преобразования числового представления времени в строку с информацией о времени предназначена функция `ctime`, которая имеет следующий прототип:

```
char *ctime(const time_t *tod);
```

где параметр `tod` должен указывать на календарное время, представленное в числовом формате. Функция возвращает адрес строки с информацией о локальном календарном времени.

Листинг 34.5. Пример использования функции `ctime`

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t t;
  char* s;
```

```
  t = time(NULL);
  s = ctime(&t);
  printf("Time: %s\n", s);

  return 0;
}
```

34.9. Преобразование представления времени из структуры в строку

Для преобразования времени, представленного структурой, в строку с информацией о времени, предназначена функция `asctime`, которая имеет следующий прототип:

```
char *asctime(const struct tm *tptr);
```

где параметр `tptr` должен указывать на календарное время, представленное структурой типа `tm`. Функция возвращает адрес строки, содержащей информацию о календарном времени.

В листинге 34.6 приведен пример использования функции `asctime`.

Листинг 34.6. Пример использования функции `asctime`

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t t;
  struct tm *p;
  char* s;

  t = time(NULL);
  p = gmtime(&t);
  s = asctime(p);
  printf("Time: %s\n", s);

  return 0;
}
```

34.10. Форматирование представления времени в строке

Для преобразования времени, представленного структурой, в отформатированную строку с информацией о времени предназначена функция `strftime`, которая имеет следующий прототип:

```
size_t strftime(char *s, size_t n, const char *format,
                                   const struct tm *tptr);
```

Опишем назначение параметров этой функции.

Параметр `s` должен указывать на массив символов, в который функция запишет отформатированную информацию о календарном времени.

Параметр `n` должен содержать длину массива, на который указывает параметр `s`.

Параметр `format` должен указывать на строку, в которой содержатся спецификаторы преобразования для представления времени в массиве, на который указывает параметр `s`. Каждый спецификатор преобразования начинается с символа `%`, за которым следует символ, указывающий формат преставления времени и даты. Спецификаторы преобразования приведены в табл. 34.1. Функция `strftime` сканирует строку `format` и для каждого спецификатора преобразования записывает в строку `s` соответствующую последовательность символов. После завершения преобразования форматов функция записывает в массив `s` пустой символ.

Если при сканировании строки `format` функция встретит длинный (multibyte) символ, отличный от символа `%`, то функция переписывает его в массив `s`.

Параметр `tptr` должен указывать на календарное время, представленное структурой типа `tm`. Это календарное время будет преобразовано в строку в соответствии с форматом, заданным параметром `format`, и записано в массив, на который указывает параметр `s`.

Таблица 34.1. *Спецификаторы преобразования календарного времени*

Спецификатор	Назначение спецификатора	Пример
%a	Сокращенное название дня недели	Sun
%A	Полное название дня недели	Sunday
%b	Сокращенное название месяца	Dec
%B	Полное название месяца	December
%c	Дата и время	Dec 2 06:55:15 1979
%d	День недели	02
%H	Час суток (длина суток – 24 часа)	16
%I	Час дня (длина дня – 12 часов)	06
%j	День года, начиная с 001	335
%m	Месяц года, начиная с 001	12
%M	Минуты часа	55
%p	AM/PM индикатор	AM
%S	Секунды минуты	15
%U	Неделя года, начиная с 00. Неделя начинается в воскресенье	48
%w	День недели, начиная с 0	6
%W	Неделя года, начиная с 00 Неделя начинается с понедельника	47
%x	Дата	Dec 2 1979
%X	Время	06:55:15
%y	Год века, начиная с 00	79
%Y	Год	1979
%Z	Название временной зоны, если оно есть	EST
%%	Знак процента	%

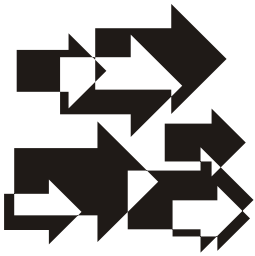
При успешном завершении функция возвращает количество символов, записанных в массив `s`. Функция завершается успешно, если значение параметра `n` больше чем количество символов, записанных в массив `s`. В противном случае функция возвращает `0`.

На форматирование времени влияет значение локальной категории `LC_TIME`. Локальные категории и работа с ними описаны в *гл. 33*.

В листинге 34.7 приведен пример использования функции `strftime`.

Листинг 34.7. Пример использования функции `strftime`

```
#include <stdio.h>
#include <time.h>
int main ()
{
  time_t t;
  struct tm *p;
  char s[80];
  char* format = "%A %B %Y";

  t = time(NULL);
  p = gmtime(&t);
  if (strftime(s, 80, format, p))
    printf("Time: %s\n", s);
  else
    printf("strftime failed.\n");

  return 0;
}
```



ЧАСТЬ IV

Стандартная библиотека языка программирования C++

Глава 35. Структура стандартной библиотеки C++

Глава 36. Обработка исключений <exception>

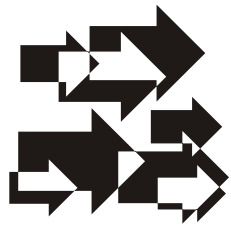
Глава 37. Классы стандартных исключений <stdexcept>

Глава 38. Динамическая идентификация типов <typeinfo>

Глава 39. Работа со строками в C++ <string>

Глава 40. Потоки ввода/вывода в C++

ГЛАВА 35



Структура стандартной библиотеки C++

Все стандартные функции языка программирования C стандартны и для языка C++. Поэтому стандартная библиотека языка программирования C++ содержит как заголовочные файлы стандартной библиотеки языка программирования C, так и новые заголовочные файлы. В отличие от заголовочных файлов языка программирования C, которые имеют расширение h, новые заголовочные файлы языка программирования C++ не имеют расширения.

Отличие между заголовочными файлами языка программирования C и новыми заголовочными файлами языка программирования C++ заключается в следующем. Все стандартные заголовочные файлы языка программирования C объявляют функции в глобальном пространстве имен, а стандартные заголовочные файлы языка программирования C++ объявляют функции в стандартном пространстве имен. Заметим, что для каждого заголовочного файла стандартной библиотеки функций языка программирования C для функций из стандартной библиотеки языка программирования C++ существует соответствующий заголовочный файл, который объявляет те же функции, но в стандартном пространстве имен. Имя такого заголовочного файла начинается с буквы "c". Например, для заголовочного файла signal.h в стандартной библиотеки языка программирования C++ существует соответствующий заголовочный файл csignal. Заметим, что некоторые компиляторы также содержат заголовочные файлы, которые объявляют стандартные функции языка программирования

C++ и в глобальном пространстве имен. Такие заголовочные файлы, как правило, имеют расширение h. Например, в заголовочном файле iostream.h могут определяться стандартные потоки ввода/вывода.

Функционально стандартная библиотека языка программирования C++ без функций, наследуемых из стандартной библиотеки языка программирования C, разбивается на следующие библиотеки специального назначения:

- поддержка языка (The Language Support Library):
  - `<exception>` — обработка исключений;
  - `<limits>` — арифметические свойства скалярных типов;
  - `<new>` — распределение и освобождение памяти;
  - `<typeinfo>` — получение информации о типах данных;
- диагностика (The Diagnostics Library):
  - `<stdexcept>` — обработка стандартных исключений;
- утилиты общего назначения (The General Utilities Library):
  - `<utility>` — шаблоны для использования в стандартной библиотеке шаблонов;
  - `<functional>` — шаблоны для построения объектов функций;
  - `<memory>` — шаблоны для распределения и освобождения памяти под объекты;
- шаблоны для работы со строками (The Standard String Templates):
  - `<string>` — шаблоны для обработки строк;
- классы и шаблоны для поддержки локализации (Localization Classes and Templates):
  - `<locale>` — поддержка локализации;
- стандартную библиотеку шаблонов (The Standard Template Library):
  - `<algorithm>` — алгоритмы над контейнерами;
  - `<bitset>` — массив бит;

- <deque> — дек;
- <iterator> — итераторы;
- <list> — список;
- <map> — отображение и мультиотображение;
- <queue> — очередь и очередь с приоритетами;
- <set> — множество и мультимножество;
- <stack> — стек;
- <vector> — вектор;

❒ поддержка числовых операций (The Standard Numerics Library):

- <complex> — комплексные числа;
- <numeric> — шаблоны функций для числовых алгоритмов над последовательностями элементов;
- <valarray> — шаблоны функций и классов для вычислений над элементами массивов;

❒ ввод/вывод (The Standard Input/Output Library):

- <fstream> — потоки для работы с файлами;
- <iomanip> — манипуляторы потоков;
- <ios> — базовые типы и функции для работы с потоками;
- <iosfwd> — объявления шаблонов и типов, используемых в других заголовочных файлах;
- <iostream> — стандартные потоки ввода/вывода;
- <istream> — потоки ввода;
- <ostream> — потоки вывода;
- <streambuf> — буфер потока;
- <sstream> — потоковые операции со строками типа basic_string;
- <strstream> — потоковые операции со строками языка программирования C.

Для использования в программе функций из библиотеки специального назначения нужно при помощи директивы `#include` включить в эту программу соответствующий файл с прототипами функций.

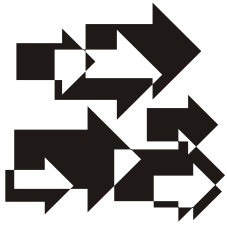
В данной книге рассматриваются не все компоненты из стандартной библиотеки языка программирования C++. Это сделано по двум причинам.

Во-первых, стандартная библиотека довольно громоздкая и подробное рассмотрение всех компонентов значительно увеличило бы объем книги.

Во-вторых, книга предназначена для начинающих и поэтому маловероятно, что подробное изложение стандартной библиотеки необходимо для таких читателей. Даже, наоборот — громоздкость может отбить охоту к изучению.

Поэтому при изложении принят компромиссный подход. Стандартная библиотека языка программирования C рассмотрена довольно подробно. Это позволяет ознакомиться со стандартными функциями, предназначенными для решения рутинных задач программирования. Из стандартной библиотеки языка программирования C++ дополнительно рассмотрены стандартные средства, наиболее часто используемые в программировании, а именно: обработка исключений, динамическая идентификация типов, обработка строк и потоки ввода/вывода. Однако следует заметить, что при должном изучении этой части стандартной библиотеки даже самостоятельное изучение остальных компонентов стандартной библиотеки языка программирования C++ не должно вызвать затруднений.

ГЛАВА 36



Обработка исключений <exception>

36.1. Классы и функции

В заголовочном файле exception объявляются следующие классы и функции, используемые при обработке исключений:

```
namespace std
{
  class exception;
  class bad exception;
  typedef void (*unexpected_handler)();
  unexpected_handler set_unexpected(unexpected_handler f)
                                                    throw();
  void unexpected();
  typedef void (*terminate_handler)();
  terminate_handler set_terminate(terminate_handler f)
                                                    throw();
  void terminate();
  bool uncaught_exception();
}
```

Подробно назначение этих классов и функций рассмотрено в следующих разделах этой главы.

36.2. Класс *exception*

Класс `exception` является базовым классом для всех стандартных исключений, определенных в стандартной библиотеке языка программирования C++. Этот класс имеет следующий интерфейс:

```
namespace std
{
  class exception
  {
  public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
  };
}
```

Строка языка программирования C, адрес которой возвращает метод `what`, не определена. Эта строка может быть определена конструкторами классов, наследуемых от класса `exception`.

Ни один из методов класса `exception` не выбрасывает исключения.

36.3. Класс *bad_exception*

Класс `bad_exception` определяет тип объектов, которые может выбросить обработчик не специфицированных исключений. Этот класс наследуется от класса `exception` и имеет следующий интерфейс:

```
namespace std
{
  class bad_exception : public exception
 {
  public:
    bad_exception() throw();
    bad_exception(const bad_exception&) throw();
    bad_exception& operator=(const bad_exception&) throw();
```

```
    virtual ~bad_exception() throw();
    virtual const char* what() const throw();
  };
}
```

Строка, адрес которой возвращает метод `what`, зависит от реализации.

36.4. Аварийное завершение программы

Для аварийного завершения программы предназначена функция `terminate`, которая имеет следующий прототип:

```
void terminate();
```

Функция `terminate` вызывает функцию, которая называется *обработчиком аварийного завершения* программы (terminate handler) и имеет следующий прототип:

```
typedef void (*terminate_handler)();
```

Обработчик аварийного завершения программы устанавливается функцией `set_terminate`, которая имеет следующий прототип:

```
terminate_ handler set_terminate(terminate_handler ph)
                                                  throw();
```

Эта функция устанавливает новый обработчик аварийного завершения программы, на который указывает параметр `ph`, и возвращает адрес предыдущего обработчика аварийного завершения программы.

Обработчик аварийного завершения, установленный по умолчанию при запуске программы, вызывает функцию `abort`, которая была описана в *разд. 21.4*.

В листинге 36.1 приведен пример установки и вызова нового обработчика аварийного завершения программы. Как правило, обработчик аварийного завершения программы используется для корректного освобождения ресурсов, захваченных этой программой.

Листинг 36.1. Установка и вызов обработчика аварийного завершения программы

```
#include <iostream>
#include <exception>
using namespace std;

int* p;
void foo()
{
  cout<< "val: " << *p << endl;
  delete p;
}
int main()
{
  set_terminate(foo);
  p = new int(10);
  terminate();  // val: 10
  return 0;
}
```

36.5. Обработка не специфицированного исключения

Если функция выбрасывает не специфицированное в ней исключение, то среда исполнения программы вызывает функцию `unexpected`, которая имеет следующий прототип:

```
void unexpected();
```

Функция `terminate` в свою очередь вызывает функцию, которая называется *обработчиком не специфицированного исключения* (unexpected handler) и имеет следующий прототип:

```
typedef void (*unexpected_handler)();
```

Обработчик не специфицированного исключения устанавливается функцией `set_unexpected`, которая имеет следующий прототип:

```
unexpected_handler set_unexpected(unexpected_handler ph)
                                                  throw();
```

Эта функция устанавливает новый обработчик не специфицированного исключения, на который указывает параметр `ph`, и возвращает адрес предыдущего обработчика не специфицированного исключения.

Обработчик не специфицированного исключения, установленный по умолчанию при запуске программы, указывает на функцию `terminate`, которая была описана в предыдущем разделе.

В листинге 36.2 приведен пример установки и вызова нового обработчика не специфицированного исключения. Как правило, обработчик не специфицированного исключения освобождает ресурсы, захваченные программой, и завершает исполнение программы.

Листинг 36.2. Установка и вызов обработчика не специфицированного исключения

```
#include <iostream>
#include <exception>
using namespace std;

int* p;
void foo()
{
  cout<< "val: " << *p << endl;
  delete p;
}
void g() throw() { throw "Unexpected exception"; }

int main()
{
  set_unexpected(foo);
  p = new int(10);
  g();  // val: 10

  return 0;
}
```

Если функция-обработчик не специфицированного исключения не выбрасывает исключения, которое специфицировано в списке

исключений функции, то среда исполнения выполняет одно из следующих действий:

- если в списке исключений функции специфицировано исключение типа `std::bad_exception`, то среда исполнения программы выбрасывает исключение этого типа и ищет обработчик этого исключения;
- если в списке исключений функции не специфицировано исключение типа `std::bad_exception`, то среда исполнения программы вызывает функцию `terminate`.

Заметим, что некоторые компиляторы могут не поддерживать обработку не специфицированных исключений и всегда вызывать в этом случае функцию `terminate`.

36.6. Определение наличия выброшенного исключения

Для определения того, существует ли в данный момент исключение, которое выброшено, но для которого еще не найден обработчик исключения, предназначена функция `uncaught_exception`, которая имеет следующий прототип:

```
bool uncaught_exception();
```

В случае если такое исключение существует, функция возвращает значение `true`, в противоположном случае — значение `false`.

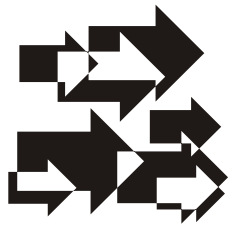
Функция `uncaught_exception` обычно используется в деструкторах, которые могут выбросить исключение. Подробнее этот вопрос рассмотрен в *разд. 15.7*.

Листинг 36.3. Определение наличия выброшенного исключения

```
#include <iostream>
#include <exception>
using namespace std;
struct Demo
{
  ~Demo()
```

```
  {
    if (uncaught_exception())
      cout << "There is uncaught exception." << endl;
    else
    {
      cout << "Exception is thrown." << endl;
      throw "Exception";
    }
  }
};
int main()
{
  try
  {
    Demo a;
    {
      Demo b;
    }  // Exception is thrown.
  }  // There is uncaught exception.
  catch(char* str)
  {
    cout << str << " is catched." << endl;
  }
  return 0;
}
```

ГЛАВА 37



Классы стандартных исключений <stdexcept>

В заголовочном файле stdexcept определены классы стандартных исключений, которые могут быть сгенерированы средой исполнения программы или функциями из стандартной библиотеки. Ниже приведены объявления этих классов:

```
namespace std
{
  class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
  class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}
```

Эти классы наследуются от класса `exception` и образуют иерархию классов, которая приведена на рис. 37.1.

Кратко опишем ситуации, в которых генерируются эти стандартные исключения:

- `logic_error` — логические ошибки в программе;
  - `domain_error` — аргумент вне области определения;
  - `invalid_argument` — неправильный аргумент функции;

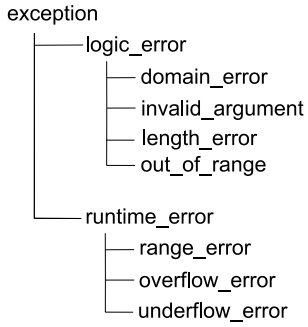


Рис. 37.1. Иерархия классов стандартных исключений

- `length_error` — длина превышает допустимую длину объекта;
- `out_of_range` — аргумент вне диапазона значений;

❒ `runtime_error` — ошибки среды исполнения программы;

- `range_error` — значение функции не имеет представления;
- `overflow_error` — при выполнении арифметической операции получено очень большое числовое значение, которое не имеет представления;
- `underflow_error` — при выполнении арифметической операции получено очень малое числовое значение, которое не имеет представления.

При рассмотрении конкретных стандартных функций будет указано, какие исключения они могут генерировать. Теперь же более подробно рассмотрим интерфейсы классов стандартных исключений. Для примера рассмотрим интерфейс класса `out_of_range`. Остальные классы стандартных исключений имеют аналогичные интерфейсы, за исключением того, что классы `logic_error` и `runtime_error` наследуются от класса `exception`.

Класс `out_of_range` имеет следующий интерфейс:

```
class out_of_range : public logic_error
{
public:
    out_of_range(const string& what_arg);
 };
```

Как видим, все члены класса `out_of_range` наследуются от класса `logic_error`, который в свою очередь наследует все свои члены от класса `exception`. Единственным параметром конструктора является объект типа `string`, который содержит описание исключения. Этот объект и возвращает функция `what`, которая наследуется классом `out_of_range` от класса `exception` через класс `logic_error`.

Тип `string` является классом, который служит для представления строк языка программирования C++. Этот класс описан в *гл. 39*.

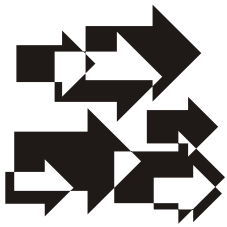
Кроме того, заметим, что стандартные исключения можно также генерировать и в пользовательских функциях. Но при этом не нужно противоречить назначению стандартных исключений. Иначе возможны недоразумения у других пользователей этой функции.

В листинге 37.1 приведен пример обработки стандартного исключения, которое генерирует оператор индексирования, перегруженный как член контейнерного класса `bitset`, предназначенного для хранения массива бит и определенного в стандартной библиотеке языка программирования C++.

