Николай Прохоренок

bhv

Язык С

САМОЕ НЕОБХОДИМОЕ

Базовый синтаксис современного языка С
Типы данных, операторы, условия и циклы
Работа с числами, массивами, строками и указателями
Создание пользовательских функций
Работа с файлами и каталогами
Многопоточные приложения
Создание статических и динамических библиотек
Библиотека MinGW-W64
Редактор Eclipse

+ Материалы на www.bhv.ru

Николай Прохоренок

Язык С

САМОЕ НЕОБХОДИМОЕ

Санкт-Петербург
«БХВ-Петербург»
2020

УДК 004.438 С
ББК 32.973.26-018.1
П84

Прохоренок Н. А.

П84 Язык С. Самое необходимое. — СПб.: БХВ-Петербург, 2020. — 480 с.: ил. — (Самое необходимое)

ISBN 978-5-9775-4116-9

Описан базовый синтаксис современного языка С: типы данных, операторы, условия, циклы, работа с числами, строками, массивами и указателями, создание пользовательских функций, модулей, статических и динамических библиотек. Рассмотрены основные функции стандартной библиотеки языка С, а также функции, применяемые только в операционной системе Windows. Для написания, компиляции и запуска программ используется редактор Eclipse, а для создания исполняемого файла — компилятор gcc.exe версии 8.2, входящий в состав популярной библиотеки MinGW-W64. Книга содержит большое количество практических примеров, помогающих начать программировать на языке С самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник. Электронный архив с примерами находится на сайте издательства.

Для программистов

УДК 004.438 С
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Марины Дамбиевой*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 05.12.19.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 38,7.
Тираж 1300 экз. Заказ № 10367.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-4116-9

Оглавление

Введение **9**

Глава 1. Установка программ под Windows **11**
1.1. Создание структуры каталогов 11
1.2. Добавление пути в переменную *PATH* 12
1.3. Работа с командной строкой 14
1.4. Установка MinGW и MSYS 14
1.5. Установка MinGW-W64 20
1.6. Установка MSYS2 и MinGW-W64 24
1.7. Установка и настройка редактора Eclipse 29
1.8. Создание проектов в редакторе Eclipse 34

Глава 2. Первые шаги **44**
2.1. Первая программа 44
2.2. Создание пустого проекта в редакторе Eclipse 47
2.3. Добавление в проект файла с программой 49
2.4. Добавление в проект заголовочного файла 50
2.5. Компиляция и запуск программы в редакторе Eclipse 53
2.6. Структура программы 57
2.7. Комментарии в программе 63
2.8. Вывод данных 65
2.9. Ввод данных 71
2.9.1. Ввод одного символа 71
2.9.2. Функция *scanf()* 72
2.9.3. Ввод строки 77
2.10. Интерактивный ввод символов 80
2.11. Получение данных из командной строки 82
2.12. Предотвращение закрытия окна консоли 84
2.13. Настройка отображения русских букв в консоли 86
2.14. Преждевременное завершение выполнения программы 89

Глава 3. Переменные и типы данных **91**
3.1. Объявление переменной 91
3.2. Именование переменных 92

3.3. Типы данных ... 93
3.4. Целочисленные типы фиксированного размера ... 97
3.5. Оператор *sizeof* и тип *size_t* ... 99
3.6. Инициализация переменных ... 100
3.7. Оператор *typedef* ... 100
3.8. Константы ... 101
3.9. Спецификаторы хранения ... 104
3.10. Области видимости переменных ... 105
3.11. Массивы ... 107
3.12. Строки ... 110
3.13. Указатели ... 111
3.14. Динамическое выделение памяти ... 118
3.14.1. Функции *malloc()* и *free()* ... 118
3.14.2. Функция *calloc()* ... 119
3.14.3. Функция *realloc()* ... 122
3.15. Структуры ... 123
3.16. Битовые поля ... 126
3.17. Объединения ... 127
3.18. Перечисления ... 129
3.19. Приведение типов ... 130

Глава 4. Операторы и циклы ... 133
4.1. Математические операторы ... 133
4.2. Побитовые операторы ... 135
4.3. Операторы присваивания ... 138
4.4. Оператор запятая ... 138
4.5. Операторы сравнения ... 139
4.6. Приоритет выполнения операторов ... 141
4.7. Оператор ветвления *if* ... 142
4.8. Оператор *?:* ... 146
4.9. Оператор выбора *switch* ... 147
4.10. Цикл *for* ... 149
4.11. Цикл *while* ... 152
4.12. Цикл *do...while* ... 152
4.13. Оператор *continue*: переход на следующую итерацию цикла ... 153
4.14. Оператор *break*: прерывание цикла ... 153
4.15. Оператор *goto* ... 154

Глава 5. Числа ... 156
5.1. Математические константы ... 159
5.2. Основные функции для работы с числами ... 160
5.3. Округление чисел ... 164
5.4. Тригонометрические функции ... 165
5.5. Преобразование строки в число ... 165
5.6. Преобразование числа в строку ... 174
5.7. Генерация псевдослучайных чисел ... 177
5.8. Бесконечность и значение *NAN* ... 179

Глава 6. Массивы ... 181
6.1. Объявление и инициализация массива ... 181
6.2. Определение количества элементов и размера массива ... 183
6.3. Получение и изменение значения элемента массива ... 184
6.4. Перебор элементов массива ... 185
6.5. Доступ к элементам массива с помощью указателя ... 186
6.6. Массивы указателей ... 189
6.7. Динамические массивы ... 189
6.8. Многомерные массивы ... 190
6.9. Поиск минимального и максимального значений ... 193
6.10. Сортировка массива ... 195
6.11. Проверка наличия значения в массиве ... 198
6.12. Копирование элементов из одного массива в другой ... 201
6.13. Сравнение массивов ... 203
6.14. Переворачивание массива ... 204

Глава 7. Символы и С-строки ... 206
7.1. Объявление и инициализация отдельного символа ... 206
7.2. Настройка локали ... 212
7.3. Изменение регистра символов ... 216
7.4. Проверка типа содержимого символа ... 218
7.5. Объявление и инициализация С-строки ... 222
7.6. Доступ к символам внутри С-строки ... 224
7.7. Определение длины строки ... 225
7.8. Перебор символов С-строки ... 226
7.9. Основные функции для работы с С-строками ... 227
7.10. Поиск и замена в С-строке ... 232
7.11. Сравнение С-строк ... 237
7.12. Форматирование С-строк ... 242

Глава 8. Широкие символы и L-строки ... 244
8.1. Объявление и инициализация широкого символа ... 245
8.2. Вывод и ввод широких символов ... 247
8.3. Изменение регистра символов ... 249
8.4. Проверка типа содержимого широкого символа ... 251
8.5. Преобразование широких символов в обычные и наоборот ... 255
8.6. Объявление и инициализация L-строки ... 256
8.7. Доступ к символам внутри L-строки ... 257
8.8. Определение длины L-строки ... 258
8.9. Перебор символов L-строки ... 259
8.10. Вывод и ввод L-строк ... 260
8.11. Преобразование С-строки в L-строку и наоборот ... 263
8.12. Преобразование кодировок ... 265
8.13. Основные функции для работы с L-строками ... 271
8.14. Поиск и замена в L-строке ... 277
8.15. Сравнение L-строк ... 282
8.16. Преобразование L-строки в число ... 286
8.17. Преобразование числа в L-строку ... 294
8.18. Типы *char16_t* и *char32_t* ... 297

Глава 9. Работа с датой и временем 300
9.1. Получение текущей даты и времени 301
9.2. Форматирование даты и времени 305
9.3. «Засыпание» программы 309
9.4. Измерение времени выполнения фрагментов кода 311

Глава 10. Пользовательские функции 312
10.1. Создание функции и ее вызов 312
10.2. Расположение объявлений и определений функций 315
10.3. Способы передачи параметров в функцию 318
10.4. Передача массивов и строк в функцию 320
10.5. Переменное количество параметров 324
10.6. Константные параметры 325
10.7. Статические переменные и функции 326
10.8. Способы возврата значения из функции 328
10.9. Указатели на функции 330
10.10. Передача в функцию и возврат данных произвольного типа 331
10.11. Рекурсия 332
10.12. Встраиваемые функции 333

Глава 11. Обработка ошибок 336
11.1. Типы ошибок 336
11.2. Предупреждающие сообщения при компиляции 337
11.3. Переменная *errno* и вывод сообщения об ошибке 338
11.4. Способы поиска ошибок в программе 341
11.5. Отладка программы в редакторе Eclipse 345

Глава 12. Чтение и запись файлов 351
12.1. Открытие и закрытие файла 351
12.2. Указание пути к файлу 354
12.3. Режимы открытия файла 356
12.4. Запись в файл 358
12.5. Чтение из файла 360
12.6. Чтение и запись двоичных файлов 363
12.7. Файлы произвольного доступа 365
12.8. Создание временных файлов 367
12.9. Перенаправление ввода/вывода 369
12.10. Работа с буфером ввода и вывода 372

Глава 13. Низкоуровневые потоки ввода и вывода 374
13.1. Открытие и закрытие файла 374
13.2. Чтение из файла и запись в файл 377
13.3. Файлы произвольного доступа 380
13.4. Создание временных файлов 381
13.5. Дескрипторы потоков ввода/вывода 382
13.6. Преобразование низкоуровневого потока в обычный 383
13.7. Создание копии потока 384
13.8. Перенаправление потоков 384

Глава 14. Работа с файловой системой 386
14.1. Преобразование пути к файлу или каталогу 386
14.2. Переименование, перемещение и удаление файла 390
14.3. Проверка прав доступа к файлу и каталогу 391
14.4. Изменение прав доступа к файлу 393
14.5. Делаем файл скрытым 394
14.6. Получение информации о файле 395
14.7. Функции для работы с дисками 399
14.8. Функции для работы с каталогами 401
14.9. Перебор объектов, расположенных в каталоге 403

Глава 15. Потоки и процессы 407
15.1. Потоки в WinAPI 407
15.1.1. Создание и завершение потока 407
15.1.2. Синхронизация потоков 412
15.2. Функции для работы с потоками, объявленные в файле process.h 417
15.3. Потоки POSIX 420
15.3.1. Создание и завершение потока 420
15.3.2. Синхронизация потоков 423
15.4. Запуск процессов 426
15.5. Получение идентификатора процесса 429

Глава 16. Создание библиотек 430
16.1. Статические библиотеки 430
16.1.1. Создание статической библиотеки из командной строки 430
16.1.2. Создание статической библиотеки в редакторе Eclipse 434
16.2. Динамические библиотеки 438
16.2.1. Создание динамической библиотеки из командной строки 438
16.2.2. Создание динамической библиотеки в редакторе Eclipse 440
16.2.3. Загрузка динамической библиотеки во время выполнения программы 443
16.2.4. Экспортируемые и внутренние функции 446
16.2.5. Функция *DllMain()* 446

Глава 17. Прочее 448
17.1. Регистрация функции, выполняемой при завершении программы 448
17.2. Выполнение системных команд 449
17.3. Получение и изменение значений системных переменных 451
17.4. Директивы препроцессора 454
17.5. Создание значка приложения 455

Заключение 459

Приложение. Описание электронного архива 461

Предметный указатель 463

Введение

Добро пожаловать в мир языка C!

Язык C — это компилируемый язык программирования высокого уровня, предназначенный для самого широкого круга задач. С его помощью можно обрабатывать различные данные, писать инструкции для микроконтроллеров, создавать драйверы, консольные, нативные и оконные приложения, и даже целые операционные системы. Язык C — настоящая легенда, один из старейших языков программирования в мире, переживший многие другие языки и оказавший влияние на синтаксис современных языков программирования, таких как Java, C#, C++, PHP и др.

C — язык кроссплатформенный, позволяющий создавать программы, которые будут работать во всех операционных системах, но для каждой операционной системы компиляцию нужно выполнять отдельно. В этой книге мы рассмотрим основы языка C применительно к 64-битной операционной системе Windows. Для создания исполняемого файла (EXE-файла) мы воспользуемся компилятором gcc.exe версии 8.2, входящим в состав популярной библиотеки MinGW-W64. Для удобства написания и запуска программы будем пользоваться редактором Eclipse.

Существует несколько стандартов языка C: C90 (ANSI C/ISO C), C99 и C11. Для того чтобы использовать правила конкретного стандарта, нужно в составе команды компиляции указать следующие флаги: `-std=c90`, `-std=c99` или `-std=c11`. Мы не будем указывать эти флаги, т. к. станем изучать современный язык C, который включает возможности стандарта C11. Узнать используемый стандарт языка C внутри программы можно с помощью макроса `__STDC_VERSION__`:

```
printf("%ld\n", __STDC_VERSION__); /* 201710 */
```

При указании флагов `-std=c99` и `-std=c11` результаты будут такими:

```
-std=c99: 199901
-std=c11: 201112
```

Получить информацию о версии компилятора позволяет макрос `__VERSION__`:

```
printf("%s\n", __VERSION__); /* 8.2.1 20181214 */
```

Давайте рассмотрим структуру книги.

В *главе 1* мы установим необходимое программное обеспечение под Windows: библиотеку MinGW и редактор Eclipse.

Глава 2 является вводной. Мы настроим среду разработки, скомпилируем и запустим первую программу — как из командной строки, так и из редактора Eclipse. Кроме того, вкратце разберемся со структурой программы, а также научимся выводить результат работы программы и получать данные от пользователя.

В *главе 3* мы познакомимся с переменными и типами данных в языке C, а в *главе 4* рассмотрим различные операторы, предназначенные для выполнения определенных действий с данными. Кроме того, в этой главе мы изучим операторы ветвления и циклы, позволяющие изменить порядок выполнения программы.

Глава 5 полностью посвящена работе с числами. Вы узнаете, какие числовые типы данных поддерживает язык C, научитесь применять различные функции, генерировать случайные числа и др.

Глава 6 познакомит с массивами в языке C. Вы научитесь создавать как одномерные массивы, так и многомерные, перебирать элементы массива, сортировать, выполнять поиск значений и др.

Глава 7 полностью посвящена работе с однобайтовыми символами и C-строками. В этой главе вы также научитесь работать с различными кодировками, настраивать локаль, выполнять форматирование строки и осуществлять поиск внутри строки. А в *главе 8* мы рассмотрим работу с широкими символами и L-строками, позволяющими использовать двухбайтовые символы из кодировки Unicode.

Глава 9 познакомит со способами работы с датой и временем.

В *главе 10* вы научитесь создавать пользовательские функции, позволяющие применять код многократно, а также объединять их в модули.

В *главе 11* мы рассмотрим способы поиска ошибок в программе и научимся отлаживать программу в редакторе Eclipse.

Главы 12–14 научат работать с файлами и каталогами, читать и записывать файлы в различных форматах.

Глава 15 познакомит с многопоточными приложениями, позволяющими значительно повысить производительность программы за счет параллельного выполнения задач несколькими потоками управления.

В *главе 16* рассматриваются способы создания статических и динамических библиотек. Статические библиотеки становятся частью программы при компиляции, а динамические библиотеки подгружаются при запуске программы или при ее выполнении.

И наконец, в *главе 17* мы рассмотрим возможность запуска функции при завершении работы приложения, научимся выполнять системные команды и получать значения переменных окружения, изучим директивы препроцессора, а также создадим значок для EXE-файла.

Все листинги из этой книги вы найдете в файле Listings.doc, электронный архив с которым можно загрузить с FTP-сервера издательства "БХВ-Петербург" по ссылке **ftp://ftp.bhv.ru/9785977541169.zip** или со страницы книги на сайте **www.bhv.ru** *(см. приложение)*.

Желаю приятного изучения и надеюсь, что книга поможет вам реализовать как самые простые, так и самые сложные приложения.

ГЛАВА 1

Установка программ под Windows

Вначале необходимо сделать два замечания.

- Во-первых, имя пользователя компьютера должно состоять только из латинских букв. Никаких русских букв и пробелов, т. к. многие программы сохраняют различные настройки и временные файлы в каталоге C:\Users\<Имя пользователя>. Если имя пользователя содержит русские буквы, то они могут быть искажены до неузнаваемости из-за неправильного преобразования кодировок, и программа не сможет сохранить настройки. Помните, что в разных кодировках русские буквы могут иметь разный код. Разработчики программ в основном работают с английским языком и ничего не знают о проблемах с кодировками, т. к. во всех однобайтовых кодировках и в кодировке UTF-8 коды латинских букв одинаковы. Так что, если хотите без проблем заниматься программированием, то от использования русских букв в имени пользователя лучше отказаться.
- Во-вторых, имена каталогов и файлов в пути не должны содержать русских букв и пробелов. Допустимы только латинские буквы, цифры, тире, подчеркивание и некоторые другие символы. С русскими буквами та же проблема, что описана в предыдущем пункте. При наличии пробелов в пути обычно требуется дополнительно заключать путь в кавычки. Если этого не сделать, то путь будет обрезан до первого встретившегося пробела. Такая проблема будет возникать при сборке и компиляции программ из командной строки.

Соблюдение этих двух простых правил позволит избежать множества проблем в дальнейшем при сборке и компиляции программ сторонних разработчиков.

1.1. Создание структуры каталогов

Перед установкой программ создадим следующую структуру каталогов:

```
book
cpp
    eclipse
    projects
    lib
```

Каталоги book и cpp лучше разместить в корне какого-либо диска. В моем случае это будет диск C:, следовательно, пути к содержимому каталогов — C:\book и C:\cpp. Можно создать каталоги в любом другом месте, но в пути не должно быть русских букв и пробелов — только латинские буквы, цифры, тире и подчеркивание. Остальных символов лучше избегать, если не хотите проблем с компиляцией и запуском программ.

В каталоге C:\book мы станем размещать наши тестовые программы и тренироваться при изучении работы с файлами и каталогами. Некоторые функции при неправильном действии без проблем могут удалить все дерево каталогов, поэтому для экспериментов мы воспользуемся отдельным каталогом, а не каталогом C:\cpp, в котором у нас будет находиться все сокровенное, — обидно, если по случайности мы удалим все установленные программы и проекты.

Внутри каталога C:\cpp у нас созданы три вложенных каталога:

- eclipse — в этот каталог мы установим редактор Eclipse;
- projects — в этом каталоге мы станем сохранять проекты из редактора Eclipse;
- lib — путь к этому каталогу мы добавим в системную переменную PATH и будем размещать в ней различные библиотеки, которые потребуются для наших программ.

1.2. Добавление пути в переменную *PATH*

Когда мы в командной строке вводим название программы без предварительного указания пути к ней, то вначале поиск программы выполняется в текущем рабочем каталоге (обычно это каталог, из которого запускается программа), а затем в путях, указанных в системной переменной PATH. Аналогично выполняется поиск библиотек динамической компоновки при запуске программы с помощью двойного щелчка на значке файла, но системные каталоги имеют более высокий приоритет, чем каталоги, указанные в переменной PATH. Пути в системной переменной PATH просматриваются слева направо до первого нахождения искомого объекта. Так что, если в путях расположено несколько объектов с одинаковыми именами, то мы получим только первый найденный объект. Поэтому, если вдруг запустилась другая программа, следует либо удалить путь, ведущий к другой программе, либо переместить новый путь в самое начало системной переменной PATH.

Давайте добавим путь к каталогу C:\cpp\lib в переменную PATH. Для того чтобы изменить системную переменную в Windows, переходим в **Параметры | Панель управления | Система и безопасность | Система | Дополнительные параметры системы**. В результате откроется окно **Свойства системы** (рис. 1.1). На вкладке **Дополнительно** нажимаем кнопку **Переменные среды**. В открывшемся окне (рис. 1.2) в списке **Системные переменные** выделяем строку с переменной Path и нажимаем кнопку **Изменить**. В следующем окне (рис. 1.3) изменяем значение в поле **Значение переменной** — для этого переходим в конец существующей строки, ставим точку с запятой, а затем вводим путь к каталогу C:\cpp\lib:

```
<Текущее значение>;C:\cpp\lib
```

Сохраняем изменения.

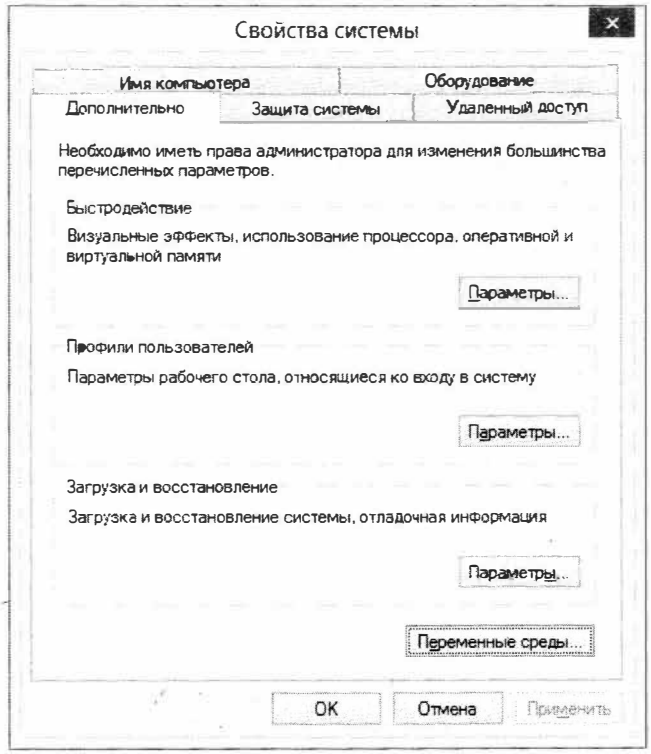


Рис. 1.1. Окно **Свойства системы**

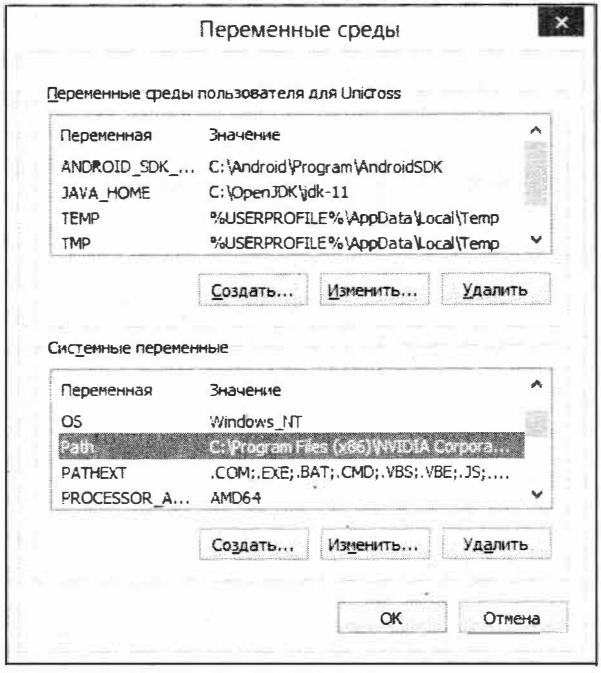


Рис. 1.2. Окно **Переменные среды**

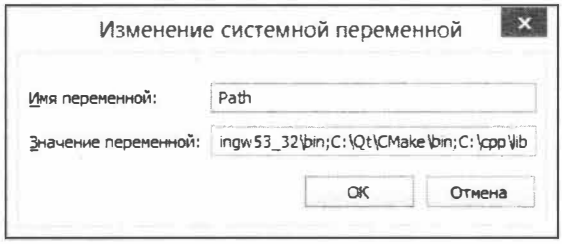


Рис. 1.3. Окно **Изменение системной переменной**

Добавлять пути в переменную PATH мы будем несколько раз, поэтому способ изменения значения этой системной переменной нужно знать наизусть.

> ***Внимание!***
> **Случайно не удалите существующее значение переменной PATH, иначе другие приложения перестанут запускаться.**

1.3. Работа с командной строкой

При изучении материала мы часто будем прибегать к помощи приложения Командная строка. Вполне возможно, что вы никогда не пользовались командной строкой и не знаете, как запустить это приложение. Давайте рассмотрим некоторые способы его запуска в Windows:

- через поиск находим приложение Командная строка;
- нажимаем комбинацию клавиш <Windows>+<R>. В открывшемся окне вводим cmd и нажимаем кнопку **OK**;
- находим файл cmd.exe в каталоге C:\Windows\System32;
- в Проводнике щелкаем правой кнопкой мыши на свободном месте списка файлов, удерживая при этом нажатой клавишу <Shift>, и из контекстного меню выбираем пункт **Открыть окно команд**;
- в Проводнике в адресной строке вводим cmd и нажимаем клавишу <Enter>.

В некоторых случаях для выполнения различных команд могут потребоваться права администратора. Для того чтобы запустить командную строку с правами администратора, через поиск находим приложение Командная строка, щелкаем на значке правой кнопкой мыши и затем выбираем пункт **Запуск от имени администратора**.

Запомните способы запуска командной строки. В дальнейшем мы просто будем говорить "запустите командную строку" без уточнения, как это сделать.

1.4. Установка MinGW и MSYS

Язык С является компилируемым языком. Для преобразования текстового файла с программой в исполняемый EXE-файл потребуется установить на компьютер специальную программу — *компилятор*. Для компиляции примеров из книги мы

воспользуемся бесплатной программой gcc.exe, входящей в состав популярной библиотеки MinGW.

Для загрузки библиотеки переходим на сайт **http://www.mingw.org/** и нажимаем кнопку **Download Installer** или переходим по ссылке **http://www.mingw.org/download/installer**. Скачиваем программу установки и запускаем файл mingw-get-setup.exe. Обратите внимание: для установки библиотеки потребуется активное подключение к Интернету. После запуска программы установки отобразится окно, показанное на рис. 1.4. Нажимаем кнопку **Install**. На следующем шаге (рис. 1.5) указываем каталог C:\MinGW и нажимаем кнопку **Continue**. После скачивания необходимых файлов отобразится окно, показанное на рис. 1.6. Нажимаем кнопку **Continue**. В следующем окне (рис. 1.7) нужно выбрать требуемые компоненты. Устанавливаем флажки **mingw32-base-bin**, **mingw32-gcc-g++-bin** и **msys-base-bin**. Затем в меню **Installation** выбираем пункт **Apply Changes** и в открывшемся окне (рис. 1.8) нажимаем кнопку **Apply**. После установки на последнем шаге (рис. 1.9) нажимаем кнопку **Close**.

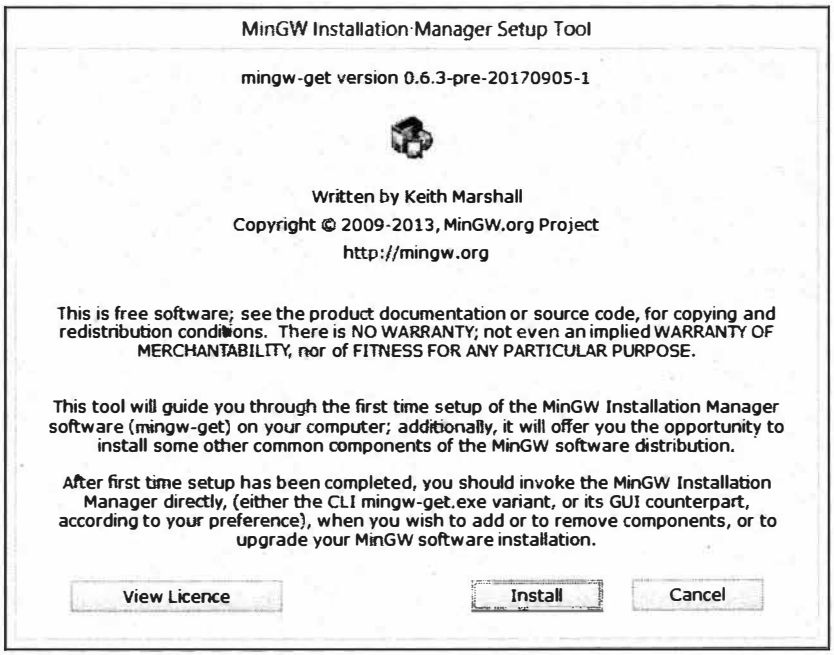


Рис. 1.4. Установка MinGW. Шаг 1

В результате библиотека MinGW будет установлена в каталог C:\MinGW. Программу gcc.exe, предназначенную для компиляции программ, написанных на языке C, можно найти в каталоге C:\MinGW\bin, а заголовочные файлы — в каталоге C:\MinGW\include.

Помимо библиотеки MinGW в каталог C:\MinGW\msys была установлена библиотека MSYS, которую редакторы используют для сборки проектов.

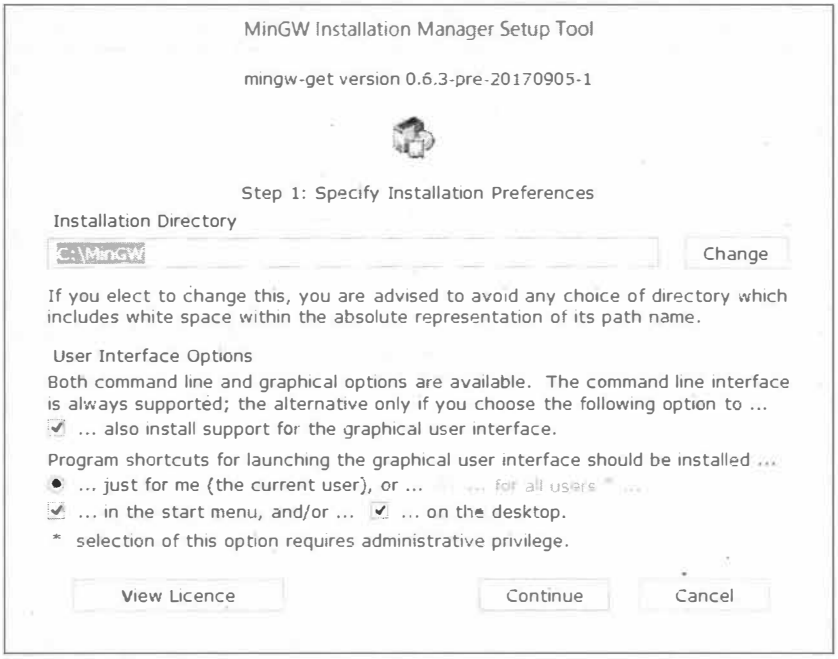


Рис. 1.5. Установка MinGW. Шаг 2

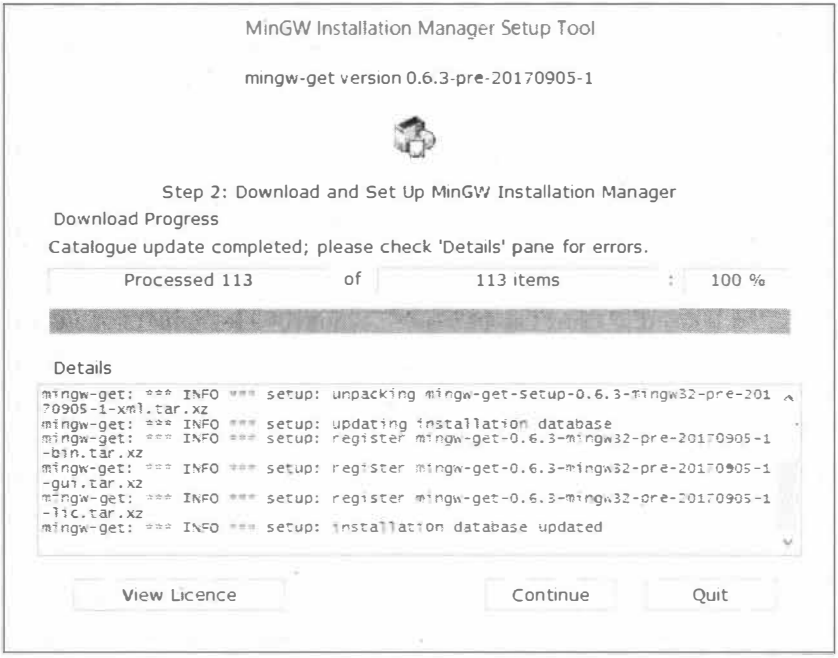


Рис. 1.6. Установка MinGW. Шаг 3

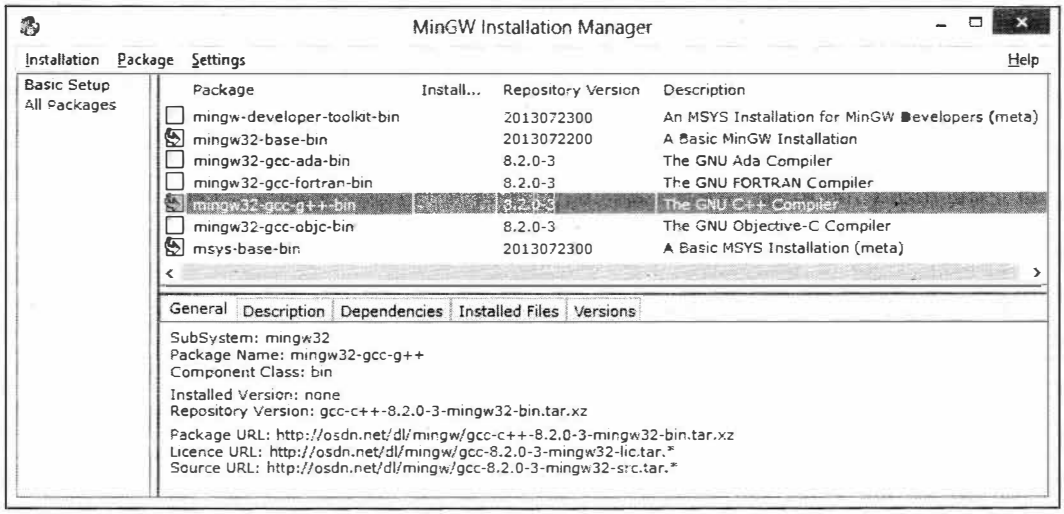


Рис. 1.7. Установка MinGW. Шаг 4

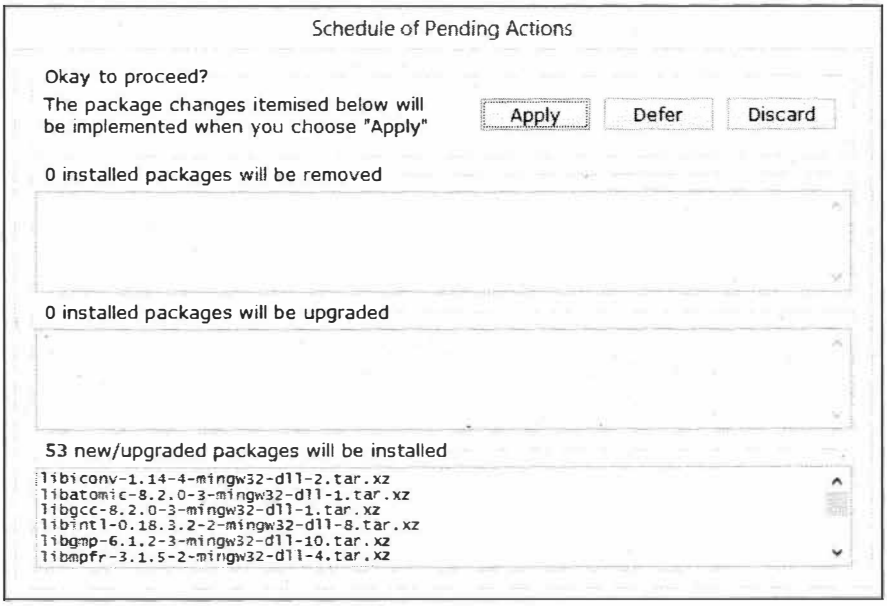


Рис. 1.8. Установка MinGW. Шаг 5

Пути к каталогам C:\MinGW\bin и C:\MinGW\msys\1.0\bin можно добавить в системную переменную PATH. Однако мы этого делать не станем, т. к. установим несколько версий компилятора. Вместо изменения переменной PATH на постоянной основе мы будем выполнять изменение в командной строке только для текущего сеанса. Продемонстрируем это на примере, а заодно проверим работоспособность компилятора. Запускаем командную строку и выполняем следующие команды:

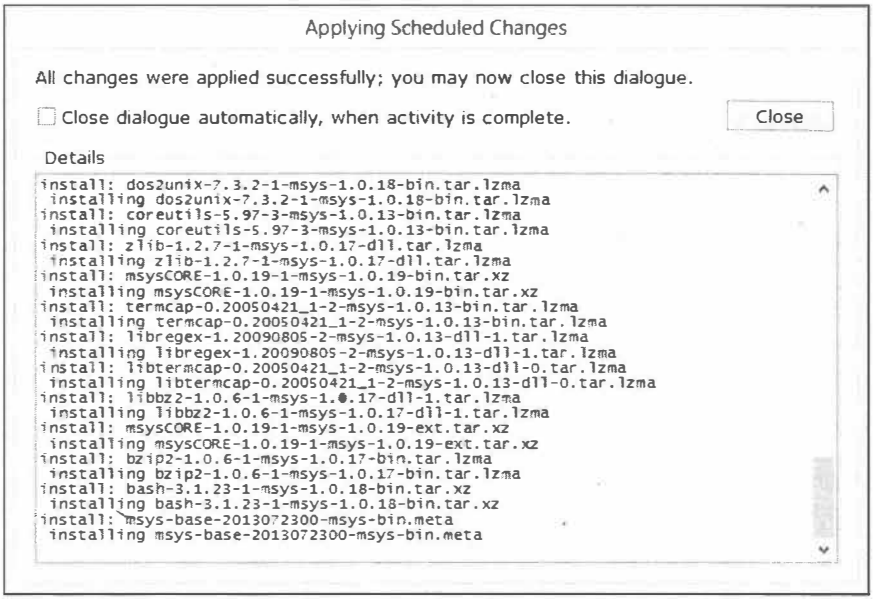


Рис. 1.9. Установка MinGW. Шаг 6

```
C:\Users\Unicross>cd C:\

C:\>set Path=C:\MinGW\bin;C:\MinGW\msys\1.0\bin;%Path%

C:\>gcc --version
gcc (MinGW.org GCC-8.2.0-3) 8.2.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

C:\>g++ --version
g++ (MinGW.org GCC-8.2.0-3) 8.2.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

Первая команда (`cd C:\`) делает текущим корневой каталог диска C:. Вторая команда (`set Path=C:\MinGW\bin;C:\MinGW\msys\1.0\bin;%Path%`) изменяет значение системной переменной `PATH` для текущего сеанса. Пути к каталогам C:\MinGW\bin и C:\MinGW\msys\1.0\bin мы добавили в самое начало переменной `PATH`. Третья команда (`gcc --version`) выводит версию программы gcc.exe. Эту программу мы будем использовать для компиляции программ, написанных на языке C. Четвертая команда (`g++ --version`) выводит версию программы g++.exe. Эту программу можно использовать для компиляции программ, написанных на языке C++. Фрагменты

перед командами означают приглашение для ввода команд. Текст после команд является результатом выполнения этих команд.

Вместо отдельных команд можно написать скрипт, который выполняет сразу несколько команд и отображает результат их работы в отдельном окне. Запускаться такой скрипт будет с помощью двойного щелчка левой кнопкой мыши на значке файла. Для создания файла потребуется текстовый редактор, позволяющий корректно работать с различными кодировками. Советую установить на компьютер редактор Notepad++. Скачать редактор можно абсолютно бесплатно со страницы **https://notepad-plus-plus.org/**. Из двух вариантов (архив и инсталлятор) советую выбрать именно инсталлятор, т. к. при установке можно будет указать язык интерфейса программы. Установка Notepad++ предельно проста и в комментариях не нуждается.

Запускаем Notepad++ и создаем новый документ. Консоль в Windows по умолчанию работает с кодировкой windows-866, поэтому мы должны и файл сохранить в этой кодировке, иначе русские буквы будут искажены. В меню **Кодировки** выбираем пункт **Кодировки | Кириллица | OEM 866**. Вводим текст скрипта (листинг 1.1) и сохраняем под названием script.bat в каталоге C:\book. Запускаем скрипт с помощью двойного щелчка левой кнопкой мыши на значке файла script.bat. Результат выполнения показан на рис. 1.10. Для закрытия окна консоли достаточно нажать любую клавишу.

Листинг 1.1. Содержимое файла script.bat

```
@echo off
title Заголовок окна
cd C:\
echo Текущий каталог: %CD%
@echo.
set Path=C:\MinGW\bin;C:\MinGW\msys\1.0\bin;%Path%
rem Вывод версии gcc.exe
echo gcc --version
@echo.
gcc --version
rem Вывод версии g++.exe
echo g++ --version
@echo.
g++ --version
pause
```

Рассмотрим инструкции из этого скрипта:

- `title` — позволяет вывести текст в заголовок окна консоли;
- `echo` — выводит текст в окно консоли;
- `@echo.` — выводит пустую строку в окно консоли;
- `%CD%` — переменная, содержащая путь к текущему рабочему каталогу;

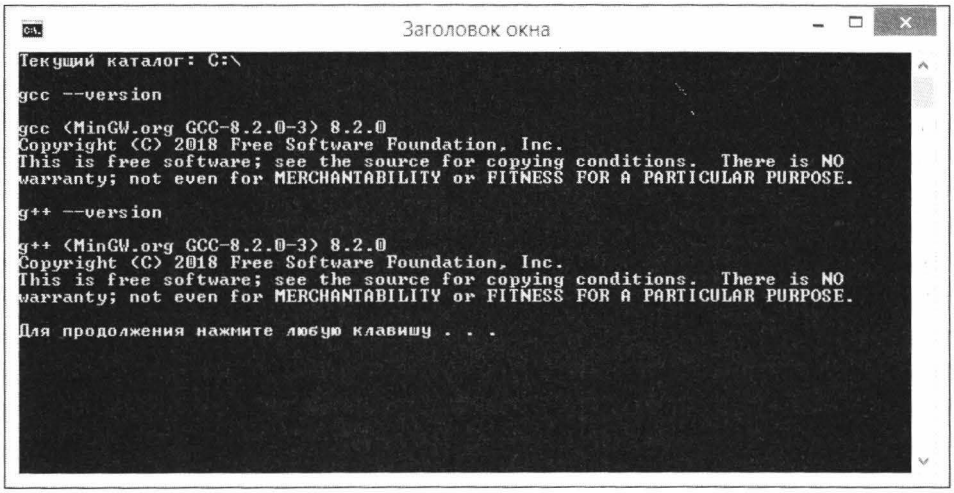


Рис. 1.10. Результат выполнения программы из листинга 1.1

- `%Path%` — переменная, содержащая значение системной переменной `PATH`;
- `rem` — вставляет комментарий, поясняющий фрагмент кода;
- `pause` — ожидает ввода любого символа от пользователя. Если эту инструкцию убрать из скрипта, то программа выполнится, и окно консоли сразу закроется, не дав нам возможности увидеть результат.

1.5. Установка MinGW-W64

Компилятор gcc.exe, установленный в предыдущем разделе в каталог C:\MinGW\bin, позволяет создавать 32-разрядные приложения. Такие приложения будут запускаться и в 32-битной операционной системе, и в 64-битной. Для того чтобы иметь возможность создавать приложения для 64-битной системы, нужно дополнительно установить библиотеку MinGW-W64.

Для загрузки библиотеки MinGW-W64 переходим на страницу **https://sourceforge.net/projects/mingw-w64/** и скачиваем файл mingw-w64-install.exe. После запуска программы установки отобразится окно, показанное на рис. 1.11. Нажимаем кнопку **Next**. В следующем окне (рис. 1.12) из списка **Version** выбираем самую новую версию (в моем случае **8.1.0**), из списка **Architecture** — пункт **x86_64**, из списка **Threads** — пункт **win32**, а из списка **Exception** — пункт **seh**. Нажимаем кнопку **Next**. На следующем шаге (рис. 1.13) задаем путь `C:\MinGW64` и нажимаем кнопку **Next**. Начнется установка библиотеки. В следующем окне (рис. 1.14) нажимаем кнопку **Next**, а на последнем шаге (рис. 1.15) — кнопку **Finish**.

В состав MinGW-W64 не входит MSYS. Программа из каталога C:\MinGW\msys настроена на библиотеку MinGW, расположенную в каталоге C:\MinGW, поэтому мы не сможем ее использовать совместно с MinGW-W64. Однако мы можем скопировать каталог C:\MinGW\msys со всем содержимым и вставить его в каталог

Рис. 1.11. Установка MinGW-W64. Шаг 1

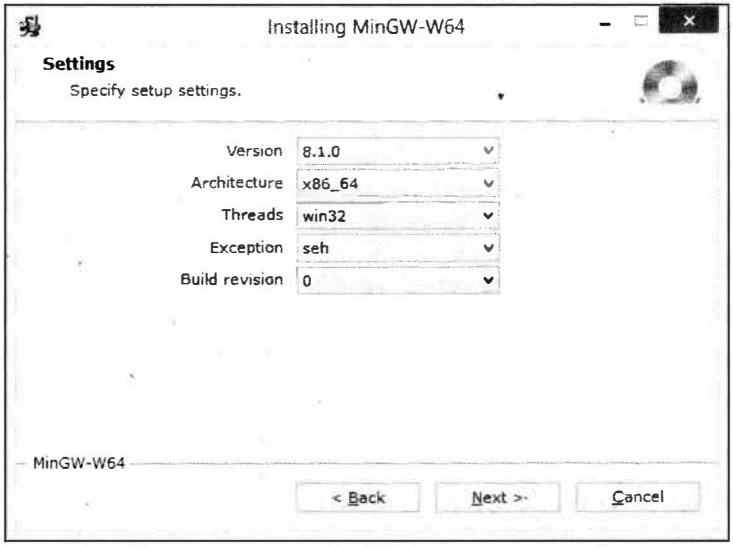


Рис. 1.12. Установка MinGW-W64. Шаг 2

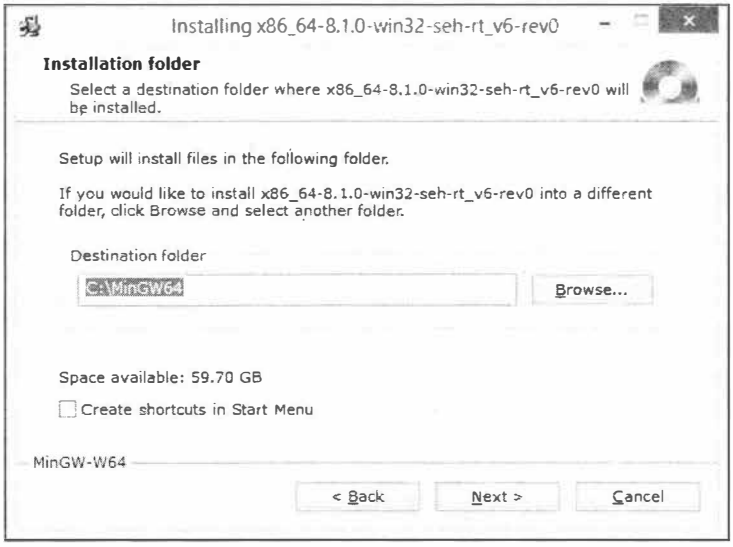


Рис. 1.13. Установка MinGW-W64. Шаг 3

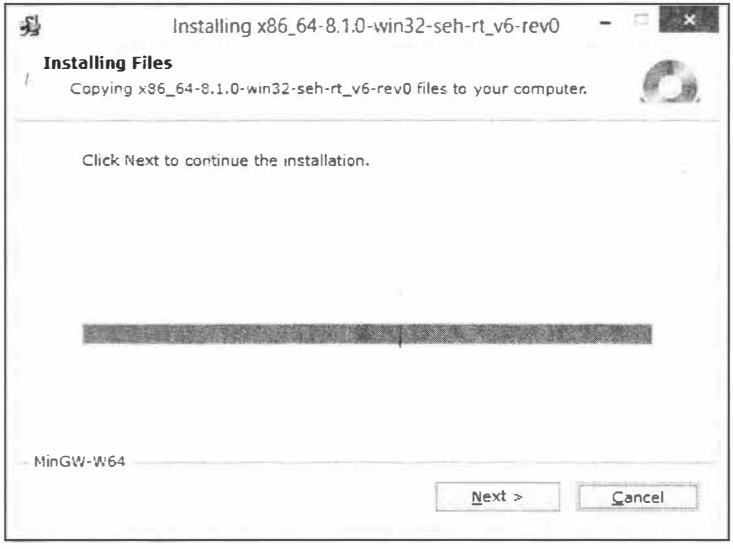


Рис. 1.14. Установка MinGW-W64. Шаг 4

Рис. 1.15. Установка MinGW-W64. Шаг 5

C:\MinGW64\mingw64\msys, а затем указать в конфигурации путь к MinGW-W64. Так мы и сделаем. В результате путь до файла make.exe должен быть таким: C:\MinGW64\mingw64\msys\1.0\bin\make.exe. Далее с помощью программы Notepad++ открываем файл C:\MinGW64\mingw64\msys\1.0\etc\fstab и находим строку:

```
C:/MinGW                /mingw
```

Изменяем ее следующим образом и сохраняем файл:

```
C:/MinGW64/mingw64                /mingw
```

Проверим правильность установки MinGW-W64. Открываем командную строку и выполняем следующие команды:

```
C:\Users\Unicross>cd C:\

C:\>set Path=C:\MinGW64\mingw64\bin;%Path%

C:\>gcc --version
gcc (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

C:\>g++ --version
g++ (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

Теперь установим компилятор для создания 32-битных приложений. Запускаем файл mingw-w64-install.exe еще раз. На втором шаге (см. рис. 1.12) из списка **Version** выбираем самую новую версию (в моем случае **8.1.0**), из списка **Architecture** — пункт **i686**, из списка **Threads** — пункт **win32**, а из списка **Exception** — пункт **dwarf**. На третьем шаге (см. рис. 1.13) задаем путь `C:\MinGW32`. После установки копируем каталог msys со всем содержимым из каталога C:\MinGW в каталог C:\MinGW32\mingw32. Далее с помощью программы Notepad++ открываем файл C:\MinGW32\mingw32\msys\1.0\etc\fstab и находим строку:

```
C:/MinGW                /mingw
```

Изменяем ее следующим образом и сохраняем файл:

```
C:/MinGW32/mingw32                /mingw
```

Проверим правильность установки. Открываем командную строку и выполняем следующие команды:

```
C:\Users\Unicross>cd C:\

C:\>set Path=C:\MinGW32\mingw32\bin;%Path%

C:\>gcc --version
gcc (i686-win32-dwarf-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

C:\>g++ --version
g++ (i686-win32-dwarf-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

1.6. Установка MSYS2 и MinGW-W64

Существует еще один способ установки компилятора. Предварительно нам нужно установить библиотеку MSYS2. Переходим на сайт **http://www.msys2.org/**, скачиваем файл msys2-x86_64-20180531.exe и запускаем его. В открывшемся окне (рис. 1.16) нажимаем кнопку **Далее**. В следующем окне (рис. 1.17) указываем каталог `C:\msys64` и нажимаем кнопку **Далее**. На следующем шаге (рис. 1.18) нажимаем кнопку **Далее**. После установки нажимаем кнопку **Далее** (рис. 1.19), а затем кнопку **Завершить** (рис. 1.20).

В итоге библиотека MSYS2 будет установлена в каталог C:\msys64. В этом каталоге расположены скрипты для запуска: msys2.exe, mingw32.exe и mingw64.exe.

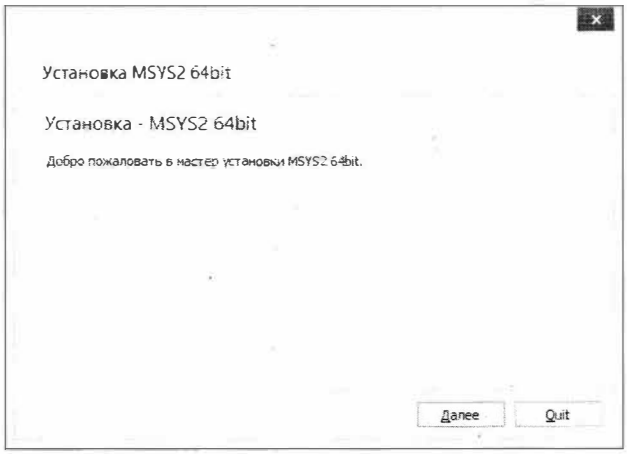


Рис. 1.16. Установка MSYS2. Шаг 1

Рис. 1.17. Установка MSYS2. Шаг 2

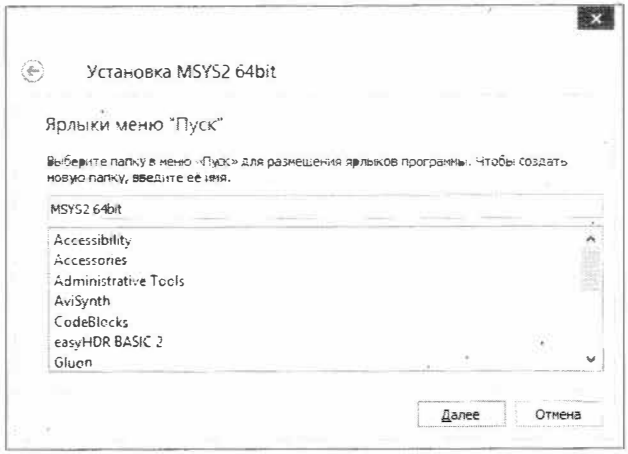


Рис. 1.18. Установка MSYS2. Шаг 3

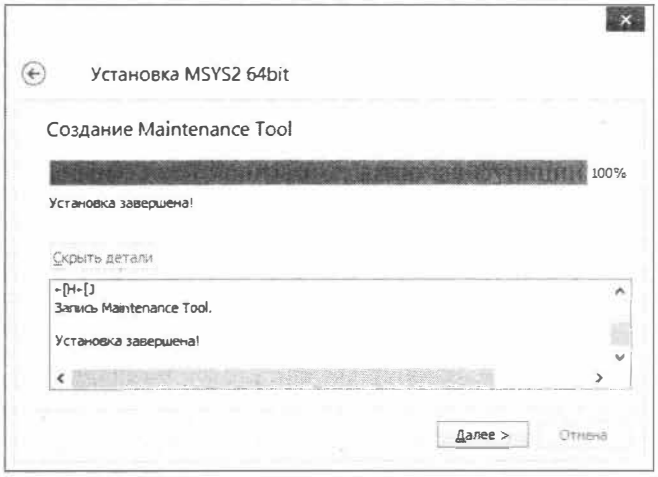


Рис. 1.19. Установка MSYS2. Шаг 4

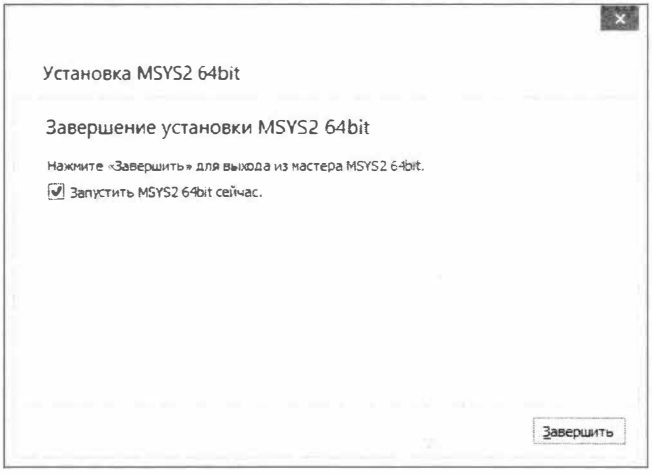


Рис. 1.20. Установка MSYS2. Шаг 5

Файл **msys2.exe** запускает командную строку, в которой мы можем установить различные библиотеки, выполнив специальные команды. Если на последнем шаге при установке вы не сбросили флажок, то это приложение запустится автоматически. Если флажок сбросили, то запускаем программу **msys2.exe** с помощью двойного щелчка левой кнопкой мыши на значке файла. Сначала обновим программу, выполнив команду (рис. 1.21):

```
pacman -Syu
```

На запрос подтверждения установки вводим букву `Y` и нажимаем клавишу <Enter>. После завершения установки закрываем окно, а затем запускаем программу **msys2.exe** снова. Вводим следующую команду:

```
pacman -Su
```

На запрос подтверждения установки вводим букву `Y` и нажимаем клавишу <Enter>.

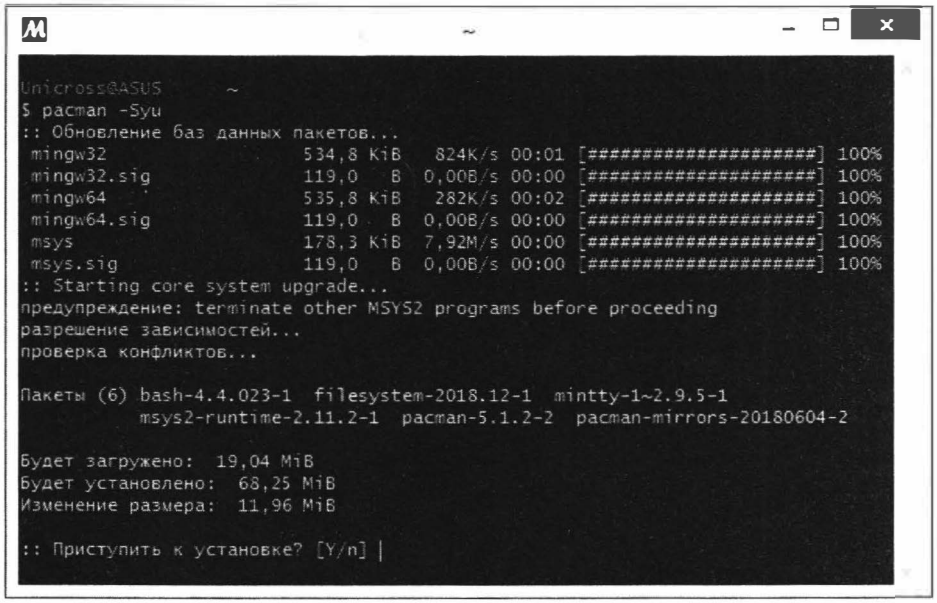


Рис. 1.21. Программа msys2.exe

Теперь можно установить библиотеку MinGW-W64. Для этого в окне вводим следующую команду:

```
pacman -S mingw-w64-x86_64-toolchain
```

Для установки всех компонентов нажимаем клавишу <Enter>, а затем на запрос подтверждения установки вводим букву Y и нажимаем клавишу <Enter>. Библиотека MinGW-W64 будет установлена в каталог C:\msys64\mingw64. Давайте проверим установку, выполнив в командной строке следующие команды:

```
C:\Users\Unicross>cd C:\

C:\>set Path=C:\msys64\mingw64\bin;%Path%

C:\>gcc --version
gcc (Rev1, Built by MSYS2 project) 8.2.1 20181214
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

C:\>g++ --version
g++ (Rev1, Built by MSYS2 project) 8.2.1 20181214
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

В командной строке мы настраивали переменную окружения PATH. Если запустить программу mingw64.exe, то переменные окружения дополнительно настраивать не нужно, достаточно сразу выполнить команду (рис. 1.22):

```
Unicross@ASUS MINGW64 ~
$ gcc --version
gcc.exe (Rev1, Built by MSYS2 project) 8.2.1 20181214
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

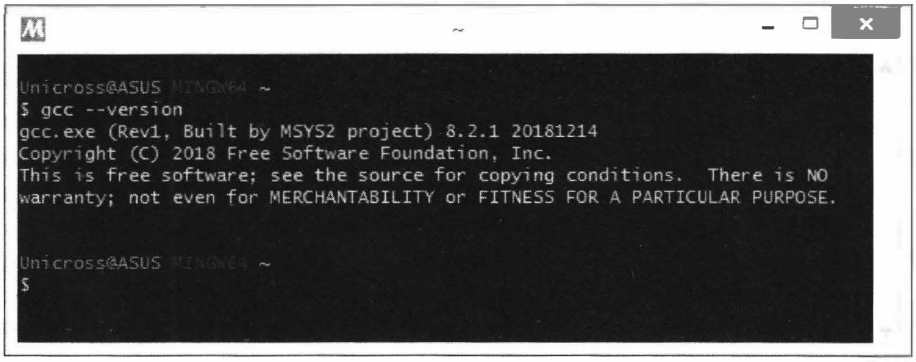


Рис. 1.22. Программа mingw64.exe

Удобство использования библиотеки MSYS2 заключается в том, что помимо установки какой-либо библиотеки, причем всего лишь одной командой, дополнительно устанавливаются все зависимости.

Давайте установим еще несколько компонентов, которые могут пригодиться:

```
pacman -S make
pacman -S mingw-w64-x86_64-cmake
pacman -S mingw-w64-x86_64-gtk3
pacman -S mingw-w64-x86_64-glade
```

Проверим установку библиотеки GTK+ (она предназначена для разработки оконных приложений на языке C). Запускаем программу C:\msys64\mingw64.exe и выполняем следующую команду:

```
pkg-config --libs gtk+-3.0
```

Результат:

```
-LC:/msys64/mingw64/lib -lgtk-3 -lgdk-3 -lgdi32 -limm32 -lshell32 -lole32
-Wl,-luuid -lwinmm -ldwmapi -lsetupapi -lcfgmgr32 -lz -lpangowin32-1.0
-lpangocairo-1.0 -lpango-1.0 -latk-1.0 -lcairo-gobject -lcairo
-lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lglib-2.0 -lintl
```

Все установленные библиотеки скомпилированы под 64-битные операционные системы. Для установки 32-битных версий библиотек нужно в команде заменить

фрагмент `x86_64` фрагментом `i686`. Пример команды для установки 32-битного компилятора:

```
pacman -S mingw-w64-i686-gcc
```

Компилятор будет установлен в каталог C:\msys64\mingw32. Для выполнения команд можно воспользоваться программой C:\msys64\mingw32.exe. Например, получим версию компилятора:

```
$ gcc --version
gcc.exe (Rev1, Built by MSYS2 project) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

Обратите внимание: мы установили только компилятор. Для того чтобы установить все компоненты, нужно воспользоваться командой:

```
pacman -S mingw-w64-i686-toolchain
```

Мы будем учиться создавать приложения на языке C под 64-битные операционные системы, поэтому можно все компоненты для 32-битного компилятора дополнительно не устанавливать, тем более что мы уже установили более новую версию компилятора в *разд. 1.5*.

1.7. Установка и настройка редактора Eclipse

В этой книге для создания и компиляции программ на языке C мы воспользуемся редактором Eclipse. Редактор удобен в использовании, он позволяет автоматически закончить слово, подчеркнуть код с ошибкой, подсветить код программы, вывести список всех функций, отладить программу, а также скомпилировать все файлы проекта всего лишь нажатием одной кнопки без необходимости использования командной строки.

Для загрузки редактора Eclipse переходим на страницу: **https://www.eclipse.org/downloads/packages/** и скачиваем архив с программой из раздела **Eclipse IDE for C/C++ Developers**. В моем случае архив называется eclipse-cpp-2018-12-R-win32-x86_64.zip. Распаковываем скачанный архив и копируем каталог eclipse с файлами редактора в каталог C:\cpp. Редактор не нуждается в установке, поэтому просто переходим в каталог C:\cpp\eclipse и запускаем файл eclipse.exe. Однако не торопитесь прямо сейчас запускать редактор.

Редактор Eclipse написан на языке Java, поэтому предварительно нужно установить на компьютер либо *Java Runtime Environment* (*JRE*), либо *Java Development Kit* (*JDK*). JRE используется для запуска приложений, а JDK — как для запуска, так и для разработки приложений на языке Java. Для того чтобы узнать, какая версия Java требуется для работы редактора Eclipse, открываем конфигурационный файл C:\cpp\eclipse\eclipse.ini и смотрим значение опции `requiredJavaVersion`. В моем случае опция имеет такое значение:

```
-Dosgi.requiredJavaVersion=1.8
```

Значение `1.8` указывает на то, что нужна Java 8. Устанавливаем на компьютер требуемую версию Java и задаем путь к ней с помощью опции `-vm`. Самый простой способ — создать ярлык для eclipse.exe, отобразить окно **Свойства** и в поле **Объект** указать такое значение (замените фрагмент `<Путь к JRE>` реальным значением):

```
C:\cpp\eclipse\eclipse.exe -vm "<Путь к JRE>\bin\server\jvm.dll"
```

Можно использовать и более новую версию, например Java 10, но не выше, т. к. в версии 11 некоторые пакеты были удалены. Пример указания пути к Java 10:

```
C:\cpp\eclipse\eclipse.exe
-vm "C:\Program Files\Java\jre-10\bin\server\jvm.dll"
```

> ***Примечание***
>
> **Все способы указания значения подробно описаны на странице http://wiki.eclipse.org/Eclipse.ini.**

При запуске редактор попросит указать каталог с рабочим пространством (рис. 1.23). Указываем `C:\cpp\projects` и нажимаем кнопку **Launch**.

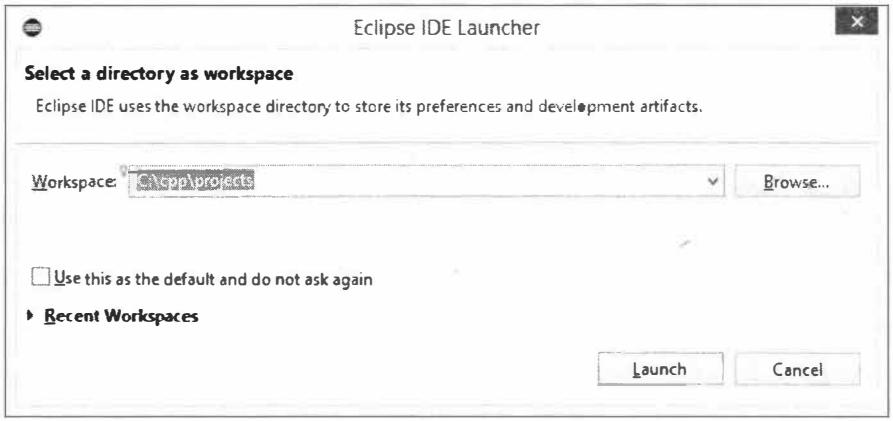


Рис. 1.23. Окно **Eclipse IDE Launcher**

Прежде чем пользоваться редактором, нужно изменить некоторые настройки по умолчанию. Начнем с кодировки файлов, в которых мы будем вводить текст программы. Для того чтобы указать кодировку по умолчанию для всех файлов, в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим в раздел **General | Workspace** (рис. 1.24). В группе **Text file encoding** устанавливаем флажок **Other** и в текстовое поле вводим `Cp1251`. Сохраняем изменения. Таким образом, мы будем пользоваться кодировкой windows-1251, которая по умолчанию используется в русской версии Windows.

Если необходимо изменить кодировку уже открытого файла, в меню **Edit** выбираем пункт **Set Encoding**. Если же потребуется указать другую кодировку для всех файлов проекта, в меню **Project** выбираем пункт **Properties**. В открывшемся окне

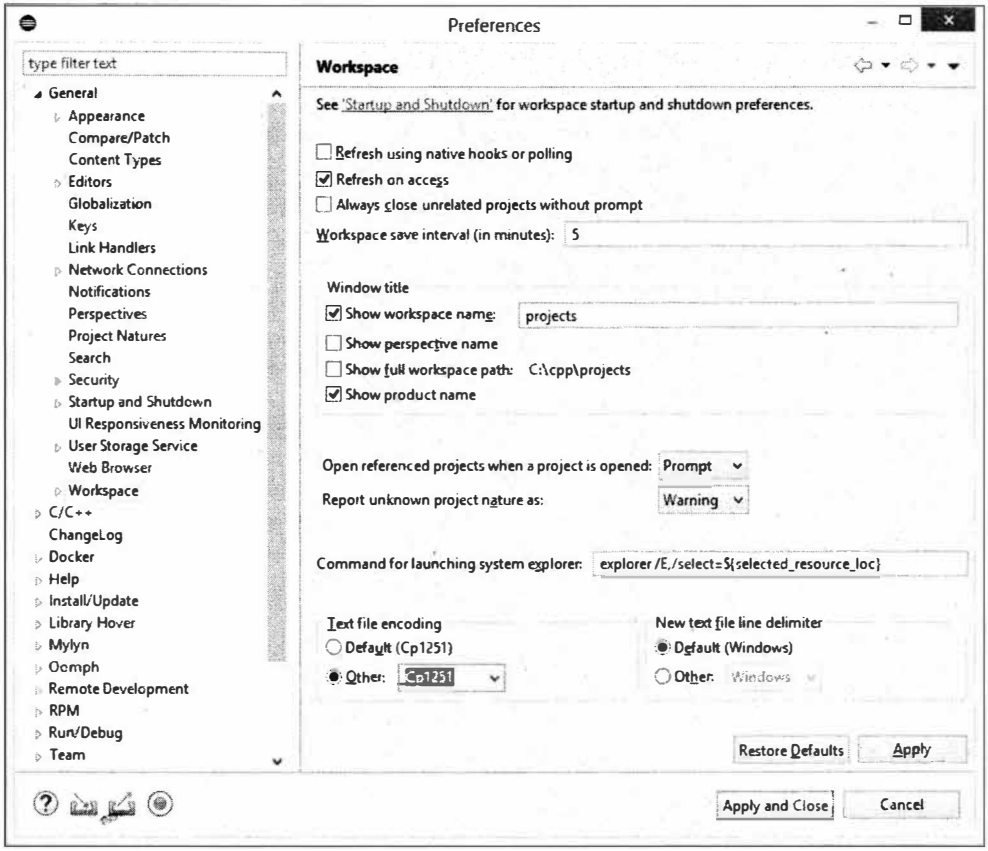


Рис. 1.24. Указание кодировки файлов по умолчанию

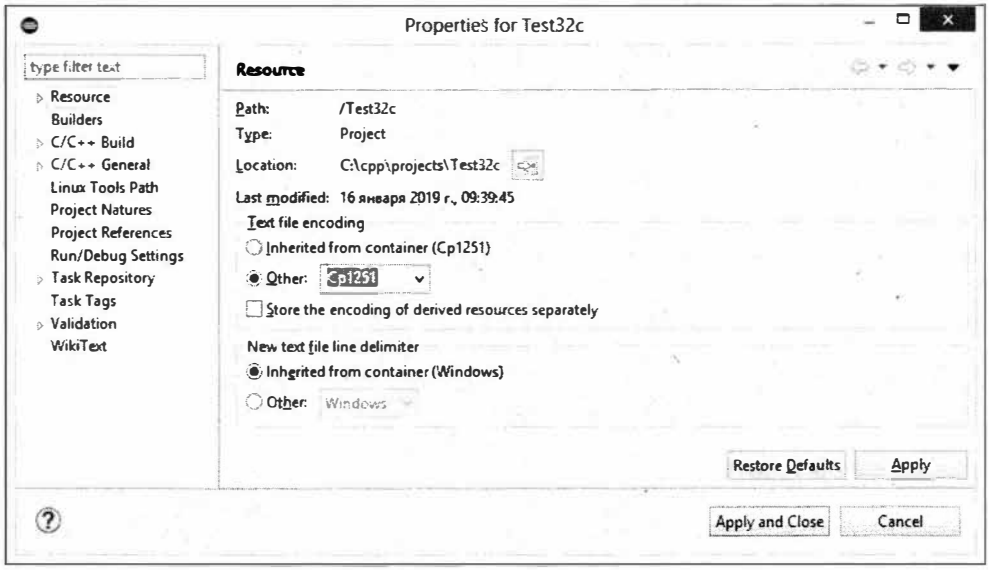


Рис. 1.25. Указание кодировки файлов для проекта

(рис. 1.25) слева в списке выбираем пункт **Resource**, а затем справа в группе **Text file encoding** указываем нужную кодировку.

По умолчанию редактор вместо пробелов вставляет символы табуляции. Нас это не устраивает. Давайте изменим настройку форматирования кода. Для этого в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим в раздел **C/C++ | Code Style | Formatter** (рис. 1.26). Нажимаем кнопку **New**. В открывшемся окне (рис. 1.27) в поле **Profile name** вводим название стиля, например `MyStyle`, а из списка выбираем пункт **K&R [built-in]**. Нажимаем кнопку **OK**. Откроется окно (рис. 1.28), в котором можно изменить настройки нашего стиля. На вкладке **Indentation** из списка **Tab policy** выбираем пункт **Spaces only**, а в поля **Indentation size** и **Tab size** вводим число 3. Сохраняем все изменения.

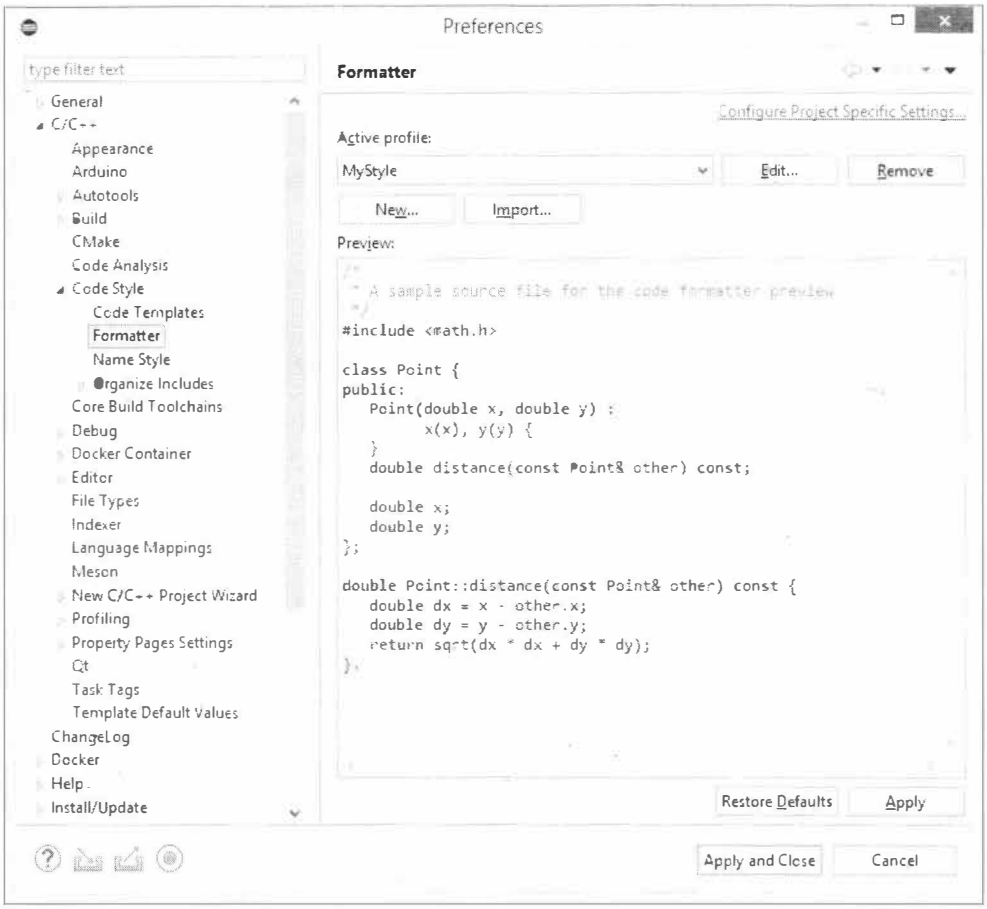


Рис. 1.26. Окно **Preferences**, раздел **Formatter**

Если необходимо изменить размер шрифта, то в меню **Window** выбираем пункт **Preferences**. В открывшемся окне переходим в раздел **General | Appearance | Colors and Fonts** (рис. 1.29). Из списка выбираем пункт **C/C++ | Editor | C/C++ Editor Text Font** и нажимаем кнопку **Edit**.

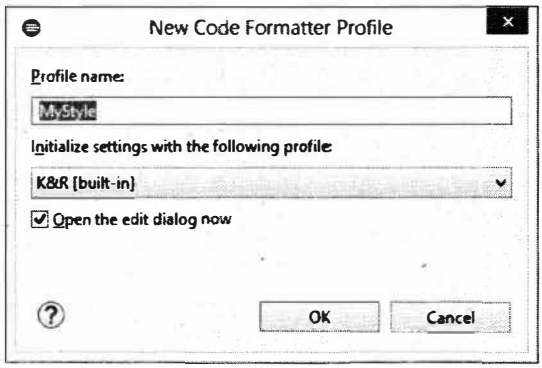


Рис. 1.27. Окно **New Code Formatter Profile**

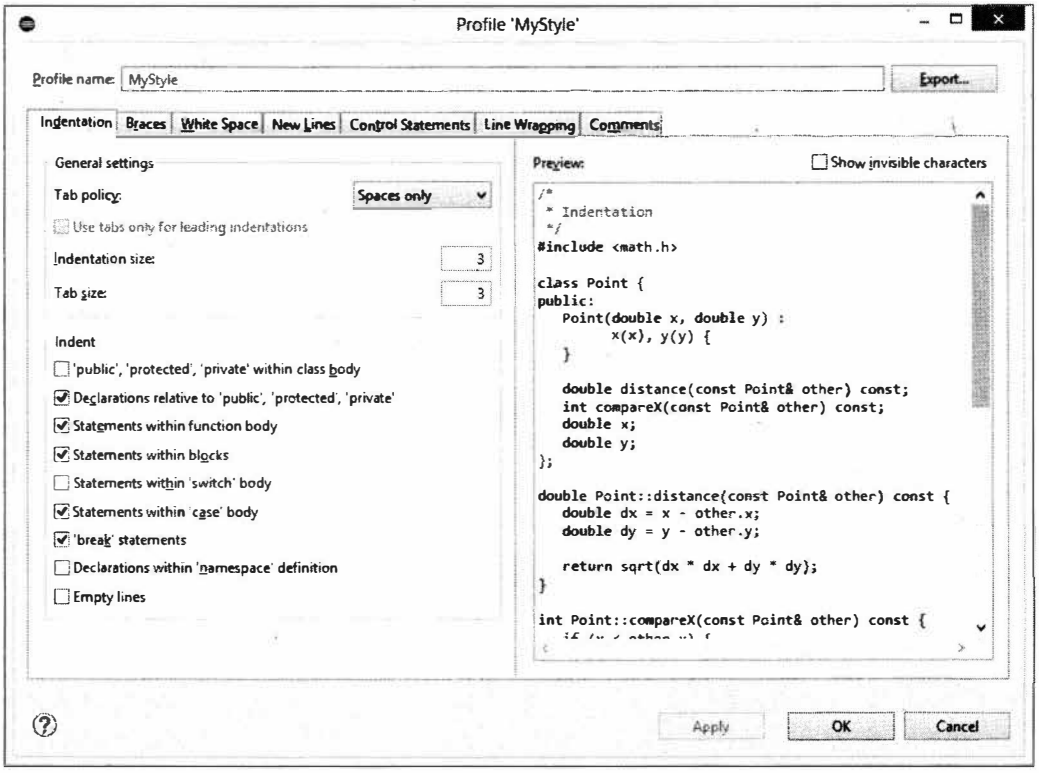


Рис. 1.28. Изменение настроек форматирования

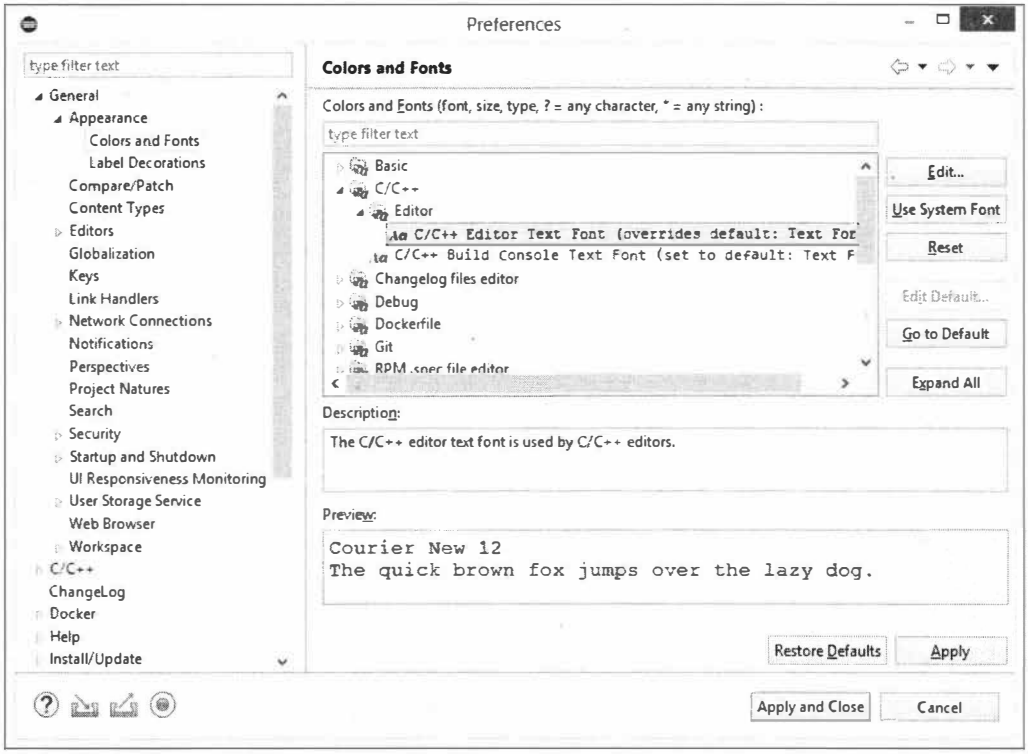


Рис. 1.29. Окно **Preferences**, раздел **Colors and Fonts**

1.8. Создание проектов в редакторе Eclipse

Давайте в редакторе Eclipse создадим два проекта. Первый проект (с названием `Test32c`) мы будем использовать для создания 32-битных программ (компилятор из каталога C:\MinGW32\mingw32\bin), а второй проект (с названием `Test64c`) — для создания 64-битных программ (компилятор из каталога C:\msys64\mingw64\bin).

Для создания проекта в меню **File** выбираем пункт **New | Project**. В открывшемся окне (рис. 1.30) в списке выбираем пункт **C/C++ | C Project** и нажимаем кнопку **Next**. На следующем шаге (рис. 1.31) в поле **Project name** вводим `Test32c`, в списке слева выбираем пункт **Executable | Hello World ANSI C Project**, а в списке справа — пункт **MinGW GCC**. Если в списке справа отсутствует пункт **MinGW GCC**, то редактор не смог автоматически обнаружить библиотеку. В этом случае следует сбросить флажок **Show project types and toolchains**, и пункт станет доступен. После нажатия кнопки **Next** отобразится окно (рис. 1.32), в котором можно ввести имя автора, информацию об авторских правах, текст, который будет выведен на консоль, и каталог, в который будут сохраняться файлы. Нажимаем кнопку **Next**. В следующем окне (рис. 1.33) нажимаем кнопку **Finish** для создания проекта.

В результате в каталоге C:\cpp\projects будет создан каталог Test32c, содержащий файлы проекта. В каталоге C:\cpp\projects\Test32c\src можно найти файл

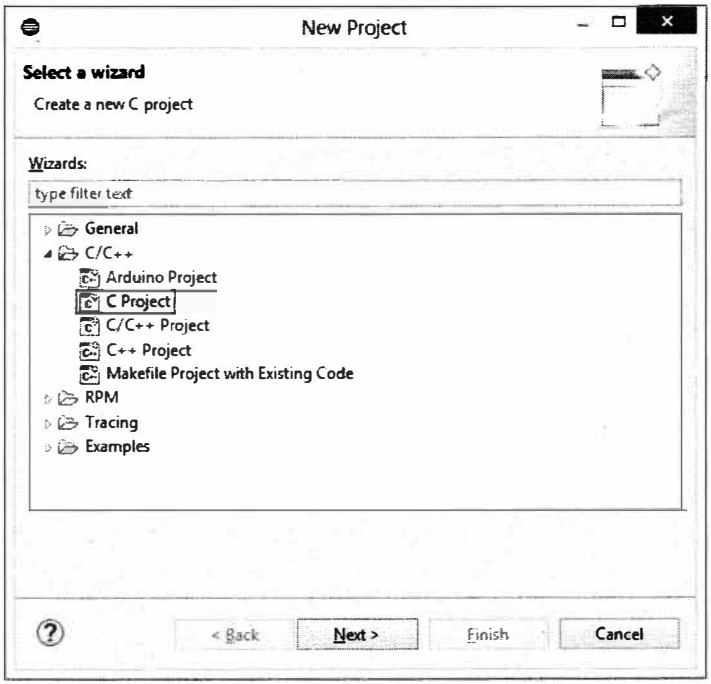


Рис. 1.30. Создание проекта. Шаг 1

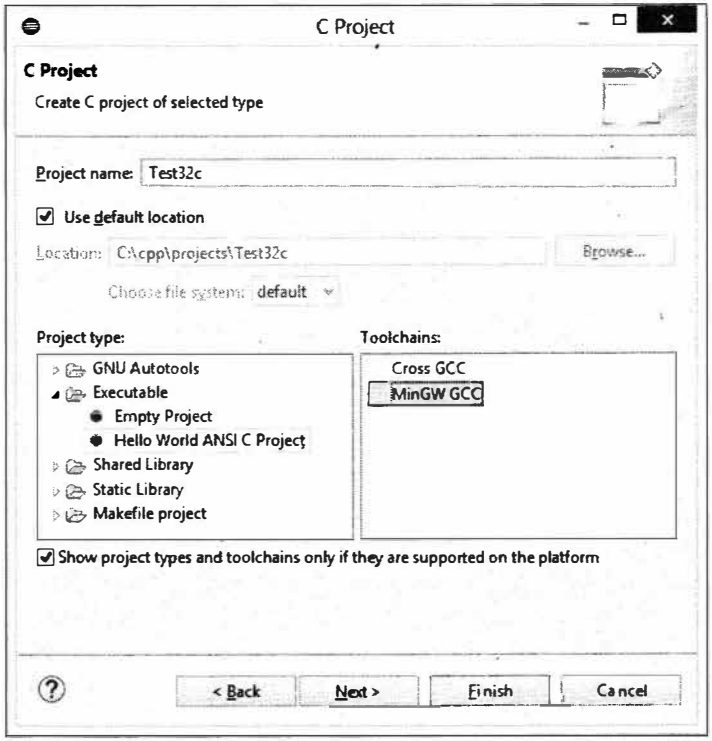


Рис. 1.31. Создание проекта. Шаг 2

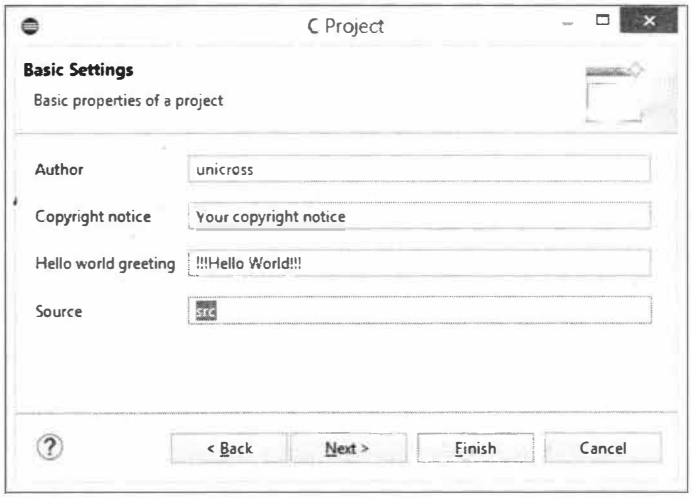


Рис. 1.32. Создание проекта. Шаг 3

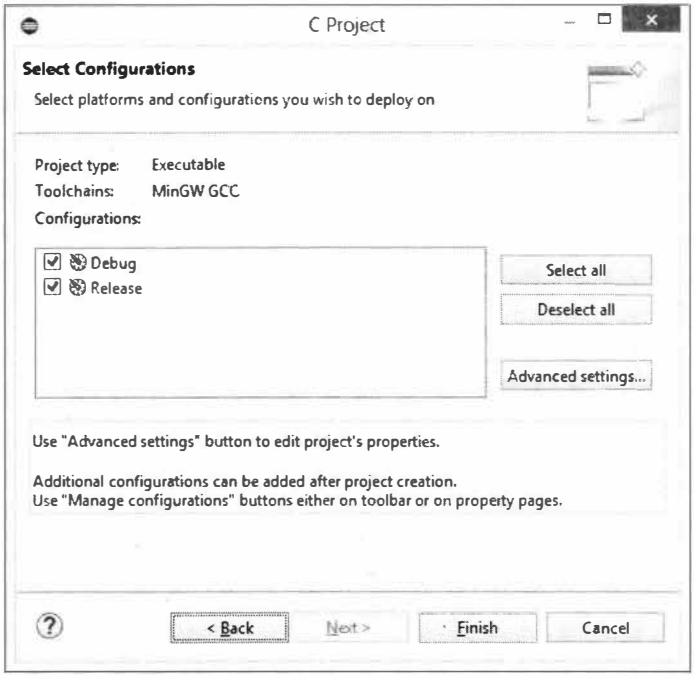


Рис. 1.33. Создание проекта. Шаг 4

Test32c.c, в который редактор уже вставил тестовую программу на языке C, выводящую надпись на консоль. Открыть этот файл можно с помощью любого текстового редактора, например с помощью Notepad++. Однако изменять его лучше в редакторе Eclipse, ведь его содержимое доступно на вкладке. Помимо файла Test32c.c редактор создал вспомогательные каталоги и файлы, названия которых начинаются с точки. В этих каталогах и файлах редактор сохраняет различные настройки. Не изменяйте и не удаляйте эти файлы.

Давайте отобразим свойства проекта и рассмотрим настройки компилятора. Для этого делаем текущей вкладку с содержимым файла Test32c.c и в меню **Project** выбираем пункт **Properties**. В открывшемся окне (рис. 1.34) в списке слева выбираем пункт **C/C++ Build | Environment**. На отобразившейся вкладке в списке прописаны следующие основные переменные окружения:

- `MINGW_HOME` — задает путь к библиотеке MinGW;
- `MSYS_HOME` — задает путь к библиотеке MSYS;
- `PATH` — в начало текущего значения переменной `PATH` добавляется путь к библиотекам (`${MINGW_HOME}\bin;${MSYS_HOME}\bin;`).

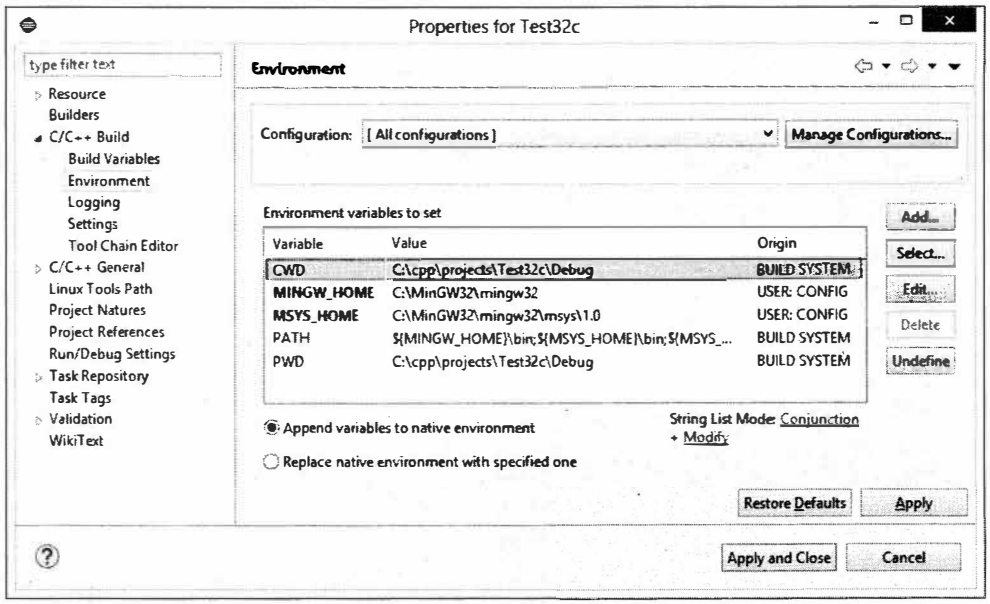


Рис. 1.34. Переменные окружения в настройках проекта

Если внимательно посмотреть на значения переменных окружения `MINGW_HOME` и `MSYS_HOME`, то можно заметить, что редактор Eclipse отдал предпочтение компилятору из каталога C:\msys64\mingw64\bin, а нам нужно, чтобы файлы проекта компилировались компилятором из каталога C:\MinGW32\mingw32\bin. Давайте изменим значения переменных окружения следующим образом (предварительно из списка **Configuration** выбираем пункт **All configurations**, а лишь затем изменяем значения переменных окружения):

- `MINGW_HOME` — указываем путь C:\MinGW32\mingw32;
- `MSYS_HOME` — указываем путь C:\MinGW32\mingw32\msys\1.0.

По умолчанию для кодирования символов в L-строках MinGW использует кодировку UTF-8, а файлы нашего проекта сохраняются в кодировке windows-1251. Если мы попробуем указать русские буквы при инициализации L-строки (`L"Строка"`), то получим ошибку. Для того чтобы избежать ошибок, нужно с помощью флага

`-finput-charset` указать компилятору кодировку исходного файла, а с помощью флага `-fexec-charset` — кодировку С-строк. Для этого в свойствах проекта из списка слева выбираем пункт **C/C++ Build | Settings**. На отобразившейся вкладке из списка **Configuration** выбираем пункт **All configurations**, а затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Miscellaneous** (рис. 1.35). В поле **Other flags** через пробел к имеющемуся значению добавляем следующие инструкции:

```
-finput-charset=cp1251 -fexec-charset=cp1251
```

Содержимое поля **Other flags** после изменения должно быть таким:

```
-c -fmessage-length=0 -finput-charset=cp1251 -fexec-charset=cp1251
```

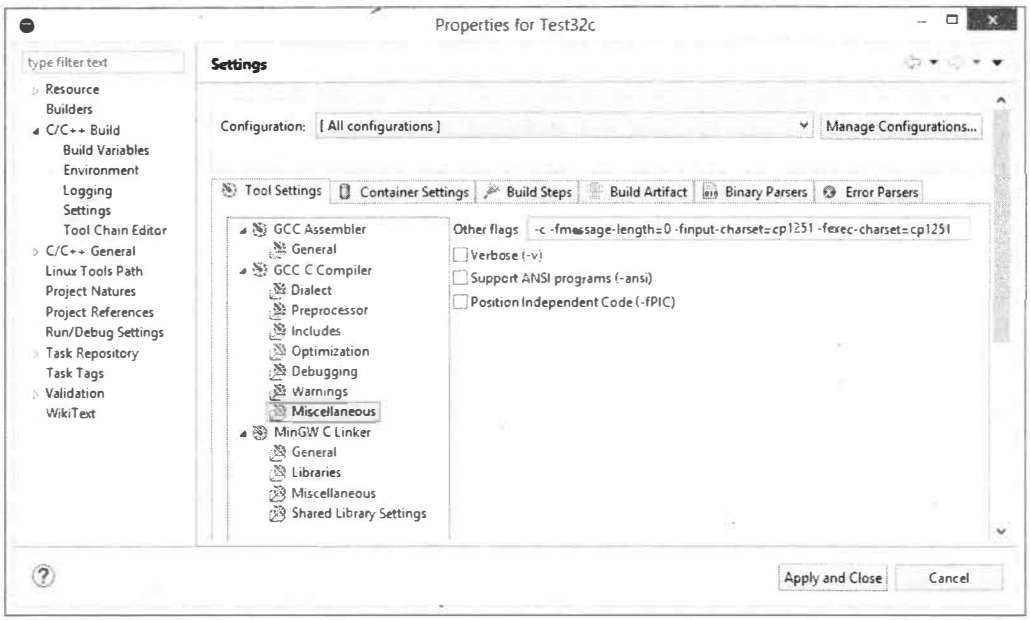


Рис. 1.35. Указание кодировки исходного файла и кодировки С-строк

Далее на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Warnings**. Устанавливаем флажок **–Wconversion** (рис. 1.36). Проверяем также установку флажка **–Wall**. Сохраняем настройки проекта, нажимая кнопку **Apply and Close**.

Для того чтобы преобразовать текстовый файл Test32c.c с программой в исполняемый EXE-файл, делаем текущей вкладку с содержимым файла Test32c.c и в меню **Project** выбираем пункт **Build Project**. В окне **Console** отобразятся инструкции компиляции и результат ее выполнения. Если компиляция прошла успешно, то никаких сообщений об ошибках в этом окне не будет:

```
12:22:53 **** Incremental Build of configuration Debug for project
Test32c ****
Info: Internal Builder is used for build
```

```
gcc -O0 -g3 -Wall -Wconversion -c -fmessage-length=0
-finput-charset=cp1251 -fexec-charset=cp1251 -o "src\\Test32c.o"
"..\\src\\Test32c.c"
gcc -o Test32c.exe "src\\Test32c.o"

12:22:53 Build Finished (took 344ms)
```

Если флаги `-Wall`, `-Wconversion`, `-finput-charset` или `-fexec-charset` отсутствуют в команде компиляции, то вы неправильно настроили свойства проекта.

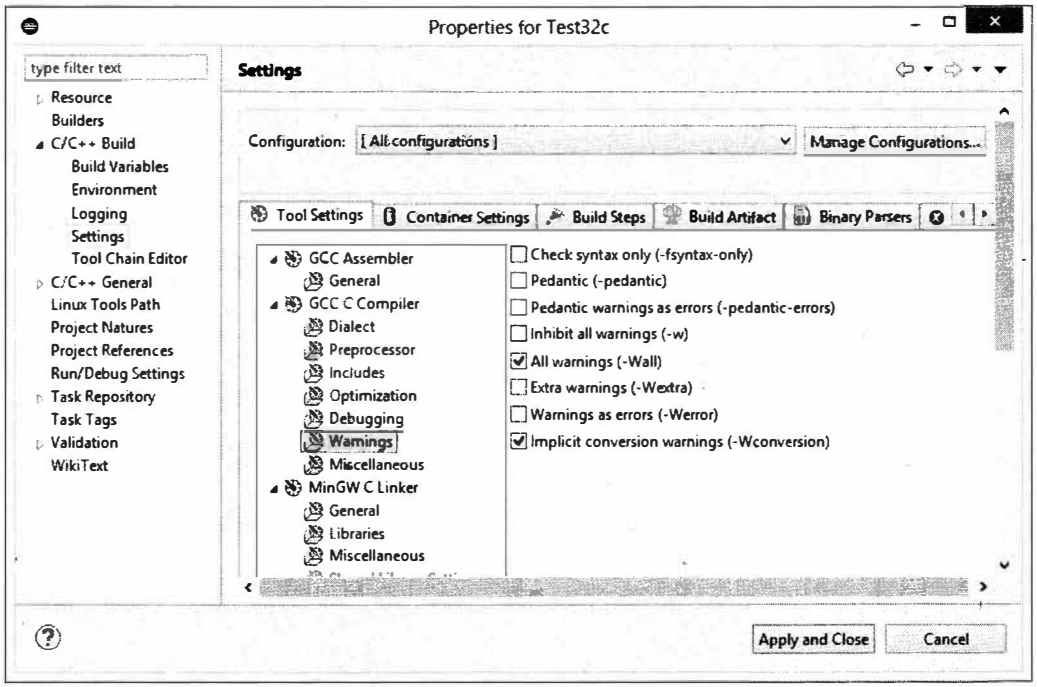


Рис. 1.36. Настройка вывода предупреждающих сообщений

В результате компиляции в каталоге C:\cpp\projects\Test32c был создан каталог Debug. Внутри этого каталога находится файл Test32c.exe, который можно запустить на выполнение с помощью двойного щелчка мыши на значке файла. Если попробовать сделать это сейчас, то окно откроется, а затем сразу закроется. Как сделать, чтобы окно сразу не закрывалось, мы рассмотрим немного позже. Сейчас же попробуем запустить программу в редакторе Eclipse. Для запуска делаем текущей вкладку с содержимым файла Test32c.c и в меню **Run** выбираем пункт **Run**. В открывшемся окне (рис. 1.37) выбираем пункт **Local C/C++ Application** и нажимаем кнопку **OK**. Результат выполнения программы отобразится в окне **Console**.

Как видите, компиляция и запуск выполняются в редакторе выбором пункта в меню. Для более быстрого доступа к этим пунктам можно воспользоваться кнопками на панели инструментов. Для того чтобы выполнить компиляцию, нажимаем кнопку с изображением молотка (рис. 1.38), а чтобы запустить программу на выполне-

ние — кнопку с зеленым кругом и белым треугольником внутри него. Причем нажатие кнопки запуска автоматически приводит к компиляции, если были произведены изменения в тексте программы. Согласитесь, очень просто и удобно.

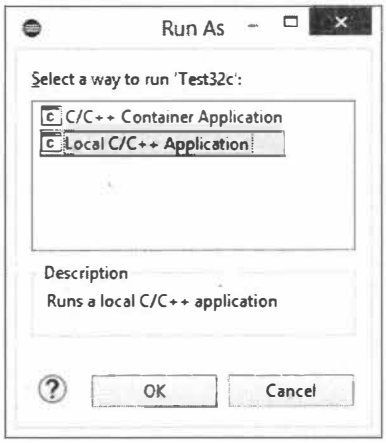


Рис. 1.37. Окно **Run As**

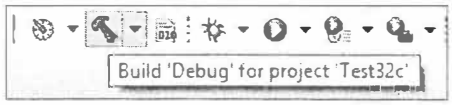


Рис. 1.38. Кнопки на панели инструментов, предназначенные для компиляции и запуска программы

По аналогии с проектом `Test32c` создаем еще один проект с названием `Test64c`. Мы будем использовать этот проект для создания 64-битных приложений. Делаем текущей вкладку с содержимым файла **Test64c.c** и в меню **Project** выбираем пункт **Properties**. В открывшемся окне (рис. 1.39) в списке слева выбираем пункт **C/C++ Build | Environment**. На отобразившейся вкладке из списка **Configuration** выбираем пункт **All configurations**, а затем проверяем значения следующих переменных

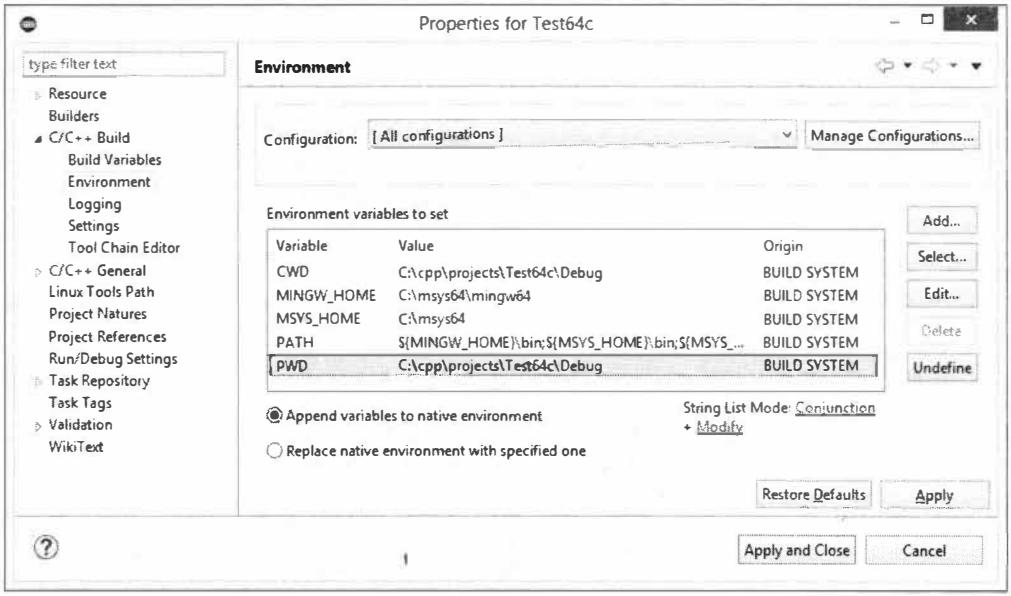


Рис. 1.39. Настройки проекта `Test64c`

окружения (в моем случае редактор Eclipse выбрал этот компилятор самостоятельно):

- `MINGW_HOME` — C:\msys64\mingw64;
- `MSYS_HOME` — C:\msys64.

Далее в свойствах проекта из списка слева выбираем пункт **C/C++ Build | Settings**. На отобразившейся вкладке из списка **Configuration** выбираем пункт **All configurations**, а затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Miscellaneous** (см. рис. 1.35). В поле **Other flags** через пробел к имеющемуся значению добавляем следующие инструкции:

```
-finput-charset=cp1251 -fexec-charset=cp1251
```

Содержимое поля **Other flags** после изменения должно быть таким:

```
-c -fmessage-length=0 -finput-charset=cp1251 -fexec-charset=cp1251
```

Затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Warnings**. Устанавливаем флажок **–Wconversion**. Проверяем также установку флажка **–Wall**. Сохраняем настройки проекта, нажимая кнопку **Apply and Close**.

Давайте сравним два проекта в окне **Project Explorer** (рис. 1.40). Во-первых, обратите внимание на пункт **Includes**. Пути поиска заголовочных файлов отличаются и соответствуют выбранным компиляторам. Во-вторых, после имени EXE-файла в проекте `Test32c` указано значение `[x86/le]`, а в проекте `Test64c` — `[amd64/le]`. Это означает, что проект `Test64c` успешно настроен на создание 64-битных программ, и мы можем начинать изучение языка C.

Рис. 1.40. Окно Project Explorer

Итак, мы установили и настроили программы, позволяющие создавать приложения на двух языках: C и C++. Язык C мы будем изучать в этой книге, начиная со следующей главы. Язык C++ в этой книге мы изучать не станем, но временами будем о нем вспоминать, т. к. он поддерживает все возможности языка C.

Рекомендую дополнительно установить бесплатную версию Visual Studio. После установки компилятор из Visual Studio можно использовать в редакторе Eclipse, но нужно будет установить плагин **C/C++ Visual C++ Support**. Для этого в редакторе Eclipse в меню **Help** выбираем пункт **Install New Software** (для установки плагина потребуется активное подключение к Интернету). В результате откроется окно, показанное на рис. 1.41. Из списка **Work with** выбираем пункт **2018-12**. После загрузки списка плагинов устанавливаем флажок напротив пункта **C/C++ Visual C++ Support**. Нажимаем кнопку **Next**. В следующем окне будут показаны только выбранные плагины. Нажимаем кнопку **Next**. На последнем шаге нужно принять лицензионное соглашение и нажать кнопку **Finish** для начала установки. После установки следует перезапустить редактор Eclipse. Теперь при создании проекта станет доступным пункт **Microsoft Visual C++** (рис. 1.42).

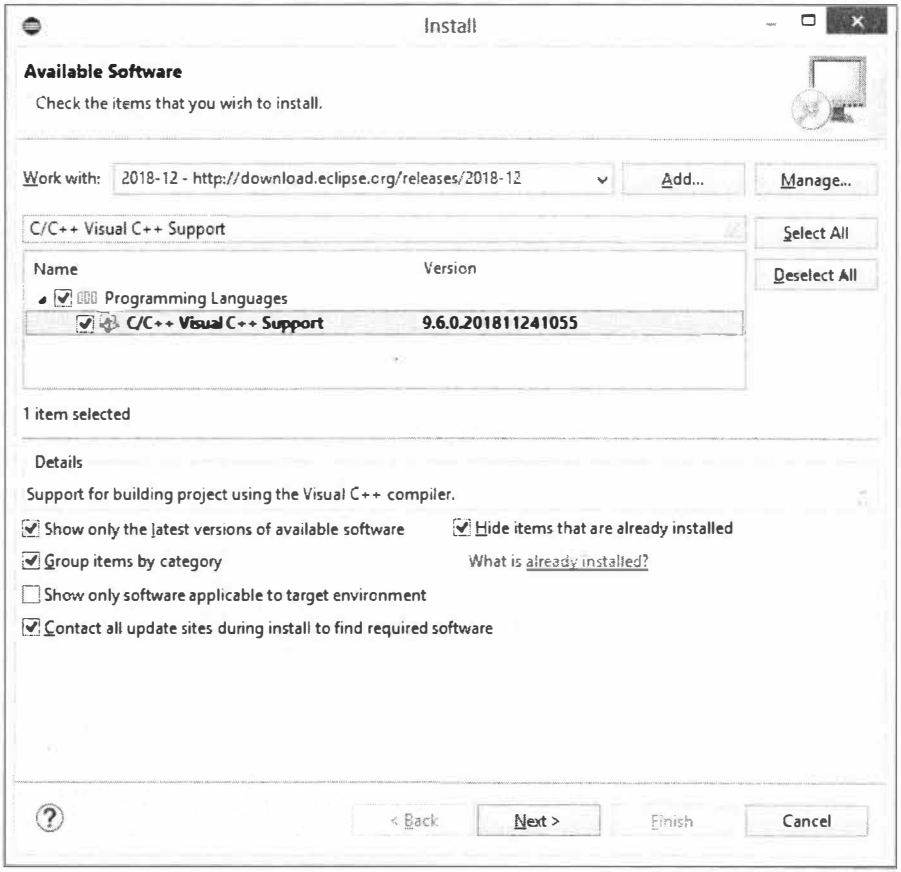


Рис. 1.41. Окно Install: выбор плагина C/C++ Visual C++ Support

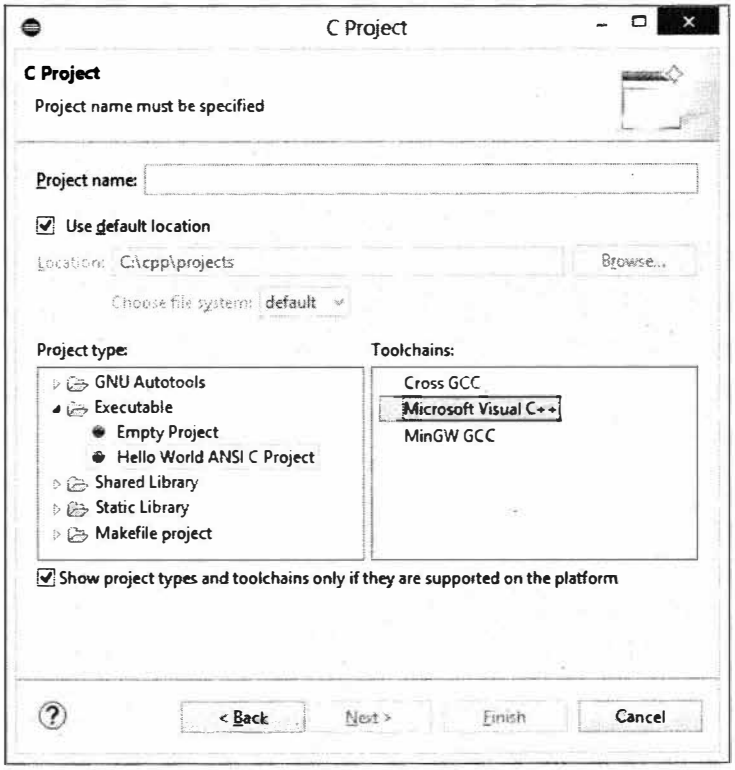
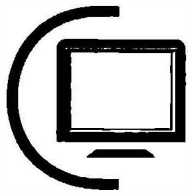


Рис. 1.42. Создание проекта с поддержкой Microsoft Visual C++



ГЛАВА 2

Первые шаги

Прежде чем мы начнем рассматривать синтаксис языка, необходимо сделать два замечания. Во-первых, не забывайте, что книги по программированию нужно не только читать, но и выполнять все приведенные в них примеры, а также экспериментировать, что-либо в этих примерах изменяя. Поэтому, если вы удобно устроились на диване и настроились просто читать, у вас практически нет шансов изучить язык! Во-вторых, помните, что прочитать книгу один раз недостаточно — ее вы должны выучить наизусть! Это основы языка! Сколько на это уйдет времени, зависит от ваших способностей и желания. Как минимум вы должны знать структуру книги.

Чем больше вы будете делать самостоятельно, тем большему научитесь. Обычно после первого прочтения многое непонятно, после второго прочтения некоторые вещи становятся понятнее, после третьего — еще лучше, ну а после *N*-го прочтения — не понимаешь, как могло быть что-то непонятно после первого прочтения. Повторение — мать учения. Наберитесь терпения, и вперед!

Итак, приступим к изучению языка C.

2.1. Первая программа

В процессе изучения основ языка C мы будем создавать *консольные приложения*. Консольное приложение — это программа, отображающая текстовую информацию и позволяющая вводить символы с клавиатуры. Консольное приложение позволит не отвлекаться на изучение среды разработки, а полностью сосредоточить внимание на изучении синтаксиса языка.

При изучении языков программирования принято начинать с программы, выводящей надпись "Hello, world!" в окно консоли. Не будем нарушать традицию и продемонстрируем, как это выглядит на языке C (листинг 2.1).

Листинг 2.1. Первая программа

```
#include <stdio.h>

int main(void) {
   printf("Hello, world!");
   return 0;
}
```

Рассмотрим структуру программы из листинга 2.1. Вся программа состоит из четырех инструкций:

- включение системного заголовочного файла stdio.h с помощью директивы препроцессора `#include`;
- описание функции `main()`;
- вывод приветствия с помощью функции `printf()`;
- возврат значения с помощью инструкции `return`.

В первой строке программы с помощью директивы `#include` включается файл stdio.h, в котором объявлена функция `printf()`, предназначенная для форматированного вывода данных в окно консоли. Так как название файла указано внутри угловых скобок, его поиск будет выполнен в путях поиска заголовочных файлов. Содержимое файла stdio.h на одной из стадий компиляции целиком вставляется вместо инструкции с директивой `#include`.

Заголовочный файл stdio.h входит в состав библиотеки MinGW и располагается в каталогах C:\MinGW32\mingw32\i686-w64-mingw32\include и C:\msys64\mingw64\x86_64-w64-mingw32\include. Заголовочный файл является обычным текстовым файлом, поэтому его содержимое можно посмотреть в любом текстовом редакторе. Посмотреть содержимое файла можно также на отдельной вкладке редактора Eclipse. Для этого внутри вкладки с программой щелкаем правой кнопкой мыши на названии файла stdio.h и из контекстного меню выбираем пункт **Open Declaration** или вставляем текстовый курсор на название файла и нажимаем клавишу <F3>.

Далее создается функция `main()`, внутри блока которой расположены все остальные инструкции. Именно функция с названием `main()` будет автоматически вызываться при запуске программы. Определение функции является составной инструкцией, поэтому после описания параметров указывается открывающая фигурная скобка. Закрывающая фигурная скобка является признаком конца блока:

```
int main(void) {

}
```

Фигурные скобки обрамляют блок, который ограничивает область видимости идентификаторов. Все идентификаторы, описанные внутри блока, видны только внутри этого блока.

Перед названием функции `main()` указывается тип возвращаемого значения. Ключевое слово `int` означает, что функция возвращает целое число. После названия функции внутри круглых скобок указывается ключевое слово `void`, которое означает, что функция не принимает параметров.

Что такое функция? *Функция* — это фрагмент кода внутри фигурных скобок, который может быть вызван из какого-либо места программы сколько угодно раз. При этом функция может возвращать какое-либо значение в место вызова или вообще ничего не возвращать, а просто выполнять какую-либо операцию.

Вывод строки `"Hello, world!"` осуществляется с помощью функции `printf()`. Текст, выводимый в консоль, указывается в кавычках внутри круглых скобок, расположенных после названия функции. Объявление функции `printf()` расположено внутри файла stdio.h, поэтому в первой строке программы мы включаем этот файл с помощью директивы `#include`. Если заголовочный файл не включить, то функция будет недоступна.

После всех инструкций указывается точка с запятой. Исключением являются составные инструкции (в нашем примере после закрывающей фигурной скобки блока функции `main()` точка с запятой не указывается) и директивы препроцессора (в нашем примере нет точки с запятой после инструкции с директивой `#include`). Указать точку с запятой после инструкции — это то же самое, что поставить точку в конце предложения.

Возвращаемое функцией `main()` значение указывается после ключевого слова `return` в самом конце блока перед закрывающей фигурной скобкой. Число `0` в данном случае означает нормальное завершение программы. Если указано другое число, то это свидетельствует о некорректном завершении программы. Согласно стандарту, внутри функции `main()` ключевое слово `return` можно не указывать. В этом случае компилятор должен самостоятельно вставить инструкцию, возвращающую значение `0`.

При запуске программы будет автоматически вызвана функция `main()` и все инструкции внутри фигурных скобок будут выполняться сверху вниз друг за другом. Как только поток выполнения дойдет до закрывающий фигурной скобки или до инструкции `return`, программа завершит работу.

Скомпилируем программу и запустим ее на исполнение. Вначале попробуем это сделать из командной строки без помощи редактора.

Запускаем Notepad++ и создаем новый документ. В меню **Кодировки** выбираем пункт **Кодировки | Кириллица | Windows-1251**. Вводим текст из листинга 2.1 и сохраняем под названием helloworld.c в каталоге C:\book.

Теперь необходимо скомпилировать программу. Для этого открываем командную строку и делаем текущим каталог C:\book:

```
C:\Users\Unicross>cd C:\book
```

Затем настраиваем переменную окружения `PATH`:

```
C:\book>set Path=C:\msys64\mingw64\bin;%Path%
```

Теперь все готово для компиляции. Выполняем следующую команду (текст команды указывается на одной строке без символа переноса):

```
C:\book>gcc -Wall -Wconversion -O3 -finput-charset=cp1251
            -fexec-charset=cp1251 -o helloworld.exe helloworld.c
```

Первое слово (`gcc`) вызывает компилятор **gcc.exe**. Флаг `-Wall` указывает выводить все предупреждающие сообщения, возникающие во время компиляции программы, флаг `-Wconversion` задает вывод предупреждений при возможной потере данных, а флаг `-O3` определяет уровень оптимизации. С помощью флага `-finput-charset` указывается кодировка файла с программой, а с помощью флага `-fexec-charset` — кодировка С-строк. Название создаваемого в результате компиляции файла (**helloworld.exe**) задается после флага `-o`. Далее указывается название исходного текстового файла с программой на языке С (**helloworld.c**).