Листинг 37.1. Обработка стандартного исключения

```
#include <iostream>
#include <bitset>
#include <stdexcept>
using namespace std;

int main()
{
  const bitset<8> x(0xFF);

  try
  {
    bool y = x[10];  // выход индекса за границы
  }
  catch (out_of_range& e)
  {
```

```
    cout << e.what() << endl; // invalid bitset<N> position
  }

  cout << "Press any key to exit." << endl;
  cin.get();

  return 0;
}
```



ГЛАВА 38

Динамическая идентификация типов <typeinfo>

38.1. Динамическая идентификация типов

Под динамической идентификацией типов (RTTI, Run-Time Type Identification) понимают определение типа объекта, на который ссылается ссылка или указывает указатель, во время исполнения программы. Если тип ссылки или указателя полиморфный, то динамическая идентификация типов позволяет определить тип реального объекта, на который ссылается эта ссылка или указывает указатель, во время исполнения программы. Если же тип ссылки или указателя не является полиморфным, то определение типа объекта выполняется статически во время компиляции программы.

Для поддержки динамической идентификации и преобразования типов объектов в заголовочном файле typeinfo определены следующие классы:

```
namespace std
{
  class type_info;
  class bad_cast;
  class bad_typeid;
}
```

Назначение этих классов описано в следующих разделах данной главы.

В заключение этого раздела заметим, что не все компиляторы поддерживают динамическую идентификацию типов по умолчанию.

38.2. Класс *type_info*

Класс `type_info` содержит описания типов, определенных в программе. Объекты этого класса хранят указатель на имя типа и код, который позволяет сравнивать типы на равенство и сортировать типы в соответствии с заданным порядком сортировки символов (collation order).

Класс `type_info` имеет следующий интерфейс:

```
namespace std
{
  class type_info
  {
  private:
    type_info(const type_info&  rhs);
    type_info& operator=(const type_info&  rhs);
  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;
  };
}
```

Из этого интерфейса видно, что в программе пользователя невозможно создать объект типа `type_info`, так как этот класс не имеет конструкторов в разделе `public`. Объект этого класса может быть создан только оператором `typeid`, который рассмотрен в следующем разделе.

Объекты класса `type_info` можно сравнивать на равенство или неравенство, используя перегруженные операторы члены класса `==` или `!=` соответственно.

Метод `before` позволяет сравнивать типы в соответствии с заданным порядком сортировки символов (collation order). Если объект, на который указывает указатель `this`, предшествует объекту, заданному параметром `rhs`, то метод `before` возвращает значение `true`, в противном случае — `false`.

Метод `name` возвращает указатель на строку языка программирования C, которая описывает тип.

38.3. Оператор *typeid*

Для динамической идентификации типа объекта в языке программирования C++ предназначен оператор `typeid`, который имеет следующий вид:

```
typeid(выражение|имя_типа)
```

Оператор `typeid` возвращает константный объект класса `type_info`, который содержит информацию о типе параметра оператора `typeid`. Поэтому при использовании оператора `typeid` в программу нужно включить заголовочный файл typeinfo.

Работает оператор `typeid` следующим образом:

- если операндом оператора `typeid` является L-value, тип которого является полиморфным классом, то оператор определяет динамический тип операнда, учитывая, что операнд может быть наследником полиморфного класса;
- если операндом оператора `typeid` не является L-value, то оператор определяет статический тип операнда, то есть наследование полиморфных классов не принимается во внимание;
- если операндом оператора `typeid` является имя типа, то оператор возвращает информацию об этом типе, причем тип должен быть полностью определен.

В листинге 38.1 приведен пример динамического определения типа объектов при помощи оператора `typeid` и использования методов класса `type_info`.

Листинг 38.1. Динамическое определение типа объекта

```
#include <iostream>
#include <typeinfo>
using namespace std;

class Base
{
  virtual f() {};
};
class Derived: public Base {};

int main()
{
  Base *p = new Base;
  Derived *q = new Derived;

  cout << (typeid(*p) == typeid(*q)) << endl;  // 0
  cout << (typeid(*p) != typeid(*q)) << endl;  // 1

  cout << (typeid(*p).before(typeid(*q))) << endl;  // 1
  delete p;
  p = q;

  cout << typeid(*p).name() << endl;  // class Derived
  cout << typeid(*q).name() << endl;  // class Derived
  delete q;

  return 0;

}
```

38.4. Обработка исключения *bad_typeid*

Если операндом оператора `typeid` является пустой указатель, то этот оператор выбрасывает исключение типа `bad_typeid`, который имеет следующий интерфейс:

```
namespace std
{
  class bad_typeid : public exception
  {
  public:
    bad_typeid() throw();
    bad_typeid(const bad_typeid&) throw();
    bad_typeid& operator=(const bad_typeid&) throw();
    virtual ~bad_typeid() throw();
    virtual const char* what() const throw();
  };
}
```

Отсюда видно, что класс `bad_typeid` наследуется от класса `exception`. Метод `what` этого класса возвращает указатель на строку языка программирования C, содержание которой зависит от реализации.

В листинге 38.2 приведен пример обработки исключения типа `bad_typeid`.

Листинг 38.2. Обработка исключения типа `bad_typeid`

```
#include <iostream>
#include <typeinfo>
using namespace std;

class Demo
{
  virtual ~Demo() {};
};
int main()
{
  Demo* d = NULL;

  try
  {
    typeid(*d);
  }
```

```
  catch (bad_typeid& e)
  {
    cout << e.what() << endl;
  }

  return 0;
}
```

38.5. Обработка исключения *bad_cast*

Если операндом оператора `dynamic_cast` является пустой указатель, то этот оператор выбрасывает исключение типа `bad_cast`, который имеет следующий интерфейс:

```
  namespace std
  {
    class bad_cast : public exception
    {
    public:
      bad_cast() throw();
      bad_cast(const bad_cast&) throw();
      bad_cast& operator=(const bad_cast&) throw();
      virtual ~bad_cast() throw();
      virtual const char* what() const throw();
    };
  }
```

Отсюда видно, что класс `bad_cast` наследуется от класса `exception`. Метод `what` этого класса возвращает указатель на строку языка программирования C, содержание которой зависит от реализации.

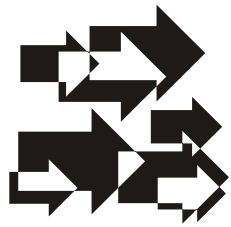
В листинге 38.3 приведен пример обработки исключения типа `bad_cast`.

Листинг 38.3. Обработка исключения типа `bad_cast`

```
#include <typeinfo.h>
#include <iostream>
```

```
using namespace std;

class Base
{
  virtual f() {};
};
class Derived: public Base {};
void f(Base& b) { Derived& rd = dynamic_cast<Derived&>(b); }

int main()
{
  Base b;
  Derived d;

  try
  {
    f(d);
    cout << "f(d)" << endl;  // f(d)
    f(b);
    cout << "f(b)" << endl;
  }
  catch (bad_cast& e)
  {
    cout << e.what() << endl;  // Bad dynamic_cast
  }

  return 0;
}
```

ГЛАВА 39



Работа со строками в C++ <string>

В заголовочном файле string определяются следующие шаблоны классов и их специализации:

```
namespace std
{
  template<class E> class char_traits;
  template<> class char_traits<char>;
  template<> class char_traits<wchar_t>;
  template<class E, class T = char_traits<E>,
    class A = allocator<E>> class basic_string;
  typedef basic_string<char> string;
  typedef basic_string<wchar_t> wstring;
}
```

Шаблон класса `char_traits` определяет характеристики объектов класса `E`, который является параметром шаблона. При работе со строками эти объекты обычно называются символами.

Для шаблона класса `char_traits` определены две явные специализации `char_traits<char>` и `char_traits<wchar_t>`, которые определяют характеристики типов `char` и `wchar_t` соответственно.

Шаблон класса `basic_string` определяет контейнер, элементами которого являются объекты класса, задаваемого параметром `E` этого шаблона. Параметры `T` и `A` задают классы, которые определяют соответственно характеристики и распределитель памяти для объектов класса `E`.

В заголовочном файле string также определяются типы `string` и `wstring`, которые являются синонимами шаблонных классов `basic_string<char>` и `basic_string<wchar_t>` соответственно, т. е. синонимами контейнерных классов, элементами которых являются символы типов `char` и `wchar_t`.

Кроме того, в заголовочном файле string определены шаблоны операторов сравнения и сцепления строк, а также функции для ввода строк и обмена строк своими значениями. Все эти операторы и функции будут рассмотрены в следующих разделах этой главы.

39.1. Характеристики символов

Характеристики (traits) это множество классов, каждый из которых обладает определенными свойствами. Эти свойства задаются набором типов и функций, которые должны быть определены в любом классе из этого множества. Характеристики позволяют единообразно обрабатывать различные классы, которые обладают одинаковыми свойствами. Характеристики могут быть определены при помощи шаблона класса. В этом случае каждый шаблонный класс, созданный из этого шаблона, обладает заданными свойствами.

Характеристики символов задаются шаблоном класса `char_traits`, который имеет следующий интерфейс:

```
template<class E> class char_traits
{
public:
  // ассоциированный тип, определяемый параметром шаблона
  typedef E char_type;
  // ассоциированные типы, определяемые реализацией
  typedef T1 int_type;
  typedef T2 pos_type;
  typedef T3 off_type;
  typedef T4 state_type;
  // функции присваивания
  static void assign(char_type& x, const char_type& y);
```

```
  static char_type* assign(
    char_type *x, size_t n, char_type y);
  // функции сравнения
  static bool eq(const char_type& x, const char_type& y);
  static bool lt(const char_type& x, const char_type& y);
  static int compare(
    const char_type *x, const char_type *y, size_t n);
  static bool eq_int_type(
    const int_type& ch1, const int_type& ch2);
  // функция определения длины символа
  static size_t length(const char_type *x);
  // функции копирования символов
  static char_type *copy(
    char_type *x, const char_type *y, size_t n);
  static char_type *move(
    char_type *x, const char_type *y, size_t n);
  // функция поиска символа
  static const char_type *find(
    const char_type *x, size_t n, const char_type& y);
  // функции преобразования символов
  static char_type to_char_type(const int_type& ch);
  static int_type to_int_type(const char_type& c);
  // функции определения конца файла
  static int_type eof();
  static int_type not_eof(const int_type& ch);
};
```

Предполагается, что параметр шаблона является типом, для которого определены конструктор по умолчанию, конструктор копирования и оператор присваивания. Кроме того, побитовое копирование объектов этого типа должно быть эквивалентно исполнению оператора присваивания.

Ассоциированные типы, определяемые реализацией, имеют следующее назначение:

- `int_type` — целочисленный тип, который допускает преобразование к типу `E` и наоборот, а также обеспечивает представление любого символа, т. е. объекта типа `E`, без потери значения;

- `pos_type` — тип для описания индикатора позиции файла;
- `off_type` — тип для описания смещения в операциях по изменению значения индикатора позиции файла;
- `state_type` — тип для описания состояния разбираемой строки длинных (multibyte) символов.

Прежде чем рассматривать функции-члены шаблона класса `char_traits`, заметим, что все они определены как статические члены класса. Теперь перейдем к описанию этих функций:

- `assign` — присваивают символы. Первая функция присваивает параметру `x` значение параметра `y`. Вторая функция присваивает `n` элементам массива `x` значение параметра `y`;
- `eq` — возвращает значение `true`, если `x` равно `y`, `false` — в остальных случаях;
- `lt` — возвращает значение `true`, если `x` меньше `y`, `false` — в остальных случаях;
- `compare` — поэлементно сравнивает массивы `x` и `y` длины `n`, возвращает:
  - отрицательное число, если для первых не равных элементов элемент массива `x` меньше элемента массива `y`;
  - ноль, если все элементы массивов равны;
  - положительное число в остальных случаях;
- `eq_int_type` — возвращает значение `true`, если `ch1` равно `ch2`, значение `false` — в остальных случаях;
- `length` — возвращает количество символов в последовательности `x` до первого символа `char_type()`;
- `copy` — копирует `n` символов из массива `x` в массив `y`. Возвращает массив `x`. Массивы не должны перекрываться;
- `move` — копирует `n` символов из массива `x` в массив `y`. Возвращает массив `x`. Массивы могут перекрываться;
- `find` — возвращает адрес первого элемента массива `x`, который равен параметру `y`;
- `to_char_type` — преобразует значение параметра `ch` к типу `char_type`;

- `to_int_type` — преобразует значение параметра `c` к типу `int_type`;
- `eof` — возвращает значение, обозначающее конец файла. Это значение не должно быть равно ни одному из значений типа `E`, которое обозначает символ;
- `not_eof` — сравнивает значение параметра `ch` со значением `eof()`. Если эти значения не равны, то функция возвращает значение параметра `ch`, в противном случае она возвращает значение, отличное от значения `eof()`.

На практике шаблон класса `char_traits` используется редко, т. к. в заголовочном файле string определены две явные специализации этого шаблона класса, а именно `char_traits<char>` и `char_traits<wchar_t>`, которые и используются при работе с символами типов `char` и `wchar_t` соответственно.

В листинге 39.1 приведен пример работы с шаблоном класса `char_traits`.

Листинг 39.1. Пример работы с шаблоном класса `char_traits`

```
#include <iostream>
#include <string.h>
#include <string>
using namespace std;

int main()
{
  typedef char_traits<char> traits;
  char ch;
  char* s = "Demo string";
  char buf[256];

  traits::assign(ch, 's');
  cout << ch << endl;              // s

  cout << traits::compare(s, s, ::strlen(s)) << endl;  // 0

  traits::copy(buf, s, ::strlen(s) + 1);
  cout << buf << endl;             // Demo string
```

```
  cout << traits::eof() << endl;  // -1

  return 0;
}
```

39.2. Строки

В языке программирования C++ под строкой понимается контейнерный объект шаблонного класса, хранящий последовательность символов:

```
template<class E, class T = char_traits<E>,
  class A = allocator<E>> class basic_string;
```

При этом тип символов, из которых состоит строка, определяется параметром `E` шаблона. Характеристики символов задаются параметром `T` шаблона класса, а параметр `A` задает класс, который управляет распределением памяти под символ типа `E`.

В заголовочном файле string также определяются типы `string` и `wstring`, которые являются синонимами шаблонных классов `basic_string<char>` и `basic_string<wchar_t>` соответственно. Эти типы позволяют работать со строками, элементами которых являются символы типов `char` и `wchar_t` соответственно. Как правило, этих типов хватает для обработки символьной информации, исключая длинные (multibyte) символы восточных алфавитов. Поэтому непосредственно с шаблоном класса `basic_string` работают редко.

Строки языка программирования C++ обладают следующей дополнительной функциональностью по сравнению со строками языка программирования C:

- автоматическое управление размером строки;
- вставка в строку символов и других строк;
- удаление из строки символов и подстрок;
- замена в строке символов и подстрок.

В следующих подразделах описаны все открытые (public) типы и члены, определенные в шаблоне класса `basic_string`.

39.2.1. Ассоциированные типы

В шаблоне класса `basic_string` определены следующие типы, определяемые параметрами шаблона класса:

```
typedef T traits_type;
typedef A allocator_type;
```

Кроме того, в шаблоне класса `basic_string` определены следующие типы, значение которых зависит от реализации шаблона класса:

```
typedef T0 iterator;
typedef T1 const_iterator;
typedef T2 size_type;
typedef T3 difference_type;
typedef reverse_iterator<iterator> reverse_iterator;
typedef reverse_iterator<const_iterator>
  const_reverse_iterator;
typedef typename allocator_type::pointer pointer;
typedef typename allocator_type::const_pointer
  const_pointer;
typedef typename allocator_type::reference reference;
typedef typename allocator_type::const_reference
  const_reference;
typedef typename allocator_type::value_type value_type;
```

Эти типы имеют следующее назначение:

- `iterator` и `const_iterator` — определяют соответственно типы переменных и константных итераторов произвольного доступа, связанных со строкой;
- `size_type` — служит для описания длин строк;
- `difference_type` — служит для описания разности адресов двух символов строки;
- `reverse_iterator` и `reverse_iterator` — определяют соответственно типы переменных и константных обратимых итераторов, связанных со строкой;
- `pointer` и `const_pointer` — описывают соответственно указатели на элементы переменных и константных строк;

- `reference` и `const_reference` — описывают соответственно ссылки на элементы переменных и константных строк;
- `value_type` — описывает тип элементов строки.

Итератор (iterator) это абстракция указателя языка программирования C++. Можно сказать, что итератор это множество классов, в каждом из которых перегружены операторы разыменования `*`, инкремента `++` и декремента `--`, которые имеют тот же смысл, что соответствующие операторы, применяемые к указателям. В стандартной библиотеке STL итераторы детально классифицированы в соответствии с набором перегруженных в них операторов и семантикой исполнения этих операторов. Здесь же только скажем, что любой итератор произвольного доступа (random access iterator) перегружает дополнительно арифметические операторы `+`, `-`, `+=` и `-=`, а также оператор индексирования `[]`. В отличие от итератора произвольного доступа в обратном итераторе оператор индексирования не обязательно может быть перегружен.

Использование типов, ассоциированных с шаблоном класса `basic_string`, обеспечивает независимость программ от значений параметров и реализации этого шаблона класса.

39.2.2. Специальное значение символьного типа

Статический член:

```
static const size_type npos = -1;
```

шаблонного класса `basic_string` определяет константу, значение которой не является действительным символом. Эта константа служит для возвращения значения, выходящего за пределы допустимых значений символов, или как специальный код.

39.2.3. Конструкторы

В шаблоне класса `basic_string` определены следующие конструкторы:

```
basic_string();
explicit basic_string(const allocator_type& al);
```

```
basic_string(size_type n, value_type c);
basic_string(size_type n, value_type c,
    const allocator_type& al);
basic_string(const value_type *s, size_type n);
basic_string(const value_type *s, size_type n,
    const allocator_type& al);
basic_string(const value_type *s);
basic_string(const value_type *s, const allocator_type& al);
basic_string(const basic_string& rhs);
basic_string(const basic_string& rhs, size_type pos,
    size_type n = npos);
basic_string(const basic_string& rhs, size_type pos,
    size_type n, const allocator_type& al);
```

Все конструкторы запоминают распределитель памяти и инициализируют последовательность символов в шаблонном классе.

Если распределитель памяти присутствует как параметр `al`, то он и запоминается. Конструкторы копирования запоминают распределитель памяти `rhs.get_allocator()`. Остальные конструкторы запоминают распределитель памяти `A()`, где `A` — параметр шаблона.

Все остальные параметры конструкторов служат для инициализации последовательности символов, хранящейся в строке, и имеют следующее назначение:

- `n` — конструкторов задает начальную длину создаваемой строки;
- `c` — задает символ, которым инициализируется строка;
- `s` — задает строку языка программирования C, которой инициализируется строка;
- `rhs` — задает другую строку, которой инициализируется создаваемая строка;
- `pos` — задает позицию в строке, начиная с которой эта строка копируется в создаваемую строку.

Если других параметров нет, то инициализируется пустая строка.

В листинге 39.2 приведены примеры инициализации строк.

Листинг 39.2. Примеры инициализации строк

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string s1(3, 'a');
  cout << s1 << endl;  // aaa

  string s2("Demo string");
  cout << s2 << endl;  // Demo string

  string s3("Demo string", 4);
  cout << s3 << endl;  // Demo

  string s4(s2);
  cout << s4 << endl;  // Demo string

  string s5(s2, 5, 6);
  cout << s5 << endl;  // string

  return 0;
}
```

39.2.4. Шаблоны конструкторов

В шаблоне класса `basic_string` определены следующие шаблоны конструкторов:

```
template <class InIt> basic_string(InIt first, InIt last);
template <class InIt> basic_string(InIt first,
    InIt last, const allocator_type& al);
```

Шаблоны конструкторов служат для инициализации строки подстрокой другого объекта класса `basic_string`. Эта подстрока задается интервалом итераторов `[first, last)` (такое обозначение говорит о том, что левая граница входит в интервал, а правая — нет). Параметр `al` задает распределитель памяти для элементов строки.

Например:

```
string s("aaa bbb");
string::iterator it_beg = s.begin(), it_end = s.end();

string d(it_beg + 1, it_end - 1);
cout << d << endl;  // aa bb
```

39.2.5. Операторы присваивания

В шаблоне класса `basic_string` перегружены оператор присваивания:

```
basic_string& operator=(const basic_string& rhs);
basic_string& operator=(const value_type *s);
basic_string& operator=(value_type c);
```

То есть строкам можно присваивать значения других строк, строк языка программирования C и отдельных символов. Например:

```
string s;
string t("Some string");

s = 'a';
s = "Demo string";
s = t;
```

39.2.6. Функции, возвращающие итераторы

В шаблоне класса `basic_string` определены следующие функции-члены, возвращающие итераторы элементов строки:

```
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;
```

Эти функции выполняют следующие действия:

- `begin` — возвращает итератор произвольного доступа, который указывает на первый элемент последовательности символов;
- `end` — возвращает итератор произвольного доступа, который указывает на элемент, следующий за последним элементом последовательности символов;
- `rbegin` — возвращает обратный итератор, который указывает на элемент, следующий за последним элементом последовательности символов;
- `rend` — возвращает обратный итератор, который указывает на первый элемент последовательности символов.

Примеры использования функций `begin` и `end` были приведены в *разд. 39.2.4*. Сейчас же приведем примеры использования функций `rbegin` и `rend`.