Если в процессе компиляции возникнут какие-либо ошибки, то они будут выведены. Эти ошибки следует исправить, а затем запустить компиляцию повторно. Если компиляция прошла успешно, то никаких сообщений не появится, а отобразится приглашение для ввода следующей команды:

```
C:\book>gcc -Wall -Wconversion -O3 -finput-charset=cp1251
            -fexec-charset=cp1251 -o helloworld.exe helloworld.c

C:\book>
```

После успешной компиляции в каталоге **C:\book** будет создан файл **helloworld.exe**, который можно запустить из командной строки следующим образом:

```
C:\book>helloworld
Hello, world!
```

Вторая строка в этом коде — это результат выполнения программы.

2.2. Создание пустого проекта в редакторе Eclipse

В *разд. 1.8* мы создали два проекта в редакторе Eclipse: `Test32c` (для создания 32-битных программ) и `Test64c` (для создания 64-битных программ). При этом мы пользовались мастером, который автоматически создал файл и вставил в него код. Давайте попробуем создать пустой проект и самостоятельно добавить в него файлы.

Для создания пустого проекта в меню **File** выбираем пункт **New | Project**. В открывшемся окне (см. рис. 1.30) в списке выбираем пункт **C/C++ | C Project** и нажимаем кнопку **Next**. На следующем шаге (рис. 2.1) в поле **Project name** вводим `HelloWorld`, в списке слева выбираем пункт **Executable | Empty Project**, а в списке справа — пункт **MinGW GCC**. Для создания проекта нажимаем кнопку **Finish**. Проект отобразится в окне **Project Explorer**.

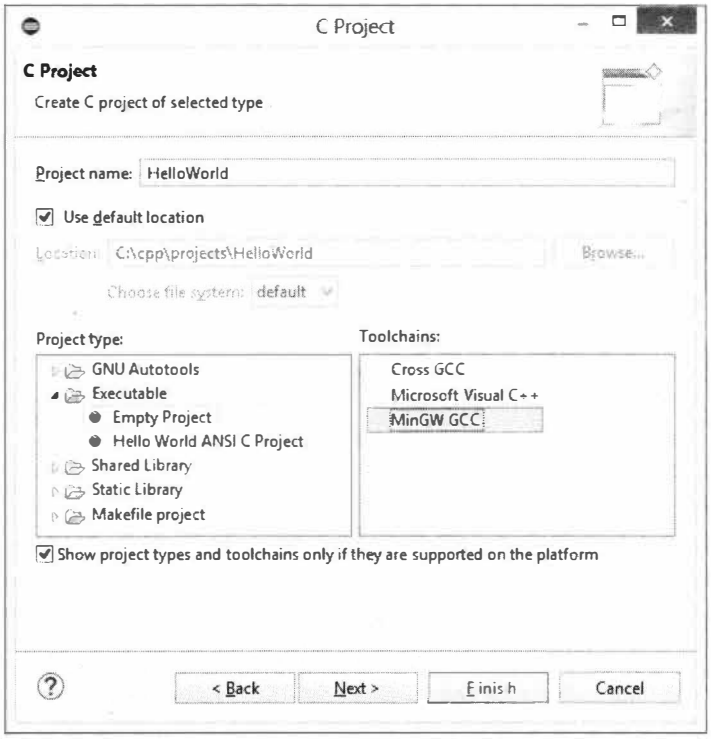


Рис. 2.1. Создание пустого проекта

Далее создадим каталог для файлов с исходным кодом. Для этого в окне **Project Explorer** щелкаем правой кнопкой мыши на названии проекта и из контекстного меню выбираем пункт **New | Source Folder** (рис. 2.2). В открывшемся окне (рис. 2.3) в поле **Folder name** вводим `src` и нажимаем кнопку **Finish**.

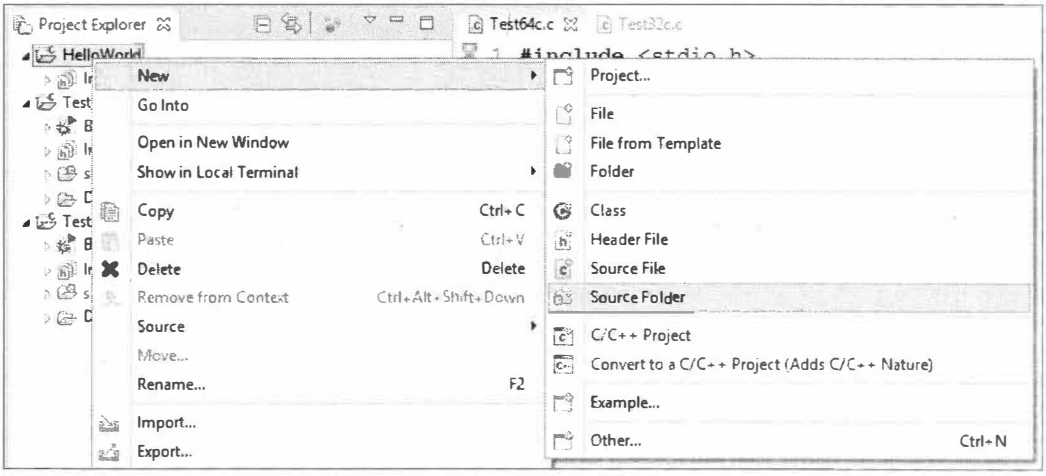


Рис. 2.2. Контекстное меню проекта

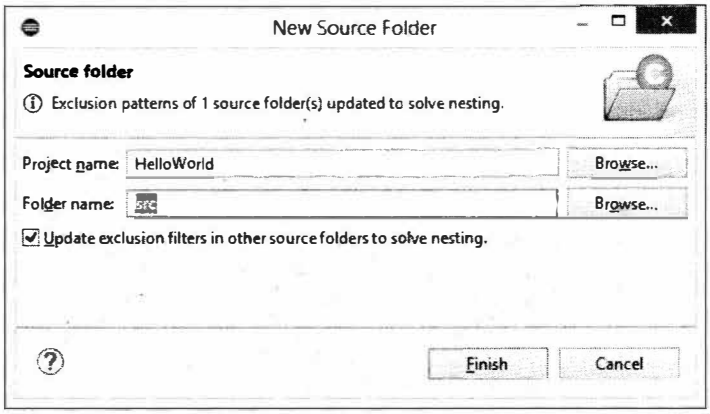


Рис. 2.3. Создание нового каталога

2.3. Добавление в проект файла с программой

Теперь добавим в созданный каталог файл для ввода текста программы. Для этого в окне **Project Explorer** щелкаем правой кнопкой мыши на названии каталога src и из контекстного меню выбираем пункт **New | Source File** (см. рис. 2.2). В открывшемся окне (рис. 2.4) в поле **Source folder** должно быть уже вставлено значение `HelloWorld/src`. В поле **Source file** вводим `HelloWorld.c` (обратите внимание: файлы с исходным кодом на языке C имеют расширение c), а из списка **Template** выбираем пункт **<None>**. Нажимаем кнопку **Finish**.

В результате файл будет создан в каталоге src и открыт на отдельной вкладке. Вводим сюда код из листинга 2.1 и сохраняем файл. Для этого в меню **File** выбираем пункт **Save** или нажимаем кнопку **Save** на панели инструментов.

Теперь проверим кодировку файла. Увидеть кодировку файлов проекта можно в свойствах проекта (см. рис. 1.25), а также выбрав в меню **Edit** пункт **Set Encoding** (рис. 2.5). Кодировка должна быть указана **Cp1251** (windows-1251).

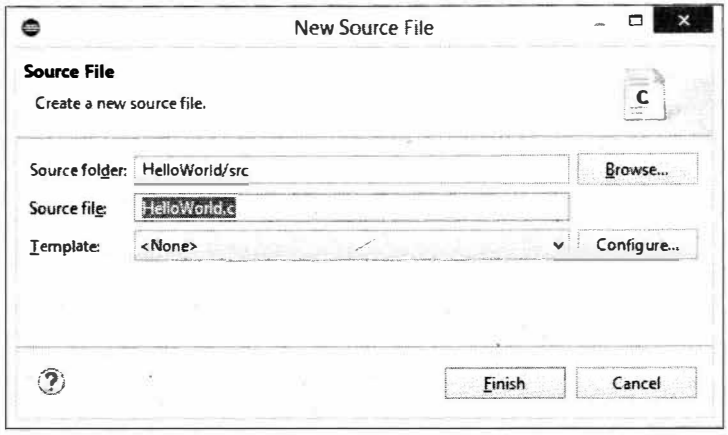


Рис. 2.4. Создание файла с программой

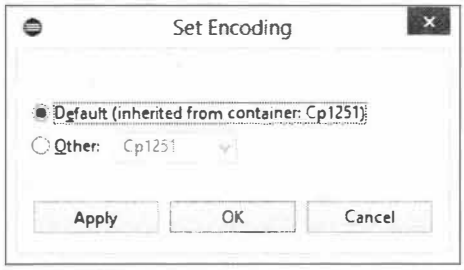


Рис. 2.5. Проверка кодировки файла

2.4. Добавление в проект заголовочного файла

Помимо файлов с исходным кодом (имеют расширение c) в проекте могут быть заголовочные файлы (имеют расширение h). В заголовочных файлах указываются прототипы функций и различные объявления. Для создания заголовочного файла в окне **Project Explorer** щелкаем правой кнопкой мыши на названии каталога src и из контекстного меню выбираем пункт **New | Header File** (см. рис. 2.2). В открывшемся окне (рис. 2.6) в поле **Source folder** должно быть уже вставлено значение `HelloWorld/src`. В поле **Header file** вводим `HelloWorld.h` (обратите внимание: заголовочные файлы имеют расширение h), а из списка **Template** выбираем пункт **Default C header template**. Нажимаем кнопку **Finish**.

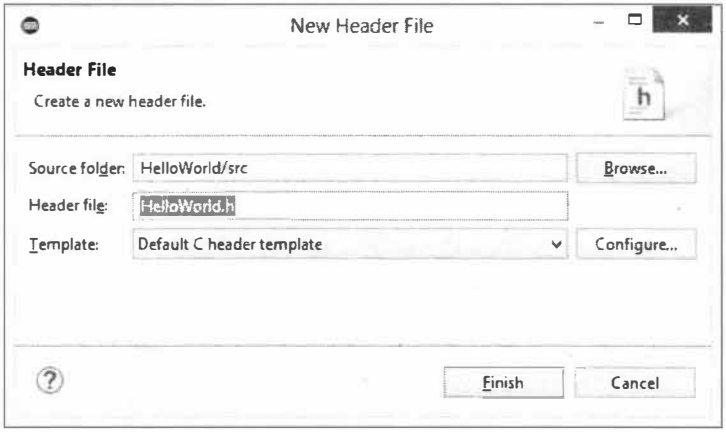


Рис. 2.6. Создание заголовочного файла

В результате файл будет создан в каталоге src и открыт на отдельной вкладке. Причем внутри файла будет вставлен код, приведенный в листинге 2.2.

Листинг 2.2. Содержимое файла HelloWorld.h

```
#ifndef HELLOWORLD_H_
#define HELLOWORLD_H_

#endif /* HELLOWORLD_H_ */
```

Текст между символами `/*` и `*/` является *комментарием*. Инструкции, начинающиеся с символа `#`, — это *директивы препроцессора*. В нашем примере их три:

- `#ifndef` — проверяет отсутствие константы с именем `HELLOWORLD_H_`;
- `#define` — создает константу с именем `HELLOWORLD_H_`;
- `#endif` — обозначает конец блока проверки отсутствия константы.

Зачем нужны эти директивы препроцессора? Заголовочный файл мы подключаем к файлу с исходным кодом с помощью директивы `#include`:

```
#include "HelloWorld.h"
```

Встретив в исходном коде директиву `#include`, компилятор вставляет все содержимое заголовочного файла на место директивы. Если мы вставим две одинаковые директивы `#include`, то содержимое заголовочного файла будет вставлено дважды. Так как объявить один идентификатор (например, глобальную переменную) дважды нельзя, компилятор выведет сообщение об ошибке. Для того чтобы этого избежать, прототипы функций и прочие объявления вкладываются в блок, ограниченный директивами `#ifndef` и `#endif`. В директиве `#ifndef` указывается константа, совпадающая с названием заголовочного файла. Все буквы в имени константы заглавные, а точка заменена символом подчеркивания. Если константа не существует (при первом включении заголовочного файла так и будет), то с помощью директивы `#define` эта константа создается и содержимое блока вставляется в исходный код. При повторном включении заголовочного файла константа уже существует, поэтому содержимое блока будет проигнорировано. Таким образом, заголовочный файл дважды вставлен не будет, а значит, и ошибки не возникнет.

Вместо этих директив в самом начале заголовочного файла можно указать директиву препроцессора `#pragma` со значением `once`, которая также препятствует повторному включению файла (в старых компиляторах директива может не поддерживаться):

```
#pragma once
// Объявление функций и др.
```

Название заголовочного файла в директиве `#include` может быть указано внутри угловых скобок:

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

или внутри кавычек:

```
#include "HelloWorld.h"
```

В первом случае заголовочный файл ищется в путях поиска заголовочных файлов. При этом текущий рабочий каталог не просматривается. Добавить каталог в пути поиска заголовочных файлов позволяет флаг `-I` в команде компиляции. Обычно с помощью угловых скобок включаются заголовочные файлы стандартной библиотеки или библиотеки стороннего разработчика.

Во втором случае мы имеем дело с заголовочным файлом, который вначале ищется в текущем рабочем каталоге (или относительно него), а затем в путях поиска заго-

ловочных файлов, как будто название указано внутри угловых скобок. Таким способом обычно включаются заголовочные файлы проекта.

Можно указать просто название заголовочного файла:

```
#include "HelloWorld.h"
```

абсолютный путь к нему:

```
#include "C:\\cpp\\projects\\HelloWorld\\src\\HelloWorld.h"
```

или относительный путь к нему:

```
#include "./HelloWorld.h"
```

Если указано только название заголовочного файла и этот файл не входит с состав проекта (или название указано внутри угловых скобок), нужно дополнительно указать место поиска заголовочных файлов. Давайте попробуем подключить файл jni.h (находится в каталоге <Путь к JDK>\include):

```
#include "C:\\Program Files\\Java\\jdk-10\\include\\jni.h"
```

Даже если мы укажем абсолютный путь к этому файлу, все равно получим сообщение об ошибке. Дело в том, что внутри файла jni.h подключается файл jni_md.h, который расположен в каталоге <Путь к JDK>\include\win32. Компилятору

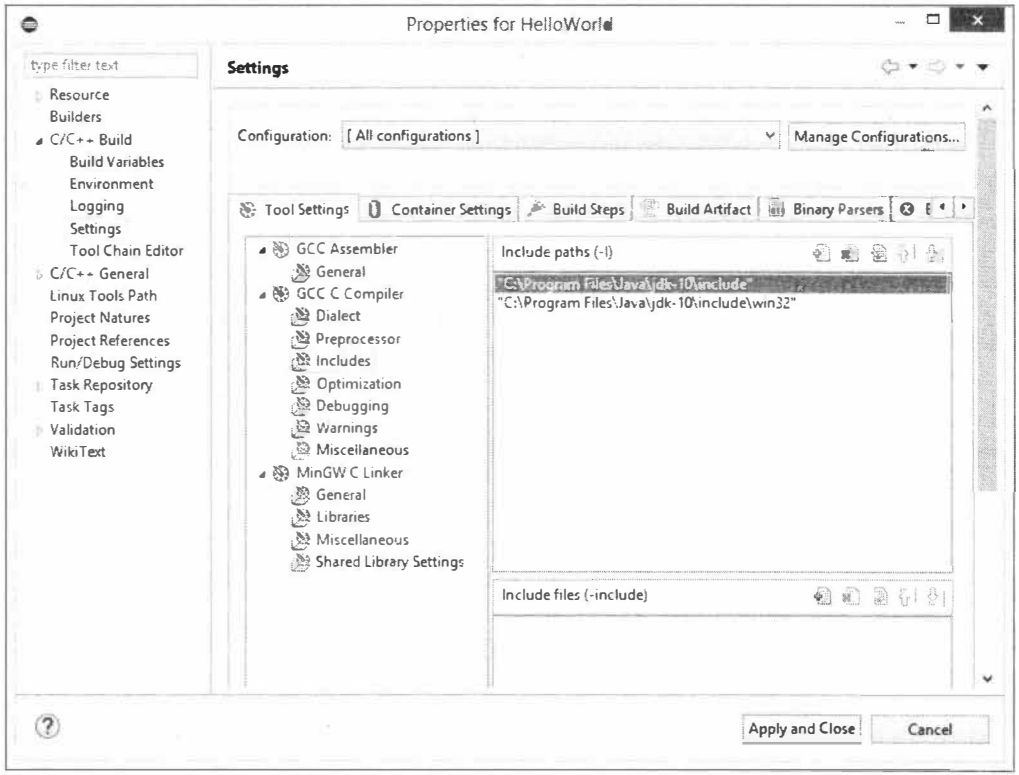


Рис. 2.7. Указание пути к заголовочным файлам

о местоположении этого файла ничего не известно. Давайте в директиве `#include` оставим только название файла, а путь попробуем указать другим способом:

```
#include <jni.h>
```

Открываем свойства проекта (в меню **Project** выбираем пункт **Properties**) и из списка слева выбираем пункт **C/C++ Build | Settings**. На отобразившейся вкладке (рис. 2.7) из списка **Configuration** выбираем пункт **All configurations**, а затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Includes**. В разделе **Include paths (-I)** нажимаем кнопку **Add**. В открывшемся окне (рис. 2.8) вводим путь к каталогу **<Путь к JDK>\include** и нажимаем кнопку **OK**. Путь будет добавлен в список. Аналогичным образом добавляем путь к каталогу **<Путь к JDK>\include\win32**. Сохраняем изменения, нажав кнопку **Apply and Close**.

Путь к заголовочным файлам в командной строке добавляется с помощью флага `-I`:

```
gcc -Wall -O3 "-IC:\\Program Files\\Java\\jdk-10\\include"
    "-IC:\\Program Files\\Java\\jdk-10\\include\\win32"
    -o helloworld.exe helloworld.c
```

В нашем случае путь содержит пробел, поэтому весь путь указывается внутри кавычек. Если пробелов нет, то кавычки можно не указывать.

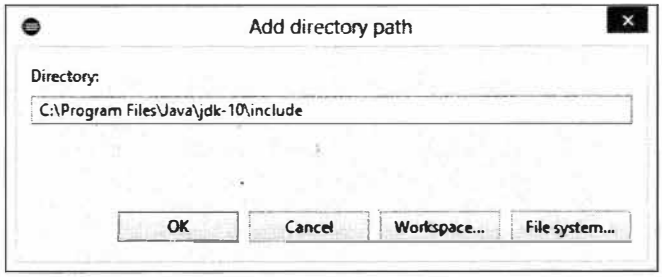


Рис. 2.8. Добавление пути к заголовочному файлу

2.5. Компиляция и запуск программы в редакторе Eclipse

Итак, файл HelloWorld.c с кодом из листинга 2.1 добавлен в проект, теперь можно скомпилировать его и запустить в редакторе Eclipse. Предварительно сохраняем файл, выбрав в меню **File** пункт **Save** или нажав комбинацию клавиш **<Ctrl>+<S>**.

Скомпилировать программу можно несколькими способами:

- в меню **Project** выбираем пункт **Build Project**;
- нажимаем кнопку с изображением молотка на панели инструментов (см. рис. 1.38).

В окне **Console** (рис. 2.9) отобразятся инструкции компиляции и результат ее выполнения. Если компиляция прошла успешно, то никаких сообщений об ошибках в этом окне не будет:

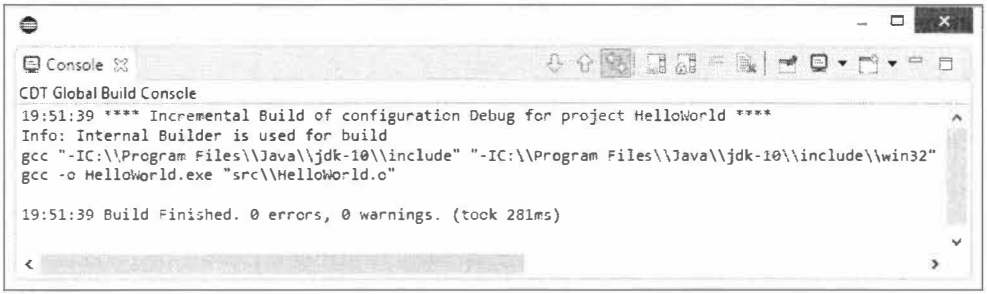


Рис. 2.9. Результат компиляции в окне Console

```
19:51:39 **** Incremental Build of configuration Debug for project
HelloWorld ****
Info: Internal Builder is used for build
gcc "-IC:\\Program Files\\Java\\jdk-10\\include" "-IC:\\Program
Files\\Java\\jdk-10\\include\\win32" -O0 -g3 -Wall -c
-fmessage-length=0 -o "src\\HelloWorld.o" "..\\src\\HelloWorld.c"
gcc -o HelloWorld.exe "src\\HelloWorld.o"

19:51:39 Build Finished. 0 errors, 0 warnings. (took 281ms)
```

Компиляция в редакторе Eclipse выполняется в два прохода. При первом проходе создается объектный файл HelloWorld.o, а на втором проходе на его основе создается EXE-файл.

По умолчанию для проекта задается режим компиляции `Debug` (отладка). В этом режиме дополнительно сохраняется информация для отладчика, и EXE-файл будет создан без оптимизаций. Когда программа уже написана и отлажена, нужно выбрать режим `Release`. Для этого в меню **Project** выбираем пункт **Build Configurations | Set Active | Release** (рис. 2.10). Кроме того, на панели инструментов рядом с кнопкой с изображением молотка расположена кнопка со стрелкой, направленной вниз. Если щелкнуть мышью на этой кнопке, то отобразится меню (рис. 2.11), в котором также можно выбрать режим компиляции.

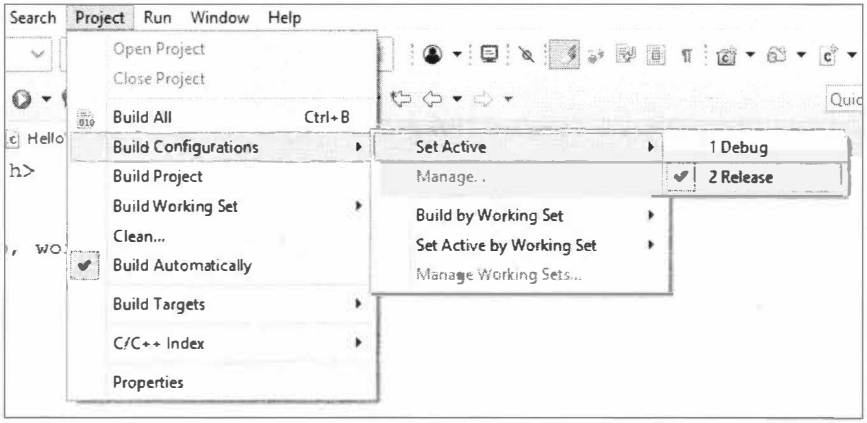


Рис. 2.10. Выбор режима компиляции в меню

Рис. 2.11. Выбор режима компиляции на панели инструментов

При создании пустого проекта в настройках режима `Release` не указан режим оптимизации и не отключен вывод отладочной информации. Давайте это исправим. Открываем свойства проекта (в меню **Project** выбираем пункт **Properties**) и из списка слева выбираем пункт **C/C++ Build | Settings**. На отобразившейся вкладке из списка **Configuration** выбираем пункт **Release**, а затем на вкладке **Tool Settings** выбираем пункт **GCC C Compiler | Optimization**. Из раскрывающегося списка **Optimization Level** (рис. 2.12) выбираем пункт **Optimize most (–O3)**. Затем выбираем пункт **GCC C Compiler | Debugging** и из раскрывающегося списка **Debug Level** выбираем пункт **None**.

Далее укажем кодировку файла и кодировку С-строк. Для этого в свойствах проекта из списка слева выбираем пункт **C/C++ Build | Settings**. На отобразившейся вкладке из списка **Configuration** выбираем пункт **All configurations**, а затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Miscellaneous** (см. рис. 1.35). В поле **Other flags** через пробел к имеющемуся значению добавляем следующие инструкции:

```
-finput-charset=cp1251 -fexec-charset=cp1251
```

Содержимое поля **Other flags** после изменения должно быть таким:

```
-c -fmessage-length=0 -finput-charset=cp1251 -fexec-charset=cp1251
```

Затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Warnings**. Устанавливаем флажок **–Wconversion**. Проверяем также установку флажка **–Wall**. Сохраняем настройки проекта, нажимая кнопку **Apply and Close**.

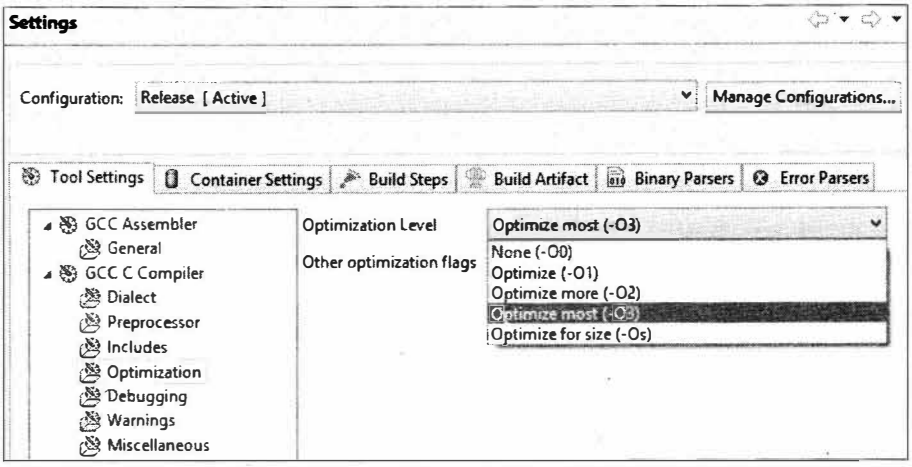


Рис. 2.12. Выбор режима оптимизации

После изменения настроек при компиляции в режиме `Release` в окне **Console** отобразятся следующие инструкции:

```
20:06:27 **** Incremental Build of configuration Release for project
HelloWorld ****
Info: Internal Builder is used for build
gcc "-IC:\\Program Files\\Java\\jdk-10\\include"
"-IC:\\Program Files\\Java\\jdk-10\\include\\win32" -O3 -Wall
-Wconversion -c -fmessage-length=0 -finput-charset=cp1251
-fexec-charset=cp1251 -o "src\\HelloWorld.o" "..\\src\\HelloWorld.c"
gcc -o HelloWorld.exe "src\\HelloWorld.o"

20:06:27 Build Finished. 0 errors, 0 warnings. (took 344ms)
```

В результате компиляции в разных режимах были созданы два EXE-файла:

```
C:\cpp\projects\HelloWorld\Debug\HelloWorld.exe
C:\cpp\projects\HelloWorld\Release\HelloWorld.exe
```

Первый файл содержит отладочную информацию и имеет размер 383 Кбайт (78,4 Кбайт, если использовать компилятор из каталога C:\MinGW64\mingw64\bin). Второй файл не содержит отладочной информации и имеет размер 357 Кбайт (52,7 Кбайт, если использовать компилятор из каталога C:\MinGW64\mingw64\bin). При компиляции второго файла была дополнительно выполнена оптимизация. Именно этот файл следует отдавать заказчикам, хотя запустить на исполнение из командной строки можно оба файла:

```
C:\book>C:\cpp\projects\HelloWorld\Debug\HelloWorld.exe
Hello, world!

C:\book>C:\cpp\projects\HelloWorld\Release\HelloWorld.exe
Hello, world!
```

Запустить программу в редакторе Eclipse можно несколькими способами:

- в меню **Run** выбираем пункт **Run**;
- в меню **Run** выбираем пункт **Run As**, а затем профиль запуска;
- нажимаем комбинацию клавиш <Ctrl>+<F11>;
- на панели инструментов нажимаем кнопку с зеленым кругом и белым треугольником внутри него (см. рис. 2.11).

Если программа была изменена, то автоматически запустится процесс компиляции, а затем результат отобразится в окне **Console**. Согласитесь, очень просто и удобно, когда компиляция и запуск выполняются нажатием всего одной кнопки.

Для изменения настроек конфигурации запуска в меню **Run** выбираем пункт **Run Configurations**. В итоге отобразится окно (рис. 2.13), в котором можно указать специальные настройки для уже созданных конфигураций или создать новую конфигурацию запуска.

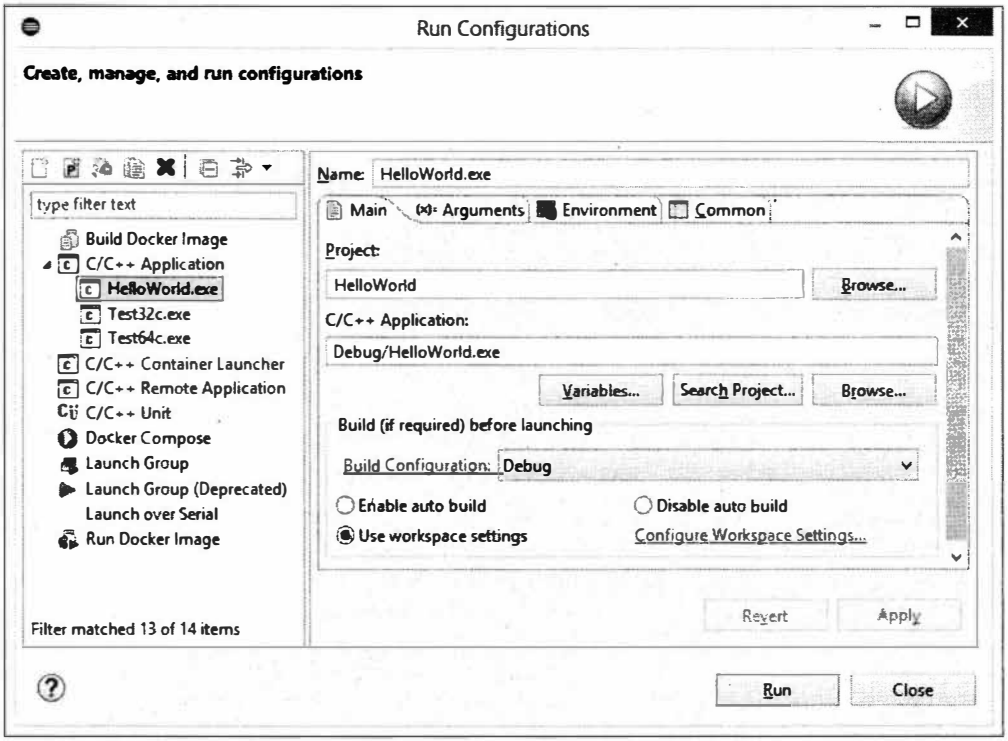


Рис. 2.13. Окно **Run Configurations**

2.6. Структура программы

Что ж, программная среда установлена, и редактор настроен. Теперь можно приступать к изучению языка С. Как вы уже знаете, программа состоит из инструкций, расположенных в текстовом файле. Структура обычной программы выглядит так:

```
<Подключение заголовочных файлов>
<Объявление глобальных переменных>
<Объявление функций и др.>
int main(void) {
   <Инструкции>
   return 0;
}
<Определения функций и др.>
```

В самом начале программы подключаются заголовочные файлы, в которых содержатся объявления идентификаторов без их реализации. Например, чтобы иметь возможность вывести данные в окно консоли, необходимо подключить файл stdio.h, в котором содержится объявление функции `printf()`. Подключение осуществляется с помощью директивы `#include`:

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

С помощью директивы `#include` можно подключать как стандартные заголовочные файлы, так и пользовательские файлы.

После подключения файлов производится объявление глобальных переменных. Переменные предназначены для хранения значений определенного типа. *Глобальные переменные* видны во всей программе, включая функции. Если объявить переменную внутри функции, то область видимости переменной будет ограничена рамками функции и в других частях программы использовать переменную нельзя. Такие переменные называются *локальными*. Объявить целочисленную переменную можно так:

```
int x;
```

В этом примере мы объявили переменную с названием `x` и указали, что она может хранить данные, имеющие тип `int`. Ключевое слово `int` означает, что переменная предназначена для хранения целого числа.

Если в команде компиляции указан флаг `-Wconversion`, то попытка присвоить переменной значение, имеющее другой тип, приведет к предупреждению. Исключением являются ситуации, в которых компилятор может произвести преобразование типов данных без потери точности. Например, целое число без проблем преобразуется в вещественное число, однако попытка преобразовать вещественное число в целое приведет к выводу предупреждающего сообщения. Пример сообщения:

```
warning: conversion to 'int' alters 'double' constant value
```

Хотя компилятор предупреждает о потери данных, программа все равно будет скомпилирована. Поэтому обращайте внимание на сообщения, которые выводятся в окне **Problems**, и на различные подчеркивания кода в редакторе Eclipse. Иначе программа будет работать, но результат выполнения окажется некорректным.

Если в команде компиляции не указан флаг `-Wconversion`, то предупреждающих сообщений о потери данных вообще не будет. Советую установить этот флаг в настройках проекта для режима `Debug`. Открываем свойства проекта (в меню **Project** выбираем пункт **Properties**) и из списка слева выбираем пункт **C/C++ Build | Settings**. На отобразившейся вкладке из списка **Configuration** выбираем пункт **Debug**, а затем на вкладке **Tool Settings** выбираем пункт **GCC C Compiler | Warnings**. Устанавливаем флажок **–Wconversion**. Проверяем также установку флажка **–Wall**. Сохраняем изменения.

Обратите внимание на то, что объявление переменной заканчивается точкой с запятой. Большинство инструкций в языке С должно заканчиваться точкой с запятой. Если точку с запятой не указать, то компилятор сообщит о синтаксической ошибке. Пример сообщения:

```
error: expected '=', ',', ';', 'asm' or '__attribute__' before 'x'
```

Для того чтобы увидеть строку, о которой сообщает компилятор, сделайте двойной щелчок мышью на сообщении в окне **Console**. В результате строка с ошибкой в программе станет активной. В нашем случае активной будет строка, расположенная сразу после объявления переменной. Обратите также внимание: на вкладке ре-

дактор пометил строку красным кругом с белым крестиком внутри. Если навести указать мыши на эту метку, то отобразится текст ошибки.

При объявлении можно сразу присвоить начальное значение переменной. Присваивание значения переменной при объявлении называется *инициализацией переменной*. Для того чтобы произвести инициализацию переменной, после ее названия указывается оператор =, а затем значение. Пример:

```
int x = 10;
```

Если глобальной переменной не присвоено значение при объявлении, то она будет иметь значение `0`. Если локальной переменной не присвоено значение, то переменная будет содержать произвольное значение. Как говорят в таком случае: переменная содержит "мусор".

Обратите внимание на то, что перед оператором = и после него вставлены пробелы. Количество пробелов может быть произвольным, или пробелов может не быть вовсе. Кроме того, допускается вместо пробелов использовать символ перевода строки или символ табуляции. Например, эти инструкции вполне допустимы:

```
int x=21;
int y=                  85;
int z
=
56;
```

Тем не менее следует придерживаться единообразия в коде и обрамлять операторы одним пробелом. Следует учитывать, что это не строгие правила, а лишь рекомендации по оформлению кода.

Как видно из предыдущего примера, инструкция может быть расположена на нескольких строках. Концом инструкции является точка с запятой, а не конец строки. Например, на одной строке может быть несколько инструкций:

```
int x = 21; int y = 85; int z = 56;
```

Из этого правила есть исключения. Например, после директив препроцессора точка с запятой не указывается. В этом случае концом инструкции является конец строки. Директиву препроцессора можно узнать по символу # перед названием директивы. Типичным примером директивы препроцессора является директива `#include`, которая используется для включения заголовочных файлов:

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

После объявления глобальных переменных могут располагаться объявления функций. Такие объявления называются *прототипами*. Схема прототипа функции выглядит следующим образом:

```
<Тип возвращаемого значения> <Название функции>(
                              [<Тип> [<Параметр 1>]
                          [, ..., <Тип> [<Параметр N>]]]);
```

Например, прототип функции, которая складывает два целых числа и возвращает их сумму, выглядит так:

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

После объявления функции необходимо описать ее реализацию, которая называется *определением функции*. Определение функции обычно располагается после определения функции `main()`. Обратите внимание на то, что объявлять прототип функции `main()` не нужно, т. к. определение этой функции обычно расположено перед определением других функций. Пример определения функции `sum()`:

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

Как видно из примера, первая строка в определении функции `sum()` совпадает с объявлением функции. Следует заметить, что в объявлении функции можно не указывать названия параметров. Достаточно указать информацию о типе данных. Таким образом, объявление функции можно записать так:

```
int sum(int, int);
```

После объявления функции ставится точка с запятой. В определении функции внутри фигурных скобок должна быть описана реализация функции. Так как в объявлении указано, что функция возвращает целочисленное значение, следовательно, после описания реализации необходимо вернуть значение. Возвращаемое значение указывается после ключевого слова `return`. Если функция не возвращает никакого значения, то перед названием функции вместо типа данных указывается ключевое слово `void`. Пример объявления функции, которая не возвращает значения:

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

Пример определения функции `print()`:

```
void print(int x) {
   printf("%d", x);
}
```

Вся реализация функции должна быть расположена внутри фигурных скобок. Открывающая скобка может находиться на одной строке с определением функции, как в предыдущем примере, или в начале следующей строки. Пример:

```
void print(int x)
{
   printf("%d", x);
}
```

Многие программисты считают такой стиль наиболее приемлемым, т. к. открывающая и закрывающая скобки расположены друг под другом. На мой же взгляд образуется лишняя пустая строка. Так как размеры экрана ограничены, при наличии пустой строки на экран помещается меньше кода и приходится чаще пользоваться полосой прокрутки. Если размещать инструкции с равным отступом, то блок кода выделяется визуально и следить за положением фигурных скобок просто излишне. Тем более, что редактор Eclipse позволяет подсветить парные скобки. Для того чтобы найти пару фигурных скобок, следует поместить указатель ввода после скобки. В результате скобки будут подсвечены. Какой стиль использовать, зависит от личного предпочтения программиста или от правил оформления кода, принятых

в определенной фирме. Главное, чтобы стиль оформления внутри одной программы был одинаковым.

Перед инструкциями внутри фигурных скобок следует размещать одинаковый отступ. В качестве отступа можно использовать пробелы или символ табуляции. При использовании пробелов размер отступа равняется трем или четырем пробелам для блока первого уровня. Для вложенных блоков количество пробелов умножают на уровень вложенности. Если для блока первого уровня вложенности использовалось три пробела, то для блока второго уровня вложенности должно использоваться шесть пробелов, для третьего уровня — девять пробелов и т. д. В одной программе не следует использовать и пробелы, и табуляцию в качестве отступа. Необходимо выбрать что-то одно и пользоваться этим во всей программе.

Закрывающая фигурная скобка обычно размещается в конце блока на отдельной строке. После скобки точка с запятой не указывается. Однако наличие точки с запятой ошибкой не является, т. к. инструкция может не содержать выражений вообще. Например, такая инструкция вполне допустима, хотя она ничего не делает:

```
;
```

Самой главной функцией в программе является функция `main()`. Именно функция с названием `main()` будет автоматически вызываться при запуске программы. Функция имеет три прототипа:

```
int main(void);
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char **penv);
```

Значение `void` внутри круглых скобок означает, что функция не принимает параметры. Этим прототипом мы пользовались в листинге 2.1.

Второй прототип применяется для получения значений, указанных при запуске программы из командной строки (см. листинг 2.10). Количество значений доступно через параметр `argc`, а сами значения через параметр `argv`. Параметр `penv` в третьем прототипе позволяет получить значения переменных окружения (см. листинг 17.3).

Перед названием функции указывается тип возвращаемого значения. Ключевое слово `int` означает, что функция возвращает целое число. Возвращаемое значение указывается после ключевого слова `return` в самом конце функции `main()`. Число `0` означает нормальное завершение программы. Если указано другое число, то это свидетельствует о некорректном завершении программы. Согласно стандарту, внутри функции `main()` ключевое слово `return` можно не указывать. В этом случае компилятор должен самостоятельно вставить инструкцию, возвращающую значение `0`. Возвращаемое значение передается операционной системе и может использоваться для определения корректности завершения программы.

Вместо безликого значения `0` можно воспользоваться макроопределением `EXIT_SUCCESS`, а для индикации некорректного завершения программы — макроопределением `EXIT_FAILURE`. Предварительно необходимо включить заголовочный файл **stdlib.h**. Определения макросов:

```
#include <stdlib.h>
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Пример:

```
return EXIT_SUCCESS;
```

В листинге 2.3 приведен пример программы, структуру которой мы рассмотрели в этом разделе. Не расстраивайтесь, если что-то показалось непонятным. Все это мы будем изучать более подробно в следующих главах книги. На этом этапе достаточно лишь представлять структуру программы и самое главное — уметь ее скомпилировать и запустить на выполнение.

Листинг 2.3. Пример программы

```
// Включение заголовочных файлов
#include <stdio.h>
#include <stdlib.h>
// Объявление глобальных переменных
int x = 21;
int y = 85;
// Объявление функций и др.
int sum(int, int);
void print(int);
// Главная функция (точка входа в программу)
int main(void) {
   int z;           // Объявление локальной переменной
   z = sum(x, y);   // Вызов функции sum()
   print(z);        // Вызов функции print()
   return EXIT_SUCCESS;
}
// Определения функций
int sum(int x, int y) {
   return x + y;
}
void print(int x) {
   printf("%d", x);
}
```

Объявление функций можно вынести в отдельный заголовочный файл и включить его с помощью директивы `#include`. В *разд. 2.4* мы создали заголовочный файл **HelloWorld.h**. Давайте им воспользуемся. Вначале добавим в программу инструкцию:

```
#include "HelloWorld.h"
```

после инструкции:

```
#include <stdlib.h>
```

Так как мы указали название заголовочного файла HelloWorld.h внутри кавычек, компилятор вначале будет искать его внутри проекта, а если не найдет, то в путях поиска заголовочных файлов.

Удаляем эти инструкции из программы и сохраняем файл:

```
// Объявление функций и др.
int sum(int, int);
void print(int);
```

Затем вставляем в файл HelloWorld.h код из листинга 2.4.

Листинг 2.4. Содержимое файла HelloWorld.h

```
#ifndef HELLOWORLD_H_
#define HELLOWORLD_H_

// Объявление функций и др.
int sum(int, int);
void print(int);

#endif /* HELLOWORLD_H_ */
```

Директивы препроцессора `#ifndef`, `#define` и `#endif` препятствуют повторному включению заголовочного файла. Если этих директив не будет, то повторное объявление идентификаторов (например, глобальных переменных) вызовет ошибку при компиляции. Вместо этих директив можно указать в самом начале файла директиву препроцессора `#pragma` со значением `once`:

```
#pragma once

// Объявление функций и др.
int sum(int, int);
void print(int);
```

2.7. Комментарии в программе

Комментарии предназначены для вставки пояснений в текст программы, и компилятор полностью их игнорирует. Внутри комментария может располагаться любой текст, включая инструкции, которые выполнять не следует. Помните, комментарии нужны программисту, а не компилятору. Вставка комментариев в код позволит через некоторое время быстро вспомнить предназначение фрагмента кода.

В языке C присутствуют два типа комментариев: *однострочный* и *многострочный*. Однострочный комментарий начинается с символов `//` и заканчивается в конце строки. Вставлять однострочный комментарий можно как в начале строки, так и после инструкции. Если символы `//` разместить перед инструкцией, то она не будет выполнена. Если символы `//` расположены внутри кавычек, то они не являются признаком начала комментария. Примеры однострочных комментариев:

```
// Это комментарий
printf("Hello, world!\n");  // Это комментарий
// printf("Hello, world!"); // Инструкция выполнена не будет
printf("// Это НЕ комментарий!!!");
```

Многострочный комментарий начинается с символов /* и заканчивается символами */. Комментарий может быть расположен как на одной строке, так и на нескольких. Кроме того, многострочный комментарий можно размещать внутри выражения, хотя это и нежелательно. Следует иметь в виду, что многострочные комментарии не могут быть вложенными, поэтому при комментировании больших блоков следует проверять, что в них не встречается закрывающая комментарий комбинация символов */. Тем не менее однострочный комментарий может быть расположен внутри многострочного комментария. Примеры многострочных комментариев:

```
/*
Многострочный комментарий
*/
printf("Hello, world!\n");  /* Это комментарий */
/* printf("Hello, world!"); // Инструкция выполнена не будет */
int x;
x = 10 /* Комментарий */ + 50 /* внутри выражения */;
printf("/* Это НЕ комментарий!!! */");
```

На заметку

Старые компиляторы поддерживали только многострочные комментарии.

Редактор Eclipse позволяет быстро добавить символы комментариев. Для того чтобы вставить однострочный комментарий в начале строки, нужно сделать текущей строку с инструкцией и нажать комбинацию клавиш <Ctrl>+</>. Если предварительно выделить сразу несколько строк, то перед всеми выделенными строками будет вставлен однострочный комментарий. Если все выделенные строки были закомментированы ранее, то комбинация клавиш <Ctrl>+</> удалит все однострочные комментарии. Для вставки многострочного комментария необходимо выделить строки и нажать комбинацию клавиш <Shift>+<Ctrl>+</>. Для удаления многострочного комментария предназначена комбинация клавиш <Shift>+<Ctrl>+<\>.

У начинающих программистов может возникнуть вопрос: зачем может потребоваться комментировать инструкции? Проблема заключается в том, что часто в логике работы программы возникают проблемы. Именно по вине программиста. Например, программа выдает результат, который является неверным. Для того чтобы найти ошибку в алгоритме работы программы, приходится отключать часть кода с помощью комментариев, вставлять инструкции вывода промежуточных результатов и анализировать их. Как говорится: разделяй и властвуй. Таким "дедовским" способом мы обычно ищем ошибки в коде. А "дедовским" мы назвали способ потому, что сейчас все редакторы предоставляют методы отладки, которые позволяют выполнять код построчно и сразу видеть промежуточные результаты. Раньше такого не было. Хотя способ и устарел, но все равно им часто пользуются.

2.8. Вывод данных

Для вывода одиночного символа в языке C применяется функция `putchar()`. Прототип функции:

```
#include <stdio.h>
int putchar(int ch);
```

Пример вывода символа:

```
putchar('w'); // w
putchar(119); // w
```

Вывести строку позволяет функция `puts()`. Прототип функции:

```
#include <stdio.h>
int puts(const char *str);
```

Функция выводит строку `str` и вставляет символ перевода строки. Пример:

```
puts("String1");
puts("String2");
```

Результат выполнения:

```
String1
String2
```

Для форматированного вывода используется функция `printf()`. Можно также воспользоваться функцией `_printf_l()`, которая позволяет дополнительно задать локаль. Прототипы функций:

```
#include <stdio.h>
int printf(const char *format, ...);
int _printf_l(const char *format, _locale_t locale, ...);
```

В параметре `format` указывается строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа `%`. Вместо спецификаторов формата подставляются значения, указанные в качестве параметров. Количество спецификаторов должно совпадать с количеством переданных параметров. В качестве значения функция возвращает количество выведенных символов. Пример вывода строки и числа:

```
printf("String\n");
printf("Count %d\n", 10);
printf("%s %d\n", "Count", 10);
```

Результат выполнения:

```
String
Count 10
Count 10
```

В первом примере строка формата не содержит спецификаторов и выводится как есть. Во втором примере внутри строки формата используется спецификатор `%d`,

предназначенный для вывода целого числа. Вместо этого спецификатора подставляется число 10, переданное во втором параметре. В третьем примере строка содержит сразу два спецификатора %s и %d. Спецификатор %s предназначен для вывода строки, а спецификатор %d — для вывода целого числа. Вместо спецификатора %s будет подставлена строка Count, а вместо спецификатора %d — число 10. Обратите внимание на то, что тип данных переданных значений должен совпадать с типом спецификатора. Если в качестве значения для спецификатора %s указать число, то это приведет к ошибке времени исполнения. На этапе компиляции будет выведено лишь предупреждающее сообщение:

```
warning: format '%s' expects argument of type 'char *', but argument 2
has type 'int' [-Wformat=]
```

Параметр locale в функции _printf_l() позволяет задать локаль. Настройки локали для разных стран различаются. Например, в одной стране принято десятичный разделитель вещественных чисел выводить в виде точки, в другой — в виде запятой. В функции printf() используются настройки локали по умолчанию. Пример:

```
// #include <locale.h>
_locale_t locale = _create_locale(LC_NUMERIC, "dutch");
printf("%.2f\n", 2.5);              // 2.50
_printf_l("%.2f\n", locale, 2.5); // 2,50
setlocale(LC_NUMERIC, "dutch");
printf("%.2f\n", 2.5);              // 2,50
_free_locale(locale);
```

Спецификаторы имеют следующий синтаксис:

```
%[<Флаги>][<Ширина>][.<Точность>][<Размер>]<Тип>
```

В параметре <Тип> могут быть указаны следующие символы:

- c — символ:

```
printf("%c", 'w');          // w
printf("%c", 119);          // w
```

- s — строка:

```
printf("%s", "String");     // String
```

- d или i — десятичное целое число со знаком:

```
printf("%d %i", 10, 30);    // 10 30
printf("%d %i", -10, -30);  // -10 -30
```

- u — десятичное целое число без знака:

```
printf("%u", 10);           // 10
```

- o — восьмеричное число без знака:

```
printf("%o %o", 10, 077);   // 12 77
printf("%#o %#o", 10, 077); // 012 077
```

- `x` — **шестнадцатеричное число без знака в нижнем регистре:**

```
printf("%x %x", 10, 0xff);   // a ff
printf("%#x %#x", 10, 0xff); // 0xa 0xff
```

- `X` — **шестнадцатеричное число без знака в верхнем регистре:**

```
printf("%X %X", 10, 0xff);   // A FF
printf("%#X %#X", 10, 0xff); // 0XA 0XFF
```

- `f` — **вещественное число в десятичном представлении:**

```
printf("%f %f", 18.65781452, 12.5); // 18.657815 12.500000
printf("%f", -18.65781452);         // -18.657815
printf("%#.0f %.0f", 100.0, 100.0); // 100. 100
```

- `e` — **вещественное число в экспоненциальной форме (буква "e" в нижнем регистре):**

```
printf("%e", 18657.81452); // 1.865781e+004
printf("%e", 0.000081452); // 8.145200e-005
```

- `E` — **вещественное число в экспоненциальной форме (буква "e" в верхнем регистре):**

```
printf("%E", 18657.81452); // 1.865781E+004
```

- `g` — **эквивалентно `f` или `e` (выбирается более короткая запись числа):**

```
printf("%g %g %g", 0.086578, 0.000086578, 1.865E-005);
// 0.086578 8.6578e-005 1.865e-005
```

- `G` — **эквивалентно `f` или `E` (выбирается более короткая запись числа):**

```
printf("%G %G %G", 0.086578, 0.000086578, 1.865E-005);
// 0.086578 8.6578E-005 1.865E-005
```

- `p` — **вывод адреса переменной:**

```
int x = 10;
printf("%p", &x);
// Значение в проекте Test32c: 0028FF2C
// Значение в проекте Test64c: 000000000023FE4C
```

- `%` — **символ процента (`%`):**

```
printf("10%%"); // 10%
```

Параметр <Ширина> задает минимальную ширину поля. Если строка меньше ширины поля, то она дополняется пробелами. Если строка не помещается в указанную ширину, то значение игнорируется и строка выводится полностью:

```
printf("'%3s'", "string");  // 'string'
printf("'%10s'", "string"); // '    string'
```

Задать минимальную ширину можно не только для строк, но и для других типов:

```
printf("'%10d'", 25);   // '        25'
printf("'%10f'", 12.5); // ' 12.500000'
```

Параметр <Точность> задает количество знаков после точки для вещественных чисел. Перед этим параметром обязательно должна стоять точка. Пример:

```
printf("'%10.5f'", 3.14159265359); // '   3.14159'
printf("'%.3f'", 3.14159265359);   // '3.142'
```

Если параметр <Точность> используется применительно к целому числу, то он задает минимальное количество цифр. Если число содержит меньшее количество цифр, то вначале числа добавляются нули. Пример:

```
printf("'%7d'", 100);        // '    100'
printf("'%.7d'", 100);       // '0000100'
printf("'%.7d'", 123456789); // '123456789'
```

Если параметр <Точность> используется применительно к строке, то он задает максимальное количество символов. Символы, которые не помещаются, будут отброшены. Пример:

```
printf("'%5.7s'", "Hello, world!");   // 'Hello, '
printf("'%15.20s'", "Hello, world!"); // '  Hello, world!'
```

Вместо минимальной ширины и точности можно указать символ *. В этом случае значения передаются через параметры функции printf() в порядке указания символов в строке формата. Примеры:

```
printf("'%*.*f'", 10, 5, 3.14159265359); // '   3.14159'
printf("'%.*f'", 3, 3.14159265359);      // '3.142'
printf("'%*s'", 10, "string");           // '    string'
```

В первом примере вместо первого символа * подставляется число 10, указанное во втором параметре, а вместо второго символа * подставляется число 5, указанное в третьем параметре. Во втором примере вместо символа * подставляется число 3, которое задает количество цифр после точки. В третьем примере символ * заменяется числом 10, которое задает минимальную ширину поля.

В параметре <Флаги> могут быть указаны следующие символы:

- # — для восьмеричных значений добавляет в начало символ 0, для шестнадцатеричных значений добавляет комбинацию символов 0x (если используется тип x) или 0X (если используется тип X), для вещественных чисел указывает всегда выводить дробную точку, даже если задано значение 0 в параметре <Точность>:

```
printf("%#o %#o", 10, 077);         // 012 077
printf("%#x %#x", 10, 0xff);        // 0xa 0xff
printf("%#X %#X", 10, 0xff);        // 0XA 0XFF
printf("%#.0f %.0f", 100.0, 100.0); // 100. 100
```

- 0 — задает наличие ведущих нулей для числового значения:

```
printf("'%7d'", 100);  // '    100'
printf("'%07d'", 100); // '0000100'
```

- - — задает выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе. Пример:

```
printf("'%5d' '%-5d'", 3, 3);   // '    3' '3    '
```

- пробел — вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус. Пример:

```
printf("'% d' '% d'", -3, 3); // '-3' ' 3'
```

- + — задает обязательный вывод знака как для отрицательных, так и для положительных чисел. Пример:

```
printf("'%+d' '%+d'", -3, 3); // '-3' '+3'
```

В параметре <Размер> могут быть указаны следующие буквы:

- h — для вывода значения переменной, имеющей тип short int. Пример:

```
short int x = 32767;
printf("%hd", x); // 32767
unsigned short int y = 65535;
printf("%hu", y); // 65535
```

  При передаче в функцию значение типа short автоматически расширяется до типа int, поэтому букву h можно вообще не указывать:

```
short x = 32767;
printf("%d", x);  // 32767
unsigned short y = 65535;
printf("%u", y);  // 65535
```

- l (буква "эль") — для вывода значения переменной, имеющей тип long int. Пример:

```
long int x = 2147483647L;
printf("%ld", x); // 2147483647
```

  Модификатор l можно использовать совместно с типами c и s, для вывода двухбайтового символа и строки, состоящей из двухбайтовых символов, соответственно. Пример:

```
wchar_t str[] = L"string";
printf("%ls", str); // string
```

- I64 — для вывода значения переменной, имеющей тип long long int. Используется в Windows. Пример:

```
long long x = 9223372036854775807LL;
printf("%I64d\n", x); // 9223372036854775807
unsigned long long y = 18446744073709551615ULL;
printf("%I64u\n", y); // 18446744073709551615
```

- ll — для вывода значения переменной, имеющей тип long long. При использовании функции printf() в Windows при компиляции в MinGW выводятся предупреждения. В Windows можно либо указать I64 вместо ll, либо использовать функцию __mingw_printf() вместо printf(). Пример:

```
long long x = 9223372036854775807LL;
__mingw_printf("%lld", x); // 9223372036854775807
```

- `L` — для вывода значения переменной, имеющей тип `long double`. Для того чтобы значение в MinGW в Windows выводилось правильно, нужно использовать функцию `__mingw_printf()` вместо `printf()`. Пример:

```
long double x = 8e+245L;
__mingw_printf("%Le\n", x); // 8.000000e+245
```

Если выводить несколько значений подряд с помощью функции `printf()`, то они все отобразятся на одной строке:

```
printf("string1");
printf("string2");
// string1string2
```

Для того чтобы значения выводились на отдельных строках, нужно воспользоваться комбинацией символов `\n`, обозначающей перевод строки:

```
printf("string1\n");
printf("string2");
/*
string1
string2 */
```

С помощью стандартного вывода можно создать индикатор выполнения процесса в окне консоли. Для того чтобы реализовать такой индикатор, нужно вспомнить, что символ перевода строки в Windows состоит из двух символов — `\r` (перевод каретки) и `\n` (перевод строки). Таким образом, используя только символ перевода каретки `\r`, можно перемещаться в начало строки и перезаписывать ранее выведенную информацию. Пример индикатора процесса показан в листинге 2.5.

Листинг 2.5. Индикатор выполнения процесса

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

int main(void) {
   printf("... 0%%");
   for (int i = 5; i < 101; i += 5) {
      Sleep(1000); // Имитация процесса
      printf("\r... %d%%", i);
      fflush(stdout);
   }
   printf("\n");
   return 0;
}
```

Открываем командную строку (в редакторе Eclipse эффекта не будет видно), компилируем программу и запускаем:

```
C:\book>gcc -Wall -Wconversion -O3 -o helloworld.exe helloworld.c

C:\book>helloworld.exe
```

Эта программа немного сложнее, чем простое приветствие из листинга 2.1. Здесь присутствует имитация процесса с помощью функции `Sleep()` (при этом программа "засыпает" на указанное количество миллисекунд). Прототип функции:

```
#include <windows.h>
VOID Sleep(DWORD dwMilliseconds);
```

Тип данных `DWORD` объявлен так:

```
typedef unsigned long DWORD;
```

Кроме того, в программе использован цикл `for`, который позволяет изменить порядок обработки инструкций. Обычно программа выполняется сверху вниз и слева направо. Инструкция за инструкцией. Цикл `for` меняет эту последовательность выполнения. Инструкции, расположенные внутри блока, выполняются несколько раз. Количество повторений зависит от выражений внутри круглых скобок. Этих выражений три, и разделены они точками с запятой. Первое выражение объявляет целочисленную переменную `i` и присваивает ей значение `5`. Второе выражение является условием продолжения повторений. Пока значение переменной `i` меньше значения `101`, инструкции внутри блока будут повторяться. Это условие проверяется на каждой итерации цикла. Третье выражение на каждой итерации цикла прибавляет значение `5` к текущему значению переменной `i`.

Так как данные перед выводом могут помещаться в буфер, вполне возможно потребуется сбросить буфер явным образом. Сделать это можно с помощью функции `fflush()`. Прототип функции:

```
#include <stdio.h>
int fflush(FILE *stream)
```

В качестве параметра указывается стандартный поток вывода `stdout`:

```
fflush(stdout);
```

2.9. Ввод данных

Выводить данные на консоль мы научились, теперь рассмотрим функции, предназначенные для получения данных от пользователя. Все эти функции объявлены в заголовочном файле stdio.h.

2.9.1. Ввод одного символа

Для ввода одного символа предназначена функция `getchar()`. Прототип функции:

```
#include <stdio.h>
int getchar(void);
```

В качестве значения функция возвращает код введенного символа. Для того чтобы символ был считан, необходимо после ввода символа нажать клавишу <Enter>. Если было введено несколько символов, то будет считан первый символ, а остальные

ные останутся в буфере. Ключевое слово `void` внутри круглых скобок означает, что функция не принимает никаких параметров. Пример получения символа:

```
int ch;
printf("ch = ");          // Вывод подсказки
fflush(stdout);           // Сброс буфера
ch = getchar();           // Получение символа
printf("%d\n", ch);       // Вывод кода
printf("%c\n", (char)ch); // Вывод символа
```

> ***Обратите внимание***
>
> **Символ, который мы получим, будет в кодировке консоли. По умолчанию кодировка в консоли windows-866, но пользователь может сменить кодировку с помощью команды `chcp <Кодировка>`. Учитывайте, что в программе строки у нас в кодировке windows-1251, а не в кодировке windows-866.**

2.9.2. Функция *scanf()*

Для получения и автоматического преобразования данных в конкретный тип (например, в целое число) предназначена функция `scanf()`. При вводе строки функция не производит никакой проверки длины строки, что может привести к переполнению буфера. Обязательно указывайте ширину при использовании спецификатора `%s` (например, `%255s`). Прототип функции:

```
#include <stdio.h>
int scanf(const char *format, ...);
```

В первом параметре указывается строка специального формата, внутри которой задаются спецификаторы, аналогичные применяемым в функции `printf()`, а также некоторые дополнительные спецификаторы. В последующих параметрах передаются адреса переменных. Функция возвращает количество произведенных присваиваний. Пример получения целого числа:

```
// #include <locale.h>
setlocale(LC_ALL, "Russian_Russia.1251"); // Настройка локали
int x = 0;
printf("Введите число: ");
fflush(stdout);               // Сброс буфера вывода
fflush(stdin);                // Очистка буфера ввода
int status = scanf("%d", &x); // Символ & обязателен!!!
if (status == 1) {
   printf("Вы ввели: %d\n", x);
}
else {
   puts("Ошибка при вводе числа");
}
printf("status = %d\n", status);
```

Обратите внимание на то, что перед названием переменной `x` указан символ `&`. В этом случае передается *адрес переменной* `x`, а *не ее значение*. При вводе строки

символ & не указывается, т. к. название переменной без квадратных скобок является ссылкой на первый элемент массива. Пример ввода строки:

```
// #include <stdlib.h>
system("chcp 1251");                          // Смена кодировки консоли
// #include <locale.h>
setlocale(LC_ALL, "Russian_Russia.1251"); // Настройка локали
char str[256] = "";
printf("Введите строку: ");
fflush(stdout);                    // Сброс буфера вывода
fflush(stdin);                     // Очистка буфера ввода
int status = scanf("%255s", str); // Символ & не указывается!!!
if (status == 1) {
   printf("Вы ввели: %s\n", str);
}
else {
   puts("Ошибка при вводе строки");
}
```

Обратите внимание: чтобы получить строку в кодировке windows-1251, мы вначале указываем нужную кодировку с помощью функции `system()`. Если кодировку не указать, то мы получим строку в кодировке консоли (по умолчанию в консоли используется кодировка windows-866). Далее мы настраиваем локаль, чтобы не было проблем при работе с русскими буквами. Не забудьте подключить заголовочные файлы **stdlib.h** и **locale.h**, в которых содержатся объявления функций `system()` и `setlocale()`.

Считывание символов производится до первого символа-разделителя, например пробела, табуляции или символа перевода строки. Поэтому после ввода строки `"Hello, world!"` переменная `str` будет содержать лишь фрагмент `"Hello,"`, а не всю строку.

В качестве примера произведем суммирование двух целых чисел, введенных пользователем (листинг 2.6).

Листинг 2.6. Суммирование двух введенных чисел

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int x = 0, y = 0;
   printf("Введите первое число: ");
   fflush(stdout);                    // Сброс буфера вывода
   if (scanf("%d", &x) != 1) {
      puts("Вы ввели не число");
      return 1;                       // Выходим из функции main()
   }
```

```
    printf("Введите второе число: ");
    fflush(stdout);                    // Сброс буфера вывода
    fflush(stdin);                     // Очистка буфера ввода
    if (scanf("%d", &y) != 1) {
       puts("Вы ввели не число");
       return 1;                       // Выходим из функции main()
    }
    printf("Сумма равна: %d\n", x + y);
    return 0;
}
```

В первых двух строках производится подключение заголовочных файлов stdio.h и locale.h. Далее внутри функции `main()` с помощью функции `setlocale()` настраивается локаль, а затем объявляются две локальные переменные: `x` и `y`. Ключевое слово `int` в начале строки означает, что объявляются целочисленные переменные. При объявлении переменным сразу присваивается начальное значение (`0`) с помощью оператора `=`. Если значение не присвоить, то переменная будет иметь произвольное значение, так называемый "мусор". Как видно из примера, на одной строке можно объявить сразу несколько переменных, разделив их запятыми.

В следующей строке с помощью функции `printf()` выводится подсказка для пользователя (`"Введите первое число: "`). Благодаря этой подсказке пользователь будет знать, что от него требуется. Далее идет вызов функции `fflush()`, которая сбрасывает данные из буфера потока вывода `stdout` в консоль. Если буфер вывода принудительно не сбросить, пользователь может не увидеть подсказку вообще.

После ввода числа и нажатия клавиши <Enter> значение будет присвоено переменной `x`. Ввод значения производится с помощью функции `scanf()`. В первом параметре с помощью спецификатора `%d` мы указываем, что хотим получить целое число. Во втором параметре передаем адрес переменной `x` в памяти компьютера, указав перед ее именем символ `&`. Благодаря этому введенное значение будет записано в переменную.

Пользователь вместо числа может ввести все что угодно, например строку, не содержащую число. В этом случае мы не получим число, а значит, и складывать будет нечего. Поэтому после получения числа с помощью оператора `if` мы проверяем значение, возвращаемое функцией `scanf()`. Функция возвращает количество успешных операций присваивания. Если количество присваиваний не равно одному, то с помощью функции `puts()` выводим сообщение об ошибке и завершаем выполнение функции `main()`, вызвав оператор `return`.

В этом примере для проверки условия используется оператор ветвления `if`. После названия оператора внутри круглых скобок указывается проверяемое выражение. Если выражение возвращает логическое значение `Истина`, то будут выполнены инструкции внутри фигурных скобок. Если выражение возвращает значение `Ложь`, то инструкции внутри фигурных скобок игнорируются и управление передается инструкции, расположенной сразу после закрывающей фигурной скобки. Проверка зна-

чения, возвращаемого функцией `scanf()`, осуществляется с помощью оператора `!=` (не равно).

Если первое число успешно получено, то выводим подсказку пользователю для ввода второго числа и сбрасываем буфер в окно консоли. Далее, перед получением второго числа, с помощью функции `fflush()` очищаем буфер потока ввода `stdin`. Если не очистить буфер ввода перед повторным получением числа, то это число может быть получено из предыдущего ввода из буфера. Например, при запросе первого числа было введено значение `"47 3"`. Функция `scanf()` получит число `47`, а второе число оставит в буфере, и оно будет доступно для следующей операции ввода. Если мы сейчас запустим код из листинга 2.6, то буфер ввода будет очищен:

```
Введите первое число: 47 3
Введите второе число: 5
Сумма равна: 52
```

Если закомментировать строку (инструкция работает только в Windows и то не во всех версиях):

```
fflush(stdin);                    // Очистка буфера ввода
```

то второе число будет получено сразу из буфера, не дожидаясь ввода пользователя, и результат будет другим:

```
Введите первое число: 47 3
Введите второе число: Сумма равна: 50
```

Далее производится получение второго числа, которое сохраняется в переменной `y`. Затем с помощью оператора `if` мы проверяем значение, возвращаемое функцией `scanf()`. Если количество присваиваний не равно одному, то с помощью функции `puts()` выводим сообщение об ошибке и завершаем выполнение функции `main()`, вызвав оператор `return`.

И, наконец, производим сложение двух чисел и выводим результат с помощью функции `printf()`.

В листинге 2.6 при первой попытке ввода букв вместо числа выводится сообщение об ошибке, и программа завершается, не давая пользователю возможности исправить свою ошибку. Предоставим пользователю три попытки ввода, прежде чем досрочно завершим программу (листинг 2.7).

Листинг 2.7. Суммирование двух чисел с обработкой ошибок ввода данных

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int x = 0, y = 0, flag = 0, count = 1;
   do {
      printf("Введите первое число: ");
```

```
      fflush(stdout);
      fflush(stdin);
      if (scanf("%d", &x) != 1) {
         puts("Вы ввели не число");
         // Сбрасываем флаг ошибки и очищаем буфер ввода
         if (feof(stdin) || ferror(stdin)) clearerr(stdin);
         int ch = 0;
         while ((ch = getchar()) != '\n' && ch != EOF);
         ++count;
         if (count > 3) {
            puts("Вы сделали три ошибки");
            return 1;
         }
      }
      else flag = 1;
   } while (!flag);
   flag = 0;
   count = 1;
   do {
      printf("Введите второе число: ");
      fflush(stdout);
      fflush(stdin);
      if (scanf("%d", &y) != 1) {
         puts("Вы ввели не число");
         // Сбрасываем флаг ошибки и очищаем буфер ввода
         if (feof(stdin) || ferror(stdin)) clearerr(stdin);
         int ch = 0;
         while ((ch = getchar()) != '\n' && ch != EOF);
         ++count;
         if (count > 3) {
            puts("Вы сделали три ошибки");
            return 1;
         }
      }
      else flag = 1;
   } while (!flag);
   printf("Сумма равна: %d\n", x + y);
   return 0;
}
```

В этом примере внутри функции `main()` объявляются четыре целочисленные локальные переменные: `x`, `y`, `flag` и `count`. В переменной `flag` мы будем сохранять текущий статус обработки ошибок, а в переменной `count` — количество допущенных ошибок подряд.

При возникновении ошибки ее необходимо обработать, а затем вывести повторный запрос на ввод числа. Вполне возможно эту процедуру нужно будет выполнить

несколько раз. Причем количество повторов заранее неизвестно. Для выполнения одних и тех же инструкций несколько раз предназначены циклы. В нашем примере используется цикл `do...while`. Инструкции внутри фигурных скобок будут выполняться до тех пор, пока логическое выражение после ключевого слова `while` является истинным. Проверка условия производится после выполнения инструкций внутри фигурных скобок, поэтому инструкции выполняются минимум один раз. Обратите внимание на логическое выражение `!flag`. Восклицательный знак, расположенный перед переменной, инвертирует значение. Например, если переменная содержит значение `Истина` (любое число, отличное от нуля), то выражение вернет значение `Ложь`. Так как проверка значения на истинность производится по умолчанию, выражение может не содержать операторов сравнения.

Внутри цикла выводим подсказку пользователю, сбрасываем буфер вывода и очищаем буфер ввода. Если буфер ввода не очистить, то значение будет автоматически передано следующей операции ввода, что породит повторную ошибку. Затем получаем значение и сохраняем его в переменной. Далее проверяем успешность получения значения. Если пользователь не ввел число, то выводим сообщение `Вы ввели не число`. Количество попыток мы отслеживаем с помощью переменной `count`. На начальном этапе она имеет значение `1`. Если пользователь не ввел число, то увеличиваем значение на `1` с помощью инструкции:

```
++count;
```

Если пользователь допустил три ошибки подряд (переменная `count` будет содержать значение больше `3`), то выводим сообщение `Вы сделали три ошибки` и выходим из функции `main()`, вызвав оператор `return`.

Если ошибки нет, выполняются инструкции, расположенные после ключевого слова `else`. В нашем случае переменной `flag` присваивается значение `1`. Это значение является условием выхода из цикла.

Далее таким же способом получаем второе число. Так как в предыдущем цикле значения переменных `flag` и `count` были изменены, перед циклом производим восстановление первоначальных значений. Иначе выход из цикла произойдет даже в случае ошибки. Если второе число успешно получено, производим вывод суммы чисел.

2.9.3. Ввод строки

Для ввода строки предназначена функция `gets()`, однако применять ее в программе не следует, т. к. функция не производит никакой проверки длины строки, что может привести к переполнению буфера. Прототип функции `gets()`:

```
#include <stdio.h>
char *gets(char *buffer);
```

Лучше получать строку посимвольно с помощью функции `getchar()` или воспользоваться функцией `fgets()`. Прототип функции `fgets()`:

```
#include <stdio.h>
char *fgets(char *buf, int maxCount, FILE *stream);
```

В качестве параметра `stream` указывается стандартный поток ввода `stdin`. Считывание производится до первого символа перевода строки или до конца потока либо пока не будет прочитано `maxCount-1` символов. Содержимое строки `buf` включает символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ добавлен не будет. Если произошла ошибка или достигнут конец потока, то функция возвращает нулевой указатель. Пример получения строки приведен в листинге 2.8.

Листинг 2.8. Получение строки с помощью функции `fgets()`

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include <stdlib.h>

int main(void) {
   system("chcp 1251"); // Смена кодировки консоли
   setlocale(LC_ALL, "Russian_Russia.1251");
   char buf[256] = "", *p = NULL;
   printf("Введите строку: ");
   fflush(stdout);
   fflush(stdin);
   p = fgets(buf, 256, stdin);
   if (p) {
      // Удаляем символ перевода строки
      size_t len = strlen(buf);
      if (len != 0 && buf[len - 1] == '\n') {
         buf[len - 1] = '\0';
      }
      // Выводим результат
      printf("Вы ввели: %s\n", buf);
   }
   else puts("Возникла ошибка");
   return 0;
}
```

В первой строке включается заголовочный файл **stdio.h**, в котором объявлены функции, предназначенные для ввода и вывода данных, во второй строке — файл **string.h**, в котором объявлена функция `strlen()`, возвращающая длину строки, в третьей строке — файл **locale.h**, в котором объявлена функция `setlocale()`, а в четвертой строке — файл **stdlib.h**, в котором объявлена функция `system()`.

Внутри функции `main()` сначала изменяется кодировка консоли и настраивается локаль. Строка, которую мы получим по умолчанию, будет в кодировке консоли (например, в кодировке windows-866 или какой-либо другой). Для того чтобы получить строку в кодировке windows-1251, мы в начале программы передаем команду смены кодировки `chcp 1251` функции `system()`.

Далее объявляется символьный массив `buf`, состоящий из 256 символов. Строки в языке C представляют собой последовательность (массив) символов, последним элементом которого является нулевой символ (`'\0'`). Обратите внимание на то, что нулевой символ (нулевой байт) не имеет никакого отношения к символу `'0'`. Коды этих символов разные.

В первой строке объявляется также указатель `p` на тип `char` и ему присваивается значение `NULL`. То, что переменная `p` является указателем, говорит символ `*` перед ее именем при объявлении. Значением указателя является адрес данных в памяти компьютера. Указатель, которому присвоено значение `NULL`, называется *нулевым указателем*. Такой указатель ни на что не указывает, пока ему не будет присвоен адрес.

В следующей строке выводится подсказка пользователю, а последующие инструкции сбрасывают буфер вывода и очищают буфер ввода. Далее мы получаем строку из стандартного потока ввода `stdin`. Поток указывается в третьем параметре функции `fgets()`. В первом параметре передается адрес символьного массива `buf`, а во втором — его размер. Если в процессе выполнения функции `fgets()` возникнет ошибка, то функция вернет нулевой указатель. В этом случае мы выводим сообщение `Возникла ошибка`. Если ошибка не возникла, то удаляем символ перевода строки, который был добавлен в символьный массив. Для этого с помощью функции `strlen()` получаем длину строки и сохраняем ее в переменной `len`. Затем проверяем значение последнего символа в строке. Если это символ `\n`, то заменяем его нулевым символом. В противном случае ничего не заменяем. После замены выводим полученную строку в окно консоли.

Обратите внимание на два момента. Во-первых, размер символьного массива и длина строки — это разные вещи. *Размер массива* — это общее количество символов, которое может хранить массив. *Длина строки* — это количество символов внутри символьного массива до первого нулевого символа. В следующем примере массив содержит 256 символов, а длина строки внутри него составляет всего три символа плюс нулевой символ:

```
char buf[256] = "abc";
printf("%s\n", buf);                  // abc
printf("%d\n", (int)sizeof(buf));     // 256
printf("%d\n", (int)strlen(buf));     // 3
```

Во-вторых, нумерация элементов массива начинается с `0`, а не с `1`. Поэтому последний символ имеет индекс на единицу меньше длины строки. Для доступа к отдельному символу нужно указать индекс внутри квадратных скобок:

```
char buf[256] = "abc";
printf("%c\n", buf[0]);               // a
printf("%c\n", buf[1]);               // b
printf("%c\n", buf[2]);               // c
printf("%c\n", buf[strlen(buf) - 1]); // c
```

2.10. Интерактивный ввод символов

Функция `getchar()` позволяет получить символ только после нажатия клавиши <Enter>. Если необходимо получить символ сразу после нажатия клавиши на клавиатуре, то можно воспользоваться функциями `_getche()` и `_getch()`. Прототипы функций:

```
#include <conio.h>
int _getche(void);
int _getch(void);
```

Функция `_getche()` возвращает код символа и выводит его на экран. При нажатии клавиши функция `_getch()` возвращает код символа, но сам символ на экран не выводится. Это обстоятельство позволяет использовать функцию `_getch()` для получения конфиденциальных данных (например, пароля). Следует учитывать, что код символа возвращается и при нажатии некоторых служебных клавиш, например <Home>, <End> и др.

В качестве примера использования функции `_getch()` создадим программу для ввода пароля (листинг 2.9). Пароль может содержать до 16 символов (цифры от `0` до `9` и латинские буквы от `a` до `z` в любом регистре). При нажатии клавиши вместо символа будем выводить звездочку. Если символ не входит в допустимый набор, выведем сообщение об ошибке и завершим программу. После нажатия клавиши <Enter> выведем на экран полученный пароль.

Листинг 2.9. Ввод пароля без отображения вводимых символов

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char passwd[17] = "";
   int flag = 0, i = 0, ch = 0;
   printf("Введите пароль: ");
   fflush(stdout);
   do {
      ch = _getch();
      if (i > 15 || ch == '\r' || ch == '\n') {
         flag = 1;
         passwd[i] = '\0';
      }
      else if (   (ch > 47 && ch < 58)   // Цифры от 0 до 9
               || (ch > 64 && ch < 91)   // Буквы от A до Z
               || (ch > 96 && ch < 123)) // Буквы от a до z
      {
         passwd[i] = (char)ch;
         printf("%s", "*");
```

```
            fflush(stdout);
            ++i;
        }
        else { // Если недопустимый символ, то выходим
            puts("\n Вы ввели недопустимый символ");
            return 0;
        }
    } while (!flag);

    printf("\n Вы ввели: %s\n", passwd);
    return 0;
}
```

ОБРАТИТЕ ВНИМАНИЕ

Код из листинга 2.9 будет работать только в консоли. Запуск в редакторе Eclipse приведет к бесконечной работе программы.

В начале программы производим подключение файлов stdio.h, conio.h и locale.h. Файл stdio.h необходим для использования функций `printf()`, `fflush()` и `puts()`, а файл conio.h — для функции `_getch()`. Далее внутри функции `main()` объявляем четыре переменные. В символьный массив `passwd`, содержащий 17 элементов (16 символов плюс `'\0'`), будут записываться введенные символы. В переменную `ch` будем записывать код нажатой клавиши. Переменная `flag` предназначена для хранения статуса ввода (если содержит значение `Истина` (любое число, отличное от `0`), то ввод пароля закончен), а целочисленная переменная `i` предназначена для хранения текущего индекса внутри массива `passwd`. Обратите внимание на то, что первый элемент массива имеет индекс `0`, а не `1`.

После объявления переменных выводим подсказку пользователю и сбрасываем буфер. Затем внутри цикла `do...while` получаем текущий символ с помощью функции `_getch()` и сохраняем его код в переменной `ch`. В следующей строке проверяем выход индекса за границы массива (`i > 15`) и нажатие клавиши <Enter>. При нажатии клавиши <Enter> передается последовательность символов `\r\n` (перевод каретки плюс перевод строки), однако функция `_getch()` может получить только один символ, поэтому проверяется наличие или символа `\r`, или символа `\n`. Между собой логические выражения разделяются с помощью оператора `||` (логическое *или*). Для того чтобы все выражение вернуло истину, достаточно, чтобы хотя бы одно из трех логических выражений было истинным. Если выражение `i > 15` является ложным, то проверяется выражение `ch == '\r'`, в противном случае все выражение считается истинным и дальнейшая проверка не осуществляется. Если выражение `ch == '\r'` является ложным, то проверяется выражение `ch == '\n'`, в противном случае все выражение считается истинным и дальнейшая проверка не осуществляется. Если выражение `ch == '\n'` является ложным, то все выражение считается ложным, в противном случае все выражение считается истинным.

Если выражение является истинным, то ввод пароля закончен. Для того чтобы выйти из цикла, присваиваем переменной `flag` значение `1`. Кроме того, вставляем нуле-

вой символ в конец массива. Он будет обозначать конец строки с введенным паролем. После этих инструкций управление передается в конец цикла и производится проверка условия `!flag`. Так как условие вернет `Ложь` (`!Истина`), происходит выход из цикла. После выхода из цикла производим вывод введенного пароля на консоль.

Если ввод пароля не закончен, то производим проверку вхождения символа в допустимый диапазон значений (цифры от `0` до `9` и латинские буквы от `a` до `z` в любом регистре). Выполнить проверку допустимости символа необходимо, т. к. функция `_getch()` возвращает также коды некоторых служебных клавиш, например <Home>, <End> и др. Если условие не соблюдается, то выводим сообщение об ошибке и возвращаем значение `0`, тем самым завершая выполнение программы.

Проверяемое условие содержит более сложное выражение, нежели в предыдущем условии. Выражение разбито на несколько мелких выражений с помощью круглых скобок. Внутри первых круглых скобок проверяется соответствие символа диапазону чисел от `0` до `9`. Цифра `0` соответствует коду `48`, а цифра `9` — коду `57`. Коды остальных цифр находятся в диапазоне между этими кодами. Для того чтобы не перечислять в выражении коды всех цифр, используют проверку двух условий, разделенных оператором `&&` (логическое `И`). Выражение `ch > 47 && ch < 58` следует читать так: если переменная `ch` содержит символ, имеющий код больше `47` и меньше `58`, то вернуть значение `Истина`, в противном случае — значение `Ложь`. Если условие внутри первых скобок является истинным, то все выражение является истинным и дальнейшая проверка не производится. Если условие внутри первых круглых скобок является ложным, то проверяется условие внутри вторых круглых скобок. Условие внутри третьих круглых скобок проверяется, только если условие внутри вторых круглых скобок является ложным. Вместо кодов символов можно указать сами символы внутри апострофов. В этом случае условие будет выглядеть так:

```
else if (   (ch >= '0' && ch <= '9')   // Цифры от 0 до 9
         || (ch >= 'A' && ch <= 'Z')   // Буквы от A до Z
         || (ch >= 'a' && ch <= 'z'))  // Буквы от a до z
{
```

Если символ входит в допустимый диапазон, то сохраняем символ в массиве, выводим звездочку в окно консоли, сбрасываем буфер, а затем увеличиваем значение в переменной `i` на единицу, тем самым перемещая указатель текущей позиции на следующий элемент массива. Выражение `++i` соответствует выражению `i = i + 1`. После этих инструкций управление передается в конец цикла и производится проверка условия `!flag`. Так как условие вернет `Истина` (`!0`), инструкции внутри цикла будут выполнены еще раз.

2.11. Получение данных из командной строки

Передать данные можно в командной строке после названия файла. Для получения этих данных в программе используется следующий формат функции `main()`:

```
int main(int argc, char *argv[]) {
   // Инструкции
   return 0;
}
```

Через первый параметр (`argc`) доступно количество аргументов, переданных в командной строке. Следует учитывать, что первым аргументом является название исполняемого файла, поэтому значение параметра `argc` не может быть меньше единицы. Через второй параметр (`argv`) доступны все аргументы в виде строки (тип `char *`). Квадратные скобки после названия второго параметра означают, что доступен массив строк. Рассмотрим получение данных из командной строки на примере (листинг 2.10).

Листинг 2.10. Получение данных из командной строки

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   printf("argc = %d\n", argc);
   for (int i = 0; i < argc; ++i) {
      printf("%s\n", argv[i]);
   }
   return 0;
}
```

Сохраняем программу в файл C:\book\test.c. Запускаем командную строку и компилируем программу:

```
C:\Users\Unicross>cd C:\book

C:\book>set Path=C:\msys64\mingw64\bin;%Path%

C:\book>gcc -Wall -Wconversion -O3 -o test.exe test.c
```

Для запуска программы вводим команду:

```
C:\book>test.exe -param1 -param2
```

В этой команде мы передаем программе test.exe некоторые данные (`-param1 -param2`). Результат выполнения программы будет выглядеть так:

```
argc = 3
test.exe
-param1
-param2
```

Первый элемент массива (`argv[0]`) не всегда будет содержать только название исполняемого файла. Если в командной строке запуск производится следующим образом:

```
C:\book>C:\book\test.exe -param1 -param2
```

то элемент будет содержать не только название файла, но и путь к нему:

```
argc = 3
C:\book\test.exe
-param1
-param2
```

Если нужно передать значение, которое содержит пробел, то это значение следует указывать внутри кавычек:

```
C:\book>test.exe x + y "x + y"
argc = 5
test.exe
x
+
y
x + y
```

В первом случае мы получили каждый символ по отдельности, а во втором — все символы вместе, указав значение внутри кавычек.

2.12. Предотвращение закрытия окна консоли

До сих пор мы запускали программу либо в командной строке, либо в редакторе Eclipse. Однако программу можно запустить и двойным щелчком мыши на значке файла с программой. Если мы попробуем это сделать сейчас с программой test.exe из предыдущего раздела, то окно консоли откроется, а затем сразу закроется. Для того чтобы окно не закрывалось, необходимо вставить инструкцию, ожидающую нажатие клавиши. Сделать это можно несколькими способами.

Первый способ заключается в использовании функции `_getch()` (листинг 2.11). Возвращаемое функцией значение можно проигнорировать. Прототип функции:

```
#include <conio.h>
int _getch(void);
```

Листинг 2.11. Использование функции `_getch()`

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   printf("Hello, world!\n");
   printf("Для закрытия окна нажмите любую клавишу ... ");
   fflush(stdout);                // Сброс буфера вывода
   fflush(stdin);                 // Очистка буфера ввода
   _getch();
   return 0;
}
```

Второй способ заключается в использовании функции `system()` (листинг 2.12). Эта функция позволяет передать команду операционной системе. Для вывода строки `Для продолжения нажмите любую клавишу` и ожидания нажатия клавиши предназначена команда `pause`. Прототип функции:

```
#include <stdlib.h>
int system(const char *command);
```

Листинг 2.12. Использование функции `system()`

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

int main(void) {
   printf("Hello, world!\n");
   fflush(stdout);                // Сброс буфера вывода
   system("pause");
   return 0;
}
```

Два предыдущих способа требовали подключения дополнительных файлов. Кроме того, функция `_getch()` может не поддерживаться компилятором, а функция `system()` выполняет много лишних операций. Вместо этих способов лучше использовать функцию `getchar()`. Она позволяет получить введенный символ. Ввод осуществляется после нажатия клавиши <Enter>. Возвращаемое функцией значение можно проигнорировать. Прототип функции:

```
#include <stdio.h>
int getchar(void);
```

При использовании функции `getchar()` следует учитывать один нюанс. Если в программе производился ввод данных, то в буфере могут остаться символы. В этом случае первый символ автоматически будет передан функции `getchar()`, и окно консоли сразу закроется. Поэтому после ввода данных необходимо дополнительно очистить буфер потока ввода с помощью функции `fflush()`. Пример использования функции `getchar()` приведен в листинге 2.13.

Листинг 2.13. Использование функции `getchar()`

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   printf("Hello, world!\n");
   printf("Для закрытия окна нажмите <Enter> ... ");
   fflush(stdout);                // Сброс буфера вывода
```

```
    fflush(stdin);                  // Очистка буфера ввода
    getchar();
    return 0;
}
```

Если нужно предотвратить закрытие окна только на этапе отладки программы, то можно создать вспомогательный файл с расширением bat и внутри него запускать программу. После инструкции запуска программы вставляем команду `pause`, которая выведет строку `Для продолжения нажмите любую клавишу` и будет ожидать нажатия клавиши. Пример:

```
@echo off
title Запуск программы test.exe
echo Результат:
@echo.
test.exe -param1 -param2
@echo.
@echo.
pause
```

Как видно из примера, после названия программы мы можем передать параметры. Если программа test.exe расположена в другом каталоге, то перед именем программы следует указать путь к ней.

2.13. Настройка отображения русских букв в консоли

Если мы сохраним файл с программой в кодировке windows-1251, то результат следующей инструкции:

```
printf("Привет, мир!");
```

в окне консоли будет выглядеть так:

```
╧Ёштхє, ьшЁ!
```

Причина искажения русских букв заключается в том, что по умолчанию в окне консоли используется кодировка windows-866, а в программе мы ввели текст в кодировке windows-1251. Коды русских букв в этих кодировках различаются, поэтому происходит искажение. При использовании командной строки пользователь может сменить кодировку вручную, выполнив команду:

```
chcp 1251
```

Однако этого недостаточно. Кроме смены кодировки необходимо изменить название шрифта, т. к. по умолчанию используются точечные шрифты, которые не поддерживают кодировку windows-1251. Для нормального отображения русских букв следует в свойствах окна выбрать шрифт Lucida Console (рис. 2.14). Все эти действия вы можете произвести на своем компьютере, однако пользователи не знают, в какой кодировке выводятся данные в вашей программе.

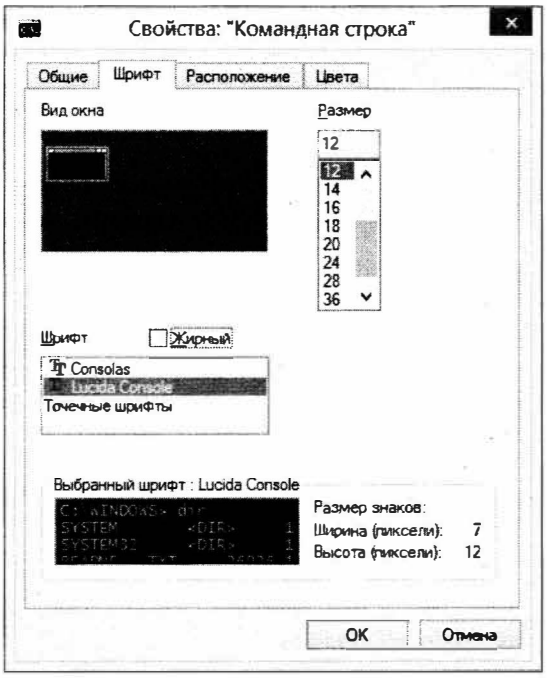


Рис. 2.14. Указание шрифта Lucida Console в свойствах окна консоли

Изменить кодировку из программы мы можем с помощью функции `system()`:

```
system("chcp 1251");
```

А вот изменить шрифт из программы проблематично. Поэтому никакой гарантии, что пользователь увидит русские буквы без искажений, нет.

Консоль в Windows по умолчанию работает с кодировкой windows-866, поэтому мы можем и файл с программой сохранить в этой кодировке. При использовании редактора Notepad++ вначале создаем новый документ, а затем в меню **Кодировки** выбираем пункт **Кодировки | Кириллица | OEM 866**. Вводим текст программы и сохраняем файл. В редакторе Eclipse кодировка задается в свойствах проекта.

Вроде все отлично, т. к. мы сохраняем файл в кодировке консоли по умолчанию:

```
C:\book>helloworld.exe
Привет, мир!
```

Однако пользователь может сменить кодировку в консоли, и мы опять получим проблему с русскими буквами:

```
C:\book>chcp 1251
Текущая кодовая страница: 1251

C:\book>helloworld.exe
ЏаЁўЃв, ¬Ёа!
```

Кроме того, при сохранении файла с программой в кодировке windows-866 мы получим множество проблем при работе с файлами и каталогами.

Преобразование кодировки в Windows производится автоматически после настройки локали (локальных настроек компьютера). Настроить локаль позволяет функция `setlocale()`. Прототип функции:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

В первом параметре указывается категория в виде числа от `0` до `5`. Вместо чисел можно использовать макроопределения `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC` и `LC_TIME`. Во втором параметре задается название локали, после которого через точку указывается кодировка файла с программой (например, `Russian_Russia.1251` или `Russian_Russia.866`).

Пример настройки локали и вывода русских букв приведен в листинге 2.14.

Листинг 2.14. Настройка локали

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   printf("Привет, мир!\n");
   return 0;
}
```

После компиляции и запуска программы русские буквы будут правильно отображаться вне зависимости от текущей кодировки консоли:

```
C:\book>chcp
Текущая кодовая страница: 866

C:\book>helloworld.exe
Привет, мир!

C:\book>chcp 1251
Текущая кодовая страница: 1251

C:\book>helloworld.exe
Привет, мир!
```

Однако программа может быть запущена на компьютере, в котором кодировка консоли не позволяет отобразить русские буквы. Единственный способ полностью решить проблему с кодировками — выводить сообщения на английском языке. Коды латинских букв во всех однобайтовых кодировках одинаковые, поэтому при использовании английского языка проблем не будет. Однако тут существует другая сложность. Пользователь может не знать английского языка. Как видите, не все так просто с кодировками при использовании консольных приложений.

2.14. Преждевременное завершение выполнения программы

В некоторых случаях может возникнуть условие, при котором дальнейшее выполнение программы лишено смысла, например, отсутствует свободная память при использовании динамической памяти. В этом случае имеет смысл вывести сообщение об ошибке и прервать выполнение программы досрочно. Для этого предназначена функция `exit()`. Прототип функции:

```
#include <stdlib.h>
void exit(int code);
```

В качестве параметра функция принимает число, которое является статусом завершения. Число `0` означает нормальное завершение программы, а любое другое число — некорректное завершение. Эти числа передаются операционной системе. Вместо чисел можно использовать макроопределения `EXIT_SUCCESS` (нормальное завершение) и `EXIT_FAILURE` (аварийное завершение). Определения макросов:

```
#include <stdlib.h>
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Пример:

```
exit(EXIT_FAILURE); // Аналогично exit(1);
```

Помимо функции `exit()` для аварийного завершения программы предназначена функция `abort()`. В этом случае завершение программы осуществляется операционной системой с выводом диалогового окна. Прототип функции:

```
#include <stdlib.h>
void abort(void);
```

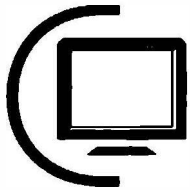
В качестве примера получим число от пользователя и выведем результат. При этом обработаем ошибку ввода (листинг 2.15).

Листинг 2.15. Преждевременное завершение выполнения программы

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int x = 0;
   printf("Введите число: ");
   fflush(stdout);
   if (scanf("%d", &x) != 1) {
      puts("Вы ввели не число");
```

```
        // Завершаем выполнение программы
        exit(EXIT_FAILURE); // Аналогично exit(1);
    }
    printf("Вы ввели: %d\n", x);
    return 0;
}
```

В этом примере вместо функции `exit()` можно было воспользоваться инструкцией `return`, т. к. завершение программы выполнялось внутри функции `main()`. Однако в больших программах основная часть кода расположена вне функции `main()`, и в этом случае инструкцией `return` для завершения всей программы уже не обойтись.



ГЛАВА 3

Переменные и типы данных

Переменные — это участки памяти, используемые программой для хранения данных. Говоря простым языком: переменная — это коробка, в которую мы можем что-то положить и из которой потом вытащить. Поскольку таких коробок может быть много, то каждая коробка подписывается (каждая переменная имеет уникальное имя внутри программы). Коробки могут быть разного размера. Например, необходимо хранить яблоко и арбуз. Согласитесь, размеры яблока и арбуза различаются. Для того чтобы поместить арбуз, мы должны взять соответствующего размера коробку. Таким образом, тип данных при объявлении переменной задает размер коробки, которую нужно подготовить, и определяет, что мы туда будем класть. Кроме того, в одну коробку мы можем положить только один предмет. Если нам нужно положить несколько яблок, то мы уже должны взять ящик (который в языке программирования называется *массивом*) и складывать туда коробки с яблоками.

3.1. Объявление переменной

Прежде чем использовать переменную, ее необходимо предварительно объявить глобально (вне функций) или локально (внутри функции). В большинстве случаев объявление переменной является сразу и ее определением. *Глобальные переменные* видны внутри всех функций в файле, а *локальные переменные* — только внутри той функции, в которой они объявлены. Для объявления переменной используется следующий формат:

```
[<Спецификатор>][ <Модификатор>] <Тип> <Переменная1>[=<Значение 1>]
                                 [, ..., <ПеременнаяN>[=<Значение N>]];
```

Пример объявления целочисленной переменной `x`:

```
int x;
```

В одной инструкции можно объявить сразу несколько переменных, указав их через запятую после названия типа данных:

```
int x, y, z;
```

На заметку

При использовании старых компиляторов все локальные переменные должны быть объявлены в самом начале функции.

3.2. Именование переменных

Каждая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр и знака подчеркивания, причем имя переменной не может начинаться с цифры. При указании имени переменной важно учитывать регистр букв: `x` и `X` — разные переменные:

```
int x = 5, X = 10;
printf("%d\n", x);  // 5
printf("%d\n", X);  // 10
```

В качестве имени переменной нельзя использовать ключевые слова. Список ключевых слов приведен в табл. 3.1 (в скобках указан стандарт, начиная с которого название стало ключевым словом). Следует учитывать, что в некоторых компиляторах могут быть определены дополнительные ключевые слова. Запоминать все ключевые слова нет необходимости, т. к. в редакторе эти слова подсвечиваются. Любая попытка использования ключевого слова вместо названия переменной приведет к ошибке при компиляции. Помимо ключевых слов следует избегать совпадений со встроенными идентификаторами.

- **Правильные имена переменных:** `x`, `y1`, `str_name`, `strName`.
- **Неправильные имена переменных:** `1y`, `ИмяПеременной`. **Последнее имя неправильное, т. к. в нем используются русские буквы.**

Таблица 3.1. Ключевые слова языка C

asm	double	int	struct	_Alignof (C11)
auto	else	long	switch	_Atomic (C11)
break	enum	register	typedef	_Bool (C99)
case	extern	restrict (C99)	union	_Complex (C99)
char	float	return	unsigned	_Generic (C11)
const	for	short	void	_Imaginary (C99)
continue	goto	signed	volatile	_Noreturn (C11)
default	if	sizeof	while	_Static_assert (C11)
do	inline (C99)	static	_Alignas (C11)	_Thread_local (C11)

3.3. Типы данных

В языке C доступны следующие элементарные типы данных.

- `_Bool` — логический тип данных. Может содержать значения `Истина` (соответствует числу `1`) или `Ложь` (соответствует числу `0`). Тип данных `_Bool` доступен, начиная со стандарта C99. Пример:

```
_Bool is_int = 1;
printf("%d\n", is_int);              // 1
printf("%d\n", !is_int);             // 0
// Выводим размер в байтах
printf("%d\n", (int) sizeof(_Bool)); // 1
```

  В языке C нет ключевых слов `true` (`Истина`) и `false` (`Ложь`). Вместо них используются числа `1` и `0` соответственно. Любое число, отличное от нуля, является `Истиной`, а число, равное нулю, — `Ложью`. Если вы хотите объявлять логические переменные так же как в языке C++, можно подключить заголовочный файл stdbool.h, в котором определены макросы `bool`, `true` и `false`:

```
#include <stdbool.h>
#define bool    _Bool
#define true    1
#define false   0
```

  Пример:

```
bool is_int = true;
printf("%d\n", is_int);              // 1
is_int = false;
printf("%d\n", is_int);              // 0
```

- `char` — код символа. Занимает 1 байт. Пример:

```
char ch = 'w';
// Выводим символ
printf("%c\n", ch);                  // w
// Выводим код символа
printf("%d\n", ch);                  // 119
// Выводим размер в байтах
printf("%d\n", (int) sizeof(char));  // 1
// #include <limits.h>
printf("%d\n", CHAR_MIN);            // -128
printf("%d\n", CHAR_MAX);            // 127
printf("%d\n", CHAR_BIT);            // 8
```

- `int` — целое число со знаком. Диапазон значений зависит от компилятора. Минимальный диапазон от `-32 768` до `32 767`. На практике диапазон от `-2 147 483 648` до `2 147 483 647` (занимает 4 байта). Пример:

```
int x = 10;
// Выводим десятичное значение
printf("%d\n", x);                   // 10
printf("%i\n", x);                   // 10
```

```
// Выводим восьмеричное значение
printf("%o\n", x);                    // 12
// Выводим шестнадцатеричное значение
printf("%x\n", x);                    // a
printf("%X\n", x);                    // A
// Выводим размер в байтах
printf("%d\n", (int) sizeof(int));    // 4
// #include <limits.h>
printf("%d\n", INT_MIN);              // -2147483648
printf("%d\n", INT_MAX);              // 2147483647
```

- `float` — вещественное число. Занимает 4 байта. Пример:

```
float x = 10.5432f;
// Выводим значение
printf("%f\n", x);                    // 10.543200
printf("%.2f\n", x);                  // 10.54
// Выводим размер в байтах
printf("%d\n", (int) sizeof(float));  // 4
```

- `double` — вещественное число двойной точности. Занимает 8 байтов. Пример:

```
double x = 10.5432;
// Выводим значение
printf("%f\n", x);                    // 10.543200
printf("%.2f\n", x);                  // 10.54
// Выводим размер в байтах
printf("%d\n", (int) sizeof(double)); // 8
```

- `void` — означает отсутствие типа. Используется в основном для того, чтобы указать, что функция не возвращает никакого значения или не принимает параметров, а также для передачи в функцию данных произвольного типа. Пример объявления функции, которая не возвращает значения и принимает указатель на данные произвольного типа:

```
void free(void *memory);
```

Перед элементарным типом данных могут быть указаны следующие модификаторы или их комбинация.

- `signed` — указывает, что символьный или целочисленный типы могут содержать отрицательные значения.

  Тип `signed char` может хранить значения от `-128` до `127`. Пример:

```
signed char min = -128, max = 127, ch = 'w';
// Выводим символ
printf("%c\n", ch);                   // w
// Выводим код
printf("%d\n", ch);                   // 119
printf("%d\n", min);                  // -128
printf("%d\n", max);                  // 127
```

```
// Выводим размер в байтах
printf("%d\n", (int) sizeof(ch));      // 1
// #include <limits.h>
printf("%d\n", SCHAR_MIN);             // -128
printf("%d\n", SCHAR_MAX);             // 127
```

Тип `signed int` **(или просто** `signed`**) соответствует типу** `int`**. Пример:**

```
signed int min = -2147483647 - 1;
signed max = 2147483647;
printf("%d\n", min);                   // -2147483648
printf("%i\n", max);                   // 2147483647
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));     // 4
```

- `unsigned` — **указывает, что символьный или целочисленный типы не могут содержать отрицательные значения.**

  Тип `unsigned char` **может содержать значения от** `0` **до** `255`**. Пример:**

```
unsigned char min = 0, max = 255;
printf("%d\n", min);                   // 0
printf("%d\n", max);                   // 255
printf("%hu\n", min);                  // 0
printf("%hu\n", max);                  // 255
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));     // 1
// #include <limits.h>
printf("%d\n", UCHAR_MAX);             // 255
```

  Диапазон значений для типа `unsigned int` **(можно указать просто** `unsigned`**) зависит от компилятора. Минимальный диапазон от** `0` **до** `65 535`**. На практике диапазон от** `0` **до** `4 294 967 295`**. Пример:**

```
unsigned int min = 0;
unsigned max = 4294967295U;
printf("%u\n", min);                   // 0
printf("%u\n", max);                   // 4294967295
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));     // 4
// #include <limits.h>
printf("%u\n", UINT_MAX);              // 4294967295
```

- `short` — **может быть указан перед целочисленным типом. Занимает 2 байта.**

  Диапазон значений у типов `short int` **(или просто** `short`**) и** `signed short int` **(или просто** `signed short`**) — от** `-32 768` **до** `32 767`**. Пример:**

```
short int min = -32768;
short max = 32767;
printf("%hd\n", min);                  // -32768
printf("%hi\n", max);                  // 32767
```

```
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));     // 2
// #include <limits.h>
printf("%d\n", SHRT_MIN);              // -32768
printf("%d\n", SHRT_MAX);              // 32767
```

Диапазон значений у типа `unsigned short int` **(или просто** `unsigned short`**) — от** `0` **до** `65 535`**. Пример:**

```
unsigned short int min = 0;
unsigned short max = 65535;
printf("%hu\n", min);                  // 0
printf("%hu\n", max);                  // 65535
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));     // 2
// #include <limits.h>
printf("%d\n", USHRT_MAX);             // 65535
```

- `long` **— может быть указан перед целочисленным типом и типом** `double`.

Диапазон значений у типов `long int` **(или просто** `long`**) и** `signed long int` **(или просто** `signed long`**) — от** `-2 147 483 648` **до** `2 147 483 647`**. Занимает 4 байта. Пример:**

```
long int min = -2147483647L - 1;
long max = 2147483647L;
printf("%ld\n", min);                  // -2147483648
printf("%li\n", max);                  // 2147483647
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));     // 4
// #include <limits.h>
printf("%ld\n", LONG_MIN);             // -2147483648
printf("%ld\n", LONG_MAX);             // 2147483647
```

Диапазон значений у типа `unsigned long int` **(или просто** `unsigned long`**) — от** `0` **до** `4 294 967 295`**. Пример:**

```
unsigned long int min = 0UL;
unsigned long max = 4294967295UL;
printf("%lu\n", min);                  // 0
printf("%lu\n", max);                  // 4294967295
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));     // 4
// #include <limits.h>
printf("%lu\n", ULONG_MAX);            // 4294967295
```

Ключевое слово `long` **перед целым типом может быть указано дважды. Диапазон значений у типов** `long long int` **(или просто** `long long`**) и** `signed long long int` **(или просто** `signed long long`**) — от** `-9 223 372 036 854 775 808` **до** `9 223 372 036 854 775 807`**. Пример:**

```
long long int min = -9223372036854775807LL - 1LL;
long long max = 9223372036854775807LL;
printf("%I64d\n", min);              // -9223372036854775808
printf("%I64i\n", max);              // 9223372036854775807
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));   // 8
// #include <limits.h>
printf("%I64d\n", LLONG_MIN);        // -9223372036854775808
printf("%I64d\n", LLONG_MAX);        // 9223372036854775807
printf("%I64d\n", LONG_LONG_MIN);    // -9223372036854775808
printf("%I64d\n", LONG_LONG_MAX);    // 9223372036854775807
```

Диапазон значений у типа `unsigned long long int` **может быть от** `0` **до** `18 446 744 073 709 551 615`. **Пример:**

```
unsigned long long int min = 0ULL;
unsigned long long max = 18446744073709551615ULL;
printf("%I64u\n", min);              // 0
printf("%I64u\n", max);              // 18446744073709551615
// Выводим размер в байтах
printf("%d\n", (int) sizeof(min));   // 8
// #include <limits.h>
printf("%I64u\n", ULLONG_MAX);       // 18446744073709551615
printf("%I64u\n", ULONG_LONG_MAX);   // 18446744073709551615
```

Ключевое слово `long` **можно указать также перед типом** `double`. **Пример:**

```
long double x = 8e+245L;
__mingw_printf("%Le\n", x);          // 8.000000e+245
printf("%d\n", (int) sizeof(x));
// Значение в проекте Test32c: 12
// Значение в проекте Test64c: 16
```

При использовании модификаторов тип `int` **подразумевается по умолчанию, поэтому тип** `int` **можно не указывать. Пример объявления переменных:**

```
short x;    // Эквивалентно: short int x;
long y;     // Эквивалентно: long int y;
signed z;   // Эквивалентно: signed int z;
unsigned k; // Эквивалентно: unsigned int k;
```

Если переменная может изменять свое значение извне, то перед модификатором указывается ключевое слово `volatile`. **Это ключевое слово предотвращает проведение оптимизации программы, при котором предполагается, что значение переменной может быть изменено только в программе.**

3.4. Целочисленные типы фиксированного размера

Вместо указания конкретного целочисленного типа можно воспользоваться макроопределениями `__int8`, `__int16`, `__int32` **и** `__int64`. **Определения макросов:**

```
#define __int8 char
#define __int16 short
#define __int32 int
#define __int64 long long
```

Пример объявления переменных:

```
__int8  x = 10;
__int16 y = 20;
__int32 z = 30;
__int64 k = 40LL;
```

Кроме того, в заголовочном файле stdint.h объявлены знаковые типы фиксированной длины `int8_t`, `int16_t`, `int32_t` **и** `int64_t`, **а также беззнаковые** `uint8_t`, `uint16_t`, `uint32_t` **и** `uint64_t` **(полный список доступных типов см. в заголовочном файле):**

```
#include <stdint.h>
typedef signed char int8_t;
typedef unsigned char   uint8_t;
typedef short  int16_t;
typedef unsigned short  uint16_t;
typedef int  int32_t;
typedef unsigned   uint32_t;
typedef long long  int64_t;
typedef unsigned long long   uint64_t;
```

Заголовочный файл stdint.h содержит также макроопределения с допустимыми диапазонами значений для этих типов:

```
#define INT8_MIN (-128)
#define INT8_MAX 127
#define INT16_MIN (-32768)
#define INT16_MAX 32767
#define INT32_MIN (-2147483647 - 1)
#define INT32_MAX 2147483647
#define INT64_MIN (-9223372036854775807LL - 1)
#define INT64_MAX 9223372036854775807LL

#define UINT8_MAX 255
#define UINT16_MAX 65535
#define UINT32_MAX 0xffffffffU  /* 4294967295U */
#define UINT64_MAX 0xffffffffffffffffULL /* 18446744073709551615ULL */
```

Пример:

```
// #include <stdint.h> или #include <inttypes.h>
int8_t min = INT8_MIN;
int8_t max = INT8_MAX;
uint8_t umax = UINT8_MAX;
printf("%d\n", min);                    // -128
```

```
printf("%d\n", max);                    // 127
printf("%d\n", umax);                   // 255
```

При использовании типов с фиксированной шириной следует учитывать, что в различных компиляторах объявления типов могут быть разными. Гарантируется только диапазон значений, но, например, при объявлении `int32_t` вместо типа `int` может быть использован тип `long`. Спецификаторы у этих типов разные. Для того чтобы правильно указать спецификатор, нужно воспользоваться следующими макроопределениями (полный список макроопределений см. в заголовочном файле inttypes.h):

```
#include <inttypes.h>
#define PRId8 "d"
#define PRId16 "d"
#define PRId32 "d"
#define PRId64 "I64d"

#define PRIu8 "u"
#define PRIu16 "u"
#define PRIu32 "u"
#define PRIu64 "I64u"
```

Пример указания спецификатора при выводе значения типа `int64_t`:

```
// #include <inttypes.h>
int64_t x = 9223372036854775807LL;
printf("%" PRId64, x); // 9223372036854775807
```

3.5. Оператор *sizeof* и тип *size_t*

После объявления переменной под нее выделяется определенная память, размер которой зависит от используемого типа данных и разрядности операционной системы. Для того чтобы сделать код машинно-независимым, следует определять размер типа с помощью оператора `sizeof`. Оператор имеет два формата:

```
<Размер> = sizeof <Переменная>;
<Размер> = sizeof (<Тип данных>);
```

Пример:

```
int x = 10;
printf("%d\n", (int) sizeof x);     // 4
printf("%d\n", (int) sizeof(x));    // 4
printf("%d\n", (int) sizeof(int)); // 4
```

Обратите внимание на то, что тип данных обязательно должен быть указан внутри круглых скобок, в то время как название переменной можно указать как внутри скобок, так без них.

Оператор `sizeof` возвращает значение типа `size_t`. Объявление этого типа в различных компиляторах может быть разным. В проекте `Test64c` объявление выглядит следующим образом:

```
#define __int64 long long
typedef unsigned __int64 size_t;
```

При выводе значения можно воспользоваться спецификатором `%zd`, но нужно использовать функцию `__mingw_printf()` вместо функции `printf()`:

```
size_t size = sizeof(int);
__mingw_printf("%zd\n", size); // 4
```

3.6. Инициализация переменных

При объявлении переменной ей можно сразу присвоить начальное значение, указав его после оператора =. Эта операция называется *инициализацией переменной*. Пример указания значения:

```
int x, y = 10, z = 30, k;
```

Переменная становится видимой сразу после объявления, поэтому на одной строке с объявлением (после запятой) эту переменную уже можно использовать для инициализации других переменных:

```
int x = 5, y = 10, z = x + y; // z равно 15
```

Инициализация *глобальных* (объявленных вне функций) переменных производится только один раз. *Локальные* (объявленные внутри функций) переменные инициализируются при каждом вызове функции, а *статические* (сохраняющие свое значение между вызовами) локальные переменные — один раз при первом вызове функции. Если при объявлении переменной значение не было присвоено, то:

- глобальные переменные автоматически получают значение `0`;
- локальным переменным значение не присваивается. Переменная будет содержать произвольное значение, так называемый "мусор";
- статические локальные переменные автоматически получают значение `0`.

Присвоить значение переменной можно уже после объявления, указав его после оператора =. Эта операция называется *присваиванием*. Пример присваивания:

```
int x;
x = 10;
```

3.7. Оператор *typedef*

Оператор `typedef` позволяет создать псевдоним для существующего типа данных. В дальнейшем псевдоним можно указывать при объявлении переменной. Оператор имеет следующий формат:

```
typedef <Существующий тип> <Псевдоним>;
```

В качестве примера создадим псевдоним для типа `long int`:

```
typedef long int lint;
lint x = 5L, y = 10L;
```

После создания псевдонима его имя можно использовать при создании другого псевдонима:

```
typedef long int lint;
typedef lint newint;
newint x = 5L, y = 10L;
```

Псевдонимы предназначены для создания машинно-независимых программ. При переносе программы на другой компьютер достаточно будет изменить одну строку. Подобный подход часто используется в стандартной библиотеке. Например, прототип функции `strlen()`, позволяющей получить длину строки, выглядит так:

```
size_t strlen(const char *str);
```

В этом прототипе тип данных `size_t`, возвращаемый функцией `strlen()`, является псевдонимом, а не новым типом. Его размер зависит от компилятора. Объявление в MinGW-W64:

```
#define __int64 long long
typedef unsigned __int64 size_t;
```

3.8. Константы

Константы — это участки памяти, значения в которых не должны изменяться во время работы программы. В более широком смысле под константой понимают любое значение, которое нельзя изменить, например `10`, `12.5`, `'W'`, `"string"`.

При объявлении константы перед типом данных указывается ключевое слово `const`:

```
const int MY_CONST = 10;
printf("%d\n", MY_CONST);  // 10
```

Обратите внимание на то, что в названии константы принято использовать буквы только в верхнем регистре. Если название константы состоит из нескольких слов, то между словами указывается символ подчеркивания. Это позволяет отличить внутри программы константу от обычной переменной. После объявления константы ее можно использовать в выражениях:

```
const int MY_CONST = 10;
int y;
y = MY_CONST + 20;
```

Присвоить значение константе можно только при объявлении. Любая попытка изменения значения в программе приведет к ошибке при компиляции:

```
error: assignment of read-only variable 'MY_CONST'
```

Создать константу можно также с помощью директивы `#define`. Значение, указанное в этой директиве, подставляется в выражение до компиляции. Название, указанное в директиве `#define`, принято называть *макроопределением* или *макросом*. Директива имеет следующий формат:

```
#define <Название макроса> <Значение>
```

Пример использования директивы `#define` приведен в листинге 3.1.

Листинг 3.1. Использование директивы #define

```
#include <stdio.h>

#define MY_CONST (-5)

int main(void) {
   int y;
   y = MY_CONST + 20;
   printf("%d\n", MY_CONST);  // -5
   printf("%d\n", y);         // 15
   return 0;
}
```

Обратите внимание на то, что оператор = не используется, и в конце инструкции точка с запятой не указывается. Если точку с запятой указать, то значение вместе с ней будет вставлено в выражение. Например, если определить макрос так:

```
#define MY_CONST 5;
```

то после подстановки значения инструкция

```
y = MY_CONST + 20;
```

будет выглядеть следующим образом:

```
y = 5; + 20;
```

Точка с запятой после цифры `5` является концом инструкции, поэтому переменной `y` будет присвоено значение `5`, а не `25`. Подобная ситуация приводит к ошибкам, которые трудно найти, т. к. в этом случае инструкция `+ 20;` не возбуждает ошибку при компиляции.

В качестве значения макроса можно указать целое выражение, например:

```
#define MY_CONST 5 + 5
```

Это значение также может привести к недоразумениям. Никакого вычисления выражения не производится. Все выражение целиком подставляется вместо названия макроса. Если инструкция выглядит так:

```
y = MY_CONST * 20;
```

то после подстановки значения инструкция примет следующий вид:

```
y = 5 + 5 * 20;
```

Приоритет оператора умножения выше приоритета оператора сложения, поэтому число `5` будет умножено на `20`, а затем к результату прибавлено число `5`. Таким образом, результат будет `105`, а не `200`, как это было бы при использовании константы:

```
const int MY_CONST = 5 + 5;
int y;
y = MY_CONST * 20; // 200
```

Для того чтобы избежать недоразумений, выражение (а также отрицательное значение) в директиве `#define` следует заключать в круглые скобки:

```
#define MY_CONST (5 + 5)
```

При подстановке инструкция будет выглядеть так:

```
y = (5 + 5) * 20;
```

Теперь мы получим число `200`, а не `105`, т. к. скобки меняют приоритет операторов.

В качестве значения макроса можно указать строку в кавычках:

```
#define ERR "Сообщение об ошибке"
```

При указании длинной строки следует учитывать, что определение макроса должно быть расположено на одной строке. Если нужно разместить значение на нескольких строках, то в конце строки необходимо добавить обратную косую черту. После косой черты не должно быть никаких символов, в том числе комментариев:

```
#define ERR "Сообщение об ошибке \
на нескольких \
строках"
```

Удалить макрос позволяет директива `#undef <Имя макроса>`:

```
#undef ERR
```

В языке C существуют встроенные макросы (до и после названия два символа подчеркивания):

- `__FILE__` — имя файла;
- `__LINE__` — номер текущей строки;
- `__DATE__` — дата компиляции файла;
- `__TIME__` — время компиляции файла.

Помимо этих макросов существует множество других, определенных в различных заголовочных файлах. Эти макросы мы рассмотрим в соответствующих разделах книги. Выведем текущие значения встроенных макросов (листинг 3.2).

Листинг 3.2. Встроенные макросы