```
string s("aaa bbb");
string::reverse_iterator it_beg = s.rbegin(),
                         it_end = s.rend();
while (it_beg != it_end)
  cout << *(it_beg++);  // bbb aaa
```

39.2.7. Получение ссылки на элемент строки

Для получения ссылки на элемент строки в шаблоне класса `basic_string` определены следующие функции:

```
reference at(size_type pos);
const_reference at(size_type pos) const;
```

Эти функции возвращают ссылку на элемент последовательности, позиция которого задана параметром `pos`. Если позиция выходит за пределы последовательности, то функции выбрасывают исключение типа `out_of_range`. Например:

```
string s("aaa");

s.at(1) = 'b';
cout << s << endl;  // aba
```

39.2.8. Оператор индексирования

В шаблоне класса basic_string перегружен оператор индексирования:

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
```

Эти операторы возвращают ссылку на элемент последовательности, позиция которого задана параметром pos. Например:

```
string s("aaa");
s[1] = 'b';  // s = "aba"
```

Если позиция выходит за пределы последовательности, то поведение оператора не определено.

39.2.9. Добавление символа в конец строки

Для добавления символа в конец строки в шаблоне класса basic_string определена функция

```
void push_back(value_type c);
```

В конец строки добавляется символ, заданный параметром c. Например:

```
string s("aaa");
s.push_back('b');  // s = "aaab"
```

39.2.10. Преобразование в строку языка C

Для преобразования объекта шаблонного класса basic_string в строку языка C предназначена функция:

```
const value_type *c_str() const;
```

Она возвращает указатель на константную строку, содержащую последовательность символов, управляемую объектом шаблонного класса basic_string. Последующий вызов не константной функции-члена шаблонного класса может сделать недействительным этот указатель.

Например:

```
string s("Demo string");
cout << s.c_str() << endl;  // Demo string
```

39.2.11. Получение адреса первого элемента строки

Для получения адреса первого элемента последовательности символов, управляемой шаблонным классом `basic_string`, предназначена функция:

```
const value_type *data() const;
```

Например:

```
string s("aaa bbb");
const char* p = s.data();

while (*p != ' ')
  cout << *p++;  // aaa
```

39.2.12. Функции, работающие с длиной строки

Для работы с длиной строки в шаблоне класса `basic_string` определены следующие функции:

```
size_type length() const;
size_type size() const;
size_type max_size() const;
void resize(size_type n, value_type c = value_type());
size_type capacity() const;
void reserve(size_type n = 0);
bool empty() const;
```

Эти функции выполняют следующие действия:

- `length` — возвращает длину строки в символах;
- `size` — также возвращает длину строки;
- `max_size` — возвращает максимально возможную длину строки;

- `resize` — устанавливает длину строки равной значению параметра `n`. Если длина строки увеличивается, то функция заполняет свободные элементы значениями параметра `c`. Если длина строки уменьшается, то вызывается функция `erase(begin() + n, end())`;
- `capacity` — возвращает количество элементов, которые могут храниться в строке, без перераспределения памяти;
- `reserve` — устанавливает емкость строки равной значению, которое не меньше значения параметра `n`;
- `empty` — возвращает `true`, если строка пустая.

Например:

```
string s("Demo string");
cout << s.length() << endl;  // 11
```

39.2.13. Соединение строк

Для присоединения к строке другой строки в шаблоне класса `basic_string` перегружен оператор `+=`:

```
basic_string& operator+=(const basic_string& rhs);
basic_string& operator+=(const value_type *s);
basic_string& operator+=(value_type c);
```

Параметром этого оператора может быть объект типа `basic_string`, строка языка программирования C или просто символ. Например:

```
string d("This");
string s(" string");

d += " is ";
d += 'a';
d += s;
cout << d << endl;  // This is a string
```

Для присоединения к строке другой строки в шаблоне класса `basic_string` также перегружены функции `append`:

```
basic_string& append(const basic_string& str);
basic_string& append(
  const basic_string& str, size_type pos, size_type n);
basic_string& append(const value_type *s, size_type n);
basic_string& append(const value_type *s);
basic_string& append(size_type n, value_type c);
template<class InIt> basic_string& append(
  InIt first, InIt last);
```

Эти функции обеспечивают более широкие возможности при соединении строк, чем оператор `+=`. Все функции `append` возвращают модифицированную строку.

Параметры функций `append` имеют следующее назначение:

- `pos` — задает позицию символа в присоединяемой строке, начиная с которого выполняется присоединение;
- `n` — задает количество присоединяемых символов.

Остальные параметры задают строку или символы, которые присоединяются к исходной строке. Например:

```
string s;
string t("a string");

s.append(t);                // a string
s.append(t, 1, 4);          // a string str
s.append(" - ");            // a string str -
s.append("yes or no", 3);   // a string str - yes
s.append(3, '!');           // a string str - yes!!!
```

Параметры шаблонной функции `append` являются итераторами строки типа `basic_string` и задают в этой строке подстроку `[first, last)`, которая присоединяется к исходной строке. Например:

```
string s("ab");
string t("-c-");

string::iterator it_beg = t.begin(), it_end = t.end();
s.append(it_beg + 1, it_end - 1);  // s = "abc"
```

39.2.14. Присваивание строк

Для присваивания строк друг другу в шаблоне класса `basic_string` перегружены функции `assign`:

```
basic_string& assign(const basic_string& str);
basic_string& assign(const basic_string& str,
    size_type pos, size_type n);
basic_string& assign(const value_type *s, size_type n);
basic_string& assign(const value_type *s);
basic_string& assign(size_type n, value_type c);
template<class InIt> basic_string& assign(
    InIt first, InIt last);
```

Функции `assign` предоставляют более широкие возможности при присваивании строк, чем оператор присваивания `=`, т. к. позволяют присваивать объекту типа `basic_string` подстроки и заполнять строку символом-заполнителем. Все функции `assign` возвращают модифицированную строку.

Параметры функций `assign` имеют следующее назначение:

- `pos` — задает позицию символа в присваиваемой строке, начиная с которого выполняется присваивание;
- `n` — задает количество присваиваемых символов.

Остальные параметры задают строку или символы, которые присваиваются исходной строке. Например:

```
string s;
string t("aaa bbb");

s.assign(t);              // aaa bbb
s.assign(t, 1, 5);        // aa bb
s.assign("aaa bbb");      // aaa bbb
s.assign("aaa bbb", 3);   // aaa
s.assign(3, '!');         // !!!
```

Параметры шаблонной функции `assign` являются итераторами строки типа `basic_string` и задают в этой строке подстроку `[first, last)`, которая присваивается исходной строке.

Например:

```
string s;
string t("aaa bbb");

string::iterator it_beg = t.begin(), it_end = t.end();
s.append(it_beg + 1, it_end - 1);  // s = "aa bb"
```

39.2.15. Вставка строк и символов в строку

Для вставки в строку другой строки типа `basic_string`, строки языка программирования C, одного или нескольких символов в шаблоне класса `basic_string` перегружены функции `erase`:

```
basic_string& insert(size_type p0, const basic_string& str);
basic_string& insert(size_type p0,
    const basic_string& str, size_type pos, size_type n);
basic_string& insert(size_type p0, const value_type *s,
    size_type n);
basic_string& insert(size_type p0, const value_type *s);
basic_string& insert(size_type p0, size_type n,
    value_type c);
iterator insert(iterator it, value_type c = value_type());
void insert(iterator it, size_type n, value_type c);
template<class InIt>
     void insert(iterator it, InIt first, InIt last);
```

Если функция `insert` возвращает значение, то это или модифицированная строка, или итератор, который указывает на вставленный символ.

Параметры функций `insert` имеют следующее назначение:

- `p0` — указывает на позицию в текущей строке, с которой начинается вставка;
- `pos` — задает позицию символа во вставляемой строке, начиная с которого выполняется вставка;
- `n` — задает количество вставляемых символов;
- `it` — итератор, указывающий на позицию в текущей строке, с которой начинается вставка.

Остальные параметры задают строку или символы, которые вставляются в исходную строку. Например:

```
string s("aa  cc");
string t("bb");

s.insert(3, t);           // s = "aa bb cc"
s.insert(3, t, 1, 1);     // s = "aa bbb cc"
s.insert(3, "<<");        // s = "aa <<bbb cc"
s.insert(8, ">>>>", 2);   // s = "aa <<bbb>> cc"
s.insert(8, 3, '-');      // s = "aa <<bbb--->> cc"

string::iterator it = s.begin();
s.insert(it + 5, 3, '-');  // s = "aa <<---bbb--->> cc"
```

Параметры `first` и `last` шаблонной функции `insert` являются итераторами вставляемой строки типа `basic_string` и задают в этой строке подстроку `[first, last)`, которая вставляется в исходную строку. Например:

```
string s("aa  cc");
string t("--bb--");

string::iterator it = s.begin(),
    it_beg = t.begin(), it_end = t.end();
s.insert(it + 3, it_beg + 2, it_end - 2);  // s = "aa bb cc"
```

39.2.16. Удаление последовательности символов из строки

Для удаления подстрок или символа из текущей строки в шаблоне класса `basic_string` перегружены функции `erase`:

```
basic_string& erase(size_type p0 = 0, size_type n = npos);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

Функции `erase` возвращают или модифицированную строку, или итератор, указывающий на символ, следующий за последним удаленным символом.

Параметры функций `erase` имеют следующее назначение:

- `p0` — указывает на позицию в текущей строке, с которой начинается удаление символов;
- `n` — задает количество удаляемых символов;
- `it` — итератор, указывающий на позицию в текущей строке, с которой начинается удаление символов;
- `first` и `last` — задают интервал `[first, last)` удаляемых символов.

Например:

```
string s("aa bbbb cc");
s.erase(3, 2);  // s = "aa bb cc"

string::iterator it_beg = s.begin(), it_end = s.end();
s.erase(it_beg + 3);  // s = "aa b cc"
s.erase(it_beg + 3, it_end - 3); // s == "aa cc"
```

39.2.17. Очистка строки

Для очистки строки в шаблоне класса `basic_string` определена функция `clear`:

```
void clear();
```

Эта функция удаляет все символы из строки.

39.2.18. Замена символов и подстрок в строке

Для замены подстроки в строке типа `basic_string` на подстроку этого же типа, или подстроку языка программирования C, а также для заполнения подстроки символом-заполнителем в шаблоне класса `basic_string` перегружены функции `replace`:

```
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str, size_type pos, size_type n);
```

```
basic_string& replace(size_type p0, size_type n0,
    const value_type *s, size_type n);
basic_string& replace(size_type p0, size_type n0,
    const value_type *s);
basic_string& replace(size_type p0, size_type n0,
    size_type n, value_type c);
basic_string& replace(iterator first0, iterator last0,
    const basic_string& str);
basic_string& replace(iterator first0, iterator last0,
    const value_type *s, size_type n);
basic_string& replace(iterator first0, iterator last0,
    const value_type *s);
basic_string& replace(iterator first0, iterator last0,
    size_type n, value_type c);
template<class InIt>
    basic_string& replace(iterator first0, iterator last0,
        InIt first, InIt last);
```

Все функции `replace` возвращают модифицированную строку.

Параметры функций `replace` имеют следующее назначение:

- `p0` — указывает на позицию в текущей строке, с которой начинается замена символов;
- `n0` — задает количество заменяемых символов;
- `pos` — задает позицию символа в строке-заместителе, начиная с которого выполняется замена;
- `n` — задает количество символов в строке-заместителе или количество символов заполнителей, используемых для замены;
- `first0` и `last0` — задают интервал `[first0, last0)` заменяемых символов в исходной строке.

Остальные параметры задают строку или символы, которые служат для замены. Например:

```
string s("--aa--");

s.replace(2, 2, string("bbb"));        // "--bbb--"
s.replace(2, 3, string("+cc+"), 1, 2); // "--cc--"
s.replace(2, 2, "bbbb", 2);            // "--bb--"
```

```
s.replace(2, 2, "ccc");                 // "--ccc--"
s.replace(2, 3, 2, 'x');                // "--xx--"

string::iterator first0 = s.begin(), last0 = s.end();

s.replace(first0 + 2, last0 - 2, string("aa"));  // "--aa--"
s.replace(first0 + 2, last0 - 2, "bb");          // "--bb--"
s.replace(first0 + 2, last0 - 2, "xxxx", 2);     // "--xx--"
s.replace(first0 + 2, last0 - 2, 2, 'a');        // "--aa--"
```

Параметры `first` и `last` шаблонной функции `replace` являются итераторами строки-заместителя типа `basic_string` и задают в этой строке подстроку `[first, last)`, которая заменяет подстроку в исходной строке. Например,

```
string s("--aa--");
string t("++bb++");
string::iterator first0 = s.begin(), last0 = s.end(),
                 first = t.begin(), last = t.end();

s.replace(first0 + 2, last0 - 2, first + 2, last - 2);
```

39.2.19. Копирование подстроки в массив символов

В шаблоне класса `basic_string` определена функция `copy`:

```
size_type copy(value_type *s, size_type n,
    size_type pos = 0) const;
```

Она копирует `n` символов из исходной строки, начиная с позиции `pos`, в массив, адрес которого задан параметром `s`. Возвращает функция количество скопированных элементов. Например:

```
char buffer[80] = "aa--aa";
string s("-bb-");

s.copy(buffer + 2, 2, 1);
cout << buffer << endl;  // aabbaa
```

39.2.20. Обмен строк

Для обмена строк в шаблоне класса `basic_string` определена функция:

```
void swap(basic_string& str);
```

Функция обменивает значение исходной строки со значением строки, заданной параметром `str`.

39.2.21. Поиск символов и подстрок в строке

Для поиска символов и подстрок в строке в шаблоне класса `basic_string` определены следующие функции:

```
size_type find(const basic_string& str,
    size_type pos = 0) const;
size_type find(const value_type *s, size_type pos,
    size_type n) const;
size_type find(const value_type *s,
    size_type pos = 0) const;
size_type find(value_type c, size_type pos = 0) const;
size_type rfind(const basic_string& str,
    size_type pos = npos) const;
size_type rfind(const value_type *s, size_type pos,
    size_type n = npos) const;
size_type rfind(const value_type *s,
    size_type pos = npos) const;
size_type rfind(value_type c, size_type pos = npos) const;
size_type find_first_of(const basic_string& str,
    size_type pos = 0) const;
size_type find_first_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_first_of(const value_type *s,
    size_type pos = 0) const;
size_type find_first_of(value_type c,
    size_type pos = 0) const;
size_type find_last_of(const basic_string& str,
    size_type pos = npos) const;
```

```
size_type find_last_of(const value_type *s,
    size_type pos, size_type n = npos) const;
size_type find_last_of(const value_type *s,
    size_type pos = npos) const;
size_type find_last_of(value_type c,
    size_type pos = npos) const;
size_type find_first_not_of(const basic_string& str,
    size_type pos = 0) const;
size_type find_first_not_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_first_not_of(const value_type *s,
    size_type pos = 0) const;
size_type find_first_not_of(value_type c,
    size_type pos = 0) const;
size_type find_last_not_of(const basic_string& str,
    size_type pos = npos) const;
size_type find_last_not_of(const value_type *s,
    size_type pos, size_type n) const;
size_type find_last_not_of(const value_type *s,
    size_type pos = npos) const;
size_type find_last_not_of(value_type c,
    size_type pos = npos) const;
```

Эти функции реализуют следующие алгоритмы:

- `find` — поиск первого символа или подстроки в текущей строке, начиная с позиции, заданной параметром `pos`, которые равны заданному символу или подстроке соответственно;
- `rfind` — обратный поиск, т. е. поиск в обратном направлении, первого символа или подстроки в текущей строке, начиная с позиции, заданной параметром `pos`, которые равны заданному символу или подстроке соответственно;
- `find_first_of` — поиск первого символа в текущей строке, начиная с позиции, заданной параметром `pos`, который равен любому из заданных символов;
- `find_last_of` — поиск последнего символа в текущей строке до позиции, заданной параметром `pos`, который равен любому из заданных символов;

- `find_first_not_of` — поиск первого символа в текущей строке, начиная с позиции, заданной параметром `pos`, который не равен ни одному из заданных символов;
- `find_last_not_of` — поиск последнего символа в текущей строке до позиции, заданной параметром `pos`, который не равен ни одному из заданных символов.

Все эти функции в случае удачи возвращают позицию найденного символа, а в случае неудачи — значение `npos`.

Параметры функций поиска имеют следующее назначение:

- `pos` — указывает на позицию символа в исходной строке, с которой начинается поиск;
- `n` — задает количество последовательных символов в заданной строке, поиск которых выполняется в исходной строке.

Остальные параметры задают строку или символ, которые ищутся в исходной строке. Например:

```
string s("--abc--");

cout << s.find(string("abc")) << endl;  // 2
cout << s.find("abc") << endl;          // 2
cout << s.find("c-b-a", 1, 2) << endl;  // 4
cout << s.find('a') << endl;            // 2
```

39.2.22. Выделение подстроки

Для выделения из исходной строки подстроки в шаблоне класса `basic_string` определена функция:

```
basic_string substr(size_type pos = 0,
    size_type n = npos) const;
```

Функция `substr` возвращает подстроку, которая начинается в исходной строке с позиции, заданной параметром `pos`, и имеет длину равную значению параметра `n`. Например:

```
string s("--abc--");
cout << s.substr(2, 3) << endl;  // abc
```

39.2.23. Сравнение строк

Для сравнения строки типа `basic_string` с другой строкой этого типа или со строкой языка программирования C в шаблоне класса `basic_string` перегружены функции `compare`:

```
int compare(const basic_string& str) const;
int compare(size_type p0, size_type n0,
    const basic_string& str);
int compare(size_type p0, size_type n0,
    const basic_string& str, size_type pos, size_type n);
int compare(const value_type *s) const;
int compare(size_type p0, size_type n0,
    const value_type *s) const;
int compare(size_type p0, size_type n0,
    const value_type *s, size_type pos) const;
```

Все эти функции возвращают следующие значения:

- меньше нуля — если строки не совпадают и первый несовпадающий символ исходной строки меньше символа заданной строки, с которым он сравнивается;
- ноль — если строки совпадают;
- больше нуля — если первый несовпадающий символ исходной строки больше символа заданной строки, с которым он сравнивается.

Параметры функций `compare` имеют следующее назначение:

- `p0` — задает позицию в исходной строке, с которой начинается сравнение;
- `n0` — задает количество сравниваемых символов.

Если эти параметры не заданы, то сравнивается вся строка. Остальные параметры задают строку, с которой выполняется сравнение. Например:

```
string s("-aa-");
cout << s.compare(s) << endl;                    // 0
cout << s.compare(1, 2, "aa") << endl;           // 0
cout << s.compare(1, 2, "+aa+", 1, 2) << endl;   // 0
```

39.2.24. Получение распределителя памяти для символа

В шаблоне класса `basic_string` определена функция:

```
allocator_type get_allocator() const;
```

Она возвращает распределитель памяти для символов, из которых состоит строка.

39.3. Шаблоны операторов

39.3.1. Соединение строк

Для соединения строки типа `basic_string` со строкой этого же типа, строкой языка программирования C или символом в заголовочном файле string перегружен шаблон оператора +:

```
namespace std
{
  template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
  template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs, const E *rhs);
  template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs, E rhs);
  template<class E, class T, class A>
    basic_string<E, T, A> operator+( const E *lhs,
        const basic_string<E, T, A>& rhs);
  template<class E, class T, class A>
    basic_string<E, T, A> operator+( E lhs,
        const basic_string<E, T, A>& rhs);
}
```

Например:

```
  cout << (string("-a") + string("b-")) << endl;  // -ab-
  cout << (string("-a") + "b-")  << endl;         // -ab-
  cout << (string("-a") + 'b' + '-')  << endl;    // -ab-
```

39.3.2. Сравнение строк

Для сравнения строк типа `basic_string` в заголовочном файле string перегружены следующие операторы сравнения:

```
namespace std
{
  template<class E, class T, class A>
    bool operator==(const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
    bool operator!=(const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
  template<class E, class T, class A>
    bool operator<(const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
  template<class E, class T, class A>
    bool operator>(const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
  template<class E, class T, class A>
    bool operator<=(const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
  template<class E, class T, class A>
    bool operator>=(const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
}
```

Эти операторы сравнения выполняют следующие действия:

- `==` — лексикографически сравнивает строки типа `basic_string`, если строки совпадают, то оператор возвращает значение `true`, в противном случае — `false`;
- `!=` — лексикографически сравнивает строки типа `basic_string`, если строки не совпадают, то оператор возвращает значение `true`, в противном случае — `false`;
- `<` — лексикографически сравнивает строки типа `basic_string`, если строка `lhs` меньше строки `rhs`, то оператор возвращает значение `true`, в противном случае — `false`;
- `>` — лексикографически сравнивает строки типа `basic_string`, если строка `lhs` больше строки `rhs`, то оператор возвращает значение `true`, в противном случае — `false`;

- <= — лексикографически сравнивает строки типа basic_string, если строка lhs не больше строки rhs, то оператор возвращает значение true, в противном случае — false;
- >= — лексикографически сравнивает строки типа basic_string, если строка lhs не меньше строки rhs, то оператор возвращает значение true, в противном случае — false.

Шаблоны операторов сравнения также перегружены для сравнения строки типа basic_string со строкой языка программирования C. Например, для оператора сравнения на равенство перегружены следующие шаблоны:

```
namespace std
{
  template<class E, class T, class A>
    bool operator==(const basic_string<E, T, A>& lhs,
        const E *rhs);
  template<class E, class T, class A>
    bool operator==(const E *lhs,
        const basic_string<E, T, A>& rhs);
}
```

Шаблоны остальных операторов сравнения строк типа basic_string со строками языка программирования C перегружены аналогично.

Ниже приведены примеры сравнения строк:

```
cout << (string("aa") <= string("ab")) << endl;  // 1
cout << (string("aa") == "aa")  << endl;         // 1
```

39.3.3. Ввод/вывод строк

Для ввода/вывода строк типа basic_string из потоков типа basic_istream в заголовочном файле string перегружены операторы сдвига (shift) >> и <<:

```
namespace std
{
  template<class E, class T, class A>
    basic_ostream<E>& operator<<(basic_ostream<E>& os,
        const basic_string<E, T, A>& str);
```

```
  template<class E, class T, class A>
    basic_istream<E>& operator>>(basic_istream<E>& is,
        basic_string<E, T, A>& str);
}
```

Перегруженный оператор >> вводит из потока в строку символы до тех пор, пока не встретит пробельный символ. При этом ведущие пробельные символы пропускаются.

Пример ввода и вывода строк из потока:

```
string s;
cout << "Input a word> : ";
cin >> s;
cout << "The word: " << s << endl;
```

39.4. Шаблоны функций

39.4.1. Ввод строк

Для ввода из потока последовательности символов в строку типа basic_string в заголовочном файле string перегружены шаблоны функции getline:

```
namespace std
{
  template<class E, class T, class A>
    basic_istream<E, T>& getline( basic_istream<E, T>& is,
        basic_string<E, T, A>& str);
  template<class E, class T, class A>
    basic_istream<E, T>& getline(basic_istream<E, T>& is,
        basic_string<E, T, A>& str, E delim);
}
```

Первая шаблонная функция getline заканчивает ввод, встретив символ перехода на новую строку \n.

Вторая шаблонная функция getline заканчивает ввод, встретив символ, заданный параметром delim.

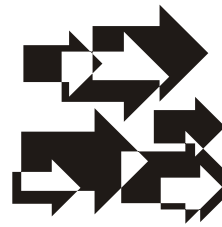
Обе функции возвращают введенную строку. Ниже приведен пример ввода и вывода строк из потока:

```
string s;
cout << "Input a line> : ";
getline(cin, s);
cout << "The line: " << s << endl;
```

39.4.2. Обмен строк

Для обмена строк типа `basic_string` своими значениями в заголовочном файле string определен шаблон функции `swap`:

```
namespace std
{
  template<class E, class T, class A>
    void swap(basic_string<E, T, A>& lhs,
        basic_string<E, T, A>& rhs);
}
```



ГЛАВА 40

Потоки ввода/вывода в C++

40.1. Структура стандартной библиотеки ввода/вывода

Как уже говорилось в *разд. 30.1, потоком* называется логический или, другими словами, программный интерфейс, который обеспечивает доступ к файлу. В языке программирования C++ потоки ввода/вывода представляются объектами шаблонных классов. Сами шаблоны потоков определены в стандартной библиотеке ввода/вывода языка программирования C++, которая организована как иерархия шаблонов классов.

В стандартной библиотеке определены два множества шаблонных классов этой иерархии: одно для работы с символами типа `char`, другое для работы с символами типа `wchar_t`. Классы для работы с символами типа `char` имеют синонимы, имена которых соответствуют именам шаблонов классов, но не содержат слова `basic` и символа `_`. Например, для шаблонного класса `basic_istream<char>` определен тип-синоним `istream`. Шаблонные классы для работы с символами типа `wchar_t` имеют те же имена, что и классы для работы с символами типа `char`, но с префиксом `w`. Например, для шаблонного класса `basic_istream<wchar>` определен тип-синоним `wistream`.

Ниже перечислены заголовочные файлы, а также классы и шаблоны классов из стандартной библиотеки ввода/вывода, которые определены в этих файлах:

- `ios` — класс `ios_base`, шаблон класса `basic_ios`;
- `ostream` — шаблон класса `basic_ostream`;
- `istream` — шаблоны классов `basic_istream` и `basic_iostream`;
- `iostream` — стандартные потоки `cin`, `cout`, `cerr`, `clog` и `wcin`, `wcout`, `wcerr`, `wclog`;
- `fstream` — шаблоны классов `basic_filebuf`, `basic_ifstream`, `basic_ofstream` и `basic_fstream`;
- `strstream` — классы `strstreambuf`, `istrstream`, `ostrstream` и `strstream`;
- `sstream` — шаблоны классов `basic_stringbuf`, `basic_istringstream`, `basic_ostringstream` и `basic_stringstream`;
- `streambuf` — шаблон класса `basic_streambuf`.

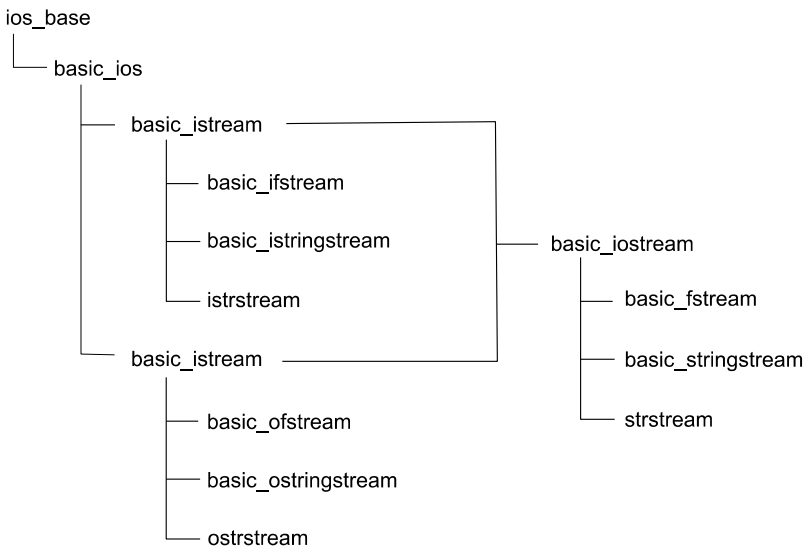


Рис. 40.1. Иерархия шаблонов классов стандартной библиотеки ввода/вывода

В следующих разделах все классы и шаблоны классов, определенные в стандартной библиотеке ввода/вывода, будут рассмотрены подробно.

40.2. Базовые классы потоков <ios>

В заголовочном файле ios определены следующие классы, шаблоны классов и типы:

```
namespace std
{
  typedef T1 streamoff;
  typedef T2 streamsize;

  class ios_base;
  template <class E, class T = char_traits<E>>
      class basic_ios;
  typedef basic_ios<char, char_traits<char>> ios;
  typedef basic_ios<wchar_t, char_traits<wchar_t>> wios;
  template <class St> class fpos;
  typedef fpos<mbstate_t> streampos;
  typedef fpos<mbstate_t> wstreampos;
}
```

Тип `streamoff` является синонимом некоторого целочисленного типа `T1`, который зависит от реализации и имеет длину, по крайней мере, 32 бита. Этот тип используется для представления смещения в операциях с индикаторами позиции потока. Этот тип не гарантирует представление любого смещения. Для обозначения неправильного смещения обычно используется значение `streamoff(-1)`.

Тип `streamsize` является синонимом некоторого целочисленного типа `T2`, который зависит от реализации и имеет длину, по крайней мере, 16 бит. Этот тип используется для представления длины в функциях-членах потоков. Этот тип не гарантирует представление любой длины.

В классе `ios_base` определены методы, которые являются общими для входных и выходных потоков и, кроме того, не зависят от параметров шаблонов.

В классе шаблоне класса `basic_ios` определены методы, которые также являются общими для входных и выходных потоков, но зависимы от параметров шаблонов.

Для специализаций basic_ios<char, char_traits<char>> и basic_ios<wchar_t, char_traits<wchar_t>> шаблонного класса basic_ios объявлены типы синонимы ios и wios, которые определяют базовые потоки ввода/вывода для работы с символами типов char и wchar_t соответственно.

Шаблон класса fpos определяет индикатор позиции потока. Для специализации fpos<mbstate_t> этого шаблона класса объявлены типы синонимы streampos и wstreampos, которые предназначены для работы с потоками типов ios и wios соответственно.

Кроме того, в заголовочном файле ios объявлены функции не члены классов, которые изменяют состояние потока. Такие функции называются *манипуляторами*.

Все эти классы и функции будут подробно рассмотрены ниже.

40.2.1. Класс *ios_base*

Объекты типа ios_base поддерживают следующую информацию о состоянии потока:

- флаги форматирования в объекте типа fmtflags;
- ширину поля ввода/вывода в объекте типа int;
- разрешение дисплея в объекте типа int;
- маску исключений в объекте типа iostate;
- стек вызовов callback функций при наступлении событий типа event;
- внутренний расширяемый массив с элементами типа long;
- внутренний расширяемый массив с элементами типа void*;
- объект типа locale, в котором хранится информация о локальности.

В этом разделе описаны все определенные в классе ios_base типы, открытые (public) члены этого класса, а также конструктор этого класса, который является защищенным (protected) членом.

40.2.1.1. Флаги

В классе `ios_base` определен тип `fmtflags`, который является синонимом вложенного типа `T1`, зависимого от реализации класса и определяющего битовую маску:

```
typedef T1 fmtflags;
```

В классе `ios_base` определены следующие флаги, которые управляют режимами работы потока и могут быть установлены в этой битовой маске:

```
static const fmtflags boolalpha, dec, fixed, hex,
    internal, left, oct, right, scientific,
    showbase, showpoint, showpos, skipws, unitbuf,
    uppercase, adjustfield, basefield, floatfield;
```

Эти флаги имеют следующее назначение:

- `boolalpha` — ввод/вывод булевых данных в текстовом формате;
- `dec` — ввод/вывод целочисленных данных в десятичном формате;
- `fixed` — ввод/вывод чисел с плавающей точкой в формате с точкой и без экспоненты;
- `hex` — ввод/вывод целочисленных данных в шестнадцатеричном формате;
- `internal` — вставка символа заполнителя внутрь поля вывода до достижения нужной ширины этого поля;
- `left` — вставка символа заполнителя в конец поля вывода до достижения нужной ширины этого поля;
- `oct` — ввод/вывод целочисленных данных в восьмеричном формате;
- `right` — вставка символа заполнителя в начало поля вывода до достижения нужной ширины этого поля;
- `scientific` — ввод/вывод чисел с плавающей точкой в формате с точкой и экспонентой;
- `showbase` — вывод перед целочисленным числом префикса, который указывает систему счисления;

- `showpoint` — вывод десятичной точки в числе с плавающей точкой;
- `showpos` — вывод знака `+` перед неотрицательными числами;
- `skipws` — пропуск ведущих пробелов при вводе;
- `unitbuf` — освобождение потока при каждой операции вывода;
- `uppercase` — вывод прописных букв в некоторых форматах, например, символ `E` в числе с плавающей точкой;
- `adjustfield` — эквивалентно `internal|left|right`;
- `basefield` — эквивалентно `dec|hex|oct`;
- `floatfield` — эквивалентно `fixed|scientific`.

Для работы с флагами в классе `ios_base` определены следующие функции:

```
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
void unsetf(fmtflags mask);
```

Эти функции выполняют следующие действия:

- `flags` — первая функция возвращает значение битовой маски; вторая функция сохраняет в битовой маске значение, заданное параметром `fmtfl`, и возвращает предыдущее значение битовой маски;
- `setf` — первая функция устанавливает биты, заданные параметром `fmtfl`, и возвращает предыдущее значение битовой маски; вторая функция сбрасывает в битовой маске флаги, установленные в параметре `mask`, затем устанавливает в битовой маске флаги, которые установлены как в параметре `fmtfl`, так и в параметре `mask`, возвращает предыдущее значение битовой маски;
- `unsetf` — сбрасывает в битовой маске флаги, установленные в параметре `mask`.

Пример установки и сброса некоторых флагов показан в листинге 40.1.

Листинг 40.1. Пример установки и сброса управляющих флагов

```
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::boolalpha);   // символьные булевы значения
  cout << false << ' ' << true << endl;  // false true
  // установить 16 с/с в качестве базовой
  cout.setf(ios_base::hex, ios_base::basefield);
  cout.setf(ios_base::showbase);     // показывать базу
  cout << 32 << endl;                // 0x20
  cout.setf(0, ios_base::showbase);  // не показывать базу
  cout << 32 << endl;                // 20

  return 0;
}
```

40.2.1.2. Состояния

В классе `ios_base` определен тип `iostate`, который является синонимом вложенного типа `T2`, зависимого от реализации класса и определяющего битовую маску исключений:

```
typedef T2 iostate;
```

В классе `ios_base` определены следующие флаги, которые отмечают состояние потока и могут быть установлены в этой битовой маске:

```
static const iostate badbit, eofbit, failbit, goodbit;
```

Эти флаги отмечают следующие состояния потока:

- `badbit` — нарушение целостности буфера потока;
- `eofbit` — обнаружен конец файла при вводе из потока;
- `failbit` — возникла ошибка при вводе данных из потока;
- `goodbit` — не установлены никакие флаги.

40.2.1.3. Режимы открытия потока

В классе `ios_base` определен тип `openmode`:

```
typedef T3 openmode;
```

Тип является синонимом вложенного типа `T3`, зависимого от реализации класса и определяющего битовую маску с режимами открытия потока. В классе `ios_base` определены следующие флаги, которые управляют режимами открытия потока и могут быть установлены в этой битовой маске:

```
static const openmode app, ate, binary, in, out, trunc;
```

Эти флаги имеют следующее назначение:

- `app` — каждая операция вывода записывает данные в конец потока;
- `ate` — открытие потока и установка индикатора позиции на его конец;
- `binary` — открытие потока в бинарном режиме;
- `in` — открытие потока для чтения данных;
- `out` — открытие потока для записи данных;
- `trunc` — открытие потока и стирание информации из файла, с которым связывается этот поток.

Использование этих флагов будет показано в дальнейшем при создании и открытии файлов.

40.2.1.4. Направления поиска

В классе `ios_base` определен тип `seekdir`:

```
typedef T4 seekdir;
```

Тип является синонимом вложенного типа `T4`, определяющего перечисление и используемого для определения направления поиска позиции в потоке. В классе `ios_base` определены следующие флаги, которые управляют направлением поиска позиции:

```
static const seekdir beg, cur, end;
```

Эти флаги имеют следующее назначение:

- `beg` — поиск позиции, начиная от начала потока;
- `cur` — поиск позиции, начиная от текущей позиции;
- `end` — поиск позиции, начиная от конца потока.

Использование этих флагов будет показано в дальнейшем при установке индикатора позиции потока.

40.2.1.5. События

В классе `ios_base` определен тип `event`:

```
typedef T5 event;
```

Тип является синонимом вложенного типа `T5`, определяющего перечисление и используемого для определения событий, могут быть переданы как аргументы *функциям обратного вызова* (callback functions). В классе `ios_base` определены следующие события:

```
static const event copyfmt_event, erase_event,
    imbue_event;
```

Эти события имеют следующее назначение:

- `copyfmt_event` — событие, возникающее при исполнении функции `copyfmt` — члене шаблона класса `basic_ios` перед копированием маски исключений потока;
- `erase_event` — событие, которое может возникнуть или в начале исполнения функции `copyfmt` — члене шаблона класса `basic_ios`, или в начале исполнения деструктора для объекта `*this`;
- `imbue_event` — событие, возникающее при исполнении функции `imbue` — члене класса `ios_base` перед выходом из этой функции.

Когда происходит одно из этих событий, вызываются функции обратного вызова, адреса которых хранятся в стеке функций обратного вызова. Вызов функций обратного вызова происходит в порядке, обратном порядку записи их адресов в стек.

Функция обратного вызова имеет тип `event_callback`, который определен следующим образом:

```
typedef void (*event_callback)(
    event ev, ios_base& ios, int idx);
```

Здесь параметры `ev` и `ios` задают соответственно событие и поток, в котором возникает это событие, а параметр `idx` задает индекс, с которым регистрируется функция обратного вызова.

Для регистрации функций обратного вызова предназначена функция `register_callback` — член класса `ios_base`, которая имеет следующий прототип:

```
void register_callback(event_callback pfn, int idx);
```

Здесь параметр `pfn` задает адрес функции обратного вызова, а параметр `idx` задает индекс, который передается функции обратного вызова как аргумент.

В листинге 40.2 приведен пример регистрации и вызова функции обратного вызова.

Листинг 40.2. Регистрация и вызов функции обратного вызова

```
#include <locale>
#include <iostream>
using namespace std;

void f(ios::event ev, ios_base& ios, int idx)
{
  cout << "event: " << ev << endl;
  cout << "index: " << idx << endl;
}

int main()
{
  cout.register_callback(f, 10);
  cout.imbue(locale(""));  // event: 1 index: 10
  cout << 12.5 << endl;    // 12,5

  return 0;
}
```

40.2.1.6. Конструктор

Класс `ios_base` имеет единственный конструктор по умолчанию:

```
ios_base();
```

Конструктор является защищенным членом этого класса. Поэтому объекты класса `ios_base` могут быть созданы только в наследуемых классах.

40.2.1.7. Оператор присваивания

В классе `ios_base` определен оператор присваивания:

```
ios_base& operator=(const ios_base& rhs);
```

Он позволяет присваивать объекты типа `ios_base`.

40.2.1.8. Класс *failer*

В классе `ios_base` определен вложенный класс `failer`, который имеет следующий интерфейс:

```
class failure: public exception
{
public:
    explicit failure(const string& what_arg);
};
```

Этот класс служит базовым классом для всех исключений, которые может выбросить метод `clear` шаблонного класса `basic_ios`. Класс `failer` наследует от базового класса `exception` метод `what`, применение которого к объекту класса `failer` будет возвращать значение `what_arg.data()`.

40.2.1.9. Класс *Init*

В классе `ios_base` определен вложенный класс:

```
class Init {};
```

Создание объекта этого класса гарантирует создание стандартных потоков ввода/вывода, независимо от того, созданы или нет эти потоки при запуске главной программы приложения.

40.2.1.10. Управление разрешением дисплея

Для управления разрешением дисплея в классе ios_base перегружены функции precision:

```
streamsize precision() const;
streamsize precision(streamsize prec);
```

Первая функция precision возвращает хранимое в потоке разрешение дисплея.

Вторая функция precision устанавливает новое разрешение дисплея, заданное параметром prec, и возвращает старое разрешение дисплея.

40.2.1.11. Управление шириной поля ввода/вывода

Для управления шириной поля ввода/вывода в классе ios_base перегружены функции width:

```
streamsize width() const;
stramsize width(streamsize wide);
```

Первая функция width возвращает хранимую в потоке ширину поля ввода/вывода.

Вторая функция width устанавливает новую ширину поля, заданную параметром wide, и возвращает старую ширину.

40.2.1.12. Управление памятью

Для управления памятью, распределяемой под расширяемые массивы потоков, в классе ios_base определены функции iword и pword:

```
long& iword(int idx);
void*& pword(int idx);
```

Функция iword распределяет память под массив, элементы которого имеют тип long, и возвращает ссылку на элемент массива с индексом, заданным параметром idx. При этом размерность созданного массива не меньше, чем значение параметра idx.

Функция pword распределяет память под массив, элементы которого имеют тип void*, и возвращает ссылку на элемент массива с

индексом, заданным параметром idx. При этом размерность созданного массива не меньше, чем значение параметра idx.

Ссылки, возвращаемые функциями iword и pword, становятся недействительными после следующего вызова этих функций соответственно, после вызова функции basic_ios::copyfmt или после разрушения потока деструктором.

Если параметр idx имеет отрицательное значение или память под массив не может быть распределена, то функции iword и pword вызывают функцию basic_ios::setstate(badbit) и возвращают недействительную ссылку.

Для получения индекса, который имеет одно и то же значение для всех потоков типа ios_base в классе ios_base, определена статическая функция xalloc:

```
static int xalloc();
```

Эта функция возвращает хранимое статическое целочисленное значение, а после этого увеличивает его на единицу.

40.2.1.13. Управление локальностью

Для работы с локальностями в классе ios_base определены функции getloc и imbue:

```
locale getloc() const;
locale imbue(const locale& loc);
```

Функция getloc возвращает хранимую в потоке локальность.

Функция imbue сохраняет в потоке локальность, заданную параметром loc, после этого устанавливает событие imbue_event и возвращает старую локальность. Пример использования функции imbue был приведен в листинге 40.2.

40.2.1.14. Синхронизация с потоками языка программирования C

Для синхронизации потоков языка программирования C++ с потоками языка программирования C в классе ios_base определена функция sync_with_stdio:

```
static bool sync_with_stdio(bool sync = true);
```

Эта функция устанавливает во флаге, управляющем синхронизацией потоков C и C++, значение, заданное параметром `sync`, и возвращает старое значение этого флага. Начальное значение флага синхронизации установлено в `true`.

Если флаг синхронизации установлен в `true`, то потоки C и C++ синхронизируют доступ к одному файлу.

Для синхронизации доступа к стандартным потокам C и C++ функция `sync_with_stdio` должна быть вызвана до выполнения любых операций над этими потоками.

40.2.2. Шаблон класса *basic_ios*

Шаблон класса определяется как `public` наследник класса `ios_base`

```
template <class E, class T = char_traits<E>>
    class basic_ios;
```

Параметр `E` шаблона задает тип символов, которые являются элементами потока, а параметр `T` определяет характеристики этих символов. Подробно характеристики символов описаны в *разд. 39.1*.

Дополнительно к информации, наследуемой от объектов типа `ios_base`, объекты класса `basic_ios<E, T>` поддерживают следующую информацию о потоке:

- символ заполнитель;
- указатель на буфер потока;
- указатель на объект типа `basic_ios<E, T>`.

В этом разделе описаны все ассоциированные с классом `basic_ios<E, T>` типы, а также открытые (`public`) и защищенные (`protected`) члены этого класса.

40.2.2.1. Ассоциированные типы

В шаблоне класса `basic_ios<E, T>` определены следующие типы, которые определяются аргументами шаблона:

```
typedef E char_type;
typedef T traits_type;
typedef typename T::int_type int_type;
typedef typename T::pos_type pos_type;
typedef typename T::off_type off_type;
```

Эти типы имеют следующее назначение:

- `char_type` — определяет тип символов, которые являются элементами потока;
- `traits_type` — определяет характеристики элементов потока;
- `int_type` — определяет целочисленный тип, который допускает преобразование к типу `char_type` и обратно без потери значения;
- `pos_type` — тип для описания индикатора позиции файла;
- `off_type` — тип для описания смещения в операциях по изменению значения индикатора позиции файла.

40.2.2.2. Конструкторы

В шаблоне класса `basic_ios<E, T>` определен открытый `(public)` конструктор:

```
explicit basic_ios(basic_streambuf<E, T> *sb);
```

Параметр `sb` этого конструктора задает буфер потока. Этот конструктор вызывает защищенную функцию-член класса `init(sb)`, которая инициализирует поток.

Также в шаблоне класса `basic_ios<E, T>` определен защищенный `(protected)` конструктор:

```
basic_ios();
```

Он создает объект этого класса без инициализации его членов.

40.2.2.3. Деструктор

В шаблоне класса `basic_ios<E, T>` определен открытый (`public`) деструктор:

```
virtual ~basic_ios();
```

Деструктор вызывает функции обратного вызова, адреса которых хранятся в стеке обратного вызова. При этом первый аргумент имеет значение erase_event. После этого деструктор разрушает объект.

40.2.2.4. Инициализация объектов

Для инициализации объектов шаблонного класса basic_ios<E, T> в шаблоне этого класса определена защищенная (protected) функция init:

```
void init(basic_streambuf<E, T> *sb);
```

Эта функция инициализирует члены шаблонного класса таким образом, что выполняются следующие условия:

- вызов функции rdbuf() возвращает указатель на буфер потока, заданный параметром sb функции init;
- вызов функции tie() возвращает пустой указатель;
- вызов функции rdstate() возвращает значение goodbit, если значение параметра sb функции init не равно нулю, в противном случае — значение badbit;
- вызов функции exceptions() возвращает значение goodbit;
- вызов функции flags(), наследуемой от класса ios_base, возвращает значение skipws|dec;
- вызов функции width(), наследуемой от класса ios_base, возвращает ноль;
- вызов функции precision(), наследуемой от класса ios_base, возвращает значение 6;
- вызов функции fill() возвращает код пробела;
- вызов функции getloc() возвращает значение locale::classic();
- вызов функции iword(), наследуемой от класса ios_base, возвращает ноль для любого значения аргумента;
- вызов функции pword(), наследуемой от класса ios_base, возвращает пустой указатель для любого значения аргумента.

40.2.2.5. Управление состоянием потока

Для управления состоянием потока в шаблоне класса `basic_ios<E, T>` определены следующие функции:

```
iostate rdstate() const;
void clear(iostate state = goodbit);
void setstate(iostate state);
bool good() const;
bool eof() const;
bool fail() const;
bool bad() const;
```

Эти функции выполняют следующие действия:

- `rdstate` — возвращает битовую маску типа `iostate`, хранимую в потоке;
- `clear` — устанавливает в битовой маске типа `iostate` потока значение `state|(rdbuf() != 0?goodbit:badbit)`, после этого, если значение `state&exceptions()` не равно нулю, выбрасывает исключение типа `failure`;
- `setstate` — вызывает функцию `clear(state|rdstate())`;
- `good` — возвращает значение `true`, если выполняется условие `rdstate()==goodbit`, в противном случае — `false`;
- `eof` — возвращает значение `true`, если в битовой маске типа `iostate` установлен бит `eofbit`, в противном случае — `false`;
- `fail` — возвращает значение `true`, если в битовой маске типа `iostate` установлен бит `failbit`, в противном случае — `false`;
- `bad` — возвращает значение `true`, если в битовой маске типа `iostate` установлен бит `badbit`, в противном случае — `false`.

40.2.2.6. Управление исключениями

Для управления состояниями потока, которые вызывают исключения, в шаблоне класса `basic_ios<E, T>` определены следующие функции:

```
iostate exceptions() const;
iostate exceptions(iostate except);
```

Первая функция exceptions возвращает битовую маску исключений, хранимую в потоке.

Вторая функция exceptions устанавливает в потоке битовую маску исключений, заданную параметром except. Если в новой маске исключений установлен вызывающий исключение бит, то функция выбрасывает исключение типа failure, в противном случае возвращает старую битовую маску исключений.

40.2.2.7. Копирование состояния потока

Для копирования состояний потока в шаблоне класса basic_ios<E, T> определена функция copyfmt:

```
basic_ios& copyfmt(const basic_ios& rhs);
```

Эта функция выполняет следующую последовательность действий:

- устанавливает событие erase_event;
- копирует в текущий поток из потока, заданного параметром rhs, следующую информацию:
  - символ заполнитель;
  - указатель на объект типа basic_ios<E, T>;
  - информацию, хранящуюся в членах-данных, наследуемых от базового класса ios_base;
- устанавливает событие copyfmt_event;
- если в маске исключений установлены биты, то функция вызывает функцию clear(rdstate()), в противном случае возвращает ссылку на текущий поток.

40.2.2.8. Перегруженные операторы

В шаблоне класса basic_ios<E, T> перегружены следующие операторы:

```
bool operator!() const;
operator void*() const;
```

Оператор ! возвращает значение fail().

Оператор преобразования к типу `void*` возвращает пустой указатель, только если выполняется условие `fail()==true`.

40.2.2.9. Установка символа заполнителя

Для установки символа заполнителя в шаблоне класса `basic_ios<E, T>` перегружены следующие функции:

```
char_type fill() const;
char_type fill(char_type ch);
```

Первая функция `fill` возвращает установленный в потоке символ заполнитель.

Вторая функция `fill` устанавливает в потоке символ заполнитель, заданный параметром `ch`, и возвращает старый символ заполнитель.

40.2.2.10. Управление буфером

Для работы с указателем на буфер потока, который хранится в потоке, в шаблоне класса `basic_ios<E, T>` перегружены следующие функции:

```
basic_streambuf<E, T> *rdbuf() const;
basic_streambuf<E, T> *rdbuf(basic_streambuf<E, T> *sb);
```

Первая функция `rdbuf` возвращает хранимый в потоке указатель на буфер потока.

Вторая функция `rdbuf` сохраняет в потоке указатель на буфер, заданный параметром `sb`, и возвращает значение старого указателя на буфер потока.

40.2.2.11. Управление потоком вывода

Для работы с указателем на поток вывода, который хранится в потоке, шаблоне класса `basic_ios<E, T>` перегружены следующие функции:

```
basic_ostream<E, T> *tie() const;
basic_ostream<E, T> *tie(basic_ostream<E, T> *str);
```

Первая функция `tie` возвращает хранимый в потоке указатель на поток вывода.

Вторая функция tie сохраняет в потоке указатель на поток вывода, заданный параметром str, и возвращает значение старого указателя на поток вывода.

40.2.2.12. Управление локальностью

Для определения локальности, хранимой в потоке, в шаблоне класса basic_ios<E, T> определена функция imbue:

```
locale imbue(const locale& loc);
```

Если указатель на буфер потока не пустой, то функция imbue вызывает функцию rdbuf()->pubimbue(loc). В любом случае функция возвращает значение ios_base::imbue(loc).

Кроме того, для работы с локальностью в шаблоне класса basic_ios<E, T> определены следующие функции:

```
char_type widen(char ch);
E widen(char ch);
char narrow(char_type ch, char dflt);
char narrow(E ch, char dflt);
```

которые здесь рассматриваться не будут.

40.2.3. Шаблон класса *fops*

Шаблон класса fops имеет следующий интерфейс:

```
template <class St> class fops
{
public:
  // конструкторы
  fpos(streamoff off);
  explicit fpos(St state);
  // функции для работы с состоянием индикатора позиции файла
  St state() const;
  void state(St state);
  // перегруженные операторы
  operator streamoff() const;
  fpos& operator-=(streamoff off);
  fpos& operator+=(streamoff off);
```

```
  streamoff operator-(const fpos& rhs) const;
  fpos operator-(streamoff off) const;
  fpos operator+(streamoff off) const;
  bool operator==(const fpos& rhs) const;
  bool operator!=(const fpos& rhs) const;
};
```

Этот шаблон класса предназначен для определения индикаторов позиции файла. Параметр `St` шаблона класса `fops` задает тип объекта, который хранит состояние строки длинных (multibyte) символов, которая хранится в буфере потока.

Объекты класса `fops<St>` содержат, по крайней мере, два объекта: объект типа `streamoff`, который задает смещение в байтах, и объект типа `St`.

Конструкторы шаблонного класса `fops<St>` выполняют следующие действия:

- первый конструктор устанавливает начальное смещение равным значению, заданному параметром `off`, и инициализирует объект-член типа `St`; начальное смещение равное `-1` считается неправильным;
- второй конструктор устанавливает начальное значение равным нулю и инициализирует объект-член типа `St` значением, заданным параметром `state`.

Для работы с состоянием строки длинных (multibyte) символов предназначены функции `state`, которые выполняют следующие действия:

- первая функция `state` возвращает состояние строки длинных символов, хранящееся в индикаторе позиции;
- вторая функция `state` устанавливает в индикаторе позиции состояние строки длинных символов, заданное параметром `state`.

Операторы, перегруженные в шаблоне класса `fops<St>`, имеют следующее назначение:

- `streamoff` — возвращает смещение, хранимое в индикаторе позиции;

- `-=` — вычитает из смещения, хранимого в текущем объекте, значение параметра `off`;
- `+=` — прибавляет к смещению, хранимому в текущем объекте, значение параметра `off`;
- `-` — первый оператор возвращает разность между смещениями, хранящимися в текущем объекте и в объекте, заданном параметром `rhs`; второй оператор возвращает объект `fops(*this)-=off`;
- `+` — возвращает объект `fops(*this)+=off`;
- `==` — возвращает значение `true`, если текущий объект равен объекту, заданному параметром `rhs`, иначе — значение `false`;
- `!=` — возвращает значение `true`, если текущий объект не равен объекту, заданному параметром `rhs`, иначе — значение `false`.

Для специализации `fpos<mbstate_t>` шаблона класса объявлены типы синонимы `streampos` и `wstreampos`, которые предназначены для работы с потоками типов `ios` и `wios` соответственно.

Здесь `mbstate_t` — синоним некоторого типа, который используется для представления состояния строки длинных символов.

40.2.4. Манипуляторы

Манипулятор — это функция, которая изменяет поток. В заголовочном файле ios определены следующие манипуляторы, которые устанавливают флаги, хранимые в потоке, заданном параметром `str`:

- `ios_base& boolalpha(ios_base& str);` — флаг `boolalpha`
- `ios_base& dec(ios_base& str);` — флаг `dec`
- `ios_base& fixed(ios_base& str);` — флаг `fixed`
- `ios_base& hex(ios_base& str);` — флаг `hex`
- `ios_base& internal(ios_base& str);` — флаг `internal`
- `ios_base& left(ios_base& str);` — флаг `left`
- `ios_base& oct(ios_base& str);` — флаг `oct`
- `ios_base& right(ios_base& str);` — флаг `right`

- `ios_base& scientific(ios_base& str);` — флаг `scientific`
- `ios_base& showbase(ios_base& str);` — флаг `showbase`
- `ios_base& showpoint(ios_base& str);` — флаг `showpoint`
- `ios_base& showpos(ios_base& str);` — флаг `showpos`
- `ios_base& skipws(ios_base& str);` — флаг `skipws`
- `ios_base& unitbuf(ios_base& str);` — флаг `unitbuf`
- `ios_base& uppercase(ios_base& str);` — флаг `uppercase`

Кроме того, в заголовочном файле ios определены следующие манипуляторы, которые сбрасывают флаги, хранимые в потоке, заданном параметром `str`:

- `ios_base& noboolalpha(ios_base& str);` — флаг `boolalpha`
- `ios_base& noshowbase(ios_base& str);` — флаг `showbase`
- `ios_base& noshowpoint(ios_base& str);` — флаг `showpoint`
- `ios_base& noskipws(ios_base& str);` — флаг `skipws`
- `ios_base& nounitbuf(ios_base& str);` — флаг `unitbuf`
- `ios_base& nouppercase(ios_base& str);` — флаг `uppercase`

В листинге 40.3 показан пример изменения состояния флагов потока при помощи манипуляторов. Сравнение с листингом 40.1 показывает, что состояние флага удобнее изменять манипулятором, чем функциями `setf` и `unsetf`, рассмотренными в *разд. 40.2.1.3*.

Листинг 40.3. Пример использования манипуляторов

```
#include <iostream>
using namespace std;

int main()
{
  cout << boolalpha << false << endl;      // false
  cout << hex << showbase << 32 << endl;   // 0x20
  cout << noshowbase << 32 << endl;        // 20

  return 0;
}
```

40.3. Базовые потоки вывода <ostream>

В заголовочном файле ostream определен шаблон класса `basic_ostream<E, T>`, который определяет базовые классы для потоков вывода, а также типы — синонимы специализаций этого шаблона:

```
namespace std
{
  template<class E, class T = char_traits<E>>
      class basic_ostream;
  typedef basic_ostream<char, char_traits<char>>
      ostream;
  typedef basic_ostream<wchar_t, char_traits<wchar_t>>
      wostream;
}
```

Кроме того, в заголовочном файле ostream перегружены шаблоны оператора `<<` для форматированного вывода символов и строк языка программирования C, а также определены шаблоны манипуляторов `endl`, `ends` и `flush`, работающих с выходными потоками.

40.3.1. Шаблон класса *basic_ostream*

Шаблон класса определяется как `virtal` и `public` наследник шаблона класса `basic_ios<E, T>`:

```
template<class E, class T = char_traits<E>>
    class basic_ostream;
```

Этот шаблон предназначен для создания шаблонных классов потоков вывода.

Параметр `E` шаблона задает тип символов, которые являются элементами потока, а параметр `T` определяет характеристики этих символов. Подробно характеристики символов описаны в *разд. 39.1*.

В этом разделе описаны все ассоциированные с классом `basic_ostream<E, T>` типы, а также открытые (public) члены этого шаблона класса.

40.3.1.1. Ассоциированные типы

В шаблоне класса `basic_ostream<E, T>` определены следующие типы, которые определяются аргументами шаблона:

```
typedef typename basic_ios<E, T>::char_type char_type;
typedef typename basic_ios<E, T>::traits_type traits_type;
typedef typename basic_ios<E, T>::int_type int_type;
typedef typename basic_ios<E, T>::pos_type pos_type;
typedef typename basic_ios<E, T>::off_type off_type;
```

Отсюда видно, что эти типы являются синонимами соответствующих типов, определенных в шаблоне класса `basic_ios<E, T>` и описанных в *разд. 40.2.2.1*.

40.3.1.2. Конструктор

В шаблоне класса `basic_ostream<E, T>` определен конструктор:

```
explicit basic_ostream(basic_streambuf<E, T> *sb);
```

Параметр `sb` этого конструктора задает буфер потока вывода. Этот конструктор вызывает защищенную функцию `init(sb)`, которая наследуется от класса `basic_ios<E, T>` и инициализирует поток.

40.3.1.3. Деструктор

В шаблоне класса `basic_ostream<E, T>` определен деструктор, который разрушает выходной поток, не выполняя при этом никаких операций над буфером потока:

```
virtual ~ostream();,
```

40.3.1.4. Перегруженный оператор вывода <<

Вывод данных из памяти в поток называется форматированным, если при выводе представление данных преобразуется из битово-

го в символьное. Для форматированного вывода в поток данных разных типов в шаблоне класса `basic_ostream<E, T>` перегружены следующие операторы:

```
basic_ostream& operator<<(bool n);
basic_ostream& operator<<(short n);
basic_ostream& operator<<(unsigned short n);
basic_ostream& operator<<(int n);
basic_ostream& operator<<(unsigned int n);
basic_ostream& operator<<(long n);
basic_ostream& operator<<(unsigned long n);
basic_ostream& operator<<(float n);
basic_ostream& operator<<(double n);
basic_ostream& operator<<(long double n);
basic_ostream& operator<<(const void *n);
```

Все эти операторы уже неоднократно использовались для вывода в стандартный поток вывода `cout` различных данных.

Также оператор `<<` перегружен для вывода в поток содержимого буфера:

```
basic_ostream& operator<<(basic_streambuf<E, T> *sb);
```

Данные из буфера выводятся в поток до тех пор, пока не выполнится одно из следующих условий:

- в буфере встретился символ конца файла;
- запись символа в поток завершилась неудачно;
- при извлечении символа из буфера произошло исключение.

Если в поток не выведено ни одного символа или при извлечении символа из буфера произошло исключение, то оператор вывода вызывает функцию `setstate(failbit)`, наследуемую от шаблонного класса `basic_ios<E, T>`. В последнем случае также повторно выбрасывает обрабатываемое исключение.

Если указатель на буфер пустой, то оператор вызывает функцию `setstate(badbit)`.

Кроме того, в шаблоне класса `basic_ostream<E, T>` перегружены следующие операторы `<<`:

```
basic_ostream& operator<<(ios_base& (*pf)(ios_base&));
basic_ostream& operator<<(
    basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_ostream& operator<<(
    basic_ostream& (*pf)(basic_ostream&));
```

Они обеспечивают правильную обработку манипуляторов потоков типа `ios_base`, `basic_ios<E, T>` и `basic_ostream<E, T>` соответственно.

Все перегруженные операторы `<<` возвращают значение `*this`.

40.3.1.5. Вывод символа

Для неформатированного вывода в поток символа (байтов) в шаблоне класса `basic_ostream<E, T>` определена функция `put`:

```
basic_ostream& put(char_type c);
```

Пример использования функции `put` приведен в листинге 40.5.

40.3.1.6. Вывод блока памяти

Для неформатированного вывода в поток блока памяти в шаблоне класса `basic_ostream<E, T>` определена функция `write`:

```
basic_ostream& write(char_type *s, streamsize n);
```

Здесь параметр `s` содержит адрес блока памяти, который будет выводиться в поток, а параметр `n` задает длину блока, т. е. количество выводимых в поток элементов. Заметим, что при выводе данные не форматируются, т. е. в поток записывается копия блока памяти.

Функция `write` возвращает значение `*this`.

Пример использования функции `write` приведен в листинге 40.13, в котором показано, как создавать бинарные файлы.

40.3.1.7. Освобождение буфера потока

Для освобождения буфера, связанного с потоком вывода, в шаблоне класса `basic_ostream<E, T>` определена функция `flush`:

```
basic_ostream& flush();
```

Освобождение буфера заключается в выводе данных из буфера, связанного с потоком вывода в этот поток.

Функция `flush` работает следующим образом. Сначала вызывается функция `rdbuf()`. Если эта функция вернула не пустой указатель, то вызывается функция `rdbuf()->pubsync()`. Если функция `rdbuf` вернула значение `-1`, то вызывается функция `setstate(badbit)`. Функция возвращает значение `*this`.

В листинге 40.4 показан пример освобождения буфера потока вывода при выводе данных в файл.

40.3.1.8. Управление индикатором позиции

Для управления индикатором позиции в шаблоне класса `basic_ostream<E, T>` определены следующие функции:

```
pos_type tellp();
basic_ostream& seekp(pos_type pos);
basic_ostream& seekp(off_type off, ios_base::seek_dir dir);
```

Функция `tellp` читает, а функции `seekp` устанавливают значение индикатора позиции для дальнейшей записи данных в поток функциями неформатного вывода.

Функция `tellp` работает следующим образом. Сначала вызывается функция `fail()`. Если эта функция вернула значение `false`, то функция `tellp` возвращает значение `rdbuf()->pubseekoff(0, cur, out)`, иначе — значение `pos_type(-1)`.

Функции `seekp` работают следующим образом. Сначала вызывается функция `fail()`. Если эта функция вернула значение `false`, то первая функция `seekp` вызывает функцию `rdbuf()->pubseekpos(pos)`, а вторая — функцию `rdbuf()->pubseekoff(off, dir)`. Обе функции `seekp` возвращают значение `*this`.

Пример работы функций `tellp` и `seekp` приведен в листинге 40.4.

Листинг 40.4. Пример освобождения буфера потока и работы с индикатором позиции файла

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main()
{
  ostream::pos_type i;
  ofstream out("test.txt");

  out << "---";
  i = out.tellp();  // запоминаем индикатор позиции
  out << ' ' << "---";
  out.flush();      // сбрасываем данные в файл

  cout << "Press any key to continue" << endl;
  cin.get();        // можно посмотреть файл

  out.seekp(i);     // устанавливаем индикатор позиции
  out.put('+');
  out.close();

  return 0;
}
```

40.3.1.9. Класс *sentry*

В шаблоне класса `basic_ostream<E, T>` определен вложенный класс `sentry`:

```
class sentry
{
public:
  explicit sentry(basic_ostream<E, T>& os);
  operator bool() const;
};
```

Класс `sentry` служит для объявления объектов, создание которых подготавливает поток, заданный параметром `os` конструктора, для форматированного и неформатированного вывода. После завершения работы конструктора вызов оператора преобразования типа `bool` возвращает значение `true`, если значение `os.good()` также равно `true`, иначе — значение `false`.

40.3.2. Манипуляторы

В заголовочном файле ostream определены следующие шаблоны манипуляторов шаблонного класса `basic_ostream<E, T>`:

```
template class<E, T>
  basic_ostream<E, T>& endl(basic_ostream<E, T>& os);
template class<E, T>
  basic_ostream<E, T>& ends(basic_ostream<E, T>& os);
template class<E, T>
  basic_ostream<E, T>& flush(basic_ostream<E, T>& os);
```

Манипулятор `endl` выводит в поток, заданный параметром `os`, символ перехода на новую строку `\n`, а затем освобождает буфер потока, вызывая функцию `os.flush()`.

Манипулятор `ends` выводит в поток, заданный параметром `os`, пустой символ `\0`.

Манипулятор `flush` освобождает буфер потока, вызывая функцию `os.flush()`.

Все манипуляторы возвращают поток, заданный параметром `os`.

40.3.3. Перегруженный шаблон оператора вывода <<

В заголовочном файле ostream перегружены следующие шаблоны оператора `<<`:

```
template<class E, class T> basic_ostream<E, T>&
  operator<<(basic_ostream<E, T>& os, E c);
template<class E, class T> basic_ostream<E, T>&
  operator<<(basic_ostream<E, T>& os, char c);
template<class T> basic_ostream<char, T>& operator<<
  (basic_ostream<char, T>& os, char c);
template<class T> basic_ostream<char, T>& operator<<(
  basic_ostream<char, T>& os, signed char c);
template<class T> basic_ostream<char, T>& operator<<(
  basic_ostream<char, T>& os, unsigned char c);
```

Соответствующие шаблонные операторы выводят в поток, заданный параметром `os`, символ, заданный параметром `c`.

Также в заголовочном файле ostream перегружены следующие шаблоны оператора <<:

```
template<class E, class T> basic_ostream<E, T>&
  operator<<(basic_ostream<E, T>& os, const E *s);
template<class E, class T> basic_ostream<E, T>&
  operator<<(basic_ostream<E, T>& os, const char *s);
template<class T> basic_ostream<char, T>& operator<<
  (basic_ostream<char, T>& os, const char *s);
template<class T> basic_ostream<char, T>& operator<<(
  basic_ostream<char, T>& os, const signed char *s);
template<class T> basic_ostream<char, T>& operator<<(
  basic_ostream<char, T>& os, const unsigned char *s);
```

Соответствующие шаблонные операторы выводят в поток, заданный параметром `os`, строку языка программирования C, заданную параметром `s`.

Если ширина поля вывода потока больше чем длина выводимых данных, то свободные позиции заполняются символом заполнителем, который задан в потоке. После завершения вывода каждый шаблонный оператор << устанавливает ширину поля вывода равной нулю.

40.4. Базовые потоки ввода и ввода/вывода <istream>

В заголовочном файле istream определены следующие шаблоны классов и типы — синонимы специализаций этих шаблонов:

```
namespace std
{
  template<class E, class T = char_traits<E>>
    class basic_istream;
  typedef basic_istream<char, char_traits<char>> istream;
  typedef basic_istream<wchar_t, char_traits<wchar_t>>
    wistream;
  template<class E, class T = char_traits<E>>
    class basic_iostream;
```

```
    typedef basic_iostream<char, char_traits<char>> iostream;
    typedef basic_iostream<wchar_t, char_traits<wchar_t>>
      wiostream;
  }
```

Шаблон класса `basic_istream<E, T>` предназначен для определения базовых потоков ввода, а типы `istream` и `wistream` являются синонимами специализаций этого шаблона класса и предназначены для определения базовых потоков ввода, работающих с символами типа `char` и `wchar_t` соответственно.

Шаблон класса `basic_iostream<E, T>` предназначен для определения базовых потоков ввода/вывода, а типы `iostream` и `wiostream` являются синонимами специализаций этого шаблона класса и предназначены для определения базовых потоков ввода, работающих с символами типа `char` и `wchar_t` соответственно.

Кроме того, в заголовочном файле istream перегружены шаблоны оператора `>>` для форматированного ввода символов и слов, а также определен шаблон манипулятора `ws`, работающего с входными потоками.

40.4.1. Шаблон класса *basic_istream*

Шаблон класса определяется как `virtal` и `public` наследник шаблона класса `basic_ios<E, T>`:

```
template <class E, class T = char_traits<E>>
  class basic_istream;
```

Этот шаблон предназначен для создания шаблонных классов потоков ввода.

Параметр `E` шаблона задает тип символов, которые являются элементами потока, а параметр `T` определяет характеристики этих символов. Подробно характеристики символов описаны в *разд. 39.1*.

В этом разделе описаны все ассоциированные с классом `basic_istream<E, T>` типы, а также открытые (`public`) члены этого шаблона класса.

40.4.1.1. Ассоциированные типы

В шаблоне класса `basic_istream<E, T>` определены следующие типы, которые определяются аргументами шаблона:

```
typedef typename basic_ios<E, T>::char_type char_type;
typedef typename basic_ios<E, T>::traits_type traits_type;
typedef typename basic_ios<E, T>::int_type int_type;
typedef typename basic_ios<E, T>::pos_type pos_type;
typedef typename basic_ios<E, T>::off_type off_type;
```

Отсюда видно, что эти типы являются синонимами соответствующих типов, определенных в шаблоне класса `basic_ios<E, T>` и описанных в *разд. 40.2.2.1*.

40.4.1.2. Конструктор

В шаблоне класса `basic_istream<E, T>` определен конструктор:

```
explicit basic_istream(basic_streambuf<E, T> *sb);
```

Параметр `sb` этого конструктора задает буфер потока ввода. Этот конструктор вызывает защищенную функцию `init(sb)`, которая наследуется от класса `basic_ios<E, T>` и инициализирует поток.

40.4.1.3. Деструктор

В шаблоне класса `basic_istream<E, T>` определен деструктор, который разрушает входной поток, не выполняя при этом никаких операций над буфером потока:

```
virtual ~istream();
```

40.4.1.4. Перегруженный оператор ввода >>

Ввод данных из потока в память называется форматированным, если при вводе представление данных преобразуется из символьного в битовое. Для форматированного ввода из потока данных разных типов в шаблоне класса `basic_istream<E, T>` перегружены следующие операторы `>>`:

```
basic_istream& operator>>(bool& n);
basic_istream& operator>>(short& n);
basic_istream& operator>>(unsigned short& n);
```

```
basic_istream& operator>>(int& n);
basic_istream& operator>>(unsigned int& n);
basic_istream& operator>>(long& n);
basic_istream& operator>>(unsigned long& n);
basic_istream& operator>>(void *& n);
basic_istream& operator>>(float& n);
basic_istream& operator>>(double& n);
basic_istream& operator>>(long double& n);
```

Все эти операторы уже неоднократно использовались для ввода различных данных из стандартного потока ввода `cin`.

Также оператор `>>` перегружен для ввода данных из потока в буфер:

```
basic_istream& operator>>(basic_streambuf<E, T> *sb);
```

Данные в буфер вводятся в поток до тех пор, пока не выполнится одно из следующих условий:

- в потоке встретился символ конца файла;
- запись символа в буфер завершилась неудачно;
- при записи символа в буфер произошло исключение.

Если из потока не введено ни одного символа, то оператор ввода вызывает функцию `setstate(failbit)`, наследуемую от шаблонного класса `basic_ios<E, T>`.

Если указатель на буфер пустой, то оператор вызывает функцию `setstate(failbit)`.

Кроме того, в шаблоне класса `basic_istream<E, T>` перегружены следующие операторы `>>`:

```
basic_istream& operator>>(ios_base& (*pf)(ios_base&));
basic_istream& operator>>(
      basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_istream& operator>>(
    basic_istream& (*pf)(basic_istream&));
```

Эти операторы обеспечивают правильную обработку манипуляторов потоков типа `ios_base`, `basic_ios<E, T>` и `basic_istream<E, T>` соответственно.

Все перегруженные операторы `>>` возвращают значение `*this`.

40.4.1.5. Ввод символов

Для неформатированного ввода из потока символа (байта) в шаблоне класса `basic_ostream<E, T>` перегружены следующие функции `get`:

```
int_type get();
basic_istream& get(char_type& c);
```

Первая функция `get` возвращает введенный из потока символ. Если символы в потоке закончились, то функция возвращает значение `traits_type::eof()`.

Вторая функция `get` записывает введенный из потока символ в параметр `c`. Если символы в потоке закончились, то в параметр `c` записывается значение `traits_type::eof()`. После этого функция возвращает значение `*this`.

Для неформатированного ввода из потока последовательности символов (байтов) в шаблоне `basic_ostream<E, T>` перегружены следующие функции `get`:

```
basic_istream& get(char_type *s, streamsize n);
basic_istream& get(char_type *s, streamsize n,
    char_type delim);
```

Обе эти перегруженные функции `get` вводят символы из текущего потока и записывают их последовательно в массив, адрес которого задан параметром `s`.

При этом первая функция `get` вводит из потока символы до тех пор, пока не выполнится одно из следующих условий:

- встретился конец файла;
- встретился символ перехода на новую строку `\n`;
- введено `n-1` символов.

Вторая функция `get` работает аналогично первой, но символ, отмечающий конец ввода, задается параметром `delim`.

Обе функции оставляют в потоке символ, отмечающий конец ввода, и после встречи этого символа записывают в конец массива символ `char_type()`, т. е. конец строки.

Каждая из этих функций get возвращает значение *this.

Для неформатированного ввода из потока последовательности символов (байтов) в шаблоне basic_ostream<E, T> также перегружены следующие функции get:

```
basic_istream& get(basic_streambuf<char_type, T> *sb);
basic_istream& get(basic_streambuf<E, T> *sb,
    char_type delim);
```

Обе эти перегруженные функции get вводят символы из текущего потока и записывают их последовательно в буфер, адрес которого задан параметром sb.

При этом первая функция get вводит из потока символы до тех пор, пока не выполнится одно из следующих условий:

- встретился конец файла;
- встретился символ перехода на новую строку \n;
- ввод символа в буфер закончился неудачей;
- при вводе символа в буфер произошло исключение.

Вторая функция get работает аналогично первой, но символ, отмечающий конец ввода, задается параметром delim.

Обе функции оставляют в потоке символ, отмечающий конец ввода, или символ, запись которого в буфер завершилась неудачей или вызвала исключение.

Каждая из этих функций get возвращает значение *this.

Все перегруженные функции get вызывают функцию setstate(failbit), если не было введено ни одного символа.

Пример использования функций put и get приведен в листинге 40.5.

Листинг 40.5. Пример использования функций put и get

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main()
{
  fstream demo("test.txt");
  char buf[80];

  demo << "aaa" << endl;
  demo << "bbb";
  demo.put('x');

  demo.seekg(0);       // устанавливаем начало файла
  demo.get(buf, 80);
  cout << buf << (char)demo.get();  // aaa'\n'

  demo.get(buf, 80, 'x');
  demo.get(buf[0]);
  cout << buf << endl;              // xbb'\n'

  demo.close();

  return 0;
}
```

40.4.1.6. Ввод строки

Для неформатированного ввода из потока последовательности символов (байт) в шаблоне `basic_ostream<E, T>` перегружены следующие функции `getline`:

```
basic_istream& getline(char_type *s, streamsize n);
basic_istream& getline(char_type *s, streamsize n,
    char_type delim);
```

Обе перегруженные функции `getline` вводят символы из текущего потока и записывают их последовательно в массив, адрес которого задан параметром `s`.

При этом первая функция `getline` вводит из потока символы до тех пор, пока не выполнится одно из следующих условий:

- встретился конец файла;
- встретился символ перехода на новую строку `\n`;
- введено `n-1` символов.

Вторая функция getline работает аналогично первой, но символ, отмечающий конец ввода, задается параметром delim.

Обе функции удаляют из потока символ, отмечающий конец ввода, и после встречи этого символа записывают в конец массива символ char_type(), т. е. конец строки.

Каждая из функций getline возвращает значение *this.

Пример использования функции getline приведен в листинге 40.6.

Листинг 40.6. Пример использования функции getline

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  fstream demo("test.txt");
  char buf[80];

  demo << "aaa aaa" << endl;
  demo << "bbb bbb";
  demo.put('x');

  demo.seekg(0);  // устанавливаем начало файла
  demo.getline(buf, 80);
  cout << buf << endl;  // aaa aaa
  demo.getline(buf, 80, 'x');
  cout << buf << endl;  // bbb bbb

  demo.close();

  return 0;
}
```

40.4.1.7. Просмотр символа в потоке

В шаблоне класса basic_ostream<E, T> определена функция peek, которая читает символ из буфера потока, но не удаляет его:

```
int_type peek();
```

Функция возвращает прочитанный элемент буфера. Если в буфере потока нет элементов, то функция `peek` возвращает значение `traits_type::eof()`.

В листинге 40.7 приведен пример использования функции `peek`.

Листинг 40.7. Пример использования функции `peek`

```
#include <iostream>
using namespace std;

int main()
{
  char c;
  int n;

  cout << "Press a key: ";
  c = cin.peek();

  if (c >= '0' && c <= '9')
  {
    cin >> n;
    cout << "It is a digit: " << n << endl;
  }
  else
  {
    cin >> c;
    cout << "It is not a digit: " << c << endl;
  }
  return 0;
}
```

40.4.1.8. Возврат символа в буфер потока ввода

Для возврата в буфер потока символа в шаблоне класса `basic_ostream<E, T>` определены функции `unget` и `putback`:

```
  basic_istream& unget();
  basic_istream& putback(char_type c);
```

Функция `unget` возвращает в поток последний прочитанный символ посредством вызова функции `rdbuf()->sungetc()`. Если вызов

функции rdbuf() возвращает пустой указатель или вызов функции sungetc() возвращает значение traits_type::eof(), то функция unget вызывает функцию setstate(badbit).

Функция putback возвращает в поток символ, заданный параметром c, посредством вызова функции rdbuf()->sputbackc(). Если вызов функции rdbuf() возвращает пустой указатель или вызов функции sputbackc() возвращает значение traits_type::eof(), то функция unget вызывает функцию setstate(badbit).

В любом случае функции unget и putback возвращают значение *this.

В листинге 40.8 приведен пример использования функций unget и putback.

Листинг 40.8. Пример использования функций unget и putback

```
#include <iostream>
using namespace std;

int main()
{
  char c;

  cout << "Press ane key: ";
  cin.get(c);
  cout << c << endl;

  cin.unget();
  cin.get(c);
  cout << c << endl;  // печатает введенный символ

  cin.putback('x');
  cin.get(c);
  cout << c << endl;  // печатает: x

  return 0;
}
```

40.4.1.9. Ввод блока памяти

Для неформатированного ввода из потока последовательности байтов в память в шаблоне класса `basic_istream<E, T>` определены функции `read` и `readsome`:

```
basic_istream& read(char_type *s, streamsize n);
streamsize readsome(char_type *s, streamsize n);
```

Функция `read` читает данные из текущего потока и последовательно записывает их в массив, адрес которого задан параметром `s`. Параметр `n` задает количество элементов, которые будут прочитаны из потока. Заметим, что при вводе данные не форматируются, т. е. в блок памяти записываются копии байтов, прочитанных из потока.