```
#include <stdio.h>

int main(void) {
   printf("%d\n", __LINE__);
   printf("%s\n", __FILE__);
   printf("%s\n", __DATE__);
   printf("%s\n", __TIME__);
   return 0;
}
```

Примерный результат выполнения:

```
4
..\src\Test64c.c
Jan 20 2019
14:35:53
```

3.9. Спецификаторы хранения

Перед модификатором и типом могут быть указаны следующие спецификаторы.

- `auto` — локальная переменная создается при входе в блок и удаляется при выходе из блока. Так как локальные переменные по умолчанию являются *автоматическими*, ключевое слово `auto` в языке C практически не используется. Пример объявления автоматической локальной переменной:

  ```
  auto int x = 10;
  ```

- `register` — является подсказкой компилятору, что переменная будет использоваться интенсивно. Для ускорения доступа значение такой переменной сохраняется в регистрах процессора. Компилятор может проигнорировать это объявление и сохранить значение в памяти. Ключевое слово может использоваться при объявлении переменной внутри блока или в параметрах функции. К глобальным переменным не применяется. Пример объявления переменной:

  ```
  register int x = 20;
  ```

- `extern` — сообщает компилятору, что переменная определена в другом месте, например в другом файле. Ключевое слово лишь объявляет переменную, а не определяет ее. Таким образом, память под переменную повторно не выделяется. Если при объявлении производится инициализация переменной, то объявление становится определением переменной. В качестве примера объявим переменную внутри функции `main()`, а определение переменной разместим после функции:

  ```
  #include <stdio.h>

  int main(void) {
     extern int x;       // Определение в другом месте
     printf("%d\n", x); // 10
     return 0;
  }
  int x = 10;            // Определение переменной x
  ```

- `static` — если ключевое слово указано перед локальной переменной, то значение будет сохраняться между вызовами функции. Инициализация статических локальных переменных производится только при первом вызове функции. При последующих вызовах используется сохраненное ранее значение.

  Создадим функцию со статической переменной. Внутри функции увеличим значение переменной на единицу, а затем выведем значение в окно консоли. Далее вызовем эту функцию несколько раз:

```
#include <stdio.h>

void func(void);

int main(void) {
   func(); // 1
   func(); // 2
   func(); // 3
   return 0;
}
void func(void) {
   static int x = 0;      // Статическая переменная
   ++x;                   // Увеличиваем значение на 1
   printf("%d\n", x);
}
```

При каждом вызове функции `func()` значение статической переменной `x` будет увеличиваться на единицу. Если убрать ключевое слово `static`, то при каждом вызове будет выводиться число `1`, т. к. автоматические локальные переменные инициализируются при входе в функцию и уничтожаются при выходе из нее.

Если ключевое слово `static` указано перед глобальной переменной, то ее значение будет видимо только в пределах файла.

3.10. Области видимости переменных

Прежде чем использовать переменную, ее необходимо предварительно объявить. До объявления переменной она не видна в программе. Объявить переменную можно глобально (вне функций) или локально (внутри функции или блока).

- *Глобальные переменные* — это переменные, объявленные в программе вне функций. Глобальные переменные видны в любой части программы, включая функции. Инициализация таких переменных производится только один раз. Если при объявлении переменной не было присвоено начальное значение, то производится автоматическая инициализация нулевым значением.
- *Локальные переменные* — это переменные, которые объявлены внутри функции или блока (области, ограниченной фигурными скобками). Локальные переменные видны только внутри функции или блока. Инициализация таких переменных производится при каждом вызове функции или входе в блок. После выхода из функции или блока локальная переменная уничтожается. Если при объявлении переменной не было присвоено начальное значение, то переменная будет содержать произвольное значение, так называемый "мусор". Исключением являются *статические локальные переменные*, которым автоматически присваивается нулевое значение и которые сохраняют значение при выходе из функции.

Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной не изменяется.

Область видимости глобальных и локальных переменных показана в листинге 3.3.

Листинг 3.3. Область видимости переменных

```
#include <stdio.h>

int x = 10; // Глобальная переменная
void func(void);

int main(void) {
   func();
   // Переменная x из функции func() здесь не видна
   // Вывод значения глобальной переменной x
   printf("%d\n", x);     // 10
   { // Блок
      int z = 30;
      printf("%d\n", z); // 30
   }
   // Переменная z здесь уже не видна!!!
   for (int i = 0; i < 10; ++i) {
      printf("%d\n", i);
   }
   // Переменная i здесь уже не видна!!!
   return 0;
}

void func(void) {
   // Переменная z здесь не видна
   int x = 5; // Локальная переменная
   // Вывод значения локальной переменной x,
   // а не одноименной глобальной
   printf("%d\n", x); // 5
}
```

Очень важно учитывать, что переменная, объявленная внутри блока, видна только в пределах блока (внутри фигурных скобок). Например, присвоим значение переменной в зависимости от некоторого условия:

```
// Неправильно!!!
if (x == 10) { // Какое-то условие
   int y;
   y = 5;
}
else {
   int y;
   y = 25;
}
// Переменная y здесь не определена!!!
```

В этом примере переменная y видна только внутри блока. После условного оператора if переменной не существует. Для того чтобы переменная была видна внутри блока и после выхода из него, необходимо поместить объявление переменной перед блоком:

```
// Правильно
int y;
if (x == 10) { // Какое-то условие
   y = 5;
}
else {
   y = 25;
}
printf("%d\n", y);
```

Если обращение к переменной внутри функции или блока производится до объявления одноименной локальной переменной, то до объявления будет использоваться глобальная переменная, а после объявления — локальная переменная (листинг 3.4). Если глобальной переменной с таким названием не существует, то будет ошибка при компиляции.

Листинг 3.4. Обращение к переменной до объявления внутри функции

```
#include <stdio.h>

int x = 10; // Глобальная переменная

int main(void) {
   // Выводится значение глобальной переменной
   printf("%d\n", x); // 10
   int x = 5;         // Локальная переменная
   // Выводится значение локальной переменной
   printf("%d\n", x); // 5
   return 0;
}
```

3.11. Массивы

Массив — это нумерованный набор переменных одного типа. Переменная в массиве называется *элементом*, а ее позиция в массиве задается *индексом*. Объем памяти (в байтах), занимаемый массивом, определяется так:

```
<Объем памяти> = sizeof (<Тип>) * <Количество элементов>
```

Объявление массива выглядит следующим образом:

```
<Тип> <Переменная>[<Количество элементов>];
```

Пример объявления массива из трех элементов, имеющих тип `long int`:

```
long arr[3];
```

При объявлении элементам массива можно присвоить начальные значения. Для этого после объявления указывается оператор =, а далее значения через запятую внутри фигурных скобок. После закрывающей фигурной скобки обязательно указывается точка с запятой. Пример инициализации массива:

```
long arr[3] = {10, 20, 30};
```

Количество значений внутри фигурных скобок может быть меньше количества элементов массива. В этом случае значения присваиваются соответствующим элементам с начала массива. Пример:

```
long arr[3] = {10, 20};
for (int i = 0; i < 3; ++i) {
   printf("%ld ", arr[i]);
} // 10 20 0
```

В этом примере первому элементу массива присваивается значение 10, второму — значение 20, а третьему элементу будет присвоено значение 0.

Если при объявлении массива указываются начальные значения, то количество элементов внутри квадратных скобок можно не указывать. Размер будет соответствовать количеству значений внутри фигурных скобок. Пример:

```
long arr[] = {10, 20, 30};
for (int i = 0; i < 3; ++i) {
   printf("%ld ", arr[i]);
} // 10 20 30
```

Если при объявлении массива начальные значения не указаны, то:

- элементам глобальных массивов автоматически присваивается значение 0;
- элементы локальных массивов будут содержать произвольные значения, так называемый "мусор".

Обращение к элементам массива осуществляется с помощью квадратных скобок, в которых указывается индекс элемента. Обратите внимание на то, что нумерация элементов массива начинается с 0, а не с 1, поэтому первый элемент имеет индекс 0. С помощью индексов можно присвоить начальные значения элементам массива уже после объявления:

```
long arr[3];
arr[0] = 10; // Первый элемент имеет индекс 0!!!
arr[1] = 20; // Второй элемент
arr[2] = 30; // Третий элемент
```

Следует учитывать, что проверка выхода указанного индекса за пределы диапазона на этапе компиляции не производится. Таким образом, можно перезаписать значение в смежной ячейке памяти и тем самым нарушить работоспособность программы или даже повредить операционную систему. Помните, что контроль корректности индекса входит в обязанности программиста.

Все элементы массива располагаются в смежных ячейках памяти. Например, если объявлен массив из трех элементов, имеющих тип int (занимает 4 байта), то адреса соседних элементов будут отличаться на 4 байта:

```
int arr[3] = {10, 20, 30};
for (int i = 0; i < 3; ++i) {
   printf("%p\n", &arr[i]);
}
```

Результат в проекте `Test64c`:

```
000000000023FE40
000000000023FE44
000000000023FE48
```

После определения массива выделяется необходимый размер памяти, а в переменной сохраняется адрес первого элемента массива. При указании индекса внутри квадратных скобок производится вычисление адреса соответствующего элемента массива. Зная адрес элемента массива, можно получить значение или перезаписать его. Иными словами, с элементами массива можно производить такие же операции, как и с обычными переменными. Пример:

```
long arr[3] = {10, 20, 30};
long x = 0;
x = arr[1] + 12;
arr[2] = x - arr[2];
printf("%ld ", x);      // 32
printf("%ld ", arr[2]); // 2
```

Массивы в языке C могут быть *многомерными*. Объявление многомерного массива имеет следующий формат:

```
<Тип> <Переменная>[<Количество элементов1>]...[<Количество элементовN>];
```

На практике наиболее часто используются *двумерные массивы*, позволяющие хранить значения ячеек таблицы, содержащей определенное количество строк и столбцов. Объявление двумерного массива выглядит так:

```
<Тип> <Переменная>[<Количество строк>][<Количество столбцов>];
```

Пример объявления двумерного массива, содержащего две строки и четыре столбца:

```
int arr[2][4]; // Две строки из 4-х элементов каждая
```

Все элементы двумерного массива располагаются в памяти друг за другом. Вначале элементы первой строки, затем второй и т. д. При инициализации двумерного массива элементы указываются внутри фигурных скобок через запятую. Элементы каждой строки также размещаются внутри фигурных скобок. Пример:

```
int arr[2][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
for (int i = 0, j = 0; i < 2; ++i) {
   for (j = 0; j < 4; ++j) {
      printf("%3d ", arr[i][j]);
   }
   printf("\n");
}
```

Для того чтобы сделать процесс инициализации наглядным, мы расположили элементы на отдельных строках. Количество элементов на строке совпадает с количеством столбцов в массиве. Результат выполнения:

```
1   2   3   4
5   6   7   8
```

Если при объявлении производится инициализация, то количество строк можно не указывать, оно будет определено автоматически:

```
int arr[][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
```

Получить или задать значение элемента можно, указав два индекса (не забывайте, что нумерация начинается с нуля):

```
int arr[2][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
printf("%d\n", arr[0][0]); // 1
printf("%d\n", arr[1][0]); // 5
printf("%d\n", arr[1][3]); // 8
```

3.12. Строки

Строка является массивом символов, последний элемент которого содержит нулевой символ (`'\0'`). Обратите внимание на то, что нулевой символ (нулевой байт) не имеет никакого отношения к символу `'0'`. Коды этих символов разные. Такие строки часто называют *С-строками*.

Объявляется С-строка так же, как и массив элементов типа `char`:

```
char str[7];
```

При инициализации можно перечислить символы внутри фигурных скобок:

```
char str[7] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
```

или указать строку внутри двойных кавычек:

```
char str[7] = "String";
```

При использовании двойных кавычек следует учитывать, что длина строки на один символ больше, т. к. в конец будет автоматически вставлен нулевой символ. Если это не предусмотреть и объявить массив из шести элементов вместо семи, то будет ошибка при работе со строкой в дальнейшем.

Если размер массива при объявлении не указать, то он будет определен автоматически в соответствии с длиной строки:

```
char str[] = "String";
```

Обратите внимание на то, что присваивать строку в двойных кавычках можно только при инициализации. Попытка присвоить строку позже приведет к ошибке:

```
char str[7];
str = "String"; // Ошибка!!!
```

Внутри строки в двойных кавычках можно указывать специальные символы (например, \n, \r и др.). Если внутри строки встречается кавычка, то ее необходимо экранировать с помощью обратного слэша:

```
char str[] = "Группа \"Кино\"\n";
```

Объявить массив строк можно следующим образом:

```
char str[][20] = {"String1", "String2", "String3"};
printf("%s\n", str[0]); // String1
printf("%s\n", str[1]); // String2
printf("%s\n", str[2]); // String3
```

3.13. Указатели

Указатель — это переменная, которая предназначена для хранения адреса. В языке C указатели часто используются в следующих случаях:

- для управления динамической памятью;
- чтобы иметь возможность изменить значение переменной внутри функции;
- для эффективной работы с массивами и др.

Объявление указателя имеет следующий формат:

```
<Тип> *<Переменная>;
```

Пример объявления указателя на тип `int`:

```
int *p = NULL;
printf("%p\n", p);                    // Вывод адреса
// Значение в проекте Test32c: 00000000
// Значение в проекте Test64c: 0000000000000000
printf("%d\n", (int) sizeof(p));      // Вывод размера
// Значение в проекте Test32c: 4
// Значение в проекте Test64c: 8
```

Очень часто символ * указывается после типа, а не перед именем переменной:

```
int* p;
```

С точки зрения компилятора эти два объявления ничем не различаются. Однако следует учитывать, что при объявлении нескольких переменных в одной инструкции символ * относится к переменной, перед которой он указан, а не к типу данных. Например, следующая инструкция объявляет указатель и переменную, а не два указателя:

```
int* p, x; // Переменная x указателем не является!!!
```

Поэтому более логично указывать символ * перед именем переменной:

```
int *p, x;
```

Для того чтобы указателю присвоить адрес переменной, необходимо при присваивании значения перед названием переменной добавить оператор &. Типы данных переменной и указателя должны совпадать. Это нужно, чтобы при адресной арифметике был известен размер данных. Пример присвоения адреса:

```
int *p = NULL, x = 10;
p = &x;
printf("%p\n", p);  // Значение в проекте Test64c: 000000000023FE44
printf("%p\n", &x); // Значение в проекте Test64c: 000000000023FE44
```

Для того чтобы получить или изменить значение, расположенное по адресу, на который ссылается указатель, необходимо выполнить операцию *разыменования указателя*. Для этого перед названием переменной указывается оператор *. Пример:

```
int *p = NULL, x = 10;
p = &x;
printf("%d\n", *p); // 10
*p = *p + 20;
printf("%d\n", *p); // 30
```

Основные операции с указателями приведены в листинге 3.5.

Листинг 3.5. Указатели

```
#include <stdio.h>

int x = 10;
int *p = NULL; // Нулевой указатель

int main(void) {
   // Присваивание указателю адреса переменной x
   p = &x;
   // Вывод адреса
   printf("%p\n", p);  // Например: 0000000000403010
   // Вывод значения
   printf("%d\n", *p); // 10
   return 0;
}
```

В этом примере при объявлении указателя ему присваивается значение NULL. Указатель, которому присвоено значение NULL или 0, называется *нулевым указателем*. Определение макроса NULL выглядит так:

```
#include <stdio.h>
#define NULL ((void *)0)
```

В данном случае можно было и не присваивать указателю значение NULL, т. к. глобальные и статические локальные указатели автоматически получают значение 0. Однако указатели, которые объявлены в локальной области видимости, будут иметь произвольное значение. Если попытаться записать какое-либо значение через такой указатель, то можно повредить операционную систему. Поэтому, согласно соглашению, указатели, которые ни на что не указывают, должны иметь значение NULL.

Значение одного указателя можно присвоить другому указателю. При этом важно учитывать, что типы указателей должны совпадать. Пример:

```
int *p1 = NULL, *p2 = NULL, x = 10;
p1 = &x;
p2 = p1;              // Копирование адреса переменной x
printf("%d\n", *p2); // 10
*p2 = 40;             // Изменение значения в переменной x
printf("%d\n", *p1); // 40
printf("%d\n", x);   // 40
```

В этом примере мы просто скопировали адрес переменной x из одного указателя в другой. Помимо копирования адреса можно создать *указатель на указатель*. Для этого при объявлении перед названием переменной указываются два оператора *:

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

Для того чтобы получить адрес указателя, используют оператор &, а для получения значения переменной, на которую ссылается указатель, применяют два оператора *. Пример:

```
int *p1 = NULL, **p2 = NULL, x = 10;
p1 = &x;
p2 = &p1;             // Указатель на указатель
printf("%d\n", **p2); // 10
**p2 = 40;            // Изменение значения в переменной x
printf("%d\n", *p1);  // 40
printf("%d\n", x);    // 40
```

При каждом вложении добавляется дополнительная звездочка:

```
int *p1 = NULL, **p2 = NULL, ***p3 = NULL, x = 10;
p1 = &x;
p2 = &p1;
p3 = &p2;
printf("%d\n", ***p3); // 10
***p3 = 40;            // Изменение значения в переменной x
printf("%d\n", **p2);  // 40
printf("%d\n", *p1);   // 40
printf("%d\n", x);     // 40
```

Подобный синтаксис трудно понять и очень просто сделать ошибку, поэтому обычно ограничиваются использованием указателя на указатель.

При инициализации указателя ему можно присвоить не только числовое значение, но и строку. Пример:

```
const char *str = "String";
printf("%s", str);      // String
```

Указатели можно сохранять в массиве. При объявлении *массива указателей* используется следующий синтаксис:

```
<Тип> *<Название массива>[<Количество элементов>];
```

Пример использования массива указателей:

```
int *p[3]; // Массив указателей из трех элементов
int x = 10, y = 20, z = 30;
p[0] = &x;
p[1] = &y;
p[2] = &z;
printf("%d\n", *p[0]); // 10
printf("%d\n", *p[1]); // 20
printf("%d\n", *p[2]); // 30
```

Объявление массива указателей на строки и вывод значений выглядит так:

```
const char *str[] = {"String1", "String2", "String3"};
printf("%s\n", str[0]); // String1
printf("%s\n", str[1]); // String2
printf("%s\n", str[2]); // String3
```

Указатели очень часто используются для обращения к элементам массива, т. к. адресная арифметика выполняется эффективнее, чем доступ по индексу. В качестве примера создадим массив из трех элементов, а затем выведем значения (листинг 3.6).

Листинг 3.6. Перебор элементов массива

```
#include <stdio.h>

#define ARR_SIZE 3

int main(void) {
   int *p = NULL, arr[ARR_SIZE] = {10, 20, 30};
   // Устанавливаем указатель на первый элемент массива
   p = arr; // Оператор & не указывается!!!
   for (int i = 0; i < ARR_SIZE; ++i) {
      printf("%d\n", *p);
      ++p; // Перемещаем указатель на следующий элемент
   }
   p = arr; // Восстанавливаем положение указателя
   // Выполняем какие-либо инструкции
   return 0;
}
```

Вначале создается константа `ARR_SIZE`, в которой сохраняется количество элементов в массиве. Если массив используется часто, то лучше сохранить его размер как константу, т. к. количество элементов нужно будет указывать при каждом переборе массива. Если в каждом цикле указывать конкретное число, то при изменении размера массива придется вручную вносить изменения во всех циклах. При объявлении константы достаточно будет изменить ее значение один раз при определении.

В первой инструкции внутри функции `main()` объявляется указатель на тип `int` и массив. Количество элементов массива задается константой `ARR_SIZE`. При объявлении массив инициализируется начальными значениями.

После объявления переменных указателю присваивается адрес первого элемента массива. Обратите внимание на то, что перед названием массива отсутствует оператор `&`, т. к. название переменной содержит адрес первого элемента. Если использовать оператор `&`, то необходимо дополнительно указать индекс внутри квадратных скобок:

```
p = &arr[0]; // Эквивалентно: p = arr;
```

Для перебора элементов массива используется цикл `for`. В первом параметре цикла задается начальное значение (`int i = 0`), во втором — условие (`i < ARR_SIZE`), а в третьем — приращение на единицу (`++i`) на каждой итерации цикла. Инструкции внутри цикла будут выполняться, пока условие является истинным (значение переменной `i` меньше количества элементов массива).

Внутри цикла выводится значение элемента, на который ссылается указатель, а затем значение указателя увеличивается на единицу (`++p`). Обратите внимание на то, что изменяется адрес, а не значение элемента массива. При увеличении значения указателя используются правила адресной арифметики, а не правила обычной арифметики. Увеличение значения указателя на единицу означает, что значение будет увеличено на размер типа. Например, если тип `int` занимает 4 байта, то при увеличении значения на единицу указатель вместо адреса `0x0012FF30` будет содержать адрес `0x0012FF34`. Значение увеличилось на `4`, а не на `1`. В нашем примере вместо двух инструкций внутри цикла можно использовать одну:

```
printf("%d\n", *p++);
```

Выражение `p++` возвращает текущий адрес, а затем увеличивает его на единицу. Символ `*` позволяет получить доступ к значению элемента по указанному адресу. Последовательность выполнения соответствует следующей расстановке скобок:

```
printf("%d\n", *(p++));
```

Если скобки расставить так:

```
printf("%d\n", (*p)++);
```

то вначале будет получен доступ к элементу массива и выведено его текущее значение, а затем будет произведено увеличение значения элемента массива. Перемещение указателя на следующий элемент не выполняется.

Получить доступ к элементу массива можно несколькими способами. Первый способ заключается в указании индекса внутри квадратных скобок. Во втором способе используется адресная арифметика совместно с разыменованием указателя. В третьем способе внутри квадратных скобок указывается название массива, а перед квадратными скобками — индекс элемента. Этот способ может показаться странным. Однако если учесть, что выражение `1[arr]` воспринимается компилятором как `*(1 + arr)`, то все встанет на свои места. Таким образом, все эти инструкции являются эквивалентными:

```
int arr[3] = {10, 20, 30};
printf("%d\n", arr[1]);     // 20
printf("%d\n", *(arr + 1)); // 20
printf("%d\n", *(1 + arr)); // 20
printf("%d\n", 1[arr]);     // 20
```

С указателем можно выполнять следующие арифметические и логические операции:

- прибавлять целое число. Число умножается на размер базового типа указателя, а затем результат прибавляется к адресу;
- вычитать целое число. Вывести значения элементов массива в обратном порядке можно так:

```
#define ARR_SIZE 3
int *p = NULL, arr[ARR_SIZE] = {10, 20, 30};
// Устанавливаем указатель на последний элемент
p = &arr[ARR_SIZE - 1];
for (int i = ARR_SIZE - 1; i >= 0; --i) {
   printf("%d\n", *p--);
}
```

- вычитать один указатель из другого. Это позволяет получить количество элементов базового типа между двумя указателями;
- сравнивать указатели между собой.

При использовании ключевого слова `const` применительно к указателям важно учитывать местоположение ключевого слова `const`. Например, следующие объявления не эквивалентны:

```
const char *p = str;
char const *p = str;
char * const p = str;
const char * const p = str;
```

Первые два объявления являются эквивалентными. В этом случае изменить значение, на которое ссылается указатель, нельзя, но указателю можно присвоить другой адрес:

```
char str1[] = "String", str2[] = "New";
const char *p = str1;
p = str2;                           // Нормально
p[0] = 's';                         // Ошибка
```

При третьем объявлении изменить значение, на которое ссылается указатель, можно, но указателю нельзя присвоить другой адрес:

```
char str1[] = "String", str2[] = "New";
char * const p = str1;
p = str2;                              // Ошибка
p[0] = 's';                            // Нормально
```

Четвертое объявление запрещает изменение значения, на которое ссылается указатель, и присвоение другого адреса:

```
char str1[] = "String", str2[] = "New";
const char * const p = str1;
p = str2;                              // Ошибка
p[0] = 's';                            // Ошибка
```

Указатели часто используются при передаче параметров в функцию. По умолчанию в функцию передается *копия значения переменной*. Если мы в этом случае изменим значение внутри функции, то это действие не затронет значения внешней переменной. Для того чтобы иметь возможность изменять значение внешней переменной, параметр функции объявляют как указатель, а при вызове передают адрес переменной (листинг 3.7).

Листинг 3.7. Передача параметров в функцию

```
#include <stdio.h>

void func1(int x);
void func2(int *x);

int main(void) {
   int y = 10;
   func1(y);                   // Передаем копию значения
   printf("%d\n", y);          // 10 (значение не изменилось!!!)
   func2(&y);                  // Передаем адрес, а не значение
   printf("%d\n", y);          // 20 (значение изменилось!!!)
   return 0;
}
void func1(int x) {
   x = x * 2;                  // Значение нигде не сохраняется
}
void func2(int *x) {
   *x = *x * 2;
}
```

При использовании функции `func1()` передача параметра осуществляется *по значению* (применяется по умолчанию). При этом создается копия значения, и все операции производятся с этой копией. Так как локальные переменные видны только

внутри тела функции, после завершения выполнения функции копия удаляется. Любые изменения значения копии не затронут значения оригинала.

При использовании функции `func2()` мы передаем адрес переменной:

```
func2(&y);
```

Внутри функции адрес присваивается указателю. Используя операцию разыменования указателя, можно изменить значение самой переменной, а не значение копии. Достигается это с помощью следующей инструкции:

```
*x = *x * 2;
```

3.14. Динамическое выделение памяти

Как вы уже знаете, при объявлении переменной необходимо указать тип данных, а для массива дополнительно задать точное количество элементов. На основе этой информации при запуске программы автоматически выделяется необходимый объем памяти. После завершения программы память автоматически освобождается. Иными словами, объем памяти необходимо знать до выполнения программы. Во время выполнения программы создать новую переменную или увеличить размер существующего массива нельзя.

Для того чтобы произвести увеличение массива во время выполнения программы, необходимо выделить достаточный объем *динамической памяти*, перенести существующие элементы, а лишь затем добавить новые элементы. Управление динамической памятью полностью лежит на плечах программиста, поэтому после завершения работы с памятью необходимо самим возвратить память операционной системе. Если этого не сделать, то участок памяти станет недоступным для дальнейшего использования. Подобные ситуации приводят к утечке памяти.

3.14.1. Функции *malloc()* и *free()*

Для выделения динамической памяти в языке C предназначена функция `malloc()`. Прототип функции:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Функция `malloc()` принимает в качестве параметра размер памяти в байтах и возвращает указатель, имеющий тип `void *`. В языке C указатель типа `void *` неявно приводится к другому типу, поэтому использовать явное приведение не нужно (в языке C++ нужно обязательно выполнять явное приведение). Если память выделить не удалось, то функция возвращает нулевой указатель. Все элементы будут иметь произвольное значение, так называемый "мусор". Для того чтобы программа была машинно-независимой, следует применять оператор `sizeof` для вычисления размера памяти, требуемого для определенного типа. Пример выделения памяти для десяти элементов типа `int`:

```
const unsigned int ARR_SIZE = 10;
int *p = malloc(ARR_SIZE * sizeof(int));
```

Освободить ранее выделенную динамическую память позволяет функция `free()`. Прототип функции:

```
#include <stdlib.h>
void free(void *memory);
```

Функция `free()` принимает в качестве параметра указатель на ранее выделенную память и освобождает ее. Пример использования функций `malloc()` и `free()` приведен в листинге 3.8.

Листинг 3.8. Функции `malloc()` и `free()`

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   const unsigned int ARR_SIZE = 10;
   int *p = malloc(ARR_SIZE * sizeof(int));
   if (!p) {                              // Проверяем на корректность
      puts("Не удалось выделить память");
      exit(1);                            // Выходим при ошибке
   }
   // Нумеруем элементы массива от 1 до 10
   for (int i = 0; i < ARR_SIZE; ++i) { // Пользуемся памятью
      p[i] = i + 1;
   }
   // Выводим значения
   for (int i = 0; i < ARR_SIZE; ++i) { // Пользуемся памятью
      printf("%d\n", p[i]);
   }
   free(p);                               // Освобождаем память
   p = NULL;                              // Обнуляем указатель
   return 0;
}
```

3.14.2. Функция *calloc()*

Вместо функции `malloc()` можно воспользоваться функцией `calloc()`. Прототип функции:

```
#include <stdlib.h>
void *calloc(size_t count, size_t elem_size);
```

В первом параметре функция `calloc()` принимает количество элементов, а во втором — размер одного элемента. Если память выделить не удалось, то функция возвращает нулевой указатель. Все элементы будут иметь значение `0`.

Используя функцию `calloc()`, следующую инструкцию из листинга 3.8:

```
int *p = malloc(ARR_SIZE * sizeof(int));
```

мы можем записать так:

```
int *p = calloc(ARR_SIZE, sizeof(int));
```

В качестве примера использования функции `calloc()` создадим двумерный массив (листинг 3.9). Для этого нам нужно создать массив указателей и в каждом элементе массива сохранить адрес строки. Память для каждой строки нужно выделить дополнительно.

Листинг 3.9. Динамическое выделение памяти под двумерный массив

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

int main(void) {
   const unsigned int ROWS = 2;     // Количество строк
   const unsigned int COLUMNS = 4; // Количество столбцов
   int i = 0, j = 0;
   // Создаем массив указателей
   int **p = calloc(ROWS, sizeof(int*));
   if (!p) exit(1);                 // Выходим при ошибке
   // Добавляем строки
   for (i = 0; i < ROWS; ++i) {
      p[i] = calloc(COLUMNS, sizeof(int));
      if (!p[i]) exit(1);           // Выходим при ошибке
   }
   // Нумеруем элементы массива
   int n = 1;
   for (i = 0; i < ROWS; ++i) {
      for (j = 0; j < COLUMNS; ++j) {
         p[i][j] = n++;
         // *(*(p + i) + j) = n++;
      }
   }
   // Выводим значения
   for (i = 0; i < ROWS; ++i) {
      for (j = 0; j < COLUMNS; ++j) {
         printf("%3d", p[i][j]);
         // printf("%3d", *(*(p + i) + j));
      }
      printf("\n");
   }
   // Освобождаем память
   for (int i = 0; i < ROWS; ++i) {
      free(p[i]);
   }
```

```
   free(p);
   p = NULL;                              // Обнуляем указатель
   return 0;
}
```

Обратите внимание: при возвращении памяти вначале освобождается память, выделенная ранее под строки, а лишь затем освобождается память, выделенная ранее под массив указателей.

Так как мы сохраняем в массиве указателей лишь адрес строки, а не саму строку, количество элементов в строке может быть произвольным. Это обстоятельство позволяет создавать так называемые *"зубчатые" двумерные массивы*.

Строки в памяти могут быть расположены в разных местах, что не позволяет эффективно получать доступ к элементам двумерного массива. Для того чтобы доступ к элементам сделать максимально быстрым, можно представить двумерный массив в виде одномерного массива (листинг 3.10).

Листинг 3.10. Представление двумерного массива в виде одномерного

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

int main(void) {
   const unsigned int ROWS = 2;     // Количество строк
   const unsigned int COLUMNS = 4; // Количество столбцов
   unsigned int i = 0, j = 0;
   int *p = calloc(ROWS * COLUMNS, sizeof(int));
   if (!p) exit(1);                 // Выходим при ошибке
   // Нумеруем элементы массива
   int n = 1;
   for (i = 0; i < ROWS; ++i) {
      for (j = 0; j < COLUMNS; ++j) {
         *(p + i * COLUMNS + j) = n++;
      }
   }
   // Выводим значения
   for (i = 0; i < ROWS; ++i) {
      for (j = 0; j < COLUMNS; ++j) {
         printf("%3d", *(p + i * COLUMNS + j));
      }
      printf("\n");
   }
   free(p);                                // Освобождаем память
   p = NULL;                               // Обнуляем указатель
   return 0;
}
```

Так как в этом случае все элементы двумерного массива расположены в смежных ячейках, мы можем получить доступ к элементам с помощью указателя и адресной арифметики. Например, пронумеруем все элементы:

```
int *p2 = p; // Сохраняем адрес первого элемента
int n = 1;
unsigned int count = ROWS * COLUMNS;
for (i = 0; i < count; ++i) {
   *p2 = n++;
   ++p2;
}
```

3.14.3. Функция *realloc()*

Функция `realloc()` выполняет перераспределение памяти. Прототип функции:

```
#include <stdlib.h>
void *realloc(void *memory, size_t newSize);
```

В первом параметре функция `realloc()` принимает указатель на ранее выделенную динамическую память, а во втором — новый требуемый размер в байтах. Функция выделит динамическую память длиной `newSize`, скопирует в нее элементы из старой области памяти, освободит старую память и вернет указатель на новую область памяти. Новые элементы будут иметь произвольные значения, так называемый "мусор". Если новая длина меньше старой длины, то лишние элементы будут удалены. Если память не может быть выделена, то функция вернет нулевой указатель, при этом старая область памяти не изменяется (в этом случае возможны утечки памяти, если значение присваивается прежнему указателю).

Если в первом параметре задать значение `NULL`, то будет выделена динамическая память и функция вернет указатель на нее. Если второй параметр имеет значение `0`, то ранее выделенная динамическая память освобождается и функция вернет нулевой указатель.

Пример использования функции `realloc()` приведен в листинге 3.11.

Листинг 3.11. Функция `realloc()`

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

int main(void) {
   unsigned int arr_size = 5;
   int *p = malloc(arr_size * sizeof(int));
   if (!p) exit(1);                  // Выходим при ошибке
   // Нумеруем элементы массива
   for (int i = 0; i < arr_size; ++i) {
      p[i] = i + 1;
   }
```

```
    // Увеличиваем количество элементов
    arr_size += 2;
    p = realloc(p, arr_size * sizeof(int));
    // Здесь возможна утечка памяти, если realloc() вернет NULL
    if (!p) exit(1);                    // Выходим при ошибке
    p[5] = 55;
    p[6] = 66;
    // Выводим значения
    for (int i = 0; i < arr_size; ++i) {
        printf("%d ", p[i]);
    } // 1 2 3 4 5 55 66
    free(p);                            // Освобождаем память
    p = NULL;                           // Обнуляем указатель
    return 0;
}
```

3.15. Структуры

Структура — это совокупность переменных (называемых *членами*, *элементами* или *полями*), объединенных под одним именем. Объявление структуры выглядит следующим образом:

```
struct [<Название структуры>] {
    <Тип данных> <Название поля 1>;
    ...
    <Тип данных> <Название поля N>;
} [<Объявления переменных через запятую>];
```

Допустимо не задавать название структуры, если после закрывающей фигурной скобки указано объявление переменной. Точка с запятой в конце объявления структуры является обязательной.

Объявление структуры только описывает новый тип данных, а не определяет переменную, поэтому память под нее не выделяется. Для того чтобы объявить переменную, ее название указывается после закрывающей фигурной скобки при объявлении структуры или отдельно с помощью названия структуры в качестве типа данных:

```
struct <Название структуры> <Названия переменных через запятую>;
```

Пример одновременного объявления структуры и переменной:

```
struct Point {
    int x;
    int y;
} point1;
```

Пример отдельного объявления переменной:

```
struct Point point2;
```

Одновременно с объявлением переменной можно выполнить инициализацию полей структуры, указав значения внутри фигурных скобок:

```
struct Point point3 = {10, 20};
```

Можно также в списке инициализации указать точку и название поля структуры, а после оператора = — присваиваемое значение:

```
struct Point point4 = {.x = 10, .y = 20};
```

После объявления переменной выделяется необходимый размер памяти. Для получения размера структуры внутри программы следует использовать оператор `sizeof`:

```
<Размер> = sizeof <Переменная>;
<Размер> = sizeof (struct <Название структуры>);
```

Пример:

```
printf("%d\n", (int) sizeof point1);        // 8
printf("%d\n", (int) sizeof(struct Point)); // 8
```

Присвоить или получить значение поля можно с помощью точечной нотации:

```
<Переменная>.<Название поля> = <Значение>;
<Значение> = <Переменная>.<Название поля>;
```

Пример:

```
point1.x = 10;
point1.y = 20;
printf("%d\n", point1.x); // 10
printf("%d\n", point1.y); // 20
```

Одну структуру можно присвоить другой структуре с помощью оператора =. В этом случае копируются значения всех полей структуры. Пример:

```
point2 = point1;
printf("%d\n", point2.x); // 10
printf("%d\n", point2.y); // 20
```

После оператора = и при передаче структуры в качестве параметра в функцию допускается указание *составного литерала*. Литерал структуры состоит из круглых скобок, внутри которых указываются ключевое слово `struct` и название структуры, и фигурных скобок, внутри которых через запятую перечисляются значения:

```
point4 = (struct Point) {30, 40};
printf("%d\n", point4.x); // 30
printf("%d\n", point4.y); // 40
```

Структуры можно вкладывать. При обращении к полю вложенной структуры дополнительно указывается название структуры родителя. В качестве примера объявим структуру `Point` (точка), а затем используем ее для описания координат прямоугольника (листинг 3.12). Структуру, описывающую прямоугольник, объявим без названия.

Листинг 3.12. Использование вложенных структур

```
#include <stdio.h>

struct Point { // Объявление именованной структуры
   int x;
   int y;
};
struct {       // Объявление структуры без названия
   struct Point top_left;
   struct Point bottom_right;
} rect, rect2 = { {10, 20}, {30, 40} };

int main(void) {
   rect.top_left.x = 0;
   rect.top_left.y = 0;
   rect.bottom_right.x = 100;
   rect.bottom_right.y = 100;
   printf("%d\n", rect.top_left.x);      // 0
   printf("%d\n", rect.top_left.y);      // 0
   printf("%d\n", rect.bottom_right.x);  // 100
   printf("%d\n", rect.bottom_right.y);  // 100
   printf("%d\n", rect2.top_left.x);     // 10
   printf("%d\n", rect2.top_left.y);     // 20
   printf("%d\n", rect2.bottom_right.x); // 30
   printf("%d\n", rect2.bottom_right.y); // 40
   printf("%d\n", (int) sizeof rect);    // 16
   return 0;
}
```

Стандарт C11 позволяет создавать *анонимные вложенные структуры*:

```
struct MyStruct {
   int a;
   struct { // Анонимная вложенная структура
      int b;
      int c;
   };
   int d;
};
```

Инициализация и доступ к полям осуществляется следующим образом:

```
struct MyStruct my_struct = {1, {2, 3}, 4};
printf("%d\n", my_struct.a); // 1
printf("%d\n", my_struct.b); // 2
printf("%d\n", my_struct.c); // 3
printf("%d\n", my_struct.d); // 4
```

Адрес структуры можно сохранить в указателе. Объявление указателя на структуру производится так же, как и на любой другой тип данных. Для получения адреса структуры используется оператор &, а для доступа к полю структуры вместо точки применяется оператор ->. Пример использования указателя на структуру приведен в листинге 3.13.

Листинг 3.13. Использование указателя на структуру

```
#include <stdio.h>

struct Point {      // Объявление структуры и переменной
   int x;
   int y;
} point1;

int main(void) {
   struct Point *p = &point1; // Объявление указателя
   p->x = 10;
   p->y = 20;
   printf("%d\n", p->x);     // 10
   printf("%d\n", p->y);     // 20
   printf("%d\n", (*p).x);   // 10
   printf("%d\n", (*p).y);   // 20
   return 0;
}
```

3.16. Битовые поля

Язык C поддерживает *битовые поля*, которые предоставляют доступ к отдельным битам, позволяя тем самым хранить в одной переменной несколько значений, занимающих указанное количество битов. Один бит может содержать только числа 0 или 1. Следует учитывать, что минимальный размер битового поля будет соответствовать типу `int`. Объявление битового поля имеет следующий формат:

```
struct [<Название битового поля>] {
   <Тип данных> [<Название поля 1>]:<Длина в битах>;
   ...
   <Тип данных> [<Название поля N>]:<Длина в битах>;
} [<Объявления переменных через запятую>];
```

Битовые поля объявляются только с типом `int`. В одной структуре можно использовать одновременно битовые поля и обычные поля. Обратите внимание на то, что название битового поля можно не указывать. Кроме того, если длина поля составляет 1 бит, то дополнительно следует указать ключевое слово `unsigned`. Пример объявления битового поля и переменной:

```
struct Status {
   unsigned int flag1:1;
```

```
    unsigned int flag2:1;
    unsigned int flag3:1;
} status = {0, 1, 1};
```

Доступ к полю осуществляется так же, как и к полю структуры:

```
printf("%d\n", status.flag1);                  // 0
printf("%d\n", status.flag2);                  // 1
printf("%d\n", status.flag3);                  // 1
status.flag1 = 1;
printf("%d\n", status.flag1);                  // 1
printf("%d\n", (int) sizeof(struct Status)); // 4
```

3.17. Объединения

Объединение — это область памяти, используемая для хранения данных разных типов. В один момент времени в этой области могут храниться данные только одного типа. Размер объединения будет соответствовать размеру более сложного типа данных. Например, если внутри объединения определены переменные, имеющие типы `int`, `float` и `double`, то размер объединения будет соответствовать размеру типа `double`. Объявление объединения имеет следующий формат:

```
union [<Название объединения>] {
    <Тип данных> <Название члена 1>;
    ...
    <Тип данных> <Название члена N>;
} [<Объявления переменных через запятую>];
```

Точка с запятой в конце объявления обязательна. Объявление только описывает новый тип данных, а не определяет переменную, поэтому память под нее не выделяется. Для того чтобы объявить переменную, ее название указывается после закрывающей фигурной скобки при объявлении объединения или отдельно с помощью названия объединения в качестве типа данных:

```
union <Название объединения> <Названия переменных через запятую>;
```

После объявления переменной компилятор выделяет необходимый размер памяти. Пример объявления объединения и переменной:

```
union myUnion {
    int x;
    float y;
    double z;
} union1;
```

Пример отдельного объявления переменной:

```
union myUnion union2;
```

Одновременно с объявлением переменной можно выполнить инициализацию первого члена объединения, указав значение внутри фигурных скобок:

```
union myUnion union3 = {10};
```

Для инициализации произвольного члена объединения нужно после точки указать его название, а после оператора = — его значение:

```
union myUnion union4 = {.z = 2.5};
```

Присвоить или получить значение можно с помощью точечной нотации:

```
<Переменная>.<Название члена> = <Значение>;
<Значение> = <Переменная>.<Название члена>;
```

Пример:

```
union1.x = 10;
printf("%d\n", union1.x); // 10
```

При использовании указателя на объединение доступ осуществляется так:

```
<Указатель> -> <Название члена> = <Значение>;
<Значение> = <Указатель> -> <Название члена>;
```

Пример использования объединений приведен в листинге 3.14.

Листинг 3.14. Объединения

```
#include <stdio.h>

union myUnion {
   int x;
   float y;
   double z;
} union1;

int main(void) {
   union1.x = 10;
   printf("%d\n", union1.x);                    // 10
   union1.z = 2.5;
   printf("%.2f\n", union1.z);                  // 2.50
   // Объявление переменной
   union myUnion union2;
   union2.x = 5;
   printf("%d\n", union2.x);                    // 5
   // Объявление указателя на объединение
   union myUnion *p = &union2;
   printf("%d\n", p->x);                        // 5
   p->x = 20;
   printf("%d\n", p->x);                        // 20
   printf("%d\n", union2.x);                    // 20
   // Определение размера объединения
   printf("%d\n", (int) sizeof(union myUnion)); // 8
   printf("%d\n", (int) sizeof(union1));        // 8
   return 0;
}
```

Стандарт C11 позволяет создавать *анонимные вложенные объединения*:

```
struct MyStruct {
   int a;
   union { // Анонимное вложенное объединение
      int b;
      double c;
   };
   int d;
};
```

Инициализация и доступ к полям осуществляется следующим образом:

```
struct MyStruct my_struct = {1, {.c = 2.5}, 4};
printf("%d\n", my_struct.a);   // 1
printf("%.2f\n", my_struct.c); // 2.50
printf("%d\n", my_struct.d);   // 4
```

3.18. Перечисления

Перечисление — это совокупность целочисленных констант, описывающих все допустимые значения переменной. Если переменной присвоить значение, не совпадающее с перечисленными константами, то компилятор выведет сообщение об ошибке. Объявление перечисления имеет следующий формат:

```
enum [<Название перечисления>] {
   <Список констант через запятую>
} [<Объявления переменных через запятую>];
```

Точка с запятой в конце объявления является обязательной. Для объявления переменной ее название указывается после закрывающей фигурной скобки при объявлении перечисления или отдельно с помощью названия перечисления в качестве типа данных:

```
enum <Название перечисления> <Названия переменных через запятую>;
```

Пример одновременного объявления перечисления и переменной:

```
enum Color {
   RED, BLUE, GREEN, BLACK
} color1;
```

Пример отдельного объявления переменной:

```
enum Color color2;
```

Константам RED, BLUE, GREEN и BLACK автоматически присваиваются целочисленные значения, начиная с нуля. Значение каждой последующей константы будет на единицу больше предыдущей. Нумерация производится слева направо. Таким образом, константа RED будет иметь значение 0, BLUE — 1, GREEN — 2, а BLACK — 3.

При объявлении перечисления константе можно присвоить другое значение. В этом случае последующая константа будет иметь значение на единицу больше того, другого значения. Пример:

```
enum Color {
   RED = 3, BLUE, GREEN = 7, BLACK
} color1;
```

В этом примере константа RED будет иметь значение 3, а не 0, BLUE — 4, GREEN — 7, а BLACK — 8. Присвоить значение переменной можно так:

```
color1 = BLACK;
```

Допустимо не задавать одновременно название перечисления и объявление переменной. В этом случае переменные, указанные внутри фигурных скобок, используются как константы. Пример:

```
enum {
   RED, BLUE, GREEN, BLACK
};
printf("%d\n", RED); // 0
```

Пример использования перечисления приведен в листинге 3.15.

Листинг 3.15. Перечисления

```
#include <stdio.h>

enum Color {
   RED, BLUE, GREEN = 5, BLACK
};

int main(void) {
   enum Color color;                // Объявление переменной
   color = BLACK;
   printf("%d\n", color);           // 6
   if (color == BLACK) {            // Проверка значения
      printf("color == BLACK\n");   // Выведет: color == BLACK
   }
   else {
      printf("color != BLACK\n");
   }
   return 0;
}
```

3.19. Приведение типов

Как вы уже знаете, при объявлении переменной необходимо указать определенный тип данных. Далее над переменной можно производить операции, предназначенные для этого типа данных. Если в выражении используются переменные, имеющие

разный тип данных, то тип результата выражения будет соответствовать наиболее сложному типу. Например, если производится сложение переменной, имеющей тип `int`, с переменной, имеющей тип `double`, то целое число будет автоматически преобразовано в вещественное. Результатом этого выражения будет значение типа `double`.

Например, после компиляции следующего фрагмента кода будет выведено предупреждающее сообщение (при условии, что в команде на компиляцию указан флаг `-Wconversion`):

```
short y1 = 1, y2 = 2;
y1 = y1 + y2;
// warning: conversion from 'int' to 'short int' may change value
```

Для того чтобы результат выполнения этого выражения не вызывал сомнений компилятора, необходимо выполнить операцию явного *приведения типов*. Формат операции:

```
(<Тип результата>)<Выражение или переменная>
```

Пример приведения типов:

```
short y1 = 1, y2 = 2;
y1 = (short)(y1 + y2);    // OK
```

В этом случае компилятор считает, что мы знаем, что делаем, и осведомлены о возможном несоответствии значения указанному типу данных. Если значение выражения будет выходить за диапазон значений типа, то произойдет усечение результата, но никакое сообщение об ошибке не появится. Давайте рассмотрим это на примере:

```
char c1 = 127, c2 = 0;
c1 = (char)(c1 + c2);     // OK. 127
c1 = 127, c2 = 10;
c1 = (char)(c1 + c2);     // Усечение. -119
```

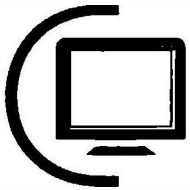
Результатом первого выражения будет число `127`, которое входит в диапазон значений типа `char`, а вот результат второго выражения — число `137` — в диапазон не входит, и происходит усечение. В итоге мы имеем число `-119`. Согласитесь, мы получили совсем не то, что хотели. Поэтому приведением типов нужно пользоваться осторожно, хотя в некоторых случаях такое усечение очень даже полезно. Например, вещественное число можно преобразовать в целое. В этом случае дробная часть просто отбрасывается:

```
float x = 1.2f;
double y = 2.5;
printf("%.2f\n", x);          // 1.20
printf("%d\n", (int)x);       // 1
printf("%.2f\n", y);          // 2.50
printf("%d\n", (int)y);       // 2
```

Рассмотрим пример, который демонстрирует частый способ применения приведения типов. В языке C деление целых чисел всегда возвращает целое число. Дробная часть при этом просто отбрасывается. Для того чтобы деление целых чисел возвращало вещественное число, необходимо преобразовать одно из целых чисел в вещественное (второе число преобразуется автоматически):

```
int x = 10, y = 3;
printf("%d\n", x / y);              // 3
printf("%.2f\n", (double)(x / y)); // 3.00
printf("%.2f\n", (double)x / y);   // 3.33
```

ГЛАВА 4



Операторы и циклы

Операторы позволяют выполнить определенные действия с данными. Например, операторы присваивания служат для сохранения данных в переменной, математические операторы предназначены для арифметических вычислений, а условные операторы позволяют в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнять его.

4.1. Математические операторы

Производить арифметические вычисления позволяют следующие операторы:

- `+` — сложение:

```
printf("%d\n", 10 + 15);              // 25
```

- `-` — вычитание:

```
printf("%d\n", 35 - 15);              // 20
```

- `-` — унарный минус:

```
int x = 10;
printf("%d\n", -x);                   // -10
```

- `*` — умножение:

```
printf("%d\n", 25 * 2);               // 50
```

- `/` — деление. Если производится деление целых чисел, то остаток отбрасывается и возвращается целое число. Деление вещественных чисел производится классическим способом. Если в выражении участвуют вещественное и целое числа, то целое число автоматически преобразуется в вещественное. Пример:

```
printf("%d\n", 10 / 3);               // 3
printf("%.3f\n", (double)(10 / 3));   // 3.000
printf("%.3f\n", 10.0 / 3.0);         // 3.333
printf("%.3f\n", 10.0 / 3);           // 3.333
printf("%.3f\n", (double)10 / 3);     // 3.333
```

Целочисленное деление на 0 приведет к неопределенности. Деление вещественного числа на 0 приведет к значению плюс или минус INFINITY (бесконечность), а деление вещественного числа 0.0 на 0.0 — к неопределенности или значению NAN (нет числа):

```
printf("%f\n", 10.0 / 0.0);           // 1.#INF00
printf("%f\n", -10.0 / 0.0);          // -1.#INF00
printf("%f\n", 0.0 / 0.0);            // -1.#IND00
// #include <math.h>
printf("%f\n", INFINITY);             // 1.#INF00
printf("%f\n", -INFINITY);            // -1.#INF00
printf("%f\n", NAN);                  // 1.#QNAN0
printf("%f\n", NAN + 1);              // 1.#QNAN0
printf("%f\n", 0 * INFINITY);         // 1.#QNAN0
printf("%f\n", INFINITY / INFINITY);  // 1.#QNAN0
printf("%f\n", -INFINITY + INFINITY); // 1.#QNAN0
```

- % — остаток от деления. Обратите внимание на то, что этот оператор нельзя применять к вещественным числам. Пример:

```
printf("%d\n", 10 % 2);               // 0 (10 - 10 / 2 * 2)
printf("%d\n", 10 % 3);               // 1 (10 - 10 / 3 * 3)
printf("%d\n", 10 % 4);               // 2 (10 - 10 / 4 * 4)
printf("%d\n", 10 % 6);               // 4 (10 - 10 / 6 * 6)
```

- ++ — оператор инкремента. Увеличивает значение переменной на 1:

```
int x = 10;
++x;                          // Эквивалентно x = x + 1;
printf("%d\n", x);            // 11
```

- -- — оператор декремента. Уменьшает значение переменной на 1:

```
int x = 10;
--x;                          // Эквивалентно x = x - 1;
printf("%d\n", x);            // 9
```

Операторы инкремента и декремента могут использоваться в постфиксной или префиксной формах:

```
x++; x--; // Постфиксная форма
++x; --x; // Префиксная форма
```

При постфиксной форме (x++) возвращается значение переменной перед операцией, а при префиксной форме (++x) — вначале производится операция и только потом возвращается значение. Продемонстрируем это на примере (листинг 4.1).

Листинг 4.1. Постфиксная и префиксная форма

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

```
int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int x = 0, y = 0;
   x = 5;
   y = x++; // y = 5, x = 6
   puts("Постфиксная форма (y = x++;):");
   printf("y = %d\n", y);
   printf("x = %d\n", x);
   x = 5;
   y = ++x; // y = 6, x = 6
   puts("Префиксная форма (y = ++x;):");
   printf("y = %d\n", y);
   printf("x = %d\n", x);
   return 0;
}
```

В итоге получим следующий результат:

```
Постфиксная форма (y = x++;):
y = 5
x = 6
Префиксная форма (y = ++x;):
y = 6
x = 6
```

Если операторы инкремента и декремента используются в сложных выражениях, то понять, каким будет результат выполнения выражения, становится сложно. Например, каким будет значение переменной `y` после выполнения этих инструкций?

```
int x = 5, y = 0;
y = ++x + ++x;
```

Для того чтобы облегчить жизнь себе и всем другим программистам, которые будут разбираться в программе, операторы инкремента и декремента лучше использовать отдельно от других операторов.

4.2. Побитовые операторы

Побитовые операторы **предназначены для манипуляции отдельными битами. Эти операторы нельзя применять к вещественным числам, логическим значениям и другим более сложным типам. Язык C поддерживает следующие побитовые операторы:**

- **`~` — двоичная инверсия. Значение каждого бита заменяется противоположным:**

```
unsigned char x = 100;          // 01100100
x = (unsigned char) ~x;         // 10011011
```

- **`&` — двоичное И:**

```
unsigned char x = 100;          // 01100100
unsigned char y = 75;           // 01001011
unsigned char z = x & y;        // 01000000
```

- | — двоичное или:

```
unsigned char x = 100;          // 01100100
unsigned char y = 75;           // 01001011
unsigned char z = x | y;        // 01101111
```

- ^ — двоичное исключающее или:

```
unsigned char x = 100;          // 01100100
unsigned char y = 250;          // 11111010
unsigned char z = x ^ y;        // 10011110
```

- << — сдвиг влево — сдвигает двоичное представление числа влево на один или более разрядов и заполняет разряды справа нулями:

```
unsigned char x = 100;          // 01100100
x = (unsigned char)(x << 1);    // 11001000
x = (unsigned char)(x << 1);    // 10010000
x = (unsigned char)(x << 2);    // 01000000
```

- >> — сдвиг вправо — сдвигает двоичное представление числа вправо на один или более разрядов и заполняет разряды слева нулями, если число положительное:

```
unsigned char x = 100;          // 01100100
x = (unsigned char)(x >> 1);    // 00110010
x = (unsigned char)(x >> 1);    // 00011001
x = (unsigned char)(x >> 2);    // 00000110
```

Если число отрицательное, то разряды слева заполняются единицами:

```
char x = -127;                  // 10000001
x = (char)(x >> 1);             // 11000000
x = (char)(x >> 2);             // 11110000
x = (char)(x << 1);             // 11100000
x = (char)(x >> 1);             // 11110000
```

Наиболее часто двоичное представление числа используется для хранения различных флагов (0 — флаг сброшен, 1 — флаг установлен). Примеры установки, снятия и проверки установки флага приведены в листинге 4.2. Для того чтобы иметь возможность видеть результат в окне консоли, дополнительно напишем функцию, позволяющую преобразовать число типа `unsigned char` в строку в двоичном формате.

Листинг 4.2. Работа с флагами

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

char *ucharToBinaryString(unsigned char x, char *str);

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
```

```
    char str[9] = "";
    const unsigned char FLAG1 = 1,  FLAG2 = 2,  FLAG3 = 4, FLAG4 = 8,
                        FLAG5 = 16, FLAG6 = 32, FLAG7 = 64, FLAG8 = 128;
    unsigned char x = 0;     // Все флаги сброшены
    printf("%s\n", ucharToBinaryString(x, str));     // 00000000
    unsigned char y = 0xFF; // Все флаги установлены
    printf("%s\n", ucharToBinaryString(y, str));     // 11111111
    // Устанавливаем флаги FLAG1 и FLAG7
    x = x | FLAG1 | FLAG7;
    printf("%s\n", ucharToBinaryString(x, str));     // 01000001
    // Устанавливаем флаги FLAG4 и FLAG5
    x = x | FLAG4 | FLAG5;
    printf("%s\n", ucharToBinaryString(x, str));     // 01011001
    // Снимаем флаги FLAG4 и FLAG5
    x = x ^ FLAG4 ^ FLAG5;
    printf("%s\n", ucharToBinaryString(x, str));     // 01000001
    // Проверка установки флага FLAG1
    if ((x & FLAG1) != 0) {
       puts("FLAG1 установлен");
    }
    printf("%s\n", ucharToBinaryString(FLAG1, str)); // 00000001
    printf("%s\n", ucharToBinaryString(FLAG2, str)); // 00000010
    printf("%s\n", ucharToBinaryString(FLAG3, str)); // 00000100
    printf("%s\n", ucharToBinaryString(FLAG4, str)); // 00001000
    printf("%s\n", ucharToBinaryString(FLAG5, str)); // 00010000
    printf("%s\n", ucharToBinaryString(FLAG6, str)); // 00100000
    printf("%s\n", ucharToBinaryString(FLAG7, str)); // 01000000
    printf("%s\n", ucharToBinaryString(FLAG8, str)); // 10000000
    return 0;
}

char *ucharToBinaryString(unsigned char x, char *str) {
    int k = 7;
    while (k >= 0) {
       if ((x & 1) != 0) {          // Проверка статуса последнего бита
          str[k] = '1';
       }
       else {
          str[k] = '0';
       }
       x = (unsigned char)(x >> 1); // Сдвиг на один разряд вправо
       --k;
    }
    str[8] = '\0';
    return str;
}
```

4.3. Операторы присваивания

Операторы присваивания предназначены для сохранения значения в переменной. Перечислим операторы присваивания, доступные в языке C.

- `=` — присваивает переменной значение. Обратите внимание на то, что хотя оператор похож на математический знак равенства, смысл у него в языке C совершенно другой. Справа от оператора присваивания может располагаться переменная, литерал или сложное выражение. Слева от оператора присваивания должна располагаться переменная, а не литерал или выражение. Пример присваивания значения:

```
int x;
x = 10;
x = 12 * 10 + 45 / 5;
12 + 45 = 45 + 5; // Так нельзя!!!
```

  В одной инструкции можно присвоить значение сразу нескольким переменным:

```
int x, y, z;
x = y = z = 2;
```

- `+=` — увеличивает значение переменной на указанную величину:

```
x += 10;      // Эквивалентно x = x + 10;
```

- `-=` — уменьшает значение переменной на указанную величину:

```
x -= 10;      // Эквивалентно x = x - 10;
```

- `*=` — умножает значение переменной на указанную величину:

```
x *= 10 + 5; // Эквивалентно x = x * (10 + 5);
```

- `/=` — делит значение переменной на указанную величину:

```
x /= 2;       // Эквивалентно x = x / 2;
```

- `%=` — делит значение переменной на указанную величину и возвращает остаток:

```
x %= 2;       // Эквивалентно x = x % 2;
```

- `&=`, `|=`, `^=`, `<<=` и `>>=` — побитовые операторы с присваиванием.