Если при вводе из потока встретился конец файла, то ввод заканчивается и функция вызывает функцию `setstate(failbit)`.

Функция `read` возвращает значение `*this`.

Функция `readsome` работает следующим образом. Сначала она определяет, сколько символов (байтов) хранится в буфере потока, а затем читает эти данные из буфера потока в массив, адрес которого задан параметром `s`. Количество прочитанных элементов равно наименьшему из значений, заданных параметром `n` и равным количеству элементов в буфере. Как и в случае с функцией `read`, данные при вводе не форматируются, т. е. в блок памяти записываются копии байтов, прочитанных из потока.

Если при определении количества элементов в буфере потока функция `rdbuf()->in_avail()` вернет отрицательное значение, то функция `readsome` вызывает функцию `setstate(eofbit)`.

В любом случае функция `readsome` возвращает количество прочитанных элементов.

40.4.1.10. Игнорирование символов

В шаблоне класса `basic_istream<E, T>` определена функция:

```
basic_istream& ignore(streamsize n = 1,
    int_type delim = traits_type::eof());
```

Она удаляет из текущего потока символы до тех пор, пока не выполнится одно из следующих условий:

- из потока удалено `n` символов;
- в потоке встретился символ, заданный параметром `delim`, который также удаляется.

Функция `ignore` возвращает значение `*this`.

40.4.1.11. Определение количества прочитанных символов

В шаблоне класса `basic_istream<E, T>` определена функция `gcount`, которая возвращает количество символов, прочитанных последней функцией неформатного ввода:

```
streamsize gcount() const;
```

Пример использования функции `gcount` приведен в листинге 40.9.

Листинг 40.9. Пример использования функции `gcount`

```
#include <iostream>
using namespace std;

int main()
{
  char line[80];

  cout << "Input a string: ";
  cin.getline(line, 80);
  cout << "String length = " << cin.gcount() - 1 << endl;

  return 0;
}
```

40.4.1.12. Управление индикатором позиции

Для управления индикатором позиции в шаблоне класса `basic_istream<E, T>` определены следующие функции:

```
pos_type tellg();
basic_istream& seekg(pos_type pos);
basic_istream& seekg(off_type off, ios_base::seek_dir dir);
```

Функция `tellg` читает, а функции `seekg` устанавливают значение индикатора позиции потока для дальнейшего чтения данных из потока функциями неформатного ввода.

Функция `tellg` работает следующим образом. Сначала вызывается функция `fail()`. Если эта функция вернула значение `false`, то функция `tellg` возвращает значение `rdbuf()-> pubseekoff(0, cur, in)`, иначе — значение `pos_type(-1)`.

Функции `seekg` работают следующим образом. Сначала вызывается функция `fail()`. Если эта функция вернула значение `false`, то первая функция `seekg` вызывает функцию `rdbuf()->pubseekpos(pos)`, а вторая — функцию `rdbuf()-> pubseekoff(off, dir)`. Обе функции `seekp` возвращают значение `*this`.

Пример работы функций `tellg` и `seekg` приведен в листинге 40.10.

Листинг 40.10. Пример работы функций `tellg` и `seekg`

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  char c;
  fstream::pos_type i;
  fstream demo("test.txt");

  demo << "---";
  i = demo.tellg();   // запоминаем индикатор позиции
  demo << 'x' << "---" << flush;
  demo.seekg(i);      // устанавливаем индикатор позиции
  c = demo.get();
  cout << c << endl;  // печатает: x
  return 0;
}
```

40.4.1.13. Синхронизация потока с файлом

В общем случае под *синхронизацией потока с файлом* понимается приведение файла в такое состояние, которое бы он имел при отсутствии буферизации ввода/вывода.

Для синхронизации потока ввода в шаблоне класса `basic_istream<E, T>` определена функция:

```
int sync();
```

Функция `sync` работает аналогично функции `flush`, определенной в шаблоне класса `basic_istream<E, T>` и рассмотренной в *разд. 4.3.1.8*.

40.4.1.14. Класс *sentry*

В шаблоне класса `basic_istream<E, T>` определен вложенный класс `sentry`:

```
class sentry
{
public:
  explicit sentry(basic_istream& is, bool noskip = false);
  operator bool() const;
};
```

Класс `sentry` служит для объявления объектов, создание которых подготавливает поток, заданный параметром `is` конструктора, для форматированного и неформатированного ввода. После завершения работы конструктора вызов оператора преобразования типа `bool` возвращает значение `true`, если значение `is.good()` также равно `true`, иначе — значение `false`.

40.4.2. Манипулятор

В заголовочном файле istream определен следующий шаблон манипулятора шаблонного класса `basic_istream<E, T>`:

```
template class<E, T>
    basic_istream<E, T>& ws(basic_istream<E, T>& is);
```

Манипулятор `ws` удаляет из потока ввода, заданного параметром `is`, все ведущие пробельные символы. Удаление символов прекращается при встрече первого не пробельного символа или при достижении конца файла. В последнем случае манипулятор вызывает функцию `setstate(eofbit)`.

40.4.3. Перегруженный шаблон оператора ввода >>

Для форматированного ввода символа в заголовочном файле istream перегружены следующие шаблоны оператора >>:

```
template<class E, class T> basic_istream<E, T>&
  operator>>(basic_istream<E, T>& is, E& c);
template<class T> basic_istream<char, T>&
  operator>>(basic_istream<char, T>& is, signed char& c);
template<class T> basic_istream<char, T>&
  operator>>(basic_istream<char, T>& is, unsigned char& c);
```

Соответствующие шаблонные операторы >> вводят из потока, заданного параметром `is`, символ, который записывают по адресу, заданному параметром `c`.

Для форматированного ввода последовательности символов в заголовочном файле istream перегружены следующие шаблоны оператора >>:

```
template<class E, class T> basic_istream<E, T>&
  operator>>(basic_istream<E, T>& is, E *s);
template<class T> basic_istream<char, T>&
 operator>>(basic_istream<char, T>& is, signed char *s);
template<class T> basic_istream<char, T>&
  operator>>(basic_istream<char, T>& is, unsigned char *s);
```

Соответствующие шаблонные операторы >> вводят из потока, заданного параметром `is`, символы и последовательно записывают их в массив, адрес которого задан параметром `s`.

При вводе все ведущие пробельные символы игнорируются. Ввод символов заканчивается при выполнении одного из следующих требований:

- введено n-1 символов, где n равно ширине поля ввода, установленного в потоке;
- встретился не пробельный символ;
- встретился конец файла.

В любом случае после завершения ввода каждый из этих операторов устанавливает ширину поля ввода равной нулю.

Если не было введено ни одного символа, то каждый из этих операторов вызывает функцию is.setstate(failbit).

40.4.4. Шаблон класса *basic_iostream*

В заголовочном файле istream также определен шаблон класса basic_iostream<E, T>, который определяет базовые классы для потоков ввода/вывода, а также типы — синонимы специализаций этого шаблона:

```
template <class E, class T = char_traits<E>>
  class basic_iostream: public basic_istream<E, T>,
      public basic_ostream<E, T>
  {
  public:
    explicit basic_iostream(basic_streambuf<E, T>& *sb);
    virtual ~basic_iostream();
  };
```

Параметр E шаблона задает тип символов, которые являются элементами потока, а параметр T определяет характеристики этих символов. Подробно характеристики символов описаны в *разд. 9.1*.

Как видно из этого определения, шаблонный класс basic_iostream<E, T> является наследником базовых потоков ввода basic_istream<E, T> и вывода basic_ostream<E, T>. Но при этом заметим, что класс basic_ios<E, T> наследуется через эти базовые классы виртуально. Поэтому шаблонный класс basic_iostream<E, T> разделяет один буфер для ввода и вывода данных.

40.5. Стандартные потоки ввода/вывода <iostream>

В заголовочном файле iostream определены следующие стандартные потоки ввода/вывода, которые работают с символами типа `char` и связаны со стандартными потоками ввода/вывода, определенными в языке программирования C:

```
namespace std
{
  extern istream cin;   // ввод из stdin
  extern ostream cout;  // вывод в stdout
  extern ostream cerr;  // вывод в stderr
  extern ostream clog;  // вывод в stderr
};
```

Также в заголовочном файле iostream определены следующие стандартные потоки ввода/вывода, которые работают с широкими символами типа `wchar_t` и связаны со стандартными потоками ввода/вывода, определенными в языке программирования C:

```
namespace std
{
  extern wistream wcin;   // ввод из stdin
  extern wostream wcout;  // вывод в stdout
  extern wostream wcerr;  // вывод в stderr
  extern wostream wclog;  // вывод в stderr
};
```

Так как стандартные потоки ввода/вывода являются объектами, то они объявляются в заголовочном файле iostream. Предполагается, что эти объекты конструируются в объектном файле, который получен из исходного файла, содержащего заголовочный файл iostream. Причем эта инициализация выполняется до начала исполнения функции `main`. Конструирование стандартных потоков гарантируется исполнением функции `basic_ios<charT,traits>::Init`. Эту функцию нужно вызывать в конструкторах статических объектов, если они работают со стандартными потоками, т. к. статические объекты инициализируются до передачи управления функции `main`.

40.6. Базовые потоки ввода/вывода в файлы <fstream>

В заголовочном файле fstream определены следующие шаблоны и типы — синонимы специализаций этих шаблонов:

```
namespace std
{
  template<class E, class T = char_traits<E>>
    class basic_ofstream;
  typedef basic_ofstream<char> ofstream;
  typedef basic_ofstream<wchar_t> wofstream;
  template<class E, class T = char_traits<E>>
    class basic_ifstream;
  typedef basic_ifstream<char> ifstream;
  typedef basic_ifstream<wchar_t> wifstream;
  template<class E, class T = char_traits<E>>
    class basic_fstream;
  typedef basic_fstream<char> fstream;
  typedef basic_fstream<wchar_t> wfstream;
};
```

Шаблоны классов `basic_ofstream<E, T>`, `basic_ifstream<E, T>` и `basic_fstream<E, T>` предназначены для создания потоков вывода, ввода и ввода/вывода в файлы. Типы `ofstream`, `wofstream`, `ifstream`, `wifstream`, `fstream` и `wfstream` являются синонимами специализаций этих шаблонов классов и предназначены для определения потоков, работающих с символами типов `<char>` и `<wchar_t>` соответственно.

В заголовочном файле fstream также определен шаблон класса `basic_filebuf<E, T>`, который используется для создания буфера, связанного с файлом. Кроме того, определены типы `filebuf` и `wfilebuf`, которые являются синонимами для специализаций этого шаблона и предназначены для создания буферов, работающих с символами типов `<char>` и `<wchar_t>` соответственно. Буфер потока создается самим потоком при его инициализации. Так как прикладные программы редко работают с буфером потока напрямую, то буферы потоков рассматриваться не будут.

40.6.1. Шаблон класса *basic_ofstream*

Шаблон класса `basic_ofstream<E, T>` имеет следующий интерфейс:

```
template <class E, class T = char_traits<E>>
  class basic_ofstream: public basic_ostream<E, T>
{
public:
  // конструкторы
  basic_ofstream();
  explicit basic_ofstream(const char *s,
      ios_base::openmode mode = ios_base::out);
  // функции-члены
  basic_filebuf<E, T> *rdbuf() const;
  void open(const char *s,
      ios_base::openmode mode = ios_base::out);
  bool is_open() const;
  void close();
};
```

Параметр `E` шаблона задает тип символов, которые являются элементами потока, а параметр `T` определяет характеристики этих символов. Подробно характеристики символов описаны в *разд. 39.1*.

Конструктор по умолчанию создает буфер потока посредством вызова конструктора `basic_filebuf<E, T>()`, а затем вызывает конструктор базового класса `basic_ostream<E, T>`, передавая ему в качестве аргумента указатель на буфер потока.

Второй конструктор также создает буфер потока посредством вызова конструктора `basic_filebuf<E, T>()`, а затем открывает файл, на имя которого указывает параметр `s`, в режиме `mode|ios_base::out`. Возможные режимы открытия файла описаны в *разд. 40.2.1.3*. Если открытие файла закончилось неудачей, то конструктор вызывает функцию `setstate(failbit)`.

Функции-члены шаблона класса `basic_ofstream<E, T>` выполняют следующие действия:

- `rdbuf` — возвращает адрес буфера потока;
- `open` — открывает файл, на имя которого указывает параметр `s`, в режиме `mode|ios_base::out`. Если открытие файла закончилось неудачей, то вызывает функцию `setstate(failbit)`;
- `is_open` — возвращает значение `true`, если файл открыт, иначе — `false`;
- `close` — закрывает файл. Если файл успешно закрыт, то возвращает указатель `this`, иначе — пустой указатель.

40.6.2. Шаблон класса *basic_ifstream*

Шаблон класса `basic_ifstream<E, T>` имеет следующий интерфейс:

```
template <class E, class T = char_traits<E>>
  class basic_ifstream : public basic_istream<E, T>
{
public:
  // конструкторы
  basic_ifstream();
  explicit basic_ifstream(const char *s,
      ios_base::openmode mode = ios_base::in);
  // функции-члены
  basic_filebuf<E, T> *rdbuf() const;
  void open(const char *s,
      ios_base::openmode mode = ios_base::in);
  bool is_open() const;
  void close();
};
```

Параметр `E` шаблона задает тип символов, которые являются элементами потока, а параметр `T` определяет характеристики этих символов. Подробно характеристики символов описаны в *разд. 39.1*.

Конструктор по умолчанию создает буфер потока посредством вызова конструктора `basic_filebuf<E, T>()`, а затем вызывает конструктор базового класса `basic_ostream<E, T>`, передавая ему в качестве аргумента указатель на буфер потока.

Второй конструктор также создает буфер потока посредством вызова конструктора `basic_filebuf<E, T>()`, а затем открывает файл, на имя которого указывает параметр `s`, в режиме `mode|ios_base::in`. Возможные режимы открытия файла описаны в *разд. 40.2.1.3*. Если открытие файла закончилось неудачей, то конструктор вызывает функцию `setstate(failbit)`.

Функции-члены шаблона класса `basic_ifstream<E, T>` выполняют следующие действия:

- `rdbuf` — возвращает адрес буфера потока;
- `open` — открывает файл, на имя которого указывает параметр `s`, в режиме `mode|ios_base::in`. Если открытие файла закончилось неудачей, то вызывает функцию `setstate(failbit)`;
- `is_open` — возвращает значение `true`, если файл открыт, иначе — `false`;
- `close` — закрывает файл. Если файл успешно закрыт, то возвращает указатель `this`, иначе — пустой указатель.

40.6.3. Шаблон класса *basic_fstream*

Шаблон класса `basic_fstream<E, T>` имеет следующий интерфейс:

```
template <class E, class T = char_traits<E>>
  class basic_fstream : public basic_iostream<E, T>
{
public:
  basic_fstream();
  explicit basic_fstream(const char *s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
  basic_filebuf<E, T> *rdbuf() const;
  void open(const char *s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
  bool is_open() const;
  void close();
};
```

Параметр `E` шаблона задает тип символов, которые являются элементами потока, а параметр `T` определяет характеристики этих символов. Подробно характеристики символов описаны в *разд. 39.1*.

Конструктор по умолчанию создает буфер потока посредством вызова конструктора `basic_filebuf<E, T>()`, а затем вызывает конструктор базового класса `basic_ostream<E, T>`, передавая ему, в качестве аргумента, указатель на буфер потока.

Второй конструктор также создает буфер потока посредством вызова конструктора `basic_filebuf<E, T>()`, а затем открывает файл, на имя которого указывает параметр `s`, в режиме `mode`. Возможные режимы открытия файла описаны в *разд. 40.2.1.3*. Если открытие файла закончилось неудачей, то конструктор вызывает функцию `setstate(failbit)`.

Функции-члены шаблона класса `basic_ifstream<E, T>` выполняют следующие действия:

- `rdbuf` — возвращает адрес буфера потока;
- `open` — открывает файл, на имя которого указывает параметр `s`, в режиме `mode`. Если открытие файла закончилось неудачей, то вызывает функцию `setstate(failbit)`;
- `is_open` — возвращает значение `true`, если файл открыт, иначе — `false`;
- `close` — закрывает файл. Если файл успешно закрыт, то возвращает указатель `this`, иначе — пустой указатель.

40.6.4. Работа с текстовыми файлами

Под текстовыми файлами понимают файлы, в которых хранится текстовая информация, т. е. каждый символ такого файла является алфавитно-цифровым, специальным или управляющим символом. При работе с текстовыми файлами в языке программирования C++ нужно использовать перегруженные операторы ввода `>>` и вывода `<<`, которые выполняют форматированный ввод/вывод данных в поток.

40.6.4.1. Создание текстового файла

Пример создания текстового файла приведен в листинге 40.11.

Листинг 40.11. Создание текстового файла

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream out;

  out.open("test.txt");

  out << "Integer: " << 10 << endl;
  out << "Double: " << 20.5 << endl;
  out << "String: " << "This is a string." << endl;

  out.close();

  return 0;
}
```

40.6.4.2. Чтение текстового файла

Пример чтения текстового файла приведен в листинге 40.12.

Листинг 40.12. Чтение текстового файла

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ifstream in;
  char buf[80];

  in.open("test.txt");
  if (!in.is_open())
```

```
  {
    cerr << "Open file failed." << endl;
    return 0;
  }

  while (!in.eof())
  {
    in.getline(buf, 80);
    cout << buf << endl;
  }

  in.close();

  return 0;
}
```

При чтении текстового файла может возникнуть следующий вопрос: какую длину буфера выбрать для хранения строки? Обычно длина буфера выбирается равной длине строки на устройстве вывода или из других ограничений, обусловленных реализацией. В конце концов, длину буфера можно выбрать равной длине файла.

40.6.4.3. Модификация текстового файла

Следует понимать, что текстовые файлы — это файлы *последовательного* доступа, т. е. содержимое файлов читается последовательно слово за словом. Поэтому для модификации текстового файла нужно содержимое этого файла переписать в другой файл, внося по ходу нужные изменения.

40.6.5. Работа с бинарными файлами

Под бинарными файлами понимаются файлы, в которых хранятся бинарные данные, т. е. данные, представленные двоичными кодами. Такие файлы обычно хранят набор записей, каждая из которых является копией содержимого структуры, определенной в прикладной программе. При работе с бинарными файлами в языке программирования C++ нужно придерживаться следующих двух правил:

- поток, связанный с бинарным файлом, должен быть открыт в бинарном режиме;

- для доступа к данным, хранящимся в бинарном файле, нужно использовать не форматирующие операции ввода/вывода.

40.6.5.1. Создание бинарного файла

Пример создания бинарного файла приведен в листинге 40.13.

Листинг 40.13. Создание бинарного файла

```
#include <iostream>
#include <fstream>
using namespace std;

struct emp
{
  int code;
  char name[20];
  double salary;
};

int main()
{
  ofstream out;     // выходной поток
  struct emp s;     // для записей файла
  int n;

  // открываем выходной поток в бинарном режиме
  out.open("demo.bin", ofstream::binary);

  cout << "Input a number of records: ";
  cin >> n;
  cout << "Input code, name and salary." << endl;
  for (int i = 0; i < n; ++i)
  {
    cout << '>';
    // вводим следующие записи с консоли
    cin >> s.code >> s.name >> s.salary;
    // пишем запись в файл
    out.write((const char*)&s, sizeof(struct emp));

  }
```

```
    // закрываем выходной поток
  out.close();

  return 0;
}
```

40.6.5.2. Чтение бинарного файла

Пример чтения бинарного файла приведен в листинге 40.14.

Листинг 40.14. Чтение бинарного файла

```
#include <iostream>
#include <fstream>
using namespace std;

struct emp
{
  int code;
  char name[20];
  double salary;
};

int main()
{
  ifstream in;      // входной поток
  struct emp s;     // для записей файла
  long length, n;  // длина файла, количество записей

  // открываем входной поток в бинарном режиме
  in.open("demo.bin");
  if(!in.is_open())
  {
    cout << "Open file failed.\n";
    return 0;
  }
  // определяем длину файла
  in.seekg(0, ios::end);
  length = in.tellg();
  in.seekg(0, ios::beg);
```

```
  // подсчитываем количество записей
  n = length / sizeof(emp);
  // читаем записи
  for (int i = 0; i < n; ++i)
  {
    in.read((char*)&s, sizeof(struct emp));
    cout << "code: " << s.code << " name: " << s.name
         << " sal: " << s.salary << endl;
  }
  in.close();

  return 0;
}
```

40.6.5.3. Модификация бинарного файла

Так как бинарные файлы содержат копии содержимого оперативной памяти, то можно определить смещение нужной записи в бинарном файле, зная длину записей и длину файла. Отсюда следует, что при модификации бинарного файла нет необходимости читать весь файл, читается, корректируется и записывается обратно в файл только нужная запись. Поэтому бинарные файлы также называются файлами прямого доступа.

Пример модификации бинарного файла приведен в листинге 40.15.

Листинг 40.15. Модификация бинарного файла

```
#include <iostream>
#include <fstream>
using namespace std;

struct emp
{
  int code;
  char name[20];
  double salary;
};
```

```
int main()
{
  fstream f;      // поток для ввода/вывода
  struct emp s;   // для записей файла
  unsigned i;     // номер записи

  // открываем поток в бинарном режиме
  f.open("demo.bin", ios::binary|ios::in|ios::out);
  if (!f.is_open())
  {
    cout << "Open file failed.\n";
    return 0;
  }

  cout << "Input an index: ";  // читаем индекс
  cin >> i;

  f.seekg(i * sizeof(struct emp));  // устанавливаем указатель
                                    на нужную запись

  f.read((char*)&s, sizeof(struct emp));  // читаем запись
                                          из файла
  if (!f.good())
  {
    cout << "Wrong index.\n";
    f.close();

    return 0;
  }
  cout << "code = " << s.code << " name = " << s.name
       << " sal = " << s.salary << endl;
  f.close();

  return 0;
}
```

40.6.6. Копирование файла

Пример копирования файлов приведен в листинге 40.16.

Листинг 40.16. Копирование файла

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ifstream in("test.txt");
  ofstream out("new_test.txt");

  if (!in.is_open())
  {
    cout << "Open file failed.\n";
    return 0;
  }
  out << in.rdbuf();     // копируем файл

  in.close();
  out .close();

  return 0;
}
```

40.7. Манипуляторы <iomanip>

Манипулятором потока называется функция, которая изменяет поток. Указатели на манипуляторы являются операндами перегруженных операторов вывода << и ввода >>, которые определены как члены шаблонов классов `basic_ostream<E, T>` и `basic_istream<E, T>` соответственно.

Условно манипуляторы подразделяются на две группы: простые манипуляторы и параметризованные манипуляторы. Далее рассмотрим эти группы манипуляторов подробнее.

40.7.1. Простые манипуляторы

Манипулятор потока, который не имеет параметров, называется *простым*. Или, если точнее, *простой манипулятор* имеет один

параметр, который является ссылкой на поток, который он изменяет. Но т. к. манипулятор вызывается через указатель перегруженным оператором ввода или вывода, то этот параметр передается манипулятору неявно.

Обычно простые манипуляторы изменяют только состояние управляющих флагов потока. Простые манипуляторы потоков, которые определены в стандартной библиотеке ввода/вывода, были рассмотрены в *разд. 40.2.4*, *40.3.2* и *40.4.2*.

Манипуляторы без параметров называются простыми потому, что очень просто определить свой простой манипулятор потока. Для этого не требуется перегружать операторы ввода и вывода. Например, простой манипулятор, определенный в листинге 40.17, выводит следующий после него текст с новой строки и выполняет горизонтальную табуляцию.

Листинг 40.17. Пример простого манипулятора

```
#include <iostream>
using namespace std;

ostream& tabl(ostream& out)
{
  out << '\n' << '\t';
  return out;
}

int main()
{
  cout << "This is an old string." << tabl
       << "This is a new string." << endl;

  return 0;
}
```

40.7.2. Параметризованные манипуляторы

Манипулятор потока, который имеет параметры, называются *параметризованным*. Далее перечислены параметризованные манипуляторы, которые объявлены в заголовочном файле iomanip:

```
namespace std
{
  T1 resetiosflags(ios_base::fmtflags mask);
  T2 setiosflags(ios_base::fmtflags mask);
  T3 setbase(int base);
  template<class E> T4 setfill(E c);
  T5 setprecision(streamsize n);
  T6 setw(streamsize n);
};
```

Здесь `T1`, ..., `T6` — некоторые типы данных, зависимые от реализации манипуляторов. Эти манипуляторы выполняют следующие действия:

- `resetiosflags` — сбрасывает флаги форматирования;
- `setiosflags` — устанавливает флаги форматирования;
- `setbase` — устанавливает систему счисления;
- `setfill` — устанавливает символ заполнитель;
- `setprecision` — устанавливает точность для плавающих чисел;
- `setw` — устанавливает ширину широкого символа заполнителя.

Пример использования параметризованного манипулятора приведен в листинге 40.18.

Листинг 41.18. Пример параметризованного манипулятора

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setbase(16) << 16 << endl;  // 10
  return 0;
}
```

Список литературы

1. Б. Керниган, Д. Ритчи. Язык программирования С, 3-е изд. — СПб.: Невский диалект, 2001. — 352 с.
2. Х. Дейтел, П. Дейтел. Как программировать на С, 4-е изд. — М.: Бином, 2005. — 912 с.
3. Б. Страуструп. Язык программирования С++, 3-е изд. — СПб. — М.: Невский диалект — Бином, 1999. — 991 с.
4. С. Б. Липпман, Ж. Лажойе. Язык программирования С++. Вводный курс, 3-е изд. — СПб. — М.: Невский диалект — ДМК Пресс, 2001. — 1104 с.
5. Х. Дейтел, П. Дейтел. Как программировать на С++. — М.: Бином, 1998. — 1024 с.
6. В. Штерн. Основы С++. Методы программной инженерии. — М.: Лори, 2003. — 880 с.
7. А. И. Голуб. Правила программирования С&С++. — М.: Бином, 1996. — 272 с.
8. Д. Вандевурд, Н. М. Джосаттис. Шаблоны С++: справочник разработчика. — М.: Вильямс, 2003. — 544 с.
9. Н. Джосьютис. С++. Стандартная библиотека. — СПб.: Питер, 2003. — 736 с.
10. П. Плаугер, А. Степанов, М. Ли, Д. Массер. STL — стандартная библиотека шаблонов С++. — СПб.: БХВ-Петербург, 2004. — 656 с.

Предметный указатель

А

Аргументы шаблона
◊ вывод 246
◊ дедукция 246
Ассоциативность операторов 46

Б

Битовое поле 110
Блок 30
◊ ввод/вывод 347
Буфер 366

В

Выражение 45

Д

Деструктор 203
Директива препроцессора 122
◊ #define 123
◊ #elif 126
◊ #else 125
◊ #endif 124
◊ #error 129
◊ #if 124
◊ #ifdef 126
◊ #ifndef 127
◊ #include 127
◊ #line 129
◊ #pragma 130
◊ #undef 124
◊ условной компиляции 126
Друзья класса 183

Е

Единица компиляции 91, 96

И

Идентификатор 27
◊ область видимости 89
Имя:
◊ переменной:
  ▫ декорирование 163
  ▫ искажение 163
◊ функции:
  ▫ декорирование 154
  ▫ искажение 154
Инкапсуляция 182

Инструкция 29,23
◊ break 64
◊ continue 64
◊ do-while 62
◊ for 63
◊ goto 65
◊ if 59
◊ if…else 59
◊ return 80
◊ switch 60
◊ while 62
◊ составная 30
Исключение 165
◊ выброс 165
◊ генерация 165
◊ неожидаемое 176
◊ непредусмотренное 176
◊ обработчик 166
◊ спецификация 175

К

Квалификатор имени объекта 158
Класс 177
◊ bad_cast 423
◊ bad_exception 408
◊ bad_typeid 421
◊ exception 408
◊ failer 466
◊ Init 466
◊ ios_base 459
◊ sentry 484, 499
◊ type_info 419
◊ абстрактный 236
◊ атрибут 179
◊ базовый 222
◊ базовый виртуальный 239
◊ вложенный 194
◊ локальный 196
◊ метод 179
◊ наследование 222
◊ обобщенный 257
◊ полиморфный 233
◊ производный 222
◊ родовой 257
◊ шаблонный 259
Комментарий 30
Конкретизация шаблона, явная 251
Константа 27
◊ именованная 37
◊ перечислимая 101
◊ с плавающей точкой 28
◊ символическая 123
◊ символьная 29
◊ строковая 29
◊ целая 28
Конструктор 197
◊ ios_base 466
◊ копирования 200
◊ по умолчанию 199
◊ явный 203

Л

Лексема 24
Литерал 27
◊ строковый 29
Локализация 382
Локальная категория 383
Локальность 382

М

Макрокоманда 123
◊ setjmp 373
Макроопределение 123

Макрос 123
◊ assert 305
◊ NULL 280
◊ offsetof 280
◊ va_arg 308
◊ va_end 308
◊ va_start 307
Манипулятор:
◊ boolalpha 477
◊ dec 477
◊ endl 485
◊ ends 485
◊ fixed 477
◊ flush 485
◊ hex 477
◊ internal 477
◊ left 477
◊ noboolalpha 478
◊ noshowbase 478
◊ noshowpoint 478
◊ noskipws 478
◊ nounitbuf 478
◊ nouppercase 478
◊ oct 477
◊ right 477
◊ scientific 478
◊ showbase 478
◊ showpoint 478
◊ showpos 478
◊ skipws 478
◊ unitbuf 478
◊ uppercase 478
◊ ws 500
◊ потока 459, 514
  ▫ параметризованный 515
  ▫ простой 514
Массив 71
◊ встроенный 77
◊ динамический 284
◊ индекс 72
◊ многомерный 75
◊ статический 77
◊ элемент 71
Метка инструкции 65
Модификатор:
◊ длины 32
◊ знака 32
◊ типа 32

H

Наследование:
◊ виртуальное 239
◊ множественное 238

O

Обработчик:
◊ аварийного завершения 409
◊ не специфицированного исключения 410
Объединение 107
◊ анонимное 108, 137
Объект 35
◊ класса 180
◊ функциональный 214
◊ функция 214
Объявление класса, предварительное 179
Оператор:
◊ const_cast 141
◊ delete 144
◊ dynamic_cast 142
◊ new 143
◊ reinterpret_cast 141
◊ static_cast 140
◊ streamoff 476

- ◊ typeid 420
- ◊ арифметические 40
- ◊ вызова функции 82
- ◊ запятая 55
- ◊ индексирования 72
- ◊ конвертер 219
- ◊ логические 44
- ◊ обращения по адресу 69
- ◊ определения адреса 69
- ◊ побитовые 41
- ◊ преобразования типа 48
- ◊ присваивания 49, 51
- ◊ разыменования 70
- ◊ сравнения 43
- ◊ точка 105
- ◊ условный 53
- ◊ Бинарный:
  - ▫ != 44
  - ▫ % 40
  - ▫ && 44
  - ▫ || 44
  - ▫ + 40
  - ▫ < 43
  - ▫ <= 43
  - ▫ == 44
  - ▫ > 43
  - ▫ >= 44
  - ▫ побитовое И 41
  - ▫ побитовое ИЛИ 41
  - ▫ побитовое исключающее ИЛИ 41
  - ▫ сдвиг влево 41
  - ▫ сдвиг вправо 41
- ◊ Препроцессора:
  - ▫ # 130
  - ▫ ## 131
- ◊ Присваивания:
  - ▫ %= 51
  - ▫ &= 51
  - ▫ *= 51
  - ▫ /= 51
  - ▫ ^= 51
  - ▫ |= 51
  - ▫ += 51
  - ▫ <<= 51
  - ▫ -= 51
  - ▫ >>= 51
- ◊ Унарный:
  - ▫ ! 44
  - ▫ * 40
  - ▫ / 40
  - ▫ + 40
  - ▫ ++ 52
  - ▫ дополнение до 1 41

Определение переменной 92

П

Перегрузка функций 149
Переменная 35
- ◊ время существования 90
- ◊ глобальная 89
- ◊ изменчивая 38
- ◊ локальная 89
- ◊ нестабильная 38
- ◊ объединение 109

Перечисление 101
- ◊ анонимное 102

Побочный эффект 56
Полиморфизм:
- ◊ включающий 233
- ◊ параметрический 262
- ◊ подтипов 233
- ◊ принудительный 48

Последовательность неявных преобразований 151
Поток 341, 456
Препроцессор 122

Преобразование (приведение) типов 47
◊ интегральное расширение 47
◊ неявное 47
◊ расширение с плавающей точкой 47
Приоритет оператора 45
Программа 23
Пространство имен 155
◊ глобальное 156
◊ локальное анонимное 156
◊ открытость 155
◊ стандартное 157
Псевдоним:
◊ имени пространства имен 162
◊ переменной 139

P

Раскрутка стека 175

C

Связывание 91
◊ динамическое 233
◊ отложенное 233
◊ позднее 233
Символ 23
◊ длинный 301
◊ разделители 26
◊ специальные 25
◊ управляющие 26
◊ широкий 135
Синхронизация потока с файлом 499
Слово 23
◊ зарезервированное 27
◊ ключевое 26
Специализация шаблона 243
◊ частичная 265
◊ явная 249, 264
Спецификатор:
◊ auto 94
◊ extern 93
◊ register 95
◊ доступа:
▫ private 181
▫ protected 181
▫ public 181
◊ классов памяти 92
◊ типа данных 33
Ссылка 138, 148
◊ константная 139
Строка 74
◊ широкая 136
Структура 103
◊ анонимная 104
◊ переменная 104
◊ поле 103
◊ тип данных 103

T

Тег:
◊ объединения 108
◊ перечисления 101
◊ структуры 103
Тип 31
◊ bool 135
◊ event 464
◊ fmtflags 460
◊ iostate 462
◊ openmode 463
◊ ptrdiff_t 279
◊ seekdir 463
◊ size_t 279
◊ streamoff 458

◊ streamsize 458
◊ wchar_t 279
◊ wchar_t 135
◊ арифметический 37
◊ базовый 31
◊ встроенный 34
◊ интегральный 36
◊ квалификатор 37
◊ композиционный 119
◊ неполный 117
◊ объединение 108
◊ плавающий 37
◊ пользовательский 101
◊ предопределенный 34
◊ простой 35
◊ пустой 37
◊ с плавающей точкой 36
◊ синоним 114
◊ скалярный 68
◊ сложный 35
◊ спецификатор 33
◊ сравнимые типы данных 119
◊ стандартный 34
◊ структура 103
◊ фундаментальный 34
◊ числовой 37
Точка последовательности 57

У

Указатель 67
◊ родовый 283
Условие продолжения цикла 62

Ф

Файл 341
◊ доступ 341
◊ заголовочный 82, 128
◊ закрытие 341
◊ исходный 96
◊ открытие 341
Функтор 87
Функция 78
◊ abort 291
◊ abs 298
◊ acos 319
◊ asctime 396
◊ asin 319
◊ atan 319
◊ atan2 319
◊ atexit 290
◊ atof 295
◊ atoi 294
◊ atol 294
◊ bad 472
◊ bsearch 288
◊ calloc 282
◊ ceil 323
◊ clear 472
◊ clock 390
◊ copyfmt 473
◊ cos 319
◊ cosh 321
◊ ctime 395
◊ difftime 392
◊ div 298
◊ ecvt 297
◊ eof 472
◊ exceptions 473
◊ exit 290
◊ exp 320
◊ fabs 324
◊ fail 472
◊ fclose 344
◊ fcvt 297
◊ feof 345

Функция *(прод.)*:
◊ ferror 344
◊ fflush 367
◊ fgetc 351
◊ fgetpos 347
◊ fgets 353
◊ fill 474
◊ flags 461
◊ flush 482
◊ fmod 324
◊ fopen 343
◊ fprintf 354
◊ fputc 350
◊ fputs 353
◊ fread 347
◊ free 283
◊ freopen 344
◊ frexp 322
◊ fscanf 360
◊ fseek 346
◊ fsetpos 346
◊ ftell 346
◊ fwrite 347
◊ gcvt 297
◊ getc 351
◊ getchar 369
◊ getenv 292
◊ getloc 468
◊ gets 369
◊ gmtime 393
◊ good 472
◊ imbue 468, 475
◊ init 471
◊ isalnum 325
◊ isalpha 325
◊ iscntrl 325
◊ isdigit 325
◊ isgraph 325
◊ islower 325
◊ isprint 325
◊ ispunct 325
◊ isspace 325
◊ isupper 325
◊ isxdigit 325
◊ itoa 297
◊ iword 467
◊ labs 298
◊ ldexp 322
◊ ldiv 299
◊ localtime 394
◊ log 320
◊ log10 320
◊ ltoa 297
◊ malloc 282
◊ mblen 301
◊ mbstowcs 303
◊ mbtowc 301
◊ memchr 336
◊ memcmp 336
◊ memcpy 336
◊ memmove 336
◊ memset 336
◊ mktime 395
◊ modf 323
◊ perror 369
◊ pow 321
◊ precision 467
◊ printf 369
◊ put 482
◊ putc 350
◊ putchar 369
◊ puts 369
◊ pword 467
◊ qsort 287
◊ rand 299
◊ rdstate 472

◊ read 496
◊ realloc 282
◊ remove 369
◊ rename 370
◊ rewind 345
◊ scanf 369
◊ seekg 498
◊ seekp 483
◊ set_terminate 409, 410
◊ setbuf 367
◊ setf 461
◊ setjmp 374
◊ setstate 472
◊ setvbuf 366
◊ signal 378
◊ sin 319
◊ sinh 321
◊ sprintf 364
◊ sqrt 321
◊ srand 300
◊ sscanf 365
◊ state 476
◊ strcat 330
◊ strchr 331
◊ strcmp 328
◊ strcoll 338
◊ strcpy 329
◊ strcspn 332
◊ strerror 339
◊ strftime 397
◊ strlen 335
◊ strncat 330
◊ strncmp 328
◊ strncpy 329
◊ strpbrk 333
◊ strrchr 331
◊ strspn 332
◊ strstr 333
◊ strtod 297
◊ strtok 334
◊ strtol 295
◊ strtoul 297
◊ strxfrm 338
◊ sync_with_stdio 468
◊ system 293
◊ tan 319
◊ tanh 321
◊ tellg 498
◊ tellp 483
◊ terminate 409
◊ tie 474
◊ time 391
◊ tmpnam 371, 372
◊ tolower 326
◊ toupper 327
◊ uncaught_exception 412
◊ unsetf 461
◊ utoa 297
◊ wcstombs 303
◊ wctomb 302
◊ width 467
◊ write 482
◊ аргументы 82
◊ виртуальная 231
◊ встроенная 147
◊ кандидат 150
◊ обобщенная 241
◊ обработки сигнала 378
◊ объявление 78
◊ определение 80
◊ осуществимая 150
◊ перегрузка 149
◊ прототип 79
◊ родовая 241
◊ сигнатура 150
◊ чисто виртуальная 236
◊ шаблонная 244

Ч

Член:
◊ класса:
  ▫ сокрытие 224
◊ объединения 107
◊ перечисления 101
◊ структуры 103

Ш

Шаблон:
◊ инстанциирование 243
◊ класса 257
  ▫ basic_fstream 506
  ▫ basic_ifstream 505
  ▫ basic_ios 469
  ▫ basic_istream 487
  ▫ basic_ofstream 504
  ▫ basic_ostream 479
  ▫ инстанциирование 259
  ▫ конкретизация 259
  ▫ первичный 265
  ▫ специализация 259
  ▫ частичная специализация 265
  ▫ явная специализация 264
◊ конкретизация 243
◊ функции 241
  ▫ специализация 249, 253
  ▫ частичное упорядочивание 254
◊ экспортируемый 252
◊ частичное упорядочивание 266