4.4. Оператор запятая

Оператор запятая позволяет разместить сразу несколько выражений внутри одной инструкции. Например, в предыдущих примерах мы использовали этот оператор для объявления нескольких переменных в одной инструкции:

```
int x, y;
```

Результат вычисления последнего выражения можно присвоить какой-либо переменной. При этом выражения должны быть расположены внутри круглых скобок, т. к. приоритет оператора "запятая" меньше приоритета оператора присваивания. Пример:

```
int x = 0, y = 0, z = 0;
x = (y = 10, z = 20, y + z);
printf("%d\n", x); // 30
printf("%d\n", y); // 10
printf("%d\n", z); // 20
```

В этом примере после объявления и инициализации переменных переменной `y` присваивается значение `10`, затем переменной `z` присваивается значение `20`, далее вычисляется выражение `y + z` и его результат присваивается переменной `x`.

4.5. Операторы сравнения

Операторы сравнения используются в логических выражениях. Перечислим операторы сравнения, доступные в языке C:

- `==` — равно;
- `!=` — не равно;
- `<` — меньше;
- `>` — больше;
- `<=` — меньше или равно;
- `>=` — больше или равно.

Логические выражения возвращают только два значения: `Истина` (соответствует числу `1`) или `Ложь` (соответствует числу `0`). Пример вывода значения логического выражения:

```
printf("%d\n", 10 == 10);        // 1 (Истина)
printf("%d\n", 10 == 5);         // 0 (Ложь)
printf("%d\n", 10 != 5);         // 1 (Истина)
printf("%d\n", 10 < 5);          // 0 (Ложь)
printf("%d\n", 10 > 5);          // 1 (Истина)
printf("%d\n", 10 <= 5);         // 0 (Ложь)
printf("%d\n", 10 >= 5);         // 1 (Истина)
```

Логическое выражение может не содержать операторов сравнения вообще. В этом случае число `0` автоматически преобразуется в ложное значение, а любое ненулевое значение (в том числе и отрицательное) — в истинное значение:

```
printf("%d\n", (_Bool)10);       // 1 (Истина)
printf("%d\n", (_Bool)-10);      // 1 (Истина)
printf("%d\n", (_Bool)12.5);     // 1 (Истина)
printf("%d\n", (_Bool)0.1);      // 1 (Истина)
printf("%d\n", (_Bool)"str");    // 1 (Истина)
printf("%d\n", (_Bool)"");       // 1 (Истина)
printf("%d\n", (_Bool)0.0);      // 0 (Ложь)
printf("%d\n", (_Bool)0);        // 0 (Ложь)
printf("%d\n", (_Bool)NULL);     // 0 (Ложь)
```

```
// #include <math.h>
printf("%d\n", (_Bool)INFINITY); // 1 (Истина)
printf("%d\n", (_Bool)NAN);      // 1 (Истина)
```

В этом примере мы воспользовались приведением к типу `_Bool`. Внутри логического выражения приведение типов выполнять не нужно, т. к. оно производится автоматически.

Следует учитывать, что оператор проверки на равенство содержит два символа `=`. Указание одного символа `=` является логической ошибкой, т. к. этот оператор используется для присваивания значения переменной, а не для проверки условия. Использовать оператор присваивания внутри логического выражения допускается, поэтому компилятор не выведет сообщение об ошибке, однако программа может выполняться некорректно. Подобную ошибку часто допускают начинающие программисты. Например, в следующем примере вместо проверки равенства числу `11` производится операция присваивания:

```
int x = 10;
if (x = 11) {
   puts("x == 11");
}
```

Выражение внутри круглых скобок присвоит переменной `x` значение `11`, а затем будет произведено сравнение. Любое число, не равное `0`, трактуется как логическое значение `Истина`, поэтому в окне консоли отобразится строка `x == 11`.

Значение логического выражения можно инвертировать с помощью оператора `!`. В этом случае, если логическое выражение возвращает `Ложь`, то `!Ложь` вернет значение `Истина`. Пример:

```
printf("%d\n",  (10 == 5));  // 0 (Ложь)
printf("%d\n", !(10 == 5));  // 1 (Истина)
```

Если оператор `!` указать дважды, то любое значение преобразуется в логическое:

```
printf("%d\n", !!10);        // 1 (Истина)
printf("%d\n", !!0);         // 0 (Ложь)
printf("%d\n", !!NULL);      // 0 (Ложь)
```

Несколько логических выражений можно объединить в одно большое с помощью следующих операторов.

- `&&` — логическое И. Логическое выражение вернет `Истина` только в случае, если оба подвыражения вернут `Истина`. Пример:

```
printf("%d\n", (10 == 10) && (5 != 3)); // 1 (Истина)
printf("%d\n", (10 == 10) && (5 == 3)); // 0 (Ложь)
```

- `||` — логическое ИЛИ. Логическое выражение вернет `Истина`, если хотя бы одно из подвыражений вернет `Истина`. Следует учитывать, что если подвыражение вернет значение `Истина`, то следующие подвыражения вычисляться не будут. Например, в логическом выражении `f1() || f2() || f3()` функция `f2()` будет

вызвана, только если функция `f1()` вернет Ложь, а функция `f3()` будет вызвана, только если функции `f1()` и `f2()` вернут Ложь. Пример использования оператора:

```
printf("%d\n", (10 == 10) || (5 != 3)); // 1 (Истина)
printf("%d\n", (10 == 10) || (5 == 3)); // 1 (Истина)
```

Результаты выполнения операторов `&&` и `||` показаны в табл. 4.1.

Таблица 4.1. Операторы `&&` и `||`

a	b	a && b	a || b
ИСТИНА	ИСТИНА	ИСТИНА	ИСТИНА
ИСТИНА	ЛОЖЬ	ЛОЖЬ	ИСТИНА
ЛОЖЬ	ИСТИНА	ЛОЖЬ	ИСТИНА
ЛОЖЬ	ЛОЖЬ	ЛОЖЬ	ЛОЖЬ

4.6. Приоритет выполнения операторов

Все операторы выполняются в порядке приоритета (табл. 4.2). Вначале вычисляется выражение, в котором оператор имеет наивысший приоритет, а затем выражение с меньшим приоритетом. Например, выражение с оператором умножения будет выполнено раньше выражения с оператором сложения, т. к. приоритет оператора умножения выше. Если приоритет операторов одинаковый, то используется порядок вычисления, определенный для конкретного оператора. Операторы присваивания и унарные операторы выполняются справа налево. Математические, побитовые и операторы сравнения, а также оператор запятая выполняются слева направо. Изменить последовательность вычисления выражения можно с помощью круглых скобок. Пример:

```
int x = 0;
x = 5 + 10 * 3 / 2;    // Умножение -> деление -> сложение
printf("%d\n", x);     // 20
x = (5 + 10) * 3 / 2; // Сложение -> умножение -> деление
printf("%d\n", x);     // 22
```

Таблица 4.2. Приоритеты операторов

Приоритет	Операторы
1 (высший)	`() [] . -> ++ --` (постфиксная форма)
2	`++ --` (префиксная форма) `~ ! * &` (разыменование) (взятие адреса) `-` (унарный минус) (приведение типа) `sizeof`
3	`* / %`
4	`+ -` (сложение и вычитание)
5	`<< >>`

Таблица 4.2 (окончание)

Приоритет	Операторы
6	< > <= >=
7	== !=
8	& (побитовое И)
9	^
10	|
11	&&
12	||
13	?:
14	= *= /= %= += -= >>= <<= &= ^= |=
15 (низший)	, (запятая)

4.7. Оператор ветвления *if*

Оператор ветвления `if` позволяет в зависимости от значения логического выражения выполнить отдельный блок программы или, наоборот, не выполнять его. Оператор имеет следующий формат:

```
if (<Логическое выражение>) {
   <Блок, выполняемый если условие истинно>
}
[else {
   <Блок, выполняемый если условие ложно>
}]
```

Если логическое выражение возвращает значение `Истина`, то выполняются инструкции, расположенные внутри фигурных скобок, сразу после оператора `if`. Если логическое выражение возвращает значение `Ложь`, то выполняются инструкции после ключевого слова `else`. Блок `else` является необязательным. Допускается не указывать фигурные скобки, если блоки состоят из одной инструкции. В качестве примера проверим число, введенное пользователем, на четность и выведем соответствующее сообщение (листинг 4.3).

Листинг 4.3. Проверка числа на четность

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int x = 0;
```

```
   printf("Введите целое число: ");
   fflush(stdout);
   int status = scanf("%d", &x);
   if (status != 1) {
      puts("Вы ввели не число");
      return 0;
   }
   else {
      if (x % 2 == 0)
         printf("%d - четное число", x);
      else
         printf("%d - нечетное число", x);
      printf("\n");
   }
   return 0;
}
```

Как видно из примера, один оператор ветвления можно вложить в другой. Первый оператор `if` проверяет отсутствие ошибки при вводе числа. Если ошибки при вводе числа нет, то выполняется блок `else`. В этом блоке находится второй оператор ветвления, который проверяет число на четность. В зависимости от условия `x % 2 == 0` выводится соответствующее сообщение. Если число делится на `2` без остатка, то оператор `%` вернет значение `0`, в противном случае — число `1`. Обратите внимание на то, что оператор ветвления не содержит фигурных скобок:

```
if (x % 2 == 0)
   printf("%d - четное число", x);
else
   printf("%d - нечетное число", x);
printf("\n");
```

В этом случае считается, что внутри блока содержится только одна инструкция, поэтому последняя инструкция к блоку `else` не относится. Она будет выполнена в любом случае, вне зависимости от условия. Для того чтобы это сделать наглядным, перед инструкциями, расположенными внутри блока, добавлено одинаковое количество пробелов. Если записать это следующим образом, то ничего не изменится:

```
if (x % 2 == 0)
printf("%d - четное число", x);
else
printf("%d - нечетное число", x);
printf("\n");
```

Однако в дальнейшем разбираться в таком коде будет неудобно. Поэтому перед инструкциями внутри блока всегда следует размещать одинаковый отступ. В качестве отступа можно использовать пробелы или символ табуляции. При использовании пробелов размер отступа равняется трем или четырем пробелам для блока пер-

вого уровня. Для вложенных блоков количество пробелов умножают на уровень вложенности: если для блока первого уровня вложенности использовались три пробела, то для блока второго уровня вложенности должны использоваться шесть пробелов, для третьего уровня — девять пробелов и т. д. В одной программе не следует применять и пробелы, и табуляцию в качестве отступа. Необходимо выбрать что-то одно и пользоваться этим во всей программе.

При отсутствии пробелов или фигурных скобок чтение программы может вызывать затруднения. Даже если знаешь приоритет выполнения операторов, всегда закрадывается сомнение, и чтение программы останавливается. Например, к какому оператору `if` принадлежит блок `else` в этом примере?

```
if (x >= 0) if (x == 0) y = 0; else y = 1;
```

Задумались? А зачем задумываться об этом, если можно сразу расставить фигурные скобки? Ведь тогда никаких сомнений вообще не будет:

```
if (x >= 0) { if (x == 0) y = 0; else y = 1; }
```

А если сделать так, то чтение и понимание программы станет мгновенным:

```
if (x >= 0) {
   if (x == 0) y = 0;
   else y = 1;
}
```

Если блок состоит из нескольких инструкций, следует указать фигурные скобки. Существует несколько стилей размещения скобок в операторе `if`:

```
// Стиль 1
if (<Логическое выражение>) {
   // Инструкции
}
else {
   // Инструкции
}
// Стиль 2
if (<Логическое выражение>) {
   // Инструкции
} else {
   // Инструкции
}
// Стиль 3
if (<Логическое выражение>)
{
   // Инструкции
}
else
{
   // Инструкции
}
```

Многие программисты считают стиль 3 наиболее приемлемым, т. к. открывающая и закрывающая скобки расположены друг под другом. На мой же взгляд образуются лишние пустые строки. Так как размеры экрана ограничены, при наличии пустой строки на экран помещается меньше кода, и приходится чаще пользоваться полосой прокрутки. Если размещать инструкции с равным отступом, то блок кода выделяется визуально, и следить за положением фигурных скобок просто излишне. Тем более что редактор кода позволяет подсветить парные скобки. Какой стиль использовать, зависит от личного предпочтения программиста или правил оформления кода, принятых в определенной компании. Главное, чтобы стиль оформления внутри одной программы был одинаковым. В этой книге мы будем пользоваться стилем 1.

В листинге 4.3 при вложении операторов мы воспользовались следующей схемой:

```
if (<Условие 1>) {
   // Инструкции
}
else {
   if (<Условие 2>) {
      // Инструкции
   }
   else {
      // Инструкции
   }
}
```

Для того чтобы проверить несколько условий, эту схему можно изменить так:

```
if (<Условие 1>) {
   // <Блок 1>
}
else if (<Условие 2>) {
   // <Блок 2>
}
// ... Фрагмент опущен ...
else if (<Условие N>) {
   // <Блок N>
}
else {
   // <Блок else>
}
```

Если `<Условие 1>` истинно, то выполняется `<Блок 1>`, а и все остальные условия пропускаются. Если `<Условие 1>` ложно, то проверяется `<Условие 2>`. Если `<Условие 2>` истинно, то выполняется `<Блок 2>`, а все остальные условия пропускаются. Если `<Условие 2>` ложно, то точно так же проверяются остальные условия. Если все условия ложны, то выполняется `<Блок else>`. В качестве примера определим, какое число от `0` до `2` ввел пользователь (листинг 4.4).

Листинг 4.4. Проверка нескольких условий

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int x = 0;
   printf("Введите число от 0 до 2: ");
   fflush(stdout);
   int status = scanf("%d", &x);
   if (status != 1) {
      puts("Вы ввели не число");
      return 0;
   }
   else {
      if (x == 0)
         puts("Вы ввели число 0");
      else if (x == 1)
         puts("Вы ввели число 1");
      else if (x == 2)
         puts("Вы ввели число 2");
      else {
         puts("Вы ввели другое число");
         printf("x = %d\n", x);
      }
   }
   return 0;
}
```

4.8. Оператор *?:*

Для проверки условия вместо оператора `if` можно использовать оператор `?:`, который имеет следующий формат:

```
<Переменная> = <Логическое выражение> ? <Выражение если Истина> :
                                        <Выражение если Ложь>;
```

Если логическое выражение возвращает значение `Истина`, то выполняется выражение, расположенное после вопросительного знака. Если логическое выражение возвращает значение `Ложь`, то выполняется выражение, расположенное после двоеточия. Результат выполнения выражений становится результатом выполнения оператора. Пример проверки числа на четность и вывода результата:

```
int x = 10;
printf("%d%s", x,
      (x % 2 == 0) ? " - четное число" : " - нечетное число");
```

Обратите внимание на то, что в качестве операндов указываются именно выражения, а не инструкции. Кроме того, выражения обязательно должны возвращать какое-либо значение, причем одинакового типа. Так как оператор возвращает значение, его можно использовать внутри выражений:

```
int x, y;
x = 0;
y = 30 + 10 / (!x ? 1 : x);   // 30 + 10 / 1
printf("%d\n", y);            // 40
x = 2;
y = 30 + 10 / (!x ? 1 : x);   // 30 + 10 / 2
printf("%d\n", y);            // 35
```

В качестве операнда можно указать функцию, которая возвращает значение:

```
int func1(int x) {
   printf("%d%s\n", x,  " - четное число");
   return 0;
}
int func2(int x) {
   printf("%d%s\n", x, " - нечетное число");
   return 0;
}
// ... Фрагмент опущен ...
int x = 10;
(x % 2 == 0) ? func1(x) : func2(x);
```

Как видно из примера, значение, возвращаемое оператором, можно проигнорировать.

4.9. Оператор выбора *switch*

Оператор выбора `switch` имеет следующий формат:

```
switch (<Выражение>) {
   case <Константа 1>:
      <Инструкции>
      break;
[...
   case <Константа N>:
      <Инструкции>
      break;]
[   default:
      <Инструкции>]
}
```

Вместо условия оператор `switch` принимает выражение. В зависимости от значения выражения выполняется один из блоков `case`, в котором указано это значение. Значением выражения должно быть целое число или символ. Если ни одно из значений не описано в блоках `case`, то выполняется блок `default` (если он указан). Обратите

внимание на то, что значения в блоках `case` не могут иметь одинаковые константы. Пример использования оператора `switch` приведен в листинге 4.5.

Листинг 4.5. Использование оператора `switch`

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int os = 0;
   printf("Какой операционной системой вы пользуетесь?\n\
1 - Windows XP\n\
2 - Windows 8\n\
3 - Windows 10\n\
4 - Другая\n\n\
Введите число, соответствующее ответу: ");
   fflush(stdout);
   int status = scanf("%d", &os);
   if (status != 1) {
      puts("Вы ввели не число");
      return 0;
   }
   else {
      switch (os) {
         case 1:
            puts("Вы выбрали - Windows XP");
            break;
         case 2:
            puts("Вы выбрали - Windows 8");
            break;
         case 3:
            puts("Вы выбрали - Windows 10");
            break;
         case 4:
            puts("Вы выбрали - Другая");
            break;
         default:
            puts("Мы не смогли определить систему");
      }
   }
   return 0;
}
```

Как видно из примера, в конце каждого блока `case` указан оператор `break`. Этот оператор позволяет досрочно выйти из оператора выбора `switch`. Если не указать

оператор `break`, то будет выполняться следующий блок `case` вне зависимости от указанного значения. В некоторых случаях это может быть полезным. Например, можно выполнить одни и те же инструкции при разных значениях, разместив инструкции в конце диапазона значений. Пример:

```
char ch = 'b';
switch (ch) {
   case 'a':
   case 'b':
   case 'c':
      puts("a, b или c");
      break;
   case 'd':
      puts("Только d");
}
```

В операторе `case` можно указать одно из значений перечисления. Пример проверки выбранного значения перечисления приведен в листинге 4.6.

Листинг 4.6. Использование оператора `switch` с перечислениями

```
#include <stdio.h>

enum Color { RED, BLUE, GREEN, BLACK };

int main(void) {
   enum Color color = GREEN;
   switch (color) {
      case RED:
         puts("RED");
         break;
      case BLUE:
         puts("BLUE");
         break;
      case GREEN:
         puts("GREEN");
         break;
      case BLACK:
         puts("BLACK");
   }
   return 0;
}
```

4.10. Цикл *for*

Операторы *циклов* позволяют выполнить одни и те же инструкции многократно. Предположим, нужно вывести все числа от `1` до `100` по одному на строке. Обычным способом пришлось бы писать 100 строк кода:

```
printf("%d\n", 1);
printf("%d\n", 2);
// ...
printf("%d\n", 100);
```

С помощью циклов то же действие можно выполнить одной строкой кода:

```
for (int i = 1; i <= 100; ++i) printf("%d\n", i);
```

Цикл `for` используется для выполнения инструкций определенное число раз. Цикл имеет следующий формат:

```
for (<Начальное значение>; <Условие>; <Приращение>) {
   <Инструкции>
}
```

Параметры имеют следующие значения:

- `<Начальное значение>` — присваивает переменной-счетчику начальное значение;
- `<Условие>` — содержит логическое выражение. Пока логическое выражение возвращает значение `Истина`, выполняются инструкции внутри цикла;
- `<Приращение>` — задает изменение переменной-счетчика на каждой итерации.

Последовательность работы цикла `for`:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие: если оно истинно, выполняются инструкции внутри цикла, а в противном случае выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в `<Приращение>`.
4. Переход к пункту 2.

Переменная-счетчик может быть объявлена как вне цикла `for`, так и в параметре `<Начальное значение>`. Если переменная объявлена в параметре, то она будет видна только внутри цикла. Кроме того, допускается объявить переменную вне цикла и сразу присвоить ей начальное значение. В этом случае параметр `<Начальное значение>` можно оставить пустым. Пример:

```
int i; // Объявление вне цикла
for (i = 1; i <= 20; ++i) {
   printf("%d\n", i);
}
// Переменная i видна вне цикла
printf("%d\n", i); // 21
// Объявление внутри цикла
for (int j = 1; j <= 20; ++j) {
   printf("%d\n", j);
}
// Переменная j НЕ видна вне цикла
int k = 1; // Инициализация вне цикла
for ( ; k <= 20; ++k) {
   printf("%d\n", k);
}
```

На заметку

При использовании старых компиляторов все локальные переменные должны быть объявлены в самом начале функции.

Цикл выполняется до тех пор, пока <Условие> не вернет Ложь. Если это не произойдет, то цикл будет бесконечным. Логическое выражение, указанное в параметре <Условие>, вычисляется на каждой итерации. Поэтому если внутри логического выражения производятся какие-либо вычисления и значение не изменяется внутри цикла, то вычисление следует вынести в параметр <Начальное значение>. В этом случае вычисление указывается после присваивания значения переменной-счетчику через запятую. Пример:

```
for (int i = 1, j = 10 + 30; i <= j; ++i) {
   printf("%d\n", i);
}
```

Выражение, указанное в параметре <Приращение>, может не только увеличивать значение переменной-счетчика, но и уменьшать его. Кроме того, значение может изменяться на любую величину. Пример:

```
// Выводим числа от 100 до 1
for (int i = 100; i > 0; --i) {
   printf("%d\n", i);
}
// Выводим четные числа от 1 до 100
for (int j = 2; j <= 100; j += 2) {
   printf("%d\n", j);
}
```

Если переменная-счетчик изменяется внутри цикла, то выражение в параметре <Приращение> можно вообще не указывать:

```
for (int i = 1; i <= 10; ) {
   printf("%d\n", i);
   ++i; // Приращение
}
```

Все параметры цикла for и инструкции внутри цикла являются необязательными. Хотя параметры можно не указывать, точки с запятой обязательно должны быть. Если все параметры не указаны, то цикл окажется бесконечным. Для того чтобы выйти из бесконечного цикла, следует использовать оператор break. Пример:

```
int i = 1;                  // <Начальное значение>
for ( ; ; ) {               // Бесконечный цикл
   if (i <= 10) {           // <Условие>
      printf("%d\n", i);
      ++i;                  // <Приращение>
   }
   else {
      break;                // Выходим из цикла
   }
}
```

4.11. Цикл *while*

Выполнение инструкций в цикле `while` продолжается до тех пор, пока логическое выражение истинно. Цикл имеет следующий формат:

```
<Начальное значение>
while (<Условие>) {
   <Инструкции>
   <Приращение>
}
```

Последовательность работы цикла `while`:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие: если оно истинно, выполняются инструкции внутри цикла, иначе выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в `<Приращение>`.
4. Переход к пункту 2.

Выведем все числа от `1` до `100`, используя цикл `while`:

```
int i = 1;                  // <Начальное значение>
while (i <= 100) {          // <Условие>
   printf("%d\n", i);       // <Инструкции>
   ++i;                     // <Приращение>
}
```

> ***Внимание!***
>
> Если `<Приращение>` **не указано, то цикл будет бесконечным**.

4.12. Цикл *do...while*

Выполнение инструкций в цикле `do...while` продолжается до тех пор, пока логическое выражение истинно. В отличие от цикла `while` условие проверяется не в начале цикла, а в конце. По этой причине инструкции внутри цикла `do...while` выполнятся минимум один раз. Цикл имеет следующий формат:

```
<Начальное значение>
do {
   <Инструкции>
   <Приращение>
} while (<Условие>);
```

Последовательность работы цикла `do...while`:

1. Переменной-счетчику присваивается начальное значение.
2. Выполняются инструкции внутри цикла.
3. Переменная-счетчик изменяется на величину, указанную в `<Приращение>`.

4. Проверяется условие: если оно истинно, происходит переход к пункту 2, а если нет — выполнение цикла завершается.

Выведем все числа от 1 до 100, используя цикл do...while:

```
int i = 1;                  // <Начальное значение>
do {
   printf("%d\n", i);       // <Инструкции>
   ++i;                     // <Приращение>
} while (i <= 100);         // <Условие>
```

> ***Внимание!***
>
> Если <Приращение> не указано, то цикл будет бесконечным.

4.13. Оператор *continue*: переход на следующую итерацию цикла

Оператор continue позволяет перейти к следующей итерации цикла до завершения выполнения всех инструкций внутри цикла. В качестве примера выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно:

```
for (int i = 1; i <= 100; ++i) {
   if (i > 4 && i < 11) continue;
   printf("%d\n", i);
}
```

4.14. Оператор *break*: прерывание цикла

Оператор break позволяет прервать выполнение цикла досрочно. Для примера выведем все числа от 1 до 100 еще одним способом:

```
int i = 1;
while (1) {
   if (i > 100) break;
   printf("%d\n", i);
   ++i;
}
```

Здесь мы в условии указали значение 1. В этом случае инструкции внутри цикла будут выполняться бесконечно. Однако использование оператора break прерывает его выполнение, как только 100 строк уже напечатано.

> ***Обратите внимание***
>
> Оператор break прерывает выполнение цикла, а не программы, т. е. далее будет выполнена инструкция, следующая сразу за циклом.

Бесконечный цикл совместно с оператором break удобно использовать для получения неопределенного заранее количества данных от пользователя. В качестве примера просуммируем неопределенное количество целых чисел (листинг 4.7).

Листинг 4.7. Суммирование неопределенного количества чисел

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int x = 0, sum = 0, status = 0;
   puts("Введите число 0 для получения результата");
   for ( ; ; ) {
      printf("Введите число: ");
      fflush(stdout);
      fflush(stdin);
      status = scanf("%d", &x);
      if (status != 1) {
         puts("Вы ввели не число");
         // Сбрасываем флаг ошибки и очищаем буфер
         if (feof(stdin) || ferror(stdin)) clearerr(stdin);
         int ch = 0;
         while ((ch = getchar()) != '\n' && ch != EOF);
         continue;
      }
      if (!x) break;
      sum += x;
   }
   printf("Сумма чисел равна: %d\n", sum);
   return 0;
}
```

4.15. Оператор *goto*

С помощью оператора безусловного перехода `goto` можно передать управление в любое место программы. Оператор имеет следующий формат:

```
goto <Метка>;
```

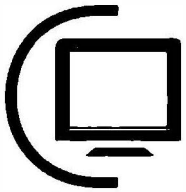
Значение в параметре `<Метка>` должно быть допустимым идентификатором. Место в программе, в которое передается управление, помечается одноименной меткой, после которой указывается двоеточие. В качестве примера имитируем цикл и выведем числа от `1` до `100`:

```
int i = 1;
BLOCK_START: {
   if (i > 100) goto BLOCK_END;
   printf("%d\n", i);
   ++i;
   goto BLOCK_START;
}
BLOCK_END:;
```

Как видно из примера, фигурные скобки можно использовать не только применительно к условным операторам, циклам, функциям, но и как отдельную конструкцию. Фрагмент кода, заключенный в фигурные скобки, называется *блоком*. Переменные, объявленные внутри блока, видны только в пределах блока.

Совет

Следует избегать использования оператора `goto`, т. к. его применение делает программу слишком запутанной и может привести к неожиданным результатам.



ГЛАВА 5

Числа

В языке C практически все элементарные типы данных (_Bool, char, int, float и double) являются числовыми. Тип _Bool может содержать только значения 1 и 0, которые соответствуют значениям Истина и Ложь. Тип char содержит код символа, а не сам символ. Поэтому значения этих типов можно использовать в одном выражении вместе со значениями, имеющими типы int, float и double. Пример:

```
_Bool a = 1;
char ch = 'w';              // 119
printf("%d\n", a + ch + 10); // 130
```

Для хранения целых чисел предназначен тип int. Диапазон значений зависит от компилятора. Минимальный диапазон — от -32 768 до 32 767. На практике — диапазон от -2 147 483 648 до 2 147 483 647 (занимает 4 байта).

С помощью ключевых слов short, long и long long можно указать точный размер типа int. При использовании этих ключевых слов тип int подразумевается по умолчанию, поэтому его можно не указывать. Тип short занимает 2 байта (диапазон от -32 768 до 32 767), тип long — 4 байта (диапазон от -2 147 483 648 до 2 147 483 647), а тип long long занимает 8 байт (диапазон значений от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807).

По умолчанию целочисленные типы являются знаковыми (signed). Знак числа хранится в старшем бите: 0 соответствует положительному числу, а 1 — отрицательному. С помощью ключевого слова unsigned можно указать, что число является только положительным. Тип unsigned char может содержать числа в диапазоне от 0 до 255, тип unsigned short — от 0 до 65 535, тип unsigned long — от 0 до 4 294 967 295, а тип unsigned long long — от 0 до 18 446 744 073 709 551 615.

При преобразовании значения из типа signed в тип unsigned следует учитывать, что знаковый бит (если число отрицательное, то бит содержит значение 1) может стать причиной очень больших чисел, т. к. старший бит у типа unsigned не содержит признака знака:

```
int x = -1;
printf("%u\n", (unsigned int)x); // 4294967295
```

Целочисленное значение задается в *десятичной*, *восьмеричной* или *шестнадцатеричной* форме. Восьмеричные числа начинаются с нуля и содержат цифры от `0` до `7`. Шестнадцатеричные числа начинаются с комбинации символов `0x` (или `0X`) и могут содержать числа от `0` до `9` и буквы от `A` до `F` (регистр букв не имеет значения). Восьмеричные и шестнадцатеричные значения преобразуются в десятичное значение. Пример:

```
int x, y, z;
x = 119;  // Десятичное значение
y = 0167; // Восьмеричное значение
z = 0xFF; // Шестнадцатеричное значение
printf("%d\n", x);                    // 119
printf("%d\n", y);                    // 119
printf("%d\n", z);                    // 255
printf("%#o\n", y);                   // 0167
printf("%#X\n", z);                   // 0XFF
```

После целочисленной константы могут быть указаны следующие суффиксы:

- без суффикса — в зависимости от значения может иметь тип `int`, `long` или `long long` (компилятор выбирает минимальный тип из указанных, позволяющий хранить значение):

```
printf("%d\n", (int) sizeof(1));                    // 4
printf("%d\n", (int) sizeof(2147483647));           // 4
printf("%d\n", (int) sizeof(2147483648));           // 8
printf("%d\n", (int) sizeof(9223372036854775807)); // 8
// warning: integer constant is so large that it is unsigned
// printf("%d\n", (int) sizeof(9223372036854775808));
```

- `L` (или `l`) — в зависимости от значения может иметь тип `long` или `long long` (компилятор выбирает минимальный тип из указанных, позволяющий хранить значение). Пример указания значения: `10L`;
- `LL` (или `ll`) — тип `long long`. Пример указания значения: `10LL`;
- `U` (или `u`) — в зависимости от значения может иметь тип `unsigned int`, `unsigned long` или `unsigned long long` (компилятор выбирает минимальный тип из указанных, позволяющий хранить значение). Пример указания значения: `10U`;
- `UL` — в зависимости от значения может иметь тип `unsigned long` или `unsigned long long`. Пример указания значения: `10UL`;
- `ULL` — тип `unsigned long long`. Пример указания значения: `10ULL`.

Пример объявления переменных с указанием суффиксов:

```
int i = 10;
unsigned int ui = 4294967295U;
printf("%d\n", i);                          // 10
printf("%u\n", ui);                         // 4294967295
printf("%d\n", (int) sizeof(10));           // 4
printf("%d\n", (int) sizeof(4294967295U)); // 4
```

```
printf("%d\n", (int) sizeof(4294967295));   // 8
long l = 2147483647L;
unsigned long ul = 4294967295UL;
printf("%ld\n", l);                         // 2147483647
printf("%lu\n", ul);                        // 4294967295
printf("%d\n", (int) sizeof(2147483647L));  // 4
printf("%d\n", (int) sizeof(4294967295UL)); // 4
printf("%d\n", (int) sizeof(4294967295L));  // 8
long long ll = 9223372036854775807LL;
unsigned long long ull = 18446744073709551615ULL;
printf("%I64d\n", ll);   // 9223372036854775807
printf("%I64u\n", ull);  // 18446744073709551615
```

Для хранения вещественных чисел предназначены типы `float` и `double`. Вещественное число может содержать точку и (или) экспоненту, начинающуюся с буквы `E` (регистр не имеет значения):

```
float x, y;
double z, k;
x = 20.0F;
y = 12.1e5F;
z = .123;
k = 47.E-5;
printf("%.5f\n", x);                // 20.00000
printf("%e\n", y);                  // 1.210000e+006
printf("%.2f\n", z);                // 0.12
printf("%g\n", k);                  // 0.00047
```

По умолчанию вещественные константы имеют тип `double`. Если необходимо изменить тип на другой, то после числа указываются следующие суффиксы:

- `F` (или `f`) — тип `float`. Пример указания значения: `12.3f`;
- `L` (или `l`) — тип `long double`. Значение должно содержать точку и (или) экспоненту, иначе тип будет `long int`. Пример указания значения: `12.3L`.

Пример указания суффиксов:

```
printf("%d\n", (int) sizeof(20.0F));  // 4
printf("%d\n", (int) sizeof(20.0));   // 8
printf("%d\n", (int) sizeof(20.0L));  // 16
```

При выполнении операций над вещественными числами следует учитывать ограничения точности вычислений. Например, результат следующей инструкции может показаться странным:

```
printf("%e", 0.3 - 0.1 - 0.1 - 0.1); // -2.775558e-017
```

Ожидаемым был бы результат `0.0`, но, как видно из примера, мы получили совсем другой результат (`-2.775558e-017`).

Если в выражении используются числа, имеющие разный тип данных, то тип результата выражения будет соответствовать наиболее сложному типу. Например,

если производится сложение переменной, имеющей тип int, с переменной, имеющей тип double, то целое число будет автоматически преобразовано в вещественное. Результатом этого выражения будет значение, имеющее тип double. Однако если результат выражения присваивается переменной типа int, то тип double будет преобразован в тип int (при этом компилятор выведет предупреждающее сообщение о возможной потере данных):

```
int x = 10, y = 0;
double z = 12.5;
y = x + z;
// warning: conversion from 'double' to 'int' may change value
printf("%d\n", y);        // 22 (тип int)
printf("%.2f\n", x + z); // 22.50 (тип double)
```

Для того чтобы компилятор не выводил предупреждающего сообщения, следует выполнить приведение типов:

```
y = (int)(x + z);
```

5.1. Математические константы

В заголовочном файле math.h определены следующие макросы, содержащие значения математических констант:

- M_PI — число пи (π). Определение макроса:

  ```
  #define M_PI        3.14159265358979323846
  ```

- M_PI_2 — значение выражения pi/2. Определение макроса:

  ```
  #define M_PI_2      1.57079632679489661923
  ```

- M_PI_4 — значение выражения pi/4. Определение макроса:

  ```
  #define M_PI_4      0.78539816339744830962
  ```

- M_1_PI — значение выражения 1/pi. Определение макроса:

  ```
  #define M_1_PI      0.31830988618379067154
  ```

- M_2_PI — значение выражения 2/pi. Определение макроса:

  ```
  #define M_2_PI      0.63661977236758134308
  ```

- M_E — значение константы e. Определение макроса:

  ```
  #define M_E         2.7182818284590452354
  ```

- M_LOG2E — значение выражения log2(e). Определение макроса:

  ```
  #define M_LOG2E     1.4426950408889634074
  ```

- M_LOG10E — значение выражения log10(e). Определение макроса:

  ```
  #define M_LOG10E    0.43429448190325182765
  ```

- M_LN2 — значение выражения ln(2). Определение макроса:

  ```
  #define M_LN2       0.69314718055994530942
  ```

- `M_LN10` — **значение выражения** `ln(10)`. **Определение макроса:**

```
#define M_LN10      2.30258509299404568402
```

- `M_2_SQRTPI` — **значение выражения** `2/sqrt(pi)`. **Определение макроса:**

```
#define M_2_SQRTPI 1.12837916709551257390
```

- `M_SQRT2` — **значение выражения** `sqrt(2)`. **Определение макроса:**

```
#define M_SQRT2     1.41421356237309504880
```

- `M_SQRT1_2` — **значение выражения** `1/sqrt(2)`. **Определение макроса:**

```
#define M_SQRT1_2  0.70710678118654752440
```

Прежде чем использовать константы, необходимо перед подключением файла math.h определить макрос с названием `_USE_MATH_DEFINES`. В Visual C без этого макроса константы недоступны. В MinGW определять макрос не нужно. Пример вывода значения констант приведен в листинге 5.1.

Листинг 5.1. Вывод значений математических констант

```
#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>

int main(void) {
   printf("%.5f\n", M_PI); // 3.14159
   printf("%.5f\n", M_E);  // 2.71828
   return 0;
}
```

5.2. Основные функции для работы с числами

Перечислим основные функции для работы с числами.

- `abs()` — возвращает абсолютное значение. Прототипы функции:

```
#include <stdlib.h>
int abs(int x);
long labs(long x);
long long llabs(long long x);
__int64 _abs64(__int64 x);
#include <math.h>
int abs(int x);
long labs(long x);
float fabsf(float x);
double fabs(double x);
long double fabsl(long double x);
```

Пример:

```
printf("%d\n", abs(-1));          // 1
```

- `imaxabs()` — **возвращает абсолютное значение. Прототип функции:**

```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
typedef long long  intmax_t;
```

Пример:

```
printf("%I64d\n", imaxabs(-5));   // 5
```

- `pow()` — **возводит число** `x` **в степень** `y`**. Прототипы функции:**

```
#include <math.h>
float powf(float x, float y);
double pow(double x, double y);
long double powl(long double x, long double y);
```

Примеры:

```
printf("%.1f\n", pow(10.0, 2));   // 100.0
printf("%.1f\n", pow(3.0, 3.0));  // 27.0
```

- `sqrt()` — **квадратный корень. Прототипы функции:**

```
#include <math.h>
float sqrtf(float x);
double sqrt(double x);
long double sqrtl(long double x);
```

Примеры использования функции:

```
printf("%.1f\n", sqrt(100.0));    // 10.0
printf("%.1f\n", sqrt(25.0));     // 5.0
```

- `exp()` — **экспонента. Прототипы функции:**

```
#include <math.h>
float expf(float x);
double exp(double x);
long double expl(long double x);
```

- `log()` — **натуральный логарифм. Прототипы функции:**

```
#include <math.h>
float logf(float x);
double log(double x);
long double logl(long double x);
```

- `log10()` — **десятичный логарифм. Прототипы функции:**

```
#include <math.h>
float log10f(float x);
double log10(double x);
long double log10l(long double x);
```

- `fmod()` — остаток от деления. Прототипы функции:

```
#include <math.h>
float fmodf(float x, float y);
double fmod(double x, double y);
long double fmodl(long double x, long double y);
```

Примеры использования функции:

```
printf("%.1f\n", fmod(100.0, 9.0));   // 1.0
printf("%.1f\n", fmod(100.0, 10.0));  // 0.0
```

- `modf()` — разделяет вещественное число `x` на целую и дробную части. В качестве значения функция возвращает дробную часть. Целая часть сохраняется в переменной, адрес которой передан во втором параметре. Прототипы функции:

```
#include <math.h>
float modff(float x, float *py);
double modf(double x, double *py);
long double modfl(long double x, long double *py);
```

Пример использования функции:

```
double x = 0.0;
printf("%.1f\n", modf(12.5, &x));   // 0.5
printf("%.1f\n", x);                // 12.0
```

- `div()` — возвращает структуру из двух полей: `quot` (результат целочисленного деления `x / y`), `rem` (остаток от деления `x % y`). Прототипы функции:

```
#include <stdlib.h>
div_t div(int x, int y);
ldiv_t ldiv(long x, long y);
lldiv_t lldiv(long long x, long long y);
```

Объявления структур:

```
typedef struct _div_t {
   int quot;
   int rem;
} div_t;
typedef struct _ldiv_t {
   long quot;
   long rem;
} ldiv_t;
typedef struct {
   long long quot;
   long long rem;
} lldiv_t;
```

Пример использования функции:

```
div_t d;
d = div(13, 2);
```

```
printf("%d\n", d.quot);  // 6 == 13 / 2
printf("%d\n", d.rem);   // 1 == 13 % 2
```

- `imaxdiv()` — возвращает структуру из двух полей: `quot` (результат целочисленного деления `x / y`), `rem` (остаток от деления `x % y`). Прототип функции:

```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t x, intmax_t y);
typedef long long intmax_t;
```

Объявление структуры `imaxdiv_t`:

```
typedef struct {
   intmax_t quot;
   intmax_t rem;
} imaxdiv_t;
```

Пример:

```
imaxdiv_t d = imaxdiv(13, 2);
printf("%I64d\n", d.quot);   // 6 == 13 / 2
printf("%I64d\n", d.rem);    // 1 == 13 % 2
```

- `__max()` — максимальное значение:

```
#include <stdlib.h>
#define __max(a,b) (((a) > (b)) ? (a) : (b))
```

Пример:

```
printf("%d\n", __max(10, 3));   // 10
```

- `fmax()` — максимальное значение. Прототипы функции:

```
#include <math.h>
float fmaxf(float a, float b);
double fmax(double a, double b);
long double fmaxl(long double a, long double b);
```

Пример:

```
printf("%.0f\n", fmax(10, 3));  // 10
```

- `__min()` — минимальное значение:

```
#include <stdlib.h>
#define __min(a,b) (((a) < (b)) ? (a) : (b))
```

Пример:

```
printf("%d\n", __min(10, 3));   // 3
```

- `fmin()` — минимальное значение. Прототипы функции:

```
#include <math.h>
float fminf(float a, float b);
double fmin(double a, double b);
long double fminl(long double a, long double b);
```

Пример:

```
printf("%.0f\n", fmin(10, 3));  // 3
```

5.3. Округление чисел

Для округления чисел предназначены следующие функции.

- `ceil()` — возвращает значение, округленное до ближайшего большего значения. Прототипы функции:

```
#include <math.h>
float ceilf(float x);
double ceil(double x);
long double ceill(long double x);
```

Примеры использования функции:

```
printf("%.1f\n", ceil(1.49));   // 2.0
printf("%.1f\n", ceil(1.5));    // 2.0
printf("%.1f\n", ceil(1.51));   // 2.0
```

- `floor()` — значение, округленное до ближайшего меньшего значения. Прототипы функции:

```
#include <math.h>
float floorf(float x);
double floor(double x);
long double floorl(long double x);
```

Примеры использования функции:

```
printf("%.1f\n", floor(1.49));  // 1.0
printf("%.1f\n", floor(1.5));   // 1.0
printf("%.1f\n", floor(1.51));  // 1.0
```

- `round()` — возвращает число, округленное до ближайшего меньшего целого для чисел с дробной частью меньше `0.5`, или значение, округленное до ближайшего большего целого для чисел с дробной частью больше или равной `0.5`. Прототипы функции:

```
#include <math.h>
float roundf(float x);
double round(double x);
long double roundl(long double x);
```

Примеры использования функции:

```
printf("%.1f\n", round(1.49));  // 1.0
printf("%.1f\n", round(1.5));   // 2.0
printf("%.1f\n", round(1.51));  // 2.0
```

5.4. Тригонометрические функции

В языке C доступны следующие тригонометрические функции.

- `sin()`, `cos()`, `tan()` — **стандартные тригонометрические функции (синус, косинус, тангенс). Угол задается в радианах. Прототипы функций:**

```
#include <math.h>
float sinf(float x);
double sin(double x);
long double sinl(long double x);
float cosf(float x);
double cos(double x);
long double cosl(long double x);
float tanf(float x);
double tan(double x);
long double tanl(long double x);
```

Пример:

```
double degrees = 90.0;
// Перевод градусов в радианы
double radians = degrees * (M_PI / 180.0);
// Перевод радианов в градусы
printf("%.1f\n", radians * (180.0 / M_PI)); // 90.0
printf("%.1f\n", sin(radians));             // 1.0
```

- `asin()`, `acos()`, `atan()` — **обратные тригонометрические функции (арксинус, арккосинус, арктангенс). Значение возвращается в радианах. Прототипы функций:**

```
#include <math.h>
float asinf(float x);
double asin(double x);
long double asinl(long double x);
float acosf(float x);
double acos(double x);
long double acosl(long double x);
float atanf(float x);
double atan(double x);
long double atanl(long double x);
```

5.5. Преобразование строки в число

Для преобразования строки в число используются следующие функции.

- `atoi()` **и** `_atoi_l()` — **преобразуют C-строку в число, имеющее тип** `int`**. Прототипы функций:**

```
#include <stdlib.h>
int atoi(const char *str);
int _atoi_l(const char *str, _locale_t locale);
```

Считывание символов производится, пока они соответствуют цифрам. Иными словами, в строке могут содержаться не только цифры. Пробельные символы в начале строки игнорируются. Функция `_atoi_l()` позволяет дополнительно задать локаль. Примеры преобразования:

```
printf("%d\n", atoi("25"));         // 25
printf("%d\n", atoi("2.5"));        // 2
printf("%d\n", atoi("5str"));       // 5
printf("%d\n", atoi("5s10"));       // 5
printf("%d\n", atoi(" \t\n 25"));   // 25
printf("%d\n", atoi("-25"));        // -25
printf("%d\n", atoi("str"));        // 0
```

Если значение выходит за диапазон значений типа `int`, то в Visual C возвращается максимальное или минимальное значение типа `int` и переменной `errno` присваивается значение `ERANGE`:

```
printf("%d\n", atoi("2147483649"));  // 2147483647
printf("%d\n", errno);               // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   puts("Выход за диапазон значений");
}
```

В MinGW будет усечение значения:

```
printf("%d\n", atoi("2147483649"));  // -2147483647
printf("%d\n", errno);               // 0
```

- `atol()` и `_atol_l()` — преобразуют С-строку в число, имеющее тип `long`. Прототипы функций:

```
#include <stdlib.h>
long atol(const char *str);
long _atol_l(const char *str, _locale_t locale);
```

Функция `_atol_l()` позволяет дополнительно задать локаль. Примеры преобразования:

```
printf("%ld\n", atol("25"));        // 25
printf("%ld\n", atol(" \n -25"));   // -25
printf("%ld\n", atol("2.5"));       // 2
printf("%ld\n", atol("5str"));      // 5
printf("%ld\n", atol("5s10"));      // 5
printf("%ld\n", atol("str"));       // 0
```

Если значение выходит за диапазон значений типа `long`, то в Visual C возвращается максимальное или минимальное значение типа `long` и переменной `errno` присваивается значение `ERANGE`:

```
printf("%ld\n", atol("2147483649")); // 2147483647
printf("%d\n", errno);               // 34
```

```
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   puts("Выход за диапазон значений");
}
```

В MinGW будет усечение значения:

```
printf("%ld\n", atol("2147483649")); // -2147483647
printf("%d\n", errno);               // 0
```

- `atoll()` — **преобразует С-строку в число, имеющее тип** `long long`. **Прототип функции:**

```
#include <stdlib.h>
long long atoll(const char *str);
```

Пример преобразования:

```
printf("%I64d\n", atoll("9223372036854775807"));
// 9223372036854775807
```

Если значение выходит за диапазон значений типа `long long`, **то возвращается максимальное или минимальное значение типа** `long long` **и переменной** `errno` **присваивается значение** `ERANGE`:

```
printf("%I64d\n", atoll("9223372036854775809"));
// 9223372036854775807
printf("%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   puts("Выход за диапазон значений");
}
```

- `_atoi64()` — **преобразует С-строку в число, имеющее тип** `__int64`. **Прототип функции:**

```
#include <stdlib.h>
__int64 _atoi64(const char *str);
```

Пример преобразования:

```
printf("%I64d\n", _atoi64("9223372036854775807"));
// 9223372036854775807
```

Если значение выходит за диапазон значений типа `__int64`, **то возвращается максимальное или минимальное значение типа** `__int64` **и переменной** `errno` **присваивается значение** `ERANGE`:

```
printf("%I64d\n", _atoi64("9223372036854775809"));
// 9223372036854775807
printf("%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   puts("Выход за диапазон значений");
}
```

- `strtol()` и `strtoul()` — преобразуют С-строку в число, имеющее типы `long` и `unsigned long` соответственно. Функции `_strtol_l()` и `_strtoul_l()` позволяют дополнительно задать локаль. Прототипы функций:

```
#include <stdlib.h>
long strtol(const char *str, char **endPtr, int radix);
long _strtol_l(const char *str, char **endPtr, int radix,
               _locale_t locale);
unsigned long strtoul(const char *str, char **endPtr, int radix);
unsigned long _strtoul_l(const char *str, char **endPtr,
                         int radix, _locale_t locale);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью числа. Указатель на этот символ доступен через параметр `endPtr`, если он не равен `NULL`. В параметре `radix` можно указать систему счисления (число от `2` до `36`). Если в параметре `radix` задано число `0`, то система счисления определяется автоматически. Пример преобразования:

```
char *p = NULL;
printf("%ld\n", strtol("25", NULL, 0));     // 25
printf("%ld\n", strtol("025", NULL, 0));    // 21
printf("%ld\n", strtol("0x25", NULL, 0));   // 37
printf("%ld\n", strtol("111", NULL, 2));    // 7
printf("%ld\n", strtol("025", NULL, 8));    // 21
printf("%ld\n", strtol("0x25", NULL, 16)); // 37
printf("%ld\n", strtol("5s10", &p, 0));     // 5
printf("%s\n", p);                          // s10
```

Если в строке нет числа, то возвращается число `0`. Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному (`LONG_MIN` или `0`) или максимальному (`LONG_MAX` или `ULONG_MAX`) значению для типа, а переменной `errno` присваивается значение `ERANGE`. Пример:

```
printf("%ld\n", strtol("str", NULL, 0));        // 0
printf("%ld\n", strtol("2147483649", NULL, 0));
// 2147483647 (соответствует LONG_MAX)
printf("%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   puts("Выход за диапазон значений");
}
```

- `strtoll()` и `strtoull()` — преобразуют С-строку в число, имеющее типы `long long` и `unsigned long long` соответственно. Прототипы функций:

```
#include <stdlib.h>
long long strtoll(const char *str, char **endPtr, int radix);
unsigned long long strtoull(const char *str, char **endPtr, int radix);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью числа. Указатель на этот символ доступен через параметр `endPtr`, если он не равен `NULL`. В параметре `radix` можно указать систему счисления (число от 2 до 36). Если в параметре `radix` задано число 0, то система счисления определяется автоматически. Пример преобразования:

```
char *p = NULL;
printf("%I64d\n", strtoll("25", NULL, 0));     // 25
printf("%I64d\n", strtoll("025", NULL, 0));    // 21
printf("%I64d\n", strtoll("0x25", NULL, 0));   // 37
printf("%I64d\n", strtoll("111", NULL, 2));    // 7
printf("%I64d\n", strtoll("025", NULL, 8));    // 21
printf("%I64d\n", strtoll("0x25", NULL, 16)); // 37
printf("%I64d\n", strtoll("5s10", &p, 0));     // 5
printf("%s\n", p);                            // s10
```

Если в строке нет числа, то возвращается 0. Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному (`LLONG_MIN` или 0) или максимальному (`LLONG_MAX` или `ULLONG_MAX`) значению для типа, а переменной `errno` присваивается значение `ERANGE`. Пример:

```
printf("%I64d\n", strtoll("str", NULL, 0));    // 0
printf("%I64d\n", strtoll("9223372036854775809", NULL, 0));
// 9223372036854775807 (соответствует LLONG_MAX).
printf("%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   puts("Выход за диапазон значений");
}
```

- `_strtoi64()` и `_strtoui64()` — преобразуют С-строку в число, имеющее типы `__int64` и `unsigned __int64` соответственно. Прототипы функций:

```
#include <stdlib.h>
__int64 _strtoi64(const char *str, char **endPtr, int radix);
unsigned __int64 _strtoui64(const char *str, char **endPtr, int radix);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе не являющемся записью числа. Указатель на этот символ доступен через параметр `endPtr`, если он не равен `NULL`. В параметре `radix` можно указать систему счисления (число от 2 до 36). Если в параметре `radix` задано число 0, то система счисления определяется автоматически. Пример преобразования:

```
char *p = NULL;
printf("%I64d\n", _strtoi64("25", NULL, 0));    // 25
printf("%I64u\n", _strtoui64("25", NULL, 0));   // 25
printf("%I64d\n", _strtoi64("025", NULL, 0));   // 21
printf("%I64d\n", _strtoi64("0x25", NULL, 0));  // 37
```

```
printf("%I64d\n", _strtoi64("111", NULL, 2));   // 7
printf("%I64d\n", _strtoi64("025", NULL, 8));   // 21
printf("%I64d\n", _strtoi64("0x25", NULL, 16)); // 37
printf("%I64d\n", _strtoi64("5s10", &p, 0));    // 5
printf("%s\n", p);                              // s10
```

Если в строке нет числа, то возвращается число 0. Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному или максимальному значению для типа, а переменной errno присваивается значение ERANGE. Пример:

```
printf("%I64d\n", _strtoi64("str", NULL, 0));        // 0
printf("%I64d\n", _strtoi64("9223372036854775809", NULL, 0));
// 9223372036854775807
printf("%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   puts("Выход за диапазон значений");
}
```

- strtoimax() и strtoumax() — преобразуют С-строку в число, имеющее типы intmax_t и uintmax_t соответственно. Прототипы функций:

```
#include <inttypes.h>
intmax_t strtoimax(const char *str, char **endPtr, int radix);
typedef long long intmax_t;
uintmax_t strtoumax(const char *str, char **endPtr, int radix);
typedef unsigned long long uintmax_t;
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью числа. Указатель на этот символ доступен через параметр endPtr, если он не равен NULL. В параметре radix можно указать систему счисления (число от 2 до 36). Если в параметре radix задано число 0, то система счисления определяется автоматически. Пример преобразования:

```
char *p = NULL;
printf("%I64u\n", strtoumax("25", NULL, 0));    // 25
printf("%I64d\n", strtoimax("25", NULL, 0));    // 25
printf("%I64d\n", strtoimax("025", NULL, 0));   // 21
printf("%I64d\n", strtoimax("0x25", NULL, 0));  // 37
printf("%I64d\n", strtoimax("111", NULL, 2));   // 7
printf("%I64d\n", strtoimax("025", NULL, 8));   // 21
printf("%I64d\n", strtoimax("0x25", NULL, 16)); // 37
printf("%I64d\n", strtoimax("5s10", &p, 0));    // 5
printf("%s\n", p);                              // s10
```

Если в строке нет числа, то возвращается 0. Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному (INTMAX_MIN или 0) или максимальному (INTMAX_MAX или UINTMAX_MAX) значению для типа, а переменной errno присваивается значение ERANGE. Пример:

```
// #include <stdint.h>
printf("%I64d\n", INTMAX_MIN);   // -9223372036854775808
printf("%I64d\n", INTMAX_MAX);   // 9223372036854775807
printf("%I64u\n", UINTMAX_MAX);  // 18446744073709551615
printf("%I64d\n", strtoimax("str", NULL, 0));   // 0
printf("%I64d\n", strtoimax("9223372036854775809", NULL, 0));
// 9223372036854775807 (соответствует INTMAX_MAX)
printf("%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   puts("Выход за диапазон значений");
}
```

- `atof()` — **преобразует С-строку в вещественное число, имеющее тип** `double`. **Функция** `_atof_l()` **позволяет дополнительно задать локаль (от локали зависит десятичный разделитель вещественных чисел). Прототипы функций:**

```
#include <stdlib.h> /* или #include <math.h> */
double atof(const char *str);
double _atof_l(const char *string, _locale_t locale);
```

Примеры преобразования:

```
printf("%.2f\n", atof("25"));             // 25.00
printf("%.2f\n", atof("2.5"));            // 2.50
printf("%.2f\n", atof("5.1str"));         // 5.10
printf("%.2f\n", atof("5s10"));           // 5.00
printf("%.2f\n", atof("str"));            // 0.00
// #include <locale.h>
_locale_t locale = _create_locale(LC_NUMERIC, "dutch");
printf("%.2f\n", atof("2,5"));            // 2.00
printf("%.2f\n", _atof_l("2,5", locale)); // 2.50
_free_locale(locale);
```

- `strtod()` — **преобразует С-строку в вещественное число, имеющее тип** `double`. **Функция** `_strtod_l()` **позволяет дополнительно задать локаль. Прототипы функций:**

```
#include <stdlib.h>
double strtod(const char *str, char **endPtr);
double _strtod_l(const char *str, char **endPtr, _locale_t locale);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью вещественного числа. Указатель на этот символ доступен через параметр `endPtr`, **если он не равен** `NULL`. **Пример преобразования:**

```
char *p = NULL;
printf("%.2f\n", strtod(" \t\n 25", NULL)); // 25.00
printf("%.2f\n", strtod("2.5", NULL));      // 2.50
```

```
printf("%.2f\n", strtod("5.1str", NULL));    // 5.10
printf("%e\n", strtod("14.5e5s10", &p));     // 1.450000e+006
printf("%s", p);                             // s10
```

Обратите внимание: от настроек локали зависит десятичный разделитель вещественных чисел:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%.2f\n", strtod("2.5", NULL));       // 2,00
printf("%.2f\n", strtod("2,5", NULL));       // 2,50
```

Если в строке нет числа, то возвращается `0`. Если число выходит за диапазон значений для типа, то возвращается значение `-HUGE_VAL` или `HUGE_VAL`, а переменной `errno` присваивается значение `ERANGE`;

- `strtof()` — преобразует С-строку в вещественное число, имеющее тип `float`. Прототип функции:

```
#include <stdlib.h>
float strtof(const char *str, char **endPtr);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью вещественного числа. Указатель на этот символ доступен через параметр `endPtr`, если он не равен `NULL`. Пример преобразования (обратите внимание: от настроек локали зависит десятичный разделитель вещественных чисел):

```
setlocale(LC_ALL, "C");
char *p = NULL;
printf("%.2f\n", strtof(" \t\n 25", NULL)); // 25.00
printf("%.2f\n", strtof("2.5", NULL));      // 2.50
printf("%.2f\n", strtof("5.1str", NULL));   // 5.10
printf("%e\n", strtof("14.5e5s10", &p));    // 1.450000e+006
printf("%s\n", p);                          // s10
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%.2f\n", strtof("2.5", NULL));      // 2,00
printf("%.2f\n", strtof("2,5", NULL));      // 2,50
```

Если в строке нет числа, то возвращается `0`. Если число выходит за диапазон значений для типа, то возвращается значение `-HUGE_VALF` или `HUGE_VALF`, а переменной `errno` присваивается значение `ERANGE`;

- `strtold()` — преобразует С-строку в вещественное число, имеющее тип `long double`. Прототип функции:

```
#include <stdlib.h>
long double strtold(const char *str, char **endPtr);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью вещественного числа. Указатель на этот символ доступен через параметр `endPtr`, если он не равен `NULL`. Пример преобразования:

```
char *p = NULL;
__mingw_printf("%.2Lf\n", strtold(" \t\n 25", NULL)); // 25.00
__mingw_printf("%.2Lf\n", strtold("2.5", NULL));      // 2.50
__mingw_printf("%.2Lf\n", strtold("5.1str", NULL));   // 5.10
__mingw_printf("%Le\n", strtold("14.5e5s10", &p));    // 1.450000e+006
printf("%s", p);                                      // s10
```

Если в строке нет числа, то возвращается `0`. Если число выходит за диапазон значений для типа, то возвращается значение `-HUGE_VALL` или `HUGE_VALL`, а переменной `errno` присваивается значение `ERANGE`.

Все рассмотренные в этом разделе функции позволяют преобразовать всю С-строку в число. Если необходимо преобразовать отдельный символ (представляющий число) в соответствующее целое число от `0` до `9`, то можно воспользоваться следующим кодом:

```
char ch = '8';
int n = ch - '0';      // Эквивалентно int n = 56 - 48;
printf("%d\n", n);     // 8
```

В переменной типа `char` хранится код символа, а не сам символ. Символы от `'0'` до `'9'` в кодировке ASCII имеют коды от `48` до `57` соответственно. Следовательно, чтобы получить целое число от `0` до `9`, достаточно вычесть из текущего символа код символа `'0'`. В качестве еще одного примера сохраним все цифры, встречающиеся в С-строке, в целочисленном массиве (листинг 5.2).

Листинг 5.2. Преобразование символов С-строки в отдельные цифры

```
#include <stdio.h>

#define ARR_SIZE 10

int main(void) {
   char str[] = "abc0123456789";
   int arr[ARR_SIZE] = {0}, index = 0;
   for (char *p = str; *p; ++p) {        // Перебираем строку
      if (*p >= '0' && *p <= '9') {
         arr[index] = *p - '0';
         ++index;
         if (index >= ARR_SIZE) break;
      }
   }
   for (int i = 0; i < ARR_SIZE; ++i) { // Выводим значения
      printf("%d\n", arr[i]);
   }
   return 0;
}
```

Для преобразования строки в число можно также воспользоваться функциями `sscanf()` и `_snscanf()`. Прототипы функций:

```
#include <stdio.h>
int sscanf(const char *str, const char *format, ...);
int _snscanf(const char *str, size_t maxCount, const char *format, ...);
```

В параметре `str` указывается С-строка, содержащая число, в параметре `maxCount` — максимальное количество просматриваемых символов в строке, а в параметре `format` — строка специального формата, внутри которой задаются спецификаторы, аналогичные применяемым в функции `printf()` *(см. разд. 2.8)*, а также некоторые дополнительные спецификаторы. В последующих параметрах передаются адреса переменных. Функции возвращают количество произведенных присваиваний. Пример получения двух целых чисел:

```
// #include <locale.h>
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "20 30 str";
int n = 0, k = 0;
int count = sscanf(str, "%d%d", &n, &k);
// int count = _snscanf(str, strlen(str), "%d%d", &n, &k);
if (count == 2) {
   printf("n = %d\n", n); // n = 20
   printf("k = %d\n", k); // k = 30
}
else puts("Не удалось преобразовать");
```

Функция `sscanf()` не производит никакой проверки длины строки, что может привести к ошибке. Обязательно указывайте ширину при использовании спецификатора `%s` (например, `%255s`). Кроме того, следует учитывать, что функция не проверяет возможность выхода значения за диапазон значений для типа, что может привести к неправильному результату. Функция никак об этом вас не предупредит, поэтому лучше использовать функции, рассмотренные в начале этого раздела.

5.6. Преобразование числа в строку

Преобразовать целое число в С-строку позволяют следующие функции:

- `_itoa()` и `_itoa_s()` — преобразуют число типа `int` в С-строку. Прототипы функций:

  ```
  #include <stdlib.h>
  char *_itoa(int val, char *dest, int radix);
  errno_t _itoa_s(int val, char *dest, size_t size, int radix);
  ```

  В параметре `val` указывается число, в параметре `dest` — указатель на символьный массив, в который будет записан результат, в параметре `size` — максимальный размер символьного массива `dest`, а в параметре `radix` — система счисления (`2`, `8`, `10` или `16`). Пример использования функции `_itoa_s()`:

  ```
  char str[100] = {0};
  int x = 255;
  ```

```
_itoa_s(x, str, 100, 10);
printf("%s\n", str);                    // 255
```

Пример вывода двоичного, восьмеричного и шестнадцатеричного значений:

```
char str[100] = {0};

// Двоичное значение
printf("%s\n", _itoa(100, str, 2));  // 1100100
// Восьмеричное значение
printf("%s\n", _itoa(10, str, 8));   // 12
// Шестнадцатеричное значение
printf("%s\n", _itoa(255, str, 16)); // ff
```

- `_ltoa()` и `_ltoa_s()` — **преобразуют число типа** `long` **в С-строку. Прототипы функций:**

```
#include <stdlib.h>
char *_ltoa(long val, char *dest, int radix);
errno_t _ltoa_s(long val, char *dest, size_t size, int radix);
```

Пример:

```
char str[100] = {0};
long x = 255;
_ltoa_s(x, str, 100, 10);
printf("%s\n", str);                    // 255
printf("%s\n", _ltoa(x, str, 16));   // ff
```

- `_ultoa()` и `_ultoa_s()` — **преобразуют число типа** `unsigned long` **в С-строку. Прототипы функций:**

```
#include <stdlib.h>
char *_ultoa(unsigned long val, char *dest, int radix);
errno_t _ultoa_s(unsigned long val, char *dest, size_t size, int radix);
```

Пример:

```
char str[100] = {0};
unsigned long x = 255;
_ultoa_s(x, str, 100, 10);
printf("%s\n", str);                    // 255
printf("%s\n", _ultoa(x, str, 16));  // ff
```

- `_i64toa()` и `_i64toa_s()` — **преобразуют число типа** `__int64` **в С-строку. Прототипы функций:**

```
#include <stdlib.h>
char *_i64toa(__int64 val, char *dest, int radix);
errno_t _i64toa_s(__int64 val, char *dest, size_t size, int radix);
```

Пример:

```
char str[100] = {0};
__int64 x = 255;
_i64toa_s(x, str, 100, 10);
```

```
printf("%s\n", str);                  // 255
printf("%s\n", _i64toa(x, str, 16)); // ff
```

- `_ui64toa()` и `_ui64toa_s()` — преобразуют число типа `unsigned __int64` в С-строку. Прототипы функций:

```
#include <stdlib.h>
char *_ui64toa(unsigned __int64 val, char *dest, int radix);
errno_t _ui64toa_s(unsigned __int64 val, char *dest, size_t size,
                   int radix);
```

Пример:

```
char str[100] = {0};
unsigned __int64 x = 255;
_ui64toa_s(x, str, 100, 10);
printf("%s\n", str);                   // 255
printf("%s\n", _ui64toa(x, str, 16)); // ff
```

Преобразовать значения элементарных типов в С-строку можно с помощью функции `sprintf()`. Существует также функция `_sprintf_l()`, которая позволяет дополнительно задать локаль. Прототипы функций:

```
#include <stdio.h>
int sprintf(char *buf, const char *format, ...);
int _sprintf_l(char *buf, const char *format, _locale_t locale, ...);
```

В параметре `format` указывается строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа `%`. Спецификаторы формата совпадают со спецификаторами, используемыми в функции `printf()` *(см. разд. 2.8)*. Вместо спецификаторов формата подставляются значения, указанные в качестве параметров. Количество спецификаторов должно совпадать с количеством переданных параметров. Результат записывается в буфер, адрес которого передается в первом параметре (`buf`). В качестве значения функция возвращает количество символов, записанных в символьный массив. Пример преобразования целого числа в С-строку:

```
char buf[50] = "";
int x = 100, count = 0;
count = sprintf(buf, "%d", x);
printf("%s\n", buf);                  // 100
printf("%d\n", count);                // 3
```

Функция `sprintf()` не производит никакой проверки размера буфера, поэтому возможно переполнение буфера. Вместо функции `sprintf()` следует использовать функцию `sprintf_s()` или `_sprintf_s_l()`. Прототипы функций:

```
#include <stdio.h>
int sprintf_s(char *buf, size_t sizeInBytes, const char *format, ...);
int _sprintf_s_l(char *buf, size_t sizeInBytes, const char *format,
                 _locale_t locale, ...);
```

Параметры `buf` и `format` аналогичны параметрам функции `sprintf()`. В параметре `sizeInBytes` указывается размер буфера. В качестве значения функции возвращают количество символов, записанных в символьный массив. Пример преобразования вещественного числа в С-строку:

```
char buf[50] = "";
int count = 0;
double pi = 3.14159265359;
count = sprintf_s(buf, 50, "%.2f", pi);
printf("%s\n", buf);                // 3.14
printf("%d\n", count);              // 4
```

5.7. Генерация псевдослучайных чисел

Для генерации псевдослучайных чисел используются следующие функции.

- `rand()` — генерирует псевдослучайное число от `0` до `RAND_MAX`. Прототип функции и определение макроса `RAND_MAX`:

  ```
  #include <stdlib.h>
  int rand(void);
  #define RAND_MAX 0x7fff
  ```

  Примеры:

  ```
  printf("%d\n", rand());   // 41
  printf("%d\n", rand());   // 18467
  printf("%d\n", RAND_MAX); // 32767
  ```

  Для того чтобы получить случайное число от `0` до определенного значения, а не до `RAND_MAX`, следует использовать оператор `%` для получения остатка от деления. Пример получения числа от `0` до `10` включительно:

  ```
  printf("%d\n", rand() % 11);
  ```

  Для того чтобы получить случайное число от `rmin` до определенного значения `rmax` (исключая это значение), например от `5` до `10` включительно, можно воспользоваться следующим кодом:

  ```
  int rmin = 5, rmax = 11;
  printf("%d\n", rmin + rand() % (rmax - rmin));
  ```

- `srand()` — настраивает генератор случайных чисел на новую последовательность. В качестве параметра обычно используется значение, возвращаемое функцией `time()` с нулевым указателем в качестве параметра. Функция `time()` возвращает количество секунд, прошедшее с 1 января 1970 г. Прототипы функций:

  ```
  #include <stdlib.h>
  void srand(unsigned int seed);
  #include <time.h>
  time_t time(time_t *time);
  ```

Пример:

```
srand( (unsigned int)time(NULL) );
printf("%d\n", rand());
printf("%d\n", rand());
printf("%d\n", rand());
```

Если функция `srand()` вызвана с одним и тем же параметром, то будет генерироваться одна и та же последовательность псевдослучайных чисел:

```
srand(100);
printf("%d\n", rand()); // 365
srand(100);
printf("%d\n", rand()); // 365
```

В качестве примера создадим генератор паролей произвольной длины (листинг 5.3). Для этого добавляем в массив `arr` все разрешенные символы, а далее в цикле получаем случайный элемент из массива. Затем записываем символ, который содержит элемент массива, в итоговый символьный массив, адрес первого элемента которого передан в качестве первого параметра. В конец символьного массива вставляем нулевой символ. Следует учитывать, что символьный массив должен быть минимум на один символ больше, чем количество символов в пароле.

Листинг 5.3. Генератор паролей

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

void passw_generator(char *pstr, int count_char);

int main(void) {
   char str[80] = "";
   srand( (unsigned int)time(NULL) );
   passw_generator(str, 8);
   printf("%s\n", str);      // Выведет примерно JMhFAE29
   passw_generator(str, 8);
   printf("%s\n", str);      // 9tpgmUts
   passw_generator(str, 10);
   printf("%s\n", str);      // MYGeIXXx2g
   return 0;
}
void passw_generator(char *pstr, int count_char) {
   const short SIZE = 60;
   char arr[60] = {'a','b','c','d','e','f','g','h','i','j',
    'k','l','m','n','p','q','r','s','t','u','v','w','x','y','z',
    'A','B','C','D','E','F','G','H','I','J','K','L',
    'M','N','P','Q','R','S','T','U','V','W',
    'X','Y','Z','1','2','3','4','5','6','7','8','9','0'};
```

```
    for (int i = 0; i < count_char; ++i) {
        *pstr = arr[rand() % SIZE];
        ++pstr;
    }
    *pstr = '\0';
}
```

5.8. Бесконечность и значение *NAN*

Целочисленное деление на `0` приведет к неопределенности, а вот с вещественными числами все обстоит несколько иначе. Деление вещественного числа на `0` приведет к значению плюс или минус `INFINITY` (бесконечность), а деление вещественного числа `0.0` на `0.0` — к неопределенности или значению `NAN` (нет числа):

```
printf("%f\n", 10.0 / 0.0);          // 1.#INF00
printf("%f\n", -10.0 / 0.0);         // -1.#INF00
printf("%f\n", 0.0 / 0.0);           // -1.#IND00
```

В заголовочном файле math.h для этих значений существуют константы `INFINITY` и `NAN`:

```
// #include <math.h>
printf("%f\n", INFINITY);            // 1.#INF00
printf("%f\n", -INFINITY);           // -1.#INF00
printf("%f\n", NAN);                 // 1.#QNAN0
printf("%f\n", NAN + 1);             // 1.#QNAN0
printf("%f\n", 0 * INFINITY);        // 1.#QNAN0
printf("%f\n", INFINITY / INFINITY); // 1.#QNAN0
printf("%f\n", -INFINITY + INFINITY); // 1.#QNAN0
```

Для проверки соответствия этим значениям нельзя использовать логические операторы. Например, значение `NAN` не равно даже самому себе:

```
printf("%d\n", NAN == NAN);          // 0
```

Для того чтобы проверить значения, нужно воспользоваться следующими функциями:

- `_finite()` и `_finitef()` — возвращают ненулевое число, если значение не равно плюс или минус бесконечность или значению `NAN`, и `0` — в противном случае. Прототипы функций:

  ```
  #include <math.h>
  int _finitef(float x);  // Только в версии x64 (MinGW-W64)
  int _finite(double x);
  ```

  Примеры:

  ```
  printf("%d\n", _finite(10.5));       // 1
  printf("%d\n", _finite(NAN));        // 0
  printf("%d\n", _finite(INFINITY));   // 0
  ```

- `_isnan()`, `_isnanf()` и макрос `isnan()` — возвращают ненулевое число, если значение равно `NAN`, и `0` — в противном случае. Прототипы функций:

```
#include <math.h>
int _isnanf(float x);  // Только в версии x64 (MinGW-W64)
int _isnan(double x);
```

Примеры:

```
printf("%d\n", isnan(10.5));          // 0
printf("%d\n", isnan(INFINITY));      // 0
printf("%d\n", isnan(NAN));           // 1
printf("%d\n", _isnan(10.5));         // 0
printf("%d\n", _isnan(INFINITY));     // 0
printf("%d\n", _isnan(NAN));          // 1
```

- `isinf()` — возвращает ненулевое число, если значение равно плюс или минус бесконечность, и `0` — в противном случае. Прототип функции:

```
#include <math.h>
#define isinf(x) (fpclassify(x) == FP_INFINITE)
```

Примеры:

```
printf("%d\n", isinf(10.5));          // 0
printf("%d\n", isinf(NAN));           // 0
printf("%d\n", isinf(INFINITY));      // 1
printf("%d\n", isinf(-INFINITY));     // 1
```

ГЛАВА 6

Массивы

Массив — это нумерованный набор переменных одного типа. Переменная в массиве называется *элементом*, а ее позиция в массиве задается *индексом*.

Все элементы массива располагаются в смежных ячейках памяти. Например, если объявлен массив из трех элементов, имеющих тип `int` (занимает 4 байта), то адреса соседних элементов будут отличаться на 4 байта:

```
int arr[3] = {10, 20, 30};
for (int i = 0; i < 3; ++i) {
   printf("%p\n", &arr[i]);
}
```

Результат в проекте `Test64c`:

```
000000000023FE40
000000000023FE44
000000000023FE48
```

6.1. Объявление и инициализация массива

Объявление массива выглядит следующим образом:

```
<Тип> <Переменная>[<Количество элементов>];
```

Пример объявления массива из трех элементов, имеющих тип `long int`:

```
long arr[3];
```

При объявлении элементам массива можно присвоить начальные значения. Для этого после объявления указывается оператор `=`, а далее значения через запятую внутри фигурных скобок. После закрывающей фигурной скобки обязательно указывается точка с запятой. Пример инициализации массива:

```
long arr[3] = {10, 20, 30};
```

Количество значений внутри фигурных скобок может быть меньше количества элементов массива. В этом случае значения присваиваются соответствующим элементам с начала массива. Пример:

```
long arr[3] = {10, 20};
for (int i = 0; i < 3; ++i) {
   printf("%ld ", arr[i]);
} // 10 20 0
```

В этом примере первому элементу массива присваивается значение 10, второму — значение 20, а третьему элементу будет присвоено значение 0.

Присвоить всем элементам массива значение 0 можно так:

```
long arr[3] = {0};
for (int i = 0; i < 3; ++i) {
   printf("%ld ", arr[i]);
} // 0 0 0
```

Стандарт C99 добавил возможность инициализации произвольных элементов массива. Для этого при инициализации массива внутри квадратных скобок указывается индекс элемента, а затем после оператора = — его значение. Можно присвоить значения нескольким элементам, указав индексы и значения через запятую. Элементы, которым не было присвоено значение при инициализации, автоматически получат значение 0. В этом примере первому элементу массива присваивается значение 5, второму — значение 0, а третьему элементу будет присвоено значение 11:

```
long arr[3] = {[0] = 5, [2] = 11};
for (int i = 0; i < 3; ++i) {
   printf("%ld ", arr[i]);
} // 5 0 11
```

Можно комбинировать инициализацию по позиции и по индексу:

```
long arr[8] = {5, [2] = 11, 15, [6] = 22, 55};
for (int i = 0; i < 8; ++i) {
   printf("%ld ", arr[i]);
} // 5 0 11 15 0 0 22 55
```

Если при объявлении массива указываются начальные значения, то количество элементов внутри квадратных скобок можно не указывать. Размер будет соответствовать количеству значений внутри фигурных скобок. Пример:

```
long arr[] = {10, 20, 30};
for (int i = 0; i < 3; ++i) {
   printf("%ld ", arr[i]);
} // 10 20 30
```

Обратите внимание: при инициализации массива значениями внутри фигурных скобок мы не можем внутри квадратных скобок указать константу, объявленную с помощью ключевого слова const. Этот код приведет к ошибке при компиляции:

```
const int ARR_SIZE = 3;
int arr[ARR_SIZE] = {1, 2, 3};
// error: variable-sized object may not be initialized
```

Однако мы можем использовать директиву препроцессора `#define`:

```
#define ARR_SIZE 3
int arr[ARR_SIZE] = {1, 2, 3}; // OK
for (int i = 0; i < ARR_SIZE; ++i) {
   printf("%d ", arr[i]);
} // 1 2 3
```

Количество элементов массива можно указать с помощью переменной или константы, объявленной с помощью ключевого слова `const`, но в этом случае нельзя выполнять инициализацию при объявлении массива:

```
const int ARR_SIZE = 2;
int arr[ARR_SIZE];
arr[0] = 1;
arr[1] = 2;
for (int i = 0; i < ARR_SIZE; ++i) {
   printf("%d ", arr[i]);
} // 1 2
```

Если при объявлении массива начальные значения не указаны, то:

- элементам глобальных массивов автоматически присваивается значение `0`;
- элементы локальных массивов будут содержать произвольные значения, так называемый «мусор».

6.2. Определение количества элементов и размера массива

Количество элементов массива задается при объявлении и не может быть изменено позже. Поэтому лучше количество элементов сохранить в каком-либо макросе и в дальнейшем указывать этот макрос, например, при переборе элементов массива. Для того чтобы получить количество элементов массива динамически при выполнении программы, нужно общий размер массива в байтах разделить на размер типа. Получить эти размеры можно с помощью оператора `sizeof`. Пример динамического определения количества элементов массива:

```
int arr[15] = {0};
printf("%d\n", (int) (sizeof(arr) / sizeof(int))); // 15
```

Можно также воспользоваться макросом `_countof()`. Определение макроса:

```
#include <stdlib.h>
#define _countof(_Array) (sizeof(_Array) / sizeof(_Array[0]))
```

Пример:

```
int arr[15] = {0};
printf("%d\n", (int)_countof(arr)); // 15
```

Если мы сохраним адрес первого элемента массива в указателе и попробуем определить количество элементов массива через него, то ничего не получится. Результатом будет размер указателя:

```
int arr[15] = {0};
int *p = arr;
printf("%d\n", (int) (sizeof(arr))); // 60
printf("%d\n", (int) (sizeof(p)));   // Размер указателя!!!
// Значение в проекте Test64c: 8
```

Объем памяти (в байтах), занимаемый массивом, определяется так:

```
<Объем памяти> = sizeof (<Массив>)
<Объем памяти> = sizeof (<Тип>) * <Количество элементов>
```

Пример:

```
int arr[3] = {10, 20, 30};
printf("%d\n", (int) sizeof(arr));     // 12
printf("%d\n", (int) sizeof(int) * 3); // 12
```

6.3. Получение и изменение значения элемента массива

Обращение к элементам массива осуществляется с помощью квадратных скобок, в которых указывается индекс элемента. Обратите внимание на то, что нумерация элементов массива начинается с `0`, а не с `1`, поэтому первый элемент имеет индекс `0`. С помощью индексов можно присвоить начальные значения элементам массива уже после объявления:

```
long arr[3];
arr[0] = 10; // Первый элемент имеет индекс 0!!!
arr[1] = 20; // Второй элемент
arr[2] = 30; // Третий элемент
```

Следует учитывать, что проверка выхода указанного индекса за пределы диапазона на этапе компиляции не производится. Таким образом, можно перезаписать значение в смежной ячейке памяти и тем самым нарушить работоспособность программы или даже повредить операционную систему. Помните, что контроль корректности индекса входит в обязанности программиста!

После определения массива выделяется необходимый размер памяти, а в переменной сохраняется адрес первого элемента массива. При указании индекса внутри квадратных скобок производится вычисление адреса соответствующего элемента массива. Зная адрес элемента массива, можно получить значение или перезаписать его. Иными словами, с элементами массива можно производить такие же операции, как и с обычными переменными. Пример:

```
long arr[3] = {10, 20, 30};
long x = 0;
x = arr[1] + 12;
```

```
arr[2] = x - arr[2];
printf("%ld ", x);       // 32
printf("%ld ", arr[2]); // 2
```

6.4. Перебор элементов массива

Для перебора элементов массива удобно использовать цикл `for`. В первом параметре переменной-счетчику присваивается значение `0` (элементы массива нумеруются с нуля), условием продолжения является значение переменной-счетчика меньше количества элементов массива. В третьем параметре указывается приращение на единицу на каждой итерации цикла. Внутри цикла доступ к элементу осуществляется с помощью квадратных скобок, внутри которых указывается переменная-счетчик. Пронумеруем все элементы массива, а затем выведем все значения в прямом и обратном порядке:

```
const short ARR_SIZE = 20;
int arr[ARR_SIZE];
// Нумеруем все элементы массива
for (int i = 0, j = 1; i < ARR_SIZE; ++i, ++j) {
   arr[i] = j;
}
// Выводим значения в прямом порядке
for (int i = 0; i < ARR_SIZE; ++i) {
   printf("%d\n", arr[i]);
}
puts("--------------------");
// Выводим значения в обратном порядке
for (int i = ARR_SIZE - 1; i >= 0; --i) {
   printf("%d\n", arr[i]);
}
```

В этом примере мы объявили количество элементов массива как постоянную величину (константа `ARR_SIZE`). Это очень удобно, т. к. размер массива приходится указывать при каждом переборе массива. Если количество элементов указывать в виде числа, то при изменении размера массива придется вручную изменять все значения. При использовании константы количество элементов достаточно будет изменить только в одном месте.

Цикл `for` всегда можно заменить циклом `while`. В качестве примера пронумеруем элементы в обратном порядке, а затем выведем все значения:

```
const short ARR_SIZE = 20;
int arr[ARR_SIZE];
// Нумеруем все элементы
int i = 0, j = ARR_SIZE;
while (i < ARR_SIZE) {
   arr[i] = j;
   ++i;
   --j;
}
```

```
// Выводим значения массива
i = 0;
while (i < ARR_SIZE) {
   printf("%d\n", arr[i]);
   ++i;
}
```

6.5. Доступ к элементам массива с помощью указателя

После определения массива в переменной сохраняется адрес первого элемента. Иными словами, название переменной является указателем, который ссылается на первый элемент массива. Поэтому доступ к элементу массива может осуществляться как по индексу, указанному внутри квадратных скобок, так и с использованием адресной арифметики. Например, следующие инструкции вывода являются эквивалентными:

```
int arr[3] = {10, 20, 30};
printf("%d\n", arr[1]);     // 20
printf("%d\n", *(arr + 1)); // 20
printf("%d\n", *(1 + arr)); // 20
printf("%d\n", 1[arr]);     // 20
```

Последняя инструкция может показаться странной. Однако если учесть, что выражение `1[arr]` воспринимается компилятором как `*(1 + arr)`, то все встанет на свои места.

При указании индекса внутри квадратных скобок каждый раз производится вычисление адреса соответствующего элемента массива. Для того чтобы сделать процесс доступа к элементу массива более эффективным, объявляют указатель и присваивают ему адрес первого элемента. Далее для доступа к элементу просто перемещают указатель на соответствующий элемент. Пример объявления указателя и присваивания ему адреса первого элемента массива:

```
int *p = NULL;
int arr[3] = {1, 2, 3};
p = arr;
printf("%d\n", *p);       // 1
printf("%d\n", *(p + 1)); // 2
printf("%d\n", *(p + 2)); // 3
```

Обратите внимание на то, что перед названием массива не указывается оператор `&`, т. к. название переменной содержит адрес первого элемента. Если использовать оператор `&`, то необходимо дополнительно указать индекс внутри квадратных скобок:

```
p = &arr[0]; // Эквивалентно: p = arr;
```

Таким же образом можно присвоить указателю адрес произвольного элемента массива, например третьего:

```
int *p = NULL;
int arr[3] = {1, 2, 3};
p = &arr[2];          // Указатель на третий элемент массива
printf("%d\n", *p);  // 3
```

Для того чтобы получить значение элемента, на который ссылается указатель, необходимо произвести операцию разыменования. Для этого перед названием указателя добавляется оператор `*`. Пример:

```
printf("%d\n", *p);
```

Указатели часто используются для перебора элементов массива. В качестве примера создадим массив из трех элементов, а затем выведем значения с помощью цикла `for`:

```
#define ARR_SIZE 3
int *p = NULL, arr[ARR_SIZE] = {1, 2, 3};
// Устанавливаем указатель на первый элемент массива
p = arr; // Оператор & не указывается!!!
for (int i = 0; i < ARR_SIZE; ++i) {
   printf("%d\n", *p);
   ++p; // Перемещаем указатель на следующий элемент
}
p = arr; // Восстанавливаем положение указателя
```

В первой строке объявляется константа `ARR_SIZE`, которая описывает количество элементов в массиве. Если массив используется часто, то лучше сохранить его размер как константу, т. к. количество элементов нужно будет указывать при каждом переборе массива. Если в каждом цикле указывать конкретное число, то при изменении размера массива придется вручную вносить изменения во всех циклах. При объявлении константы достаточно будет изменить ее значение один раз при инициализации.

В следующей строке объявляется указатель на тип `int` и массив. Количество элементов массива задается константой `ARR_SIZE`. При объявлении массив инициализируется начальными значениями. Обратите внимание: при инициализации массива значениями внутри фигурных скобок мы не можем внутри квадратных скобок указать константу, объявленную с помощью ключевого слова `const`. Этот код приведет к ошибке при компиляции:

```
const int ARR_SIZE = 3;
int arr[ARR_SIZE] = {1, 2, 3};
// error: variable-sized object may not be initialized
```

Поэтому для определения константы мы использовали директиву препроцессора `#define`, а не ключевое слово `const`.

Внутри цикла `for` выводится значение элемента, на который ссылается указатель, а затем значение указателя увеличивается на единицу (`++p;`). Обратите внимание на то, что изменяется адрес, а не значение элемента массива. При увеличении значения указателя используются правила адресной арифметики, а не правила обычной

арифметики. Увеличение значения указателя на единицу приведет к прибавлению к адресу размера типа данных. Например, если тип `int` занимает 4 байта, то при увеличении значения на единицу указатель вместо адреса `0x0012FF30` будет содержать адрес `0x0012FF34`. Значение увеличилось на `4`, а не на `1`. В нашем примере вместо двух инструкций внутри цикла можно использовать одну:

```
printf("%d\n", *p++);
```

Выражение `p++` возвращает текущий адрес, а затем увеличивает его на единицу. Символ `*` позволяет получить доступ к значению элемента по указанному адресу. Последовательность выполнения соответствует следующей расстановке скобок:

```
printf("%d\n", *(p++));
```

Если скобки расставить так:

```
printf("%d\n", (*p)++);
```

то вначале будет получен доступ к элементу массива и выведено его текущее значение, а затем произведено увеличение значения элемента массива. Перемещение указателя на следующий элемент не выполняется.

Указатель можно использовать в качестве переменной-счетчика в цикле `for`. В этом случае начальным значением будет адрес первого элемента массива, а условием продолжения — адрес меньше адреса первого элемента плюс количество элементов. Приращение осуществляется аналогично обычному, только вместо классической арифметики применяется адресная арифметика. Пример:

```
#define ARR_SIZE 3
int arr[ARR_SIZE] = {1, 2, 3};
for (int *p = arr, *p2 = arr + ARR_SIZE; p < p2; ++p) {
   printf("%d\n", *p);
}
```

Вывести все значения массива с помощью цикла `while` и указателя можно так:

```
#define ARR_SIZE 3
int *p = NULL, i = ARR_SIZE, arr[ARR_SIZE] = {1, 2, 3};
p = arr;
while (i-- > 0) {
   printf("%d\n", *p++);
}
p = arr;
```

В указателе можно сохранить адрес *составного литерала*, который удобно передавать в качестве параметра в функцию. Составной литерал состоит из круглых скобок, внутри которых указываются тип данных и количество элементов внутри квадратных скобок. После круглых скобок указываются фигурные скобки, внутри которых через запятую перечисляются значения:

```
const long *p = (long [3]) {1, 2, 3};
printf("%ld ", *p);        // 1
printf("%ld ", *(p + 1));  // 2
printf("%ld ", *(p + 2));  // 3
```

Количество элементов внутри квадратных скобок можно не указывать. Размер массива будет соответствовать количеству элементов внутри фигурных скобок:

```
const long *p = (long []) {1, 2, 3};
```

6.6. Массивы указателей

Указатели можно сохранять в массиве. При объявлении массива указателей используется следующий синтаксис:

```
<Тип> *<Название массива>[<Количество элементов>];
```

Пример использования массива указателей:

```
int *p[3]; // Массив указателей из трех элементов
int x = 10, y = 20, z = 30;
p[0] = &x;
p[1] = &y;
p[2] = &z;
printf("%d\n", *p[0]); // 10
printf("%d\n", *p[1]); // 20
printf("%d\n", *p[2]); // 30
```

6.7. Динамические массивы

При объявлении массива необходимо задать точное количество элементов. На основе этой информации при запуске программы автоматически выделяется необходимый объем памяти. Иными словами, размер массива необходимо знать до выполнения программы. Во время выполнения программы увеличить размер существующего массива нельзя.

Для того чтобы произвести увеличение массива во время выполнения программы, необходимо выделить достаточный объем динамической памяти с помощью функции `malloc()` или `calloc()`, перенести существующие элементы, а лишь затем добавить новые элементы. Для перераспределения динамической памяти можно воспользоваться функцией `realloc()`.

Управление динамической памятью полностью лежит на плечах программиста, поэтому после завершения работы с памятью необходимо самим возвратить память операционной системе с помощью функции `free()`. Если память не возвратить операционной системе, то участок памяти станет недоступным для дальнейшего использования. Подобные ситуации приводят к утечке памяти.

Функции `malloc()`, `calloc()`, `realloc()` и `free()` применительно к массивам мы уже рассматривали в *разд. 3.14*, поэтому не будем повторяться. Просто откройте этот раздел и прочитайте еще раз.

6.8. Многомерные массивы

Массивы в языке С могут быть *многомерными*. Объявление многомерного массива имеет следующий формат:

```
<Тип> <Переменная>[<Количество элементов1>]...[<Количество элементовN>];
```

На практике наиболее часто используются *двумерные массивы*, позволяющие хранить значения ячеек таблицы, содержащей определенное количество строк и столбцов. Объявление двумерного массива выглядит так:

```
<Тип> <Переменная>[<Количество строк>][<Количество столбцов>];
```

Пример объявления двумерного массива, содержащего две строки и четыре столбца:

```
int arr[2][4]; // Две строки из 4-х элементов каждая
```

Все элементы двумерного массива располагаются в памяти друг за другом. Вначале элементы первой строки, затем второй и т. д. При инициализации двумерного массива элементы указываются внутри фигурных скобок через запятую. Элементы каждой строки также размещаются внутри фигурных скобок. Пример:

```
int arr[2][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
for (int i = 0, j = 0; i < 2; ++i) {
   for (j = 0; j < 4; ++j) {
      printf("%3d ", arr[i][j]);
   }
   printf("\n");
}
```

Для того чтобы сделать процесс инициализации наглядным, мы расположили элементы на отдельных строках. Количество элементов на строке совпадает с количеством столбцов в массиве. Результат выполнения:

```
  1   2   3   4
  5   6   7   8
```

Если при объявлении производится инициализация, то количество строк можно не указывать, оно будет определено автоматически:

```
int arr[][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
```

Получить или задать значение элемента можно, указав два индекса (не забывайте, что нумерация начинается с нуля):

```
int arr[2][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
```

```
printf("%d\n", arr[0][0]); // 1
printf("%d\n", arr[1][0]); // 5
printf("%d\n", arr[1][3]); // 8
```

Для того чтобы вывести все значения массива, необходимо использовать вложенные циклы. В качестве примера пронумеруем все элементы массива, а затем выведем все значения:

```
const int ROWS = 4;     // Количество строк
const int COLUMNS = 5; // Количество столбцов
int i = 0, j = 0, n = 1, arr[ROWS][COLUMNS];
// Нумеруем все элементы массива
for (i = 0; i < ROWS; ++i) {
   for (j = 0; j < COLUMNS; ++j) {
      arr[i][j] = n;
      ++n;
   }
}
// Выводим значения
for (i = 0; i < ROWS; ++i) {
   for (j = 0; j < COLUMNS; ++j) {
      printf("%3d", arr[i][j]);
   }
   printf("\n");
}
```

Результат:

```
  1  2  3  4  5
  6  7  8  9 10
 11 12 13 14 15
 16 17 18 19 20
```

Все элементы многомерного массива располагаются в памяти друг за другом. Поэтому для перебора массива можно использовать указатель. В качестве примера выведем все элементы двумерного массива с помощью цикла `for` и указателя:

```
#define ROWS 4
#define COLUMNS 5
int arr[ROWS][COLUMNS] = {
   {1, 2, 3, 4, 5},
   {6, 7, 8, 9, 10},
   {11, 12, 13, 14, 15},
   {16, 17, 18, 19, 20}
};
for (int *p = arr[0], *p2 = arr[0]+ROWS*COLUMNS; p < p2; ++p) {
   printf("%d\n", *p);
}
```

Обратите внимание на инициализацию указателя:

```
int *p = arr[0]
```

Мы сохраняем в указателе адрес первого элемента массива. Эта инструкция эквивалентна такой инструкции:

```
int *p = &arr[0][0]
```

Следующая инструкция является неверной, т. к. `arr` содержит адрес массива, а не адрес первого элемента двумерного массива:

```
int *p = arr
```

Для того чтобы такая инициализация указателя стала корректной, нужно объявить указатель следующим образом:

```
int (*p)[COLUMNS] = arr;
printf("%d\n", p[0][0]);      // 1
printf("%d\n", p[3][4]);      // 20
```

Можно также объявить указатель на двумерный массив, в этом случае при инициализации перед названием двумерного массива нужно указать оператор `&`:

```
int (*p)[ROWS][COLUMNS] = &arr;
printf("%d\n", (*p)[0][0]);  // 1
printf("%d\n", (*p)[3][4]);  // 20
```

При динамическом создании двумерного массива вначале создается массив указателей (см. листинг 3.9):

```
int **p = calloc(ROWS, sizeof(int*));
```

Затем в этот массив добавляются строки:

```
for (i = 0; i < ROWS; ++i) {
   p[i] = calloc(COLUMNS, sizeof(int));
}
```

Сохранить значение в элементе массива можно так:

```
p[i][j] = n++;
```

или так:

```
*(*(p + i) + j) = n++;
```

Получить значение и вывести его можно так:

```
printf("%3d", p[i][j]);
```

или так:

```
printf("%3d", *(*(p + i) + j));
```

После окончания работы с динамическим двумерным массивом нужно не забыть освободить память:

```
for (int i = 0; i < ROWS; ++i) {
   free(p[i]);
}
free(p);
p = NULL;                              // Обнуляем указатель
```

6.9. Поиск минимального и максимального значений

Для того чтобы в массиве найти минимальное значение, следует присвоить переменной min значение первого элемента массива, а затем сравнить с остальными элементами. Если значение текущего элемента меньше значения переменной min, то присваиваем значение текущего элемента переменной min.

Для того чтобы найти максимальное значение, следует присвоить переменной max значение первого элемента массива, а затем сравнить с остальными элементами. Если значение текущего элемента больше значения переменной max, то присваиваем значение текущего элемента переменной max.

Пример поиска минимального и максимального значения приведен в листинге 6.1.

Листинг 6.1. Поиск минимального и максимального значений

```
#include <stdio.h>

#define ARR_SIZE 5

int min(int *pArr, int length);
int max(int *pArr, int length);
void min_max(int *pArr, int length, int *pMin, int *pMax);

int main(void) {
   int arr[ARR_SIZE] = {2, 5, 6, 1, 3};
   printf("min = %d\n", min(arr, ARR_SIZE));
   printf("max = %d\n", max(arr, ARR_SIZE));
   int _min = 0, _max = 0;
   min_max(arr, ARR_SIZE, &_min, &_max);
   printf("min = %d\n", _min);
   printf("max = %d\n", _max);
   _min = 0;
   min_max(arr, ARR_SIZE, &_min, NULL);
   printf("min = %d\n", _min);
   _max = 0;
   min_max(arr, ARR_SIZE, NULL, &_max);
   printf("max = %d\n", _max);
   return 0;
}
int min(int *pArr, int length) {
   int min = pArr[0];
   for (int i = 0; i < length; ++i) {
      if (pArr[i] < min) min = pArr[i];
   }
```

```
    return min;
}
int max(int *pArr, int length) {
    int max = pArr[0];
    for (int i = 0; i < length; ++i) {
        if (pArr[i] > max) max = pArr[i];
    }
    return max;
}
void min_max(int *pArr, int length, int *pMin, int *pMax) {
    if (!pArr || length < 1) return;
    int *p = pArr, min = pArr[0], max = pArr[0];
    for (int i = 0; i < length; ++i, ++p) {
        if (*p < min) min = *p;
        if (*p > max) max = *p;
    }
    if (pMin) *pMin = min;
    if (pMax) *pMax = max;
}
```

В этом примере мы создали три функции: `min()`, `max()` и `min_max()`. Функции в первом параметре принимают адрес первого элемента массива, а во втором — длину массива. Внутри функции `min()` выполняется поиск минимального значения, а внутри функции `max()` — максимального значения. Найденные значения функции возвращают с помощью оператора `return`. Как только внутри функции встречается оператор `return` или поток доходит до конца блока функции, управление передается в место вызова функции. При этом становится доступно значение, указанное в операторе `return`. Если функция ничего не возвращает, то при определении перед именем функции указывается ключевое слово `void`. В этом случае оператор `return` использовать не нужно. Хотя его можно и указать, например, чтобы досрочно выйти из функции:

```
if (!pArr || length < 1) return;
```

Если `pArr` является нулевым указателем или длина массива меньше `1`, то просто выходим из функции. При этом после оператора `return` значение не указывается.

Функция `min_max()` выполняет поиск и минимального значения, и максимального значения массива одновременно. Однако из функции с помощью оператора `return` мы можем вернуть только одно значение, поэтому вместо возврата значения в третьем и четвертом параметрах функция принимает адреса переменных, в которые будут записаны найденные значения. Для того чтобы передать в функцию адрес переменной, перед ее именем указывается оператор `&`:

```
int _min = 0, _max = 0;
min_max(arr, ARR_SIZE, &_min, &_max);
```

Внутри функции `min_max()` мы проверяем наличие адреса переменной, поэтому можем передать значение `NULL` вместо адреса, если какое-либо значение нам не нужно. В этом примере мы хотим получить только минимальное значение:

```
_min = 0;
min_max(arr, ARR_SIZE, &_min, NULL);
```

Внутри функций `min()` и `max()` мы используем доступ к элементу массива по индексу, указанному внутри квадратных скобок: `pArr[i]`. При этом каждый раз положение элемента вычисляется относительно начала массива: `*(pArr + i)`. Чем больше таких обращений, тем менее эффективна программа. Для того чтобы сделать программу более быстрой и эффективной, внутри функции `min_max()` на каждой итерации мы перемещаем указатель, используя адресную арифметику. В этом случае никаких дополнительных вычислений положения элемента внутри массива не производится.

6.10. Сортировка массива

Сортировка массива — это упорядочивание элементов по возрастанию или убыванию значений. Сортировка применяется при выводе значений, а также при подготовке массива к частому поиску значений. Поиск по отсортированному массиву производится гораздо быстрее, т. к. не нужно каждый раз просматривать все значения массива.

Для упорядочивания элементов массива в учебных целях очень часто применяется метод, называемый *пузырьковой сортировкой* (листинг 6.2). При этом методе наименьшее значение как бы «всплывает» в начало массива, а наибольшее значение «погружается» в конец массива. Сортировка выполняется в несколько проходов. При каждом проходе последовательно сравниваются значения двух элементов, которые расположены рядом. Если значение первого элемента больше второго, то значения элементов меняются местами. Для сортировки массива из пяти элементов необходимо максимум четыре прохода и десять сравнений. Если после прохода не было ни одной перестановки, то сортировку можно прервать. В этом случае для сортировки ранее уже отсортированного массива нужен всего один проход.

Листинг 6.2. Пузырьковая сортировка по возрастанию

```
#include <stdio.h>

#define ARR_SIZE 5

int main(void) {
   int arr[ARR_SIZE] = {10, 5, 6, 1, 3};
   int j = 0, tmp = 0, k = ARR_SIZE - 2;
   _Bool is_swap = 0;
   for (int i = 0; i <= k; ++i) {
      is_swap = 0;
      for (j = k; j >= i; --j) {
         if (arr[j] > arr[j + 1]) {
            tmp = arr[j + 1];
            arr[j + 1] = arr[j];
```

```
            arr[j] = tmp;
            is_swap = 1;
         }
      }
      // Если перестановок не было, то выходим
      if (!is_swap) break;
   }
   // Выводим отсортированный массив
   for (int i = 0; i < ARR_SIZE; ++i) {
      printf("%d\n", arr[i]);
   }
   return 0;
}
```

В качестве еще одного примера произведем сортировку по убыванию (листинг 6.3). Для того чтобы пример был более полезным, изменим направление проходов.

Листинг 6.3. Пузырьковая сортировка по убыванию

```
#include <stdio.h>

#define ARR_SIZE 5

int main(void) {
   int arr[ARR_SIZE] = {5, 6, 1, 10, 3};
   int j = 0, tmp = 0;
   _Bool is_swap = 0;
   for (int i = ARR_SIZE - 1; i >= 1; --i) {
      is_swap = 0;
      for (j = 0; j < i; ++j) {
         if (arr[j] < arr[j + 1]) {
            tmp = arr[j + 1];
            arr[j + 1] = arr[j];
            arr[j] = tmp;
            is_swap = 1;
         }
      }
      if (!is_swap) break;
   }
   // Выводим отсортированный массив
   for (int i = 0; i < ARR_SIZE; ++i) {
      printf("%d\n", arr[i]);
   }
   return 0;
}
```

Для сортировки массива лучше воспользоваться стандартной функцией `qsort()`. Прототип функции:

```
#include <stdlib.h>
void qsort(void *base, size_t numOfElements, size_t sizeOfElements,
           int (*PtFuncCompare)(const void *, const void *));
```

В параметре `base` **указывается адрес первого элемента массива, в параметре** `numOfElements` **— количество элементов массива, а в параметре** `sizeOfElements` **— размер каждого элемента в байтах. Адрес пользовательской функции (указывается название функции без круглых скобок и параметров), внутри которой производится сравнение двух элементов, передается в последнем параметре. Прототип пользовательской функции сравнения должен выглядеть так:**

```
int <Название функции>(const void *arg1, const void *arg2);
```

При сортировке по возрастанию функция должна возвращать отрицательное значение, если `arg1` **меньше** `arg2`**, положительное значение, если** `arg1` **больше** `arg2`**, или** `0`**, если значения равны. Внутри функции необходимо выполнить приведение указателя** `void *` **к определенному типу. Пример использования функции** `qsort()` **приведен в листинге 6.4.**

Листинг 6.4. Сортировка массива с помощью функции `qsort()`

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

#define ARR_SIZE 5

int compare(const void *arg1, const void *arg2) {
   if (*(int *)arg1 < *(int *)arg2) return -1;
   if (*(int *)arg1 > *(int *)arg2) return 1;
   return 0;
}
int main(void) {
   int arr[ARR_SIZE] = {10, 5, 6, 1, 3};
   qsort(arr, ARR_SIZE, sizeof(int), compare);
   for (int i = 0; i < ARR_SIZE; ++i) {
      printf("%d\n", arr[i]);
   }
   return 0;
}
```

Пример функции сравнения для сортировки по убыванию:

```
int compare(const void *arg1, const void *arg2) {
   if (*(int *)arg1 > *(int *)arg2) return -1;
   if (*(int *)arg1 < *(int *)arg2) return 1;
   return 0;
}
```

Можно также внутри функции сравнения вычесть одно значение из другого, но нужно учитывать, что при вычитании больших значений знак может поменяться на противоположный, что приведет к неправильной сортировке. Пример функции сравнения для сортировки по возрастанию:

```
int compare(const void *arg1, const void *arg2) {
   return *(int *)arg1 - *(int *)arg2;
}
```

Для того чтобы произвести сортировку по убыванию, достаточно поменять значения местами:

```
int compare(const void *arg1, const void *arg2) {
   return *(int *)arg2 - *(int *)arg1;
}
```

6.11. Проверка наличия значения в массиве

Если массив не отсортирован, то проверка наличия значения в массиве сводится к перебиранию всех элементов от начала до конца. При наличии первого вхождения можно прервать поиск. Пример поиска первого вхождения приведен в листинге 6.5.

Листинг 6.5. Поиск первого вхождения элемента

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

#define ARR_SIZE 5

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int arr[ARR_SIZE] = {10, 5, 6, 1, 6}, index = -1;
   int search_key = 6;
   for (int i = 0; i < ARR_SIZE; ++i) {
      if (arr[i] == search_key) {
         index = i;
         break;
      }
   }
   if (index != -1) {
      printf("index = %d\n", index);
   }
   else puts("Элемент не найден");
   return 0;
}
```

Если значение находится в начале массива, то поиск будет произведен достаточно быстро, но если значение расположено в конце массива, то придется просматривать

весь массив. Если массив состоит из 100 000 элементов, то нужно будет сделать 100 000 сравнений.

Поиск по отсортированному массиву производится гораздо быстрее, т. к. не нужно каждый раз просматривать все значения массива. Наиболее часто применяется *бинарный поиск* (листинг 6.6), при котором массив делится пополам и производится сравнение значения элемента, расположенного в середине массива, с искомым значением. Если искомое значение меньше значения элемента, то пополам делится первая половина массива, а если больше, то пополам делится вторая половина. Далее таким же образом производится деление оставшейся части массива. Поиск заканчивается, когда будет найдено первое совпадение или начальный индекс станет больше конечного индекса. Таким образом, на каждой итерации отбрасывается половина оставшихся элементов. Поиск значения, которое расположено в конце массива, состоящего из 100 000 элементов, будет произведен всего за 17 шагов. Однако если поиск производится один раз, то затраты на сортировку сведут на нет все преимущества бинарного поиска. В этом случае прямой перебор элементов может стать эффективнее.

Листинг 6.6. Бинарный поиск в отсортированном массиве

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

#define ARR_SIZE 5

int compare(const void *arg1, const void *arg2);
int search(int key, const int *arr, int count);

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int arr[ARR_SIZE] = {10, 5, 6, 1, 3};
   // Сортируем по возрастанию
   qsort(arr, ARR_SIZE, sizeof(int), compare);
   // Производим поиск
   int index = search(6, arr, ARR_SIZE);
   if (index != -1) {
      printf("index = %d\n", index);
   }
   else puts("Элемент не найден");
   return 0;
}
// Сортировка по возрастанию
int compare(const void *arg1, const void *arg2) {
   if (*(int *)arg1 < *(int *)arg2) return -1;
   if (*(int *)arg1 > *(int *)arg2) return 1;
   return 0;
}
```

```
// Бинарный поиск в отсортированном массиве
int search(int key, const int *arr, int count) {
   if (count < 1 || !arr) return -1;
   int start = 0, end = count - 1, i = 0;
   while (start <= end) {
      i = (start + end) / 2;
      if (arr[i] == key) return i;
      else if (arr[i] < key) start = i + 1;
      else if (arr[i] > key) end   = i - 1;
   }
   return -1;
}
```

Для проверки наличия элемента в массиве можно также воспользоваться стандартной функцией `bsearch()`. Прототип функции:

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t numOfElements, size_t sizeOfElements,
              int (*PtFuncCompare)(const void *, const void *));
```

В параметре `key` передается адрес переменной, в которой хранится искомое значение, в параметре `base` указывается адрес первого элемента массива, в параметре `numOfElements` — количество элементов массива, а в параметре `sizeOfElements` — размер каждого элемента в байтах. Адрес пользовательской функции (указывается название функции без круглых скобок и параметров), внутри которой производится сравнение двух элементов, передается в последнем параметре. Прототип пользовательской функции сравнения должен выглядеть так:

```
int <Название функции>(const void *arg1, const void *arg2);
```

Функция должна возвращать отрицательное значение, если `arg1` меньше `arg2`, положительное значение, если `arg1` больше `arg2`, или `0`, если значения равны. Внутри функции необходимо выполнить приведение указателя `void *` к определенному типу. Если значение не найдено, то возвращается нулевой указатель, в противном случае указатель ссылается на найденный элемент. Пример использования функции `bsearch()` приведен в листинге 6.7.

Листинг 6.7. Проверка наличия значения в массиве с помощью функции `bsearch()`

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

#define ARR_SIZE 5

int compare(const void *arg1, const void *arg2) {
   if (*(int *)arg1 < *(int *)arg2) return -1;
```

```
   if (*(int *)arg1 > *(int *)arg2) return 1;
   return 0;
}
int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int arr[ARR_SIZE] = {10, 5, 6, 1, 3}, search_key = 6;
   // Сортируем по возрастанию
   qsort(arr, ARR_SIZE, sizeof(int), compare);
   // Производим поиск
   int *p = (int *)bsearch(&search_key, arr, ARR_SIZE,
                           sizeof(int), compare);
   if (p) {
      int index = (int) (p - arr);
      printf("index = %d\n", index);
   }
   else puts("Элемент не найден");
   return 0;
}
```

6.12. Копирование элементов из одного массива в другой

Для копирования элементов из одного массива в другой используются следующие функции.

- `memcpy()` — **копирует первые** `size` **байтов из массива** `src` **в массив** `dst`. **В качестве значения функция возвращает указатель на массив** `dst`. **Если массив** `src` **длиннее массива** `dst`, **то произойдет переполнение буфера. Если указатели пересекаются, то поведение функции непредсказуемо. Прототип функции:**

```
#include <string.h>
void *memcpy(void *dst, const void *src, size_t size);
```

Пример использования функции:

```
#define ARR1_SIZE 5
#define ARR2_SIZE 3
int arr1[ARR1_SIZE] = {0}, arr2[ARR2_SIZE] = {1, 2, 3};
int *p = 0, i = 0;
// Копируем все элементы массива arr2
p = (int *)memcpy(arr1, arr2, sizeof arr2);
for (i = 0; i < ARR1_SIZE; ++i) {
   printf("%d ", arr1[i]);
} // 1 2 3 0 0
printf("\n");
if (!p) exit(1);
// Перемещаем указатель на четвертый элемент массива
p += 3;
```

```
// Копируем только два первых элемента массива arr2
memcpy(p, arr2, 2 * sizeof(int));
for (i = 0; i < ARR1_SIZE; ++i) {
   printf("%d ", arr1[i]);
} // 1 2 3 1 2
printf("\n");
```

Вместо функции `memcpy()` лучше использовать функцию `memcpy_s()`. Прототип функции:

```
#include <string.h>
errno_t memcpy_s(void *dst, size_t dstSize, const void *src,
                 size_t maxCount);
```

Функция `memcpy_s()` копирует первые `maxCount` байтов из массива `src` в массив `dst`. В параметре `dstSize` указывается размер массива `dst` в байтах. Если копирование прошло успешно, функция возвращает значение `0`. Пример:

```
#define ARR1_SIZE 5
#define ARR2_SIZE 3
int arr1[ARR1_SIZE] = {0}, arr2[ARR2_SIZE] = {1, 2, 3};
// Копируем только два первых элемента массива arr2
memcpy_s(arr1, sizeof arr1, arr2, 2 * sizeof(int));
for (int i = 0; i < ARR1_SIZE; ++i) {
   printf("%d ", arr1[i]);
} // 1 2 0 0 0
printf("\n");
```

- `memmove()` — копирует первые `size` байтов из массива `src` в массив `dst`. В качестве значения функция возвращает указатель на массив `dst`. Если массив `src` длиннее массива `dst`, то произойдет переполнение буфера. Основное отличие функции `memmove()` от `memcpy()` в выполнении корректных действий при пересечении указателей. Прототип функции:

```
#include <string.h>
void *memmove(void *dst, const void *src, size_t size);
```

Пример:

```
#define ARR_SIZE 5
int arr[ARR_SIZE] = {1, 2, 3, 4, 5}, *p = 0;
p = arr + 2;
// Указатели пересекаются
memmove(p, arr, 3 * sizeof(int));
for (int i = 0; i < ARR_SIZE; ++i) {
   printf("%d ", arr[i]);
} // 1 2 1 2 3 - копирование произведено корректно
printf("\n");
```

Вместо функции `memmove()` лучше использовать функцию `memmove_s()`. Прототип функции:

```
#include <string.h>
errno_t memmove_s(void *dst, size_t dstSize, const void *src,
                  size_t maxCount);
```

Функция `memmove_s()` копирует первые `maxCount` байтов из массива `src` в массив `dst`. В параметре `dstSize` указывается размер массива `dst` в байтах. Если копирование прошло успешно, функция возвращает значение `0`. Пример:

```
#define ARR_SIZE 5
int arr[ARR_SIZE] = {1, 2, 3, 4, 5}, *p = 0;
p = arr + 2;
memmove_s(p, sizeof arr, arr, 3 * sizeof(int));
for (int i = 0; i < ARR_SIZE; ++i) {
   printf("%d ", arr[i]);
} // 1 2 1 2 3
printf("\n");
```

6.13. Сравнение массивов

Для сравнения массивов предназначена функция `memcmp()`. Она сравнивает первые `size` байтов массивов `buf1` и `buf2`. В качестве значения функция возвращает:

- отрицательное число — если `buf1` меньше `buf2`;
- `0` — если массивы равны;
- положительное число — если `buf1` больше `buf2`.

Прототип функции:

```
#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t size);
```

Пример:

```
int arr1[3] = {1, 2, 3}, arr2[3] = {1, 2, 3}, x = 0;
x = memcmp(arr1, arr2, sizeof arr2);
printf("%d\n", x);                  // 0
arr1[2] = 2; // arr1[] = {1, 2, 2}, arr2[] = {1, 2, 3}
x = memcmp(arr1, arr2, sizeof arr2);
printf("%d\n", x);                  // -1
arr1[2] = 4; // arr1[] = {1, 2, 4}, arr2[] = {1, 2, 3}
x = memcmp(arr1, arr2, sizeof arr2);
printf("%d\n", x);                  // 1
```

Функция `memcmp()` производит сравнение с учетом регистра символов. Если необходимо произвести сравнение без учета символов, то можно воспользоваться функциями `_memicmp()` и `_memicmp_l()`. Для сравнения русских букв следует настроить локаль. Прототипы функций:

```
#include <string.h>
#include <locale.h>
```

```
int _memicmp(const void *buf1, const void *buf2, size_t size);
int _memicmp_l(const void *buf1, const void *buf2, size_t size,
               _locale_t locale);
```

Предназначение параметров и возвращаемое значение такое же, как у функции `memcmp()`. Функция `_memicmp_l()` позволяет дополнительно задать локаль. Пример использования функций:

```
setlocale(LC_ALL, "Russian_Russia.1251");  // Настройка локали
char str1[] = "абв", str2[] = "АБВ";
int x = 0;
x = _memicmp(str1, str2, sizeof str2);
printf("%d\n", x);                 // 0
x = memcmp(str1, str2, sizeof str2);
printf("%d\n", x);                 // 1
_locale_t locale = _create_locale(LC_ALL, "Russian_Russia.1251");
x = _memicmp_l(str1, str2, sizeof str2, locale);
printf("%d\n", x);                 // 0
_free_locale(locale);
```

6.14. Переворачивание массива

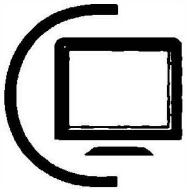
В языке C нет функции, позволяющей изменить порядок следования элементов массива, а в некоторых случаях это может пригодиться. Давайте потренируемся и напишем функцию `reverse()`, позволяющую перевернуть массив, состоящий из целых чисел (листинг 6.8).

Листинг 6.8. Переворачивание массива

```
#include <stdio.h>

#define ARR_SIZE 5

void reverse(int *pArr, int length);
void print_array(int *pArr, int length);

int main(void) {
   int arr[ARR_SIZE] = {1, 2, 3, 4, 5};
   print_array(arr, ARR_SIZE); // 1 2 3 4 5
   reverse(arr, ARR_SIZE);
   print_array(arr, ARR_SIZE); // 5 4 3 2 1
   reverse(arr, ARR_SIZE);
   print_array(arr, ARR_SIZE); // 1 2 3 4 5
   return 0;
}
void reverse(int *pArr, int length) {
   if (!pArr || length < 1) return;
   int tmp = 0;
```

```
    for (int i = 0, j = length - 1; i < j; ++i, --j) {
        tmp = pArr[i];
        pArr[i] = pArr[j];
        pArr[j] = tmp;
    }
}
void print_array(int *pArr, int length) {
    if (!pArr || length < 1) return;
    for (int i = 0; i < length; ++i) {
        printf("%d ", pArr[i]);
    }
    printf("\n");
}
```

Вначале переменной `i` присваиваем индекс первого элемента, а переменной `j` — индекс последнего элемента. На каждой итерации цикла будем увеличивать значение переменной `i` и уменьшать значение переменной `j`. Цикл будет выполняться, пока `i` меньше `j`. На каждой итерации цикла мы просто меняем местами значения двух элементов массива, предварительно сохраняя значение в промежуточной переменной `tmp`.



ГЛАВА 7

Символы и С-строки

В языке С доступны два типа строк: *С-строка* и *L-строка*. С-строка является массивом однобайтовых символов (тип `char`), последний элемент которого содержит нулевой символ (`'\0'`). Обратите внимание на то, что нулевой символ (нулевой байт) не имеет никакого отношения к символу `'0'`. Коды этих символов разные. L-строка является массивом широких символов (тип `wchar_t`), последний элемент которого содержит нулевой символ.

7.1. Объявление и инициализация отдельного символа

Для хранения символа используется тип `char`. Переменной, имеющей тип `char`, можно присвоить числовое значение (код символа) или указать символ внутри апострофов. Обратите внимание на то, что использовать кавычки нельзя, т. к. в этом случае вместо одного символа будет два: собственно сам символ плюс нулевой символ. Пример:

```
char ch1, ch2;
ch1 = 119; // Буква w
ch2 = 'w'; // Буква w
// Выводим символ
printf("%c\n", ch1);                    // w
printf("%c\n", ch2);                    // w
// Выводим код символа
printf("%d\n", ch1);                    // 119
printf("%d\n", ch2);                    // 119
```

Внутри апострофов можно указать *специальные символы* — комбинации знаков, соответствующих служебным или непечатаемым символам. Перечислим специальные символы, доступные в языке С:

- `\0` — нулевой символ;
- `\n` — перевод строки;
- `\r` — возврат каретки;

- \t — горизонтальная табуляция;
- \v — вертикальная табуляция;
- \a — звуковой сигнал;
- \b — возврат на один символ;
- \f — перевод формата;
- \' — апостроф;
- \" — кавычка;
- \? — знак вопроса;
- \\ — обратная косая черта;
- \N — восьмеричное значение N;
- \xN — шестнадцатеричное значение N.

Пример указания восьмеричного и шестнадцатеричного значений:

```
char ch1, ch2;
ch1 = '\167'; // Восьмеричное значение
ch2 = '\x77'; // Шестнадцатеричное значение
printf("%c\n", ch1);                    // w
printf("%c\n", ch2);                    // w
```

По умолчанию тип char является знаковым и позволяет хранить диапазон значений от -128 до 127. Если перед типом указано ключевое слово unsigned, то диапазон будет от 0 до 255. Когда переменной присваивается символ внутри апострофов, он автоматически преобразуется в соответствующий целочисленный код. Пример вывода размера и диапазона значений:

```
// Выводим размер в байтах
printf("%d\n", (int) sizeof(char));    // 1
// #include <limits.h>
printf("%d\n", CHAR_MIN);              // -128
printf("%d\n", CHAR_MAX);              // 127
printf("%d\n", UCHAR_MAX);             // 255
printf("%d\n", CHAR_BIT);              // 8
```

Тип char занимает в памяти 1 байт (8 бит). Если тип является знаковым, то старший бит содержит признак знака: 0 соответствует положительному числу, а 1 — отрицательному. Если тип является беззнаковым, то признак знака не используется. Это следует учитывать при преобразовании знакового типа в беззнаковый, т. к. старший бит станет причиной больших значений:

```
char ch1 = -1;
unsigned char ch2 = (unsigned char)ch1;
printf("%u\n", ch2);                    // 255
```

Символы, имеющие код меньше 128 (занимают 7 бит), соответствуют кодировке ASCII. Коды этих символов одинаковы практически во всех однобайтовых кодировках. В состав кодировки ASCII входят цифры, буквы латинского алфавита,

Таблица 7.1. Коды основных символов в кодировке ASCII

Символ	dec	oct	hex	Символ	dec	oct	hex	Символ	dec	oct	hex
\0	0	0	0	;	59	73	3b	^	94	136	5e
\a	7	7	7	<	60	74	3c	_	95	137	5f
\b	8	10	8	=	61	75	3d	`	96	140	60
\t	9	11	9	>	62	76	3e	a	97	141	61
\n	10	12	a	?	63	77	3f	b	98	142	62
\v	11	13	b	@	64	100	40	c	99	143	63
\f	12	14	c	A	65	101	41	d	100	144	64
\r	13	15	d	B	66	102	42	e	101	145	65
пробел	32	40	20	C	67	103	43	f	102	146	66
!	33	41	21	D	68	104	44	g	103	147	67
"	34	42	22	E	69	105	45	h	104	150	68
#	35	43	23	F	70	106	46	i	105	151	69
$	36	44	24	G	71	107	47	j	106	152	6a
%	37	45	25	H	72	110	48	k	107	153	6b
&	38	46	26	I	73	111	49	l	108	154	6c
'	39	47	27	J	74	112	4a	m	109	155	6d
(	40	50	28	K	75	113	4b	n	110	156	6e
)	41	51	29	L	76	114	4c	o	111	157	6f
*	42	52	2a	M	77	115	4d	p	112	160	70
+	43	53	2b	N	78	116	4e	q	113	161	71
,	44	54	2c	O	79	117	4f	r	114	162	72
-	45	55	2d	P	80	120	50	s	115	163	73
.	46	56	2e	Q	81	121	51	t	116	164	74
/	47	57	2f	R	82	122	52	u	117	165	75
0	48	60	30	S	83	123	53	v	118	166	76
1	49	61	31	T	84	124	54	w	119	167	77
2	50	62	32	U	85	125	55	x	120	170	78
3	51	63	33	V	86	126	56	y	121	171	79
4	52	64	34	W	87	127	57	z	122	172	7a
5	53	65	35	X	88	130	58	{	123	173	7b
6	54	66	36	Y	89	131	59	|	124	174	7c
7	55	67	37	Z	90	132	5a	}	125	175	7d
8	56	70	38	[	91	133	5b	~	126	176	7e
9	57	71	39	\	92	134	5c	DEL	127	177	7f
:	58	72	3a	]	93	135	5d				

знаки препинания и некоторые служебные символы (например, перенос строки, табуляция и т. д.). Коды основных символов в кодировке ASCII в десятичном (dec), восьмеричном (oct) и шестнадцатеричном (hex) виде приведены в табл. 7.1.

Восьмой бит предназначен для кодирования символов национальных алфавитов. У типа char эти символы имеют отрицательные значения (старший бит содержит признак знака). Таким образом, тип char позволяет закодировать всего 256 символов.

Для кодирования букв русского языка предназначено пять кодировок — windows-1251 (cp1251), windows-866 (cp866), iso8859-5, koi8-r и mac-cyrillic. Проблема заключается в том, что код одной и той же русской буквы в этих кодировках может быть разным. Из-за этого возникает множество проблем. Например, при выводе русских букв в консоли может отобразиться нечитаемый текст. Причина искажения русских букв заключается в том, что по умолчанию в окне консоли используется кодировка cp866. Коды русских букв и некоторых символов в кодировках cp866 и cp1251 в десятичном (unsigned и signed), восьмеричном (oct) и шестнадцатеричном (hex) виде приведены в табл. 7.2. Обратите внимание на то, что коды букв "ё" и "Ё" выпадают из последовательности кодов.

***Таблица 7.2.** Коды русских букв и некоторых символов в кодировках cp866 и cp1251*

Символ	cp866				cp1251			
	unsigned	signed	oct	hex	unsigned	signed	oct	hex
А	128	–128	200	80	192	–64	300	c0
Б	129	–127	201	81	193	–63	301	c1
В	130	–126	202	82	194	–62	302	c2
Г	131	–125	203	83	195	–61	303	c3
Д	132	–124	204	84	196	–60	304	c4
Е	133	–123	205	85	197	–59	305	c5
Ё	240	–16	360	f0	168	–88	250	a8
Ж	134	–122	206	86	198	–58	306	c6
З	135	–121	207	87	199	–57	307	c7
И	136	–120	210	88	200	–56	310	c8
Й	137	–119	211	89	201	–55	311	c9
К	138	–118	212	8a	202	–54	312	ca
Л	139	–117	213	8b	203	–53	313	cb
М	140	–116	214	8c	204	–52	314	cc
Н	141	–115	215	8d	205	–51	315	cd
О	142	–114	216	8e	206	–50	316	ce
П	143	–113	217	8f	207	–49	317	cf

Таблица 7.2 (продолжение)

Символ	cp866				cp1251			
	unsigned	signed	oct	hex	unsigned	signed	oct	hex
Р	144	–112	220	90	208	–48	320	d0
С	145	–111	221	91	209	–47	321	d1
Т	146	–110	222	92	210	–46	322	d2
У	147	–109	223	93	211	–45	323	d3
Ф	148	–108	224	94	212	–44	324	d4
Х	149	–107	225	95	213	–43	325	d5
Ц	150	–106	226	96	214	–42	326	d6
Ч	151	–105	227	97	215	–41	327	d7
Ш	152	–104	230	98	216	–40	330	d8
Щ	153	–103	231	99	217	–39	331	d9
Ъ	154	–102	232	9a	218	–38	332	da
Ы	155	–101	233	9b	219	–37	333	db
Ь	156	–100	234	9c	220	–36	334	dc
Э	157	–99	235	9d	221	–35	335	dd
Ю	158	–98	236	9e	222	–34	336	de
Я	159	–97	237	9f	223	–33	337	df
а	160	–96	240	a0	224	–32	340	e0
б	161	–95	241	a1	225	–31	341	e1
в	162	–94	242	a2	226	–30	342	e2
г	163	–93	243	a3	227	–29	343	e3
д	164	–92	244	a4	228	–28	344	e4
е	165	–91	245	a5	229	–27	345	e5
ё	241	–15	361	f1	184	–72	270	b8
ж	166	–90	246	a6	230	–26	346	e6
з	167	–89	247	a7	231	–25	347	e7
и	168	–88	250	a8	232	–24	350	e8
й	169	–87	251	a9	233	–23	351	e9
к	170	–86	252	aa	234	–22	352	ea
л	171	–85	253	ab	235	–21	353	eb
м	172	–84	254	ac	236	–20	354	ec
н	173	–83	255	ad	237	–19	355	ed
о	174	–82	256	ae	238	–18	356	ee

Таблица 7.2 (продолжение)

Символ	cp866				cp1251			
	unsigned	signed	oct	hex	unsigned	signed	oct	hex
п	175	–81	257	af	239	–17	357	ef
р	224	–32	340	e0	240	–16	360	f0
с	225	–31	341	e1	241	–15	361	f1
т	226	–30	342	e2	242	–14	362	f2
у	227	–29	343	e3	243	–13	363	f3
ф	228	–28	344	e4	244	–12	364	f4
х	229	–27	345	e5	245	–11	365	f5
ц	230	–26	346	e6	246	–10	366	f6
ч	231	–25	347	e7	247	–9	367	f7
ш	232	–24	350	e8	248	–8	370	f8
щ	233	–23	351	e9	249	–7	371	f9
ъ	234	–22	352	ea	250	–6	372	fa
ы	235	–21	353	eb	251	–5	373	fb
ь	236	–20	354	ec	252	–4	374	fc
э	237	–19	355	ed	253	–3	375	fd
ю	238	–18	356	ee	254	–2	376	fe
я	239	–17	357	ef	255	–1	377	ff
Є	242	–14	362	f2	170	–86	252	aa
є	243	–13	363	f3	186	–70	272	ba
°	248	–8	370	f8	176	–80	260	b0
№	252	–4	374	fc	185	–71	271	b9
„	–	–	–	–	132	–124	204	84
…	–	–	–	–	133	–123	205	85
Є	–	–	–	–	136	–120	210	88
‰	–	–	–	–	137	–119	211	89
‹	–	–	–	–	145	–111	221	91
’	–	–	–	–	146	–110	222	92
“	–	–	–	–	147	–109	223	93
”	–	–	–	–	148	–108	224	94
–	–	–	–	–	150	–106	226	96
—	–	–	–	–	151	–105	227	97
™	–	–	–	–	153	–103	231	99

Таблица 7.2 (окончание)

Символ	cp866				cp1251			
	unsigned	signed	oct	hex	unsigned	signed	oct	hex
¦	–	–	–	–	166	–90	246	a6
§	–	–	–	–	167	–89	247	a7
©	–	–	–	–	169	–87	251	a9
«	–	–	–	–	171	–85	253	ab
¬	–	–	–	–	172	–84	254	ac
®	–	–	–	–	174	–82	256	ae
±	–	–	–	–	177	–79	261	b1
»	–	–	–	–	187	–69	273	bb

7.2. Настройка локали

При изменении регистра русских букв может возникнуть проблема. Для того чтобы ее избежать, необходимо правильно настроить *локаль* (совокупность локальных настроек системы). Для настройки локали используется функция `setlocale()`. Прототип функции:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

В первом параметре указывается категория в виде числа от `0` до `5` или соответствующий числу макрос:

- `0` — `LC_ALL` — устанавливает локаль для всех категорий;
- `1` — `LC_COLLATE` — для сравнения строк;
- `2` — `LC_CTYPE` — для перевода символов в нижний или верхний регистр;
- `3` — `LC_MONETARY` — для отображения денежных единиц;
- `4` — `LC_NUMERIC` — для форматирования дробных чисел;
- `5` — `LC_TIME` — для форматирования вывода даты и времени.

Во втором параметре задается название локали в виде строки, например: `rus`, `russian` или `Russian_Russia`. В этом случае будет задана кодировка, используемая в системе по умолчанию:

```
char *pLocale = setlocale(LC_ALL, "Russian_Russia");
if (pLocale) {
   printf("%s\n", pLocale); // Russian_Russia.1251
}
else {
   puts("Не удалось настроить локаль");
}
```

Для того чтобы использовать другую кодировку, нужно указать ее название после точки, например ".1251" или "Russian_Russia.1251":

```
char *pLocale = setlocale(LC_ALL, "Russian_Russia.1251");
if (pLocale) {
   printf("%s\n", pLocale); // Russian_Russia.1251
}
else {
   puts("Не удалось настроить локаль");
}
```

Вместо названия можно указать пустую строку. В этом случае будет использоваться локаль, настроенная в системе:

```
char *pLocale = setlocale(LC_ALL, "");
if (pLocale) {
   printf("%s\n", pLocale); // Russian_Russia.1251
}
```

В качестве значения функция возвращает указатель на строку с названием локали, соответствующей заданной категории, или нулевой указатель в случае ошибки. Если во втором параметре передан нулевой указатель, то функция возвращает указатель на строку с названием текущей локали:

```
char *pLocale = setlocale(LC_ALL, NULL);
if (pLocale) {
   printf("%s\n", pLocale); // C
}
```

Для настройки локали можно также использовать функцию `_wsetlocale()`. Прототип функции:

```
#include <locale.h>
wchar_t *_wsetlocale(int category, const wchar_t *locale);
```

Параметры аналогичны одноименным параметрам функции `setlocale()`, но во втором параметре указывается L-строка с названием локали и возвращается указатель на L-строку:

```
wchar_t *pLocale = _wsetlocale(LC_ALL, L"Russian_Russia.1251");
if (pLocale) {
   wprintf(L"%s\n", pLocale); // Russian_Russia.1251
}
```

После настройки локали многие функции будут работать не так, как раньше. Например, от настроек локали зависит вывод функции `printf()`:

```
printf("%.2f\n", 2.5);            // 2.50
setlocale(LC_NUMERIC, "dutch");
printf("%.2f\n", 2.5);            // 2,50
```

Некоторые функции позволяют указать локаль в качестве параметра, например, функция `_printf_l()`:

```
int _printf_l(const char *format, _locale_t locale, ...);
```

Во втором параметре функция принимает структуру `_locale_t` с настройками локали. Создать эту структуру позволяет функция `_create_locale()`. Прототип функции:

```
#include <locale.h>
_locale_t _create_locale(int category, const char *locale);
```

Все параметры аналогичны одноименным параметрам функции `setlocale()`. Если структуру создать не удалось, то функция возвращает нулевой указатель.

В Visual C вместо функции `_create_locale()` можно воспользоваться функцией `_wcreate_locale()`. Обратите внимание: функция недоступна в MinGW. Прототип функции:

```
#include <locale.h>
_locale_t _wcreate_locale(int category, const wchar_t *locale);
```

Создать структуру `_locale_t` с настройками текущей локали позволяет функция `_get_current_locale()`. Прототип функции:

```
#include <locale.h>
_locale_t _get_current_locale(void);
```

Когда структура больше не нужна, ее следует освободить с помощью функции `_free_locale()`. Прототип функции:

```
#include <locale.h>
void _free_locale(_locale_t locale);
```

Пример настройки локали и вывода текущего названия локали приведен в листинге 7.1.

Листинг 7.1. Настройка локали

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

int main(void) {
   char *pLocale = setlocale(LC_ALL, NULL);
   if (pLocale) {
      printf("%s\n", pLocale); // C
   }
   printf("%.2f\n", 3.14);      // 3.14
   pLocale = setlocale(LC_NUMERIC, "dutch");
   if (pLocale) {
      printf("%s\n", pLocale); // Dutch_Netherlands.1252
   }
   printf("%.2f\n", 3.14);      // 3,14
   pLocale = setlocale(LC_ALL, "Russian_Russia");
   if (pLocale) {
      printf("%s\n", pLocale); // Russian_Russia.1251
   }
```

```
   pLocale = setlocale(LC_ALL, "Russian_Russia.866");
   if (pLocale) {
      printf("%s\n", pLocale); // Russian_Russia.866
   }

   _locale_t locale_c = _create_locale(LC_NUMERIC, "C");
   _printf_l("%.2f\n", locale_c, 3.14);  // 3.14
   _free_locale(locale_c);
   _locale_t locale_de = _create_locale(LC_NUMERIC, "dutch");
   _printf_l("%.2f\n", locale_de, 3.14); // 3,14
   _free_locale(locale_de);
   return 0;
}
```

Функция `localeconv()` **позволяет получить информацию о способе форматирования вещественных чисел и денежных сумм для текущей локали. Прототип функции:**

```
#include <locale.h>
struct lconv *localeconv(void);
```

Функция возвращает указатель на структуру `lconv`**. Пример вывода символа десятичного разделителя для локали** `Russian_Russia.1251`**:**

```
setlocale(LC_ALL, "Russian_Russia.1251");
struct lconv *p = localeconv();
if (p) {
   printf("%s\n", p->decimal_point);
}
```

Объявление структуры `lconv` **выглядит следующим образом:**

```
struct lconv {
  char *decimal_point;     // Десятичный разделитель (",")
  char *thousands_sep;     // Разделитель тысяч (" ")
  char *grouping;          // Способ группировки значений
  char *int_curr_symbol;   // Название валюты ("RUB")
  char *currency_symbol;   // Символ валюты ("р.")
  char *mon_decimal_point; // Десятичный разделитель для
                           // денежных сумм (",")
  char *mon_thousands_sep; // Разделитель тысяч для денежных
                           // сумм (" ")
  char *mon_grouping;    // Способ группировки для денежных сумм
  char *positive_sign;   // Положительный знак для денежных сумм
  char *negative_sign;   // Отрицательный знак для денежных сумм ("-")
  char int_frac_digits; // Количество цифр в дробной части для
                        // денежных сумм в международном формате (2)
  char frac_digits;     // Количество цифр в дробной части для
                        // денежных сумм в национальном формате (2)
  char p_cs_precedes;   // 1 - если символ валюты перед значением
                        // 0 - если символ валюты после значения
```

```
  char p_sep_by_space;  // 1 - если символ валюты отделяется пробелом
                        // 0 - в противном случае
                        // p_cs_precedes и p_sep_by_space применяются
                        // для положительных значений
  char n_cs_precedes;   // 1 - если символ валюты перед значением
                        // 0 - если символ валюты после значения
  char n_sep_by_space;  // 1 - если символ валюты отделяется пробелом
                        // 0 - в противном случае
                        // n_cs_precedes и n_sep_by_space применяются
                        // для отрицательных значений
  char p_sign_posn;     // Позиция символа положительного значения
  char n_sign_posn;     // Позиция символа отрицательного значения
};
```

В Visual C доступны также следующие поля структуры `lconv`:

```
wchar_t *_W_decimal_point;
wchar_t *_W_thousands_sep;
wchar_t *_W_int_curr_symbol;
wchar_t *_W_currency_symbol;
wchar_t *_W_mon_decimal_point;
wchar_t *_W_mon_thousands_sep;
wchar_t *_W_positive_sign;
wchar_t *_W_negative_sign;
```

7.3. Изменение регистра символов

Для изменения регистра символов предназначены следующие функции.

- `toupper()` **и** `_toupper_l()` **— возвращают код символа в верхнем регистре. Если преобразования регистра не было, то код символа возвращается без изменений. Прототипы функций:**

```
#include <ctype.h>
int toupper(int ch);
int _toupper_l(int ch, _locale_t locale);
```

Пример:

```
// #include <string.h>
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%c\n", (char)toupper('w'));                 // W
printf("%c\n", (char)toupper((unsigned char)'б')); // Б
char str[] = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
for (int i = 0, len = (int) strlen(str); i < len; ++i) {
   str[i] = (char)toupper((unsigned char)str[i]);
}
printf("%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

- `tolower()` и `_tolower_l()` — возвращают код символа в нижнем регистре. Если преобразования регистра не было, то код символа возвращается без изменений. Прототипы функций:

```
#include <ctype.h>
int tolower(int ch);
int _tolower_l(int ch, _locale_t locale);
```

Пример:

```
// #include <string.h>
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%c\n", (char)tolower('w'));                 // w
printf("%c\n", (char)tolower('W'));                 // w
printf("%c\n", (char)tolower((unsigned char)'Б')); // б
char str[] = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
for (int i = 0, len = (int) strlen(str); i < len; ++i) {
   str[i] = (char)tolower((unsigned char)str[i]);
}
printf("%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

- `_strupr()` — заменяет все буквы в С-строке `str` соответствующими прописными буквами. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <string.h>
char *_strupr(char *str);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
_strupr(str);
printf("%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

Вместо функции `_strupr()` лучше использовать функцию `_strupr_s()`. Прототип функции:

```
#include <string.h>
errno_t _strupr_s(char *str, size_t strSize);
```

В параметре `strSize` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[40] = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
_strupr_s(str, 40);
printf("%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

- `_strlwr()` — заменяет все буквы в С-строке `str` соответствующими строчными буквами. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <string.h>
char *_strlwr(char *str);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
_strlwr(str);
printf("%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

Вместо функции `_strlwr()` лучше использовать функцию `_strlwr_s()`. Прототип функции:

```
#include <string.h>
errno_t _strlwr_s(char *str, size_t strSize);
```

В параметре `strSize` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[40] = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
_strlwr_s(str, 40);
printf("%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

7.4. Проверка типа содержимого символа

Для проверки типа содержимого символа предназначены следующие функции.

- `isdigit()` и `_isdigit_l()` — возвращают ненулевое значение, если символ является десятичной цифрой, в противном случае — `0`. Прототипы функций:

```
#include <ctype.h>
int isdigit(int ch);
int _isdigit_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре `locale`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isdigit('w'));                // 0
printf("%d\n", isdigit('2'));                // 1
printf("%d\n", isdigit((unsigned char)'б')); // 0
```

- `isxdigit()` и `_isxdigit_l()` — возвращают ненулевое значение, если символ является шестнадцатеричной цифрой (число от `0` до `9` или буква от `A` до `F`; регистр не имеет значения), в противном случае — `0`. Прототипы функций:

```
#include <ctype.h>
int isxdigit(int ch);
int _isxdigit_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре locale. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isxdigit('8')); // 128
printf("%d\n", isxdigit('a')); // 128
printf("%d\n", isxdigit('F')); // 128
printf("%d\n", isxdigit('g')); // 0
```

- isalpha() и _isalpha_l() — возвращают ненулевое значение, если символ является буквой, в противном случае — 0. Прототипы функций:

```
#include <ctype.h>
int isalpha(int ch);
int _isalpha_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре locale. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isalpha('w'));                // 258
printf("%d\n", isalpha('2'));                // 0
printf("%d\n", isalpha((unsigned char)'б')); // 258
printf("%d\n", isalpha((unsigned char)'Б')); // 257
```

- isspace() и _isspace_l() — возвращают ненулевое значение, если символ является пробельным символом (пробелом, табуляцией, переводом строки или возвратом каретки), в противном случае — 0. Прототипы функций:

```
#include <ctype.h>
int isspace(int ch);
int _isspace_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре locale. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isspace('w'));                // 0
printf("%d\n", isspace('\n'));               // 8
printf("%d\n", isspace('\t'));               // 8
printf("%d\n", isspace((unsigned char)'б')); // 0
```

- isalnum() и _isalnum_l() — возвращают ненулевое значение, если символ является буквой или цифрой, в противном случае — 0. Прототипы функций:

```
#include <ctype.h>
int isalnum(int ch);
int _isalnum_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре locale. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isalnum('w'));                // 258
```

```
printf("%d\n", isalnum('8'));                 // 4
printf("%d\n", isalnum('\t'));                // 0
printf("%d\n", isalnum((unsigned char)'б')); // 258
```

- `islower()` и `_islower_l()` — возвращают ненулевое значение, если символ является буквой в нижнем регистре, в противном случае — `0`. Прототипы функций:

```
#include <ctype.h>
int islower(int ch);
int _islower_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре `locale`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", islower('w'));                 // 2
printf("%d\n", islower('8'));                 // 0
printf("%d\n", islower('\t'));                // 0
printf("%d\n", islower((unsigned char)'б')); // 2
printf("%d\n", islower((unsigned char)'Б')); // 0
```

- `isupper()` и `_isupper_l()` — возвращают ненулевое значение, если символ является буквой в верхнем регистре, в противном случае — `0`. Прототипы функций:

```
#include <ctype.h>
int isupper(int ch);
int _isupper_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре `locale`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isupper('W'));                 // 1
printf("%d\n", isupper('8'));                 // 0
printf("%d\n", isupper('\t'));                // 0
printf("%d\n", isupper((unsigned char)'б')); // 0
printf("%d\n", isupper((unsigned char)'Б')); // 1
```

- `ispunct()` и `_ispunct_l()` — возвращают ненулевое значение, если символ является символом пунктуации, в противном случае — `0`. Прототипы функций:

```
#include <ctype.h>
int ispunct(int ch);
int _ispunct_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре `locale`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", ispunct('8')); // 0
printf("%d\n", ispunct('f')); // 0
printf("%d\n", ispunct(',')); // 16
printf("%d\n", ispunct('.')); // 16
printf("%d\n", ispunct(' ')); // 0
```

- `isprint()` и `_isprint_l()` — возвращают ненулевое значение, если символ является печатаемым (включая пробел), в противном случае — `0`. Прототипы функций:

```
#include <ctype.h>
int isprint(int ch);
int _isprint_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре `locale`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isprint('8'));                // 4
printf("%d\n", isprint('\x5'));              // 0
printf("%d\n", isprint(' '));                // 64
printf("%d\n", isprint((unsigned char)'б')); // 258
```

- `isgraph()` и `_isgraph_l()` — возвращают ненулевое значение, если символ является печатаемым (пробел печатаемым не считается), в противном случае — `0`. Прототипы функций:

```
#include <ctype.h>
int isgraph(int ch);
int _isgraph_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре `locale`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isgraph('8'));                // 4
printf("%d\n", isgraph('\x5'));              // 0
printf("%d\n", isgraph(' '));                // 0
printf("%d\n", isgraph((unsigned char)'б')); // 258
```

- `iscntrl()` и `_iscntrl_l()` — возвращают ненулевое значение, если символ является непечатаемым, в противном случае — `0`. Прототипы функций:

```
#include <ctype.h>
int iscntrl(int ch);
int _iscntrl_l(int ch, _locale_t locale);
```

Для русских букв необходимо настроить локаль или указать ее в параметре `locale`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", iscntrl('8'));   // 0
printf("%d\n", iscntrl('\x5')); // 32
printf("%d\n", iscntrl(' '));   // 0
```

- `isblank()` — возвращает ненулевое значение, если символ является пробельным (пробел, горизонтальная табуляция и др.), в противном случае — `0`. Прототип функции:

```
#include <ctype.h>
int isblank(int ch);
```

Для русских букв необходимо настроить локаль. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", isblank('8'));   // 0
printf("%d\n", isblank('\t'));  // 1
printf("%d\n", isblank(' '));   // 1
```

Обратите внимание на то, что перед русскими буквами указывается операция приведения к типу `unsigned char`. Если это не сделать, то производится попытка преобразования к типу `unsigned int`. Поскольку коды русских букв по умолчанию имеют отрицательные значения, знаковый бит станет причиной большого значения, которое выходит за рамки диапазона значений для типа `char`. Так, для буквы "б" значение будет равно `4294967265`. Пример:

```
printf("%u\n", (unsigned int)(unsigned char)'б'); // 225
printf("%u\n", (unsigned int)'б');                // 4294967265
```

7.5. Объявление и инициализация С-строки

С-строка является массивом символов (тип `char`), последний элемент которого содержит нулевой символ (`'\0'`). Обратите внимание на то, что нулевой символ (нулевой байт) не имеет никакого отношения к символу `'0'`. Коды этих символов разные.

Объявляется С-строка так же, как и массив элементов типа `char`:

```
char str[7];
```

При инициализации С-строки можно перечислить символы внутри фигурных скобок:

```
char str[7] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
printf("%s\n", str);   // String
```

или указать строку внутри двойных кавычек:

```
char str[7] = "String";
printf("%s\n", str);   // String
```

При использовании двойных кавычек следует учитывать, что длина строки на один символ больше, т. к. в конец будет автоматически вставлен нулевой символ. Если это не предусмотреть и объявить массив из шести элементов вместо семи, то это приведет к ошибке.

Если размер массива при объявлении не указать, то он будет определен автоматически в соответствии с длиной строки:

```
char str[] = "String";
printf("%d\n", (int) sizeof(str)); // 7
printf("%s\n", str);               // String
```

Обратите внимание на то, что присваивать строку в двойных кавычках можно только при инициализации. Попытка присвоить строку позже приведет к ошибке:

```
char str[7];
str = "String"; // Ошибка!!!
```

В этом случае нужно воспользоваться функцией `strcpy()` или `strcpy_s()`:

```
char str[7];
strcpy_s(str, 7, "String");
printf("%s\n", str); // String
```

Внутри строки в двойных кавычках можно указывать специальные символы (например, `\n`, `\r` и др.), которые мы уже рассматривали в *разд. 7.1*. Если внутри строки встречается кавычка, то ее необходимо экранировать с помощью обратного слэша:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "Группа \"Кино\"\n";
printf("%s", str);   // Группа "Кино"
```

Если внутри строки встречается обратный слэш, то его необходимо экранировать. Это обязательно следует учитывать при указании пути к файлу:

```
char str1[] = "C:\\temp\\new\\file.txt"; // Правильно
char str2[] = "C:\temp\new\file.txt";    // Неправильно!!!
printf("%s\n", str1);   // C:\temp\new\file.txt
printf("%s\n", str2);   // Строка с ошибками!!!
```

Обратите внимание на вторую строку. В этом пути присутствуют сразу три специальных символа: `\t`, `\n` и `\f`. После преобразования специальных символов путь будет выглядеть следующим образом:

```
C:<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```

Разместить С-строку при инициализации на нескольких строках нельзя. Переход на новую строку вызовет синтаксическую ошибку:

```
char str[] = "string1
string2"; // Ошибка!!!
```

Для того чтобы разместить С-строку на нескольких строках, следует перед символом перевода строки указать символ \:

```
char str[] = "string1 \
string2 \
string3"; // После символа \ не должно быть никаких символов
printf("%s\n", str);   // string1 string2 string3
```

Кроме того, можно воспользоваться неявной *конкатенацией*. Если строки расположены подряд внутри одной инструкции, то они объединяются в одну большую строку. Пример:

```
char str[] = "string1"
"string2" "string3";
printf("%s\n", str);   // string1string2string3
```

Строку можно присвоить указателю, но в этом случае строка будет доступна только для чтения. Попытка изменить какой-либо символ внутри строки приведет к ошибке при выполнении:

```
const char *p = "string";
printf("%s\n", p); // string
```

Мы можем объявить несколько указателей и присвоить им одинаковую строку, но все они будут ссылаться на один и тот же адрес. Иными словами, строка, указанная в программе внутри двойных кавычек, в памяти существует в единственном экземпляре. Если бы была возможность изменить такую строку, то она изменилась бы во всех указателях. Поэтому строка, присвоенная указателю, доступна только для чтения:

```
const char *p1 = "string";
const char *p2 = "string";
const char *p3 = "string";
printf("%p %p %p\n", p1, p2, p3);
// 0000000000409030 0000000000409030 0000000000409030
```

Объявление массива строк выглядит так:

```
char str[][20] = {"String1", "String2", "String3"};
printf("%s\n", str[0]);   // String1
printf("%s\n", str[1]);   // String2
printf("%s\n", str[2]);   // String3
```

или так:

```
char *str[] = {"String1", "String2", "String3"};
printf("%s\n", str[0]);   // String1
printf("%s\n", str[1]);   // String2
printf("%s\n", str[2]);   // String3
```

Обратите внимание: при использовании первого способа мы можем изменить символы в строках, а вот при использовании второго способа попытка изменения символа приведет к ошибке при выполнении программы. Так что эти способы объявления массива строк не эквивалентны.

7.6. Доступ к символам внутри С-строки

После определения С-строки в переменной сохраняется адрес первого символа. Иными словами, название переменной является указателем, который ссылается на первый символ строки. Поэтому доступ к символу в строке может осуществляться как по индексу (нумерация начинается с нуля), указанному внутри квадратных скобок, так и с использованием адресной арифметики. Например, следующие две инструкции вывода являются эквивалентными:

```
char str[] = "строка";
printf("%c\n", str[1]);     // т
printf("%c\n", *(str + 1)); // т
```

Символ можно не только получить таким образом, но и изменить:

```
char str[] = "строка";
str[0] = 'C';         // Изменение с помощью индекса
*(str + 1) = 'T';     // Изменение с помощью указателя
printf("%s\n", str); // СТрока
```

Обратите внимание на то, что отдельный символ указывается внутри апострофов, а не внутри кавычек. Если указать символ внутри кавычек, то вместо одного символа будет два: собственно сам символ плюс нулевой символ.

Объявить указатель и присвоить ему адрес строки можно следующим образом:

```
char str[] = "строка";
char *p = NULL;
p = str;
*p = 'C';
++p;       // Перемещаем указатель на второй символ
*p = 'T';
printf("%s\n", str); // СТрока
```

Обратите внимание на то, что перед названием строки не указывается оператор &, т. к. название переменной содержит адрес первого символа. Если использовать оператор &, то необходимо дополнительно указать индекс внутри квадратных скобок:

```
p = &str[0]; // Эквивалентно: p = str;
```

При инициализации указателя ему можно присвоить строку. Такие строки нельзя изменять, поэтому обычно перед типом указывают ключевое слово `const`. Пример:

```
const char *str = "String";
printf("%s\n", str); // String
```

7.7. Определение длины строки

Получить длину строки можно с помощью функции `strlen()`. Прототип функции:

```
#include <string.h>
size_t strlen(const char *str);
```

Функция `strlen()` возвращает количество символов без учета нулевого символа. Тип `size_t` объявлен как беззнаковое целое:

```
typedef unsigned __int64 size_t;
#define __int64 long long
```

Пример:

```
char str[] = "строка";
int len = (int)strlen(str);
printf("%d\n", len);                // 6
printf("%d\n", (int) sizeof(str)); // 7
```

```
char *p = str;
printf("%d\n", (int) sizeof(p));
// Значение в проекте Test64c: 8
```

Обратите внимание: длина строки и размер символьного массива — это разные вещи. *Длина строки* — это количество символов с начала массива до первого нулевого символа. *Размер символьного массива* — это общее количество символов, доступное под строку. Мы можем получить размер символьного массива с помощью оператора `sizeof`, указав имя переменной, но если присвоить ее адрес указателю, то мы получим размер указателя, а не символьного массива.

7.8. Перебор символов С-строки

Для перебора символов удобно использовать цикл `for`. В первом параметре переменной-счетчику присваивается значение `0` (символы в строке нумеруются с нуля), условием продолжения является значение переменной-счетчика меньше количества символов в строке. В третьем параметре цикла `for` указывается приращение на единицу на каждой итерации цикла. Выведем все символы строки в прямом и обратном порядке:

```
char str[] = "String";
int len = (int)strlen(str);
// Выводим символы в прямом порядке
for (int i = 0; i < len; ++i) {
   printf("%c\n", str[i]);
}
puts("--------------------");
// Выводим символы в обратном порядке
for (int i = len - 1; i >= 0; --i) {
   printf("%c\n", str[i]);
}
```

В этом примере количество символов сохраняется в переменной `len` вне цикла. Если функцию `strlen()` указать внутри условия, то вычисление количества символов будет выполняться на каждой итерации цикла. Поэтому количество символов лучше получать вне цикла или присваивать значение переменной в первом параметре цикла `for`. Пример:

```
char str[] = "String";
for (int i = 0, len = (int)strlen(str); i < len; ++i) {
   printf("%c\n", str[i]);
}
```

Перебор символов С-строки с помощью указателя и цикла `for` выполняется так:

```
char str[] = "String";
for (char *p = str; *p; ++p) {
   printf("%c\n", *p);
}
```

В этом примере начальным значением является адрес первого символа. Условием продолжения цикла служит значение, на которое ссылается указатель. Любой символ трактуется как Истина, кроме нулевого символа, который имеет код 0. Так как С-строка завершается нулевым символом, то этот символ вернет значение Ложь и цикл завершится. В третьем параметре цикла for указывается приращение указателя на единицу на каждой итерации цикла. В этом случае используются правила адресной арифметики.

Вместо цикла for всегда можно использовать цикл while:

```
char str[] = "String";
char *p = str;
while (*p) {
   printf("%c\n", *p++);
}
p = str; // Восстанавливаем положение указателя
```

Вначале объявляются строка и указатель, которому присваивается адрес первого символа. Цикл while выполняется до тех пор, пока значение, на которое ссылается указатель, не равно нулевому символу. Внутри цикла while выводится символ, на который ссылается указатель, а затем указатель перемещается на следующий символ (*p++). Выражение p++ возвращает текущий адрес, а затем увеличивает его. Символ * позволяет получить доступ к символу по указанному адресу. Последовательность выполнения соответствует следующей расстановке скобок: *(p++) или двум инструкциям: *p и p = p + 1.

7.9. Основные функции для работы с С-строками

Перечислим основные функции для работы с С-строками.

- strlen() — возвращает количество символов в С-строке без учета нулевого символа. Прототип функции:

  ```
  #include <string.h>
  size_t strlen(const char *str);
  ```

  Тип size_t объявлен как беззнаковое целое число:

  ```
  typedef unsigned __int64 size_t;
  #define __int64 long long
  ```

  Пример:

  ```
  char str[7] = "строка";
  int len = (int)strlen(str);
  printf("%d\n", len);                // 6, а не 7
  ```

  Определить общий размер массива можно с помощью оператора sizeof. Оператор возвращает размер в байтах. Так как тип char занимает 1 байт, следовательно, размер в байтах будет равен размеру массива. Пример:

```
char str[20] = "строка";
printf("%d\n", (int) sizeof(str)); // 20
```

- `strnlen()` — возвращает количество символов в С-строке без учета нулевого символа. Прототип функции:

```
#include <string.h>
size_t strnlen(const char *str, size_t maxCount);
```

Во втором параметре указывается размер символьного массива. Если нулевой символ не встретился в числе `maxCount` символов, то возвращается значение `maxCount`. Пример:

```
char str[7] = "строка";
int len = (int)strnlen(str, 7);
printf("%d\n", len);                // 6
```

- `strcpy()` — копирует символы из С-строки `source` в С-строку `dest` и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку `dest`. Если строка `source` длиннее строки `dest`, то произойдет переполнение буфера. Прототип функции:

```
#include <string.h>
char *strcpy(char *dest, const char *source);
```

Пример использования функции:

```
char str[7];
strcpy(str, "String");
printf("%s\n", str); // String
```

Вместо функции `strcpy()` лучше использовать функцию `strcpy_s()`. Прототип функции:

```
#include <string.h>
errno_t strcpy_s(char *dest, rsize_t sizeInBytes, const char *source);
```

Функция `strcpy_s()` копирует символы из строки `source` в строку `dest` и вставляет нулевой символ. В параметре `sizeInBytes` указывается максимальное количество элементов массива `dest`. Пример:

```
#define SIZE 7
char str[SIZE];
strcpy_s(str, SIZE, "String");
printf("%s\n", str); // String
```

- `strncpy()` — копирует первые `count` символов из С-строки `source` в С-строку `dest`. В качестве значения функция возвращает указатель на строку `dest`. Если строка `source` длиннее строки `dest`, то произойдет переполнение буфера. Прототип функции:

```
#include <string.h>
char *strncpy(char *dest, const char *source, size_t count);
```

Если количество символов в строке source меньше числа count, то строка dest будет дополнена нулевыми символами, а если больше или равно, то копируются только count символов, при этом нулевой символ автоматически не вставляется. Пример копирования шести символов:

```
char str[7];
strncpy(str, "String", 6);
str[6] = '\0'; // Нулевой символ автоматически не вставляется
printf("%s\n", str); // String
```

Вместо функции strncpy() лучше использовать функцию strncpy_s(). Прототип функции:

```
#include <string.h>
errno_t strncpy_s(char *dest, size_t sizeInBytes,
                  const char *source, size_t maxCount);
```

Функция strncpy_s() копирует первые maxCount символов из строки source в строку dest и вставляет нулевой символ. В параметре sizeInBytes указывается максимальное количество элементов массива dest. Пример:

```
#define SIZE 7
char str[SIZE];
strncpy_s(str, SIZE, "String", 5);
printf("%s\n", str); // Strin
```

- strcat() — копирует символы из С-строки source в конец С-строки dest и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку dest. Если строка dest имеет недостаточный размер, то произойдет переполнение буфера. Прототип функции:

```
#include <string.h>
char *strcat(char *dest, const char *source);
```

Пример использования функции:

```
char str[20] = "ст";
strcat(str, "ро");
strcat(str, "ка");
printf("%s\n", str); // строка
```

Обратите внимание на то, что перед копированием в строке dest обязательно должен быть нулевой символ. Локальные переменные автоматически не инициализируются, поэтому этот код приведет к ошибке:

```
char str[20];           // Локальная переменная
strcat(str, "строка");  // Ошибка, нет нулевого символа!
```

Для того чтобы избавиться от ошибки, нужно произвести инициализацию строки нулями следующим образом:

```
char str[20] = {0};     // Инициализация нулями
strcat(str, "строка");  // Все нормально
printf("%s\n", str);    // строка
```

Вместо функции `strcat()` лучше использовать функцию `strcat_s()`. Прототип функции:

```
#include <string.h>
errno_t strcat_s(char *dest, rsize_t sizeInBytes, const char *source);
```

Функция `strcat_s()` копирует символы из строки `source` в конец строки `dest`. В параметре `sizeInBytes` указывается максимальное количество элементов массива `dest`. Пример:

```
#define SIZE 20
char str[SIZE] = "ст";
strcat_s(str, SIZE, "ро");
strcat_s(str, SIZE, "ка");
printf("%s\n", str); // строка
```

- `strncat()` — копирует первые `count` символов из С-строки `source` в конец строки `dest` и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку `dest`. Обратите внимание на то, что перед копированием в строке `dest` обязательно должен быть нулевой символ. Если строка `dest` имеет недостаточный размер, то произойдет переполнение буфера. Прототип функции:

```
#include <string.h>
char *strncat(char *dest, const char *source, size_t count);
```

Пример использования функции:

```
char str[20] = "ст";
strncat(str, "ром", 2);
strncat(str, "какао", 2);
printf("%s\n", str); // строка
```

Вместо функции `strncat()` лучше использовать функцию `strncat_s()`. Прототип функции:

```
#include <string.h>
errno_t strncat_s(char *dest, size_t sizeInBytes,
                  const char *source, size_t maxCount);
```

Функция `strncat_s()` копирует первые `maxCount` символов из строки `source` в конец строки `dest`. В параметре `sizeInBytes` указывается максимальное количество элементов массива `dest`. Пример:

```
#define SIZE 20
char str[SIZE] = "ст";
strncat_s(str, SIZE, "ром", 2);
strncat_s(str, SIZE, "какао", 2);
printf("%s\n", str); // строка
```

- `_strdup()` — создает копию строки в динамической памяти и возвращает указатель на нее. Прототип функции:

```
#include <string.h>
char *_strdup(const char *str);
```

Память выделяется с помощью функции `malloc()`, поэтому после использования строки следует освободить память с помощью функции `free()`. Пример:

```
// #include <stdlib.h>
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка", *p = NULL;
p = _strdup(str);
if (p) {
   printf("%s\n", p); // строка
   free(p);
}
```

- `strtok()` — разделяет С-строку `str` на подстроки, используя в качестве разделителей символы из С-строки `delim`. При первом вызове указываются оба параметра. В качестве значения возвращается указатель на первую подстроку или нулевой указатель, если символы-разделители не найдены в С-строке `str`. Для того чтобы получить последующие подстроки, необходимо каждый раз вызывать функцию `strtok()`, указывая в первом параметре нулевой указатель, а во втором параметре те же самые символы-разделители. Обратите внимание: функция изменяет исходную строку `str`, вставляя вместо символов-разделителей нулевой символ. Прототип функции:

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

Пример использования функции:

```
char str[] = "a,b.c=d", *p = NULL;
p = strtok(str, ",.=");
while (p) {
   printf("%s-", p);
   p = strtok(NULL, ",.=");
} // Результат: a-b-c-d-
printf("\n");
// Функция изменяет исходную строку!!!
for (int i = 0, len = (int)sizeof(str); i < len; ++i) {
   printf("%d ", str[i]);
} // 97 0 98 0 99 0 100 0
printf("\n");
```

Вместо функции `strtok()` лучше использовать функцию `strtok_s()`. Прототип функции:

```
#include <string.h>
char *strtok_s(char *str, const char *delim, char **context);
```

Первые два параметра аналогичны параметрам функции `strtok()`. В параметре `context` передается адрес указателя. Обратите внимание: функция изменяет исходную строку `str`, вставляя вместо символов-разделителей нулевой символ. Пример:

```
char str[] = "a b c,d", *p = NULL, *context = NULL;
p = strtok_s(str, " ,", &context);
while (p) {
   printf("%s-", p);
   p = strtok_s(NULL, " ,", &context);
} // Результат: a-b-c-d-
printf("\n");
// Функция изменяет исходную строку!!!
for (int i = 0, len = (int)sizeof(str); i < len; ++i) {
   printf("%d ", str[i]);
} // 97 0 98 0 99 0 100 0
printf("\n");
```

Практически все функции без суффикса _s, которые мы рассмотрели в этом разделе, в Visual C выводят предупреждающее сообщение `warning C4996`, т. к. по умолчанию функции не производят никакой проверки корректности данных. В этом случае возможно переполнение буфера. Забота о корректности данных полностью лежит на плечах программиста. Если вы уверены в своих действиях, то можно подавить вывод предупреждающих сообщений. Сделать это можно двумя способами:

- определить макрос с названием `_CRT_SECURE_NO_WARNINGS` в самом начале программы, перед подключением заголовочных файлов. Пример:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
```

- вставить прагму `warning`. Пример:

```
#pragma warning( disable : 4996 )
```

> ***Обратите внимание***
>
> После директив `#define` и `#pragma` точка с запятой не указывается.

7.10. Поиск и замена в С-строке

Для поиска и замены в С-строке предназначены следующие функции.

- `strchr()` — ищет в С-строке `str` первое вхождение символа `ch`. В качестве значения функция возвращает указатель на первый найденный символ в С-строке `str` или нулевой указатель, если символ не найден. Прототип функции:

```
#include <string.h>
char *strchr(const char *str, int ch);
```

Пример использования функции:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка строка", *p = NULL;
p = strchr(str, 'к');
```

```
if (p) {
   printf("Индекс: %d\n", (int)(p - str));
} // Индекс: 4
```

- `strrchr()` — **ищет в С-строке** `str` **последнее вхождение символа** `ch`. **В качестве значения функция возвращает указатель на символ или нулевой указатель, если символ не найден. Прототип функции:**

```
#include <string.h>
char *strrchr(const char *str, int ch);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка строка", *p = NULL;
p = strrchr(str, 'к');
if (p) {
   printf("Индекс: %d\n", (int)(p - str));
} // Индекс: 11
```

- `memchr()` — **ищет в строке** `str` **первое вхождение символа** `ch`. **Максимально просматривается** `maxCount` **символов. В качестве значения функция возвращает указатель на первый найденный символ в строке** `str` **или нулевой указатель, если символ не найден. Прототип функции:**

```
#include <string.h>
void *memchr(const void *str, int ch, size_t maxCount);
```

Пример:

```
char str[] = "строка строка", *p = NULL;
p = memchr(str, 'к', strlen(str));
if (p) {
   printf("Индекс: %d\n", (int)(p - str));
} // Индекс: 4
```

- `strpbrk()` — **ищет в С-строке** `str` **первое вхождение одного из символов (нулевой символ не учитывается), входящих в С-строку** `control`. **В качестве значения функция возвращает указатель на первый найденный символ или нулевой указатель, если символы не найдены. Прототип функции:**

```
#include <string.h>
char *strpbrk(const char *str, const char *control);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка строка", *p = NULL;
p = strpbrk(str, "кр");
if (p) {
   printf("Индекс: %d\n", (int)(p - str));
} // Индекс: 2 (индекс первой буквы "р")
```

- `strcspn()` — возвращает индекс первого символа в С-строке `str`, который совпадает с одним из символов, входящих в С-строку `control`. Прототип функции:

```
#include <string.h>
size_t strcspn(const char *str, const char *control);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка строка";
int index = (int)strcspn(str, "кр");
printf("Индекс: %d\n", index);
// Индекс: 2 (индекс первой буквы "р")
index = (int)strcspn(str, "ф");
printf("Индекс: %d\n", index);
// Индекс: 13 (длина строки, если ничего не найдено)
```

- `strspn()` — возвращает индекс первого символа в С-строке `str`, который не совпадает ни с одним из символов, входящих в С-строку `control`. Прототип функции:

```
#include <string.h>
size_t strspn(const char *str, const char *control);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка строка";
int index = (int)strspn(str, "окстр");
printf("Индекс: %d\n", index);
// Индекс: 5 ("а" не входит в "окстр")
```

- `strstr()` — ищет в С-строке `str` первое вхождение целого фрагмента из С-строки `subStr`. В качестве значения функция возвращает указатель на первое вхождение фрагмента в строку или нулевой указатель — в противном случае. Прототип функции:

```
#include <string.h>
char *strstr(const char *str, const char *subStr);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка строка", *p = NULL;
p = strstr(str, "ока");
if (p) {
   printf("Индекс: %d\n", (int)(p - str));
} // Индекс: 3
```

- `_strset()` — заменяет все символы в С-строке `str` указанным символом `ch`. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <string.h>
char *_strset(char *str, int ch);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка";
_strset(str, '*');
printf("'%s'\n", str); // '******'
```

Вместо функции `_strset()` лучше использовать функцию `_strset_s()`. Прототип функции:

```
#include <string.h>
errno_t _strset_s(char *str, size_t strSize, int ch);
```

В параметре `strSize` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[10] = "строка";
_strset_s(str, 10, '*');
printf("'%s'\n", str); // '******'
```

- `_strnset()` — заменяет первые `count` символов в С-строке `str` указанным символом `ch`. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <string.h>
char *_strnset(char *str, int ch, size_t count);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка";
_strnset(str, '*', 5);
printf("'%s'\n", str); // '*****а'
```

Вместо функции `_strnset()` лучше использовать функцию `_strnset_s()`. Прототип функции:

```
#include <string.h>
errno_t _strnset_s(char *str, size_t strSize, int ch, size_t count);
```

В параметре `strSize` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[10] = "строка";
_strnset_s(str, 10, '*', 4);
printf("'%s'\n", str); // '****ка'
```

- `memset()` — заменяет первые `count` элементов массива `dest` символом `ch`. В качестве значения функция возвращает указатель на массив `dest`. Прототип функции:

```
#include <string.h>
void *memset(void *dest, int ch, size_t count);
```

Пример:

```
char str[] = "String";
memset(str, '*', 4);
printf("'%s'\n", str); // '****ng'
```

- `_strrev()` — **меняет порядок следования символов внутри С-строки на противоположный, как бы переворачивает строку. Функция возвращает указатель на строку** `str`. **Прототип функции:**

```
#include <string.h>
char *_strrev(char *str);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка";
_strrev(str);
printf("%s\n", str); // акортс
```

- `_strupr()` — **заменяет все символы в С-строке** `str` **соответствующими прописными буквами. Функция возвращает указатель на строку** `str`. **Прототип функции:**

```
#include <string.h>
char *_strupr(char *str);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
_strupr(str);
printf("%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

Вместо функции `_strupr()` **лучше использовать функцию** `_strupr_s()`. **Прототип функции:**

```
#include <string.h>
errno_t _strupr_s(char *str, size_t strSize);
```

В параметре `strSize` **указывается размер строки. Функция возвращает значение** `0`, **если операция успешно выполнена, или код ошибки. Пример:**

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[40] = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
_strupr_s(str, 40);
printf("%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

- `_strlwr()` — **заменяет все символы в С-строке** `str` **соответствующими строчными буквами. Функция возвращает указатель на строку** `str`. **Прототип функции:**

```
#include <string.h>
char *_strlwr(char *str);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
_strlwr(str);
printf("%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

Вместо функции `_strlwr()` лучше использовать функцию `_strlwr_s()`. Прототип функции:

```
#include <string.h>
errno_t _strlwr_s(char *str, size_t strSize);
```

В параметре `strSize` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[40] = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
_strlwr_s(str, 40);
printf("%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

7.11. Сравнение С-строк

Для сравнения С-строк предназначены следующие функции.

- `strcmp()` — сравнивает С-строку `str1` с С-строкой `str2` без учета настроек локали и возвращает одно из значений:
  - отрицательное число — если `str1` меньше `str2`;
  - `0` — если строки равны;
  - положительное число — если `str1` больше `str2`.

  Сравнение производится с учетом регистра символов. Прототип функции:

```
#include <string.h>
int strcmp(const char *str1, const char *str2);
```

  Пример:

```
char str1[] = "абв", str2[] = "абв", str3[] = "АБВ";
printf("%d\n", strcmp(str1, str2)); // 0
printf("%d\n", strcmp(str1, str3)); // 1
str1[2] = 'б'; // str1[] = "абб", str2[] = "абв";
printf("%d\n", strcmp(str1, str2)); // -1
str1[2] = 'г'; // str1[] = "абг", str2[] = "абв";
printf("%d\n", strcmp(str1, str2)); // 1
```

- `strncmp()` — сравнивает `count` первых символов в С-строках `str1` и `str2`. Если нулевой байт встретится раньше, то значение параметра `count` игнорируется. Функция возвращает одно из значений:

- отрицательное число — если `str1` меньше `str2`;
- `0` — если строки равны;
- положительное число — если `str1` больше `str2`.

Сравнение производится с учетом регистра символов. Прототип функции:

```
#include <string.h>
int strncmp(const char *str1, const char *str2, size_t count);
```

Пример:

```
char str1[] = "абв", str2[] = "абг";
printf("%d\n", strncmp(str1, str2, 2)); // 0
printf("%d\n", strncmp(str1, str2, 3)); // -1
```

- `strcoll()` — функция аналогична функции `strcmp()`, но сравнение производится с учетом значения `LC_COLLATE` в текущей локали. Например, буква "ё" в диапазон между буквами "е" и "ж" не попадает, т. к. буква "ё" в кодировке windows-125 имеет код `-72`, буква "е" — `-27`, а буква "ж" — `-26`. Если сравнение производится с помощью функции `strcmp()`, то буква "е" будет больше буквы "ё" (`-27 > -72`). При использовании функции `strcoll()` с настройкой локали `Russian_Russia.125` буква "ё" попадет в диапазон между буквами "е" и "ж", т. к. используются локальные настройки по алфавиту. Если локаль не настроена, то эта функция эквивалентна функции `strcmp()`. Сравнение производится с учетом регистра символов. Можно также воспользоваться функцией `_strcoll_l()`, которая позволяет задать локаль в третьем параметре. Прототипы функций:

```
#include <string.h>
int strcoll(const char *str1, const char *str2);
int _strcoll_l(const char *str1, const char *str2, _locale_t locale);
```

Пример:

```
char str1[] = "е", str2[] = "ё";
printf("%d\n", str1[0]);              // -27 (е)
printf("%d\n", str2[0]);              // -72 (ё)
printf("%d\n", strcmp(str1, str2));  // 1
printf("%d\n", strcoll(str1, str2)); // 1
// Без настройки локали: е (код -27) больше ё (код -72)
// Настройка локали
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", strcmp(str1, str2));  // 1
printf("%d\n", strcoll(str1, str2)); // -1
// После настройки локали: е меньше ё
```

- `_strncoll()` и `_strncoll_l()` — сравнивают `maxCount` первых символов в С-строках `str1` и `str2`. Сравнение производится с учетом регистра символов. Функции аналогичны функции `strncmp()`, но сравнение выполняется с учетом значения `LC_COLLATE` в текущей локали. Прототипы функций:

```
#include <string.h>
int _strncoll(const char *str1, const char *str2, size_t maxCount);
int _strncoll_l(const char *str1, const char *str2,
                size_t maxCount, _locale_t locale);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str1[] = "абв", str2[] = "абг";
printf("%d\n", _strncoll(str1, str2, 2)); // 0
printf("%d\n", _strncoll(str1, str2, 3)); // -1
```

- `strxfrm()` — **преобразует С-строку** `source` **в строку специального формата и записывает ее в** `dest`. **В конец вставляется нулевой символ. Записывается не более** `maxCount` **символов. Если количество символов** `maxCount` **меньше необходимого количества символов после преобразования, то содержимое** `dest` **не определено. В качестве значения функция возвращает число необходимых символов. Для того чтобы просто получить количество необходимых символов (без учета нулевого символа), в параметре** `maxCount` **указывается число** `0`, **а в параметре** `dest` **передается нулевой указатель. Можно также воспользоваться функцией** `_strxfrm_l()`, **которая дополнительно позволяет задать локаль в последнем параметре. Прототипы функций:**

```
#include <string.h>
size_t strxfrm(char *dest, const char *source, size_t maxCount);
size_t _strxfrm_l(char *dest, const char *source,
                  size_t maxCount, _locale_t locale);
```

Функция `strcoll()`, **прежде чем произвести сравнение, неявно выполняет преобразование переданных строк в строки специального формата. Функция** `strxfrm()` **позволяет произвести такое преобразование явным образом. Строка преобразуется с учетом значения** `LC_COLLATE` **в текущей локали. Если локаль не настроена, то просто выполняется копирование. В дальнейшем строки специального формата можно сравнивать с помощью функции** `strcmp()`. **Результат сравнения будет соответствовать результату сравнения с помощью функции** `strcoll()`. **Функцию** `strxfrm()` **следует использовать, если сравнение строк производится многократно. Пример:**

```
setlocale(LC_ALL, "Russian_Russia.1251");
int x = 0;
char str1[] = "е", str2[] = "ё";
char buf1[10] = {0}, buf2[10] = {0};
x = (int)strxfrm(NULL, str1, 0);
printf("%d\n", x);                    //  6 (нужен буфер 6 + 1)
strxfrm(buf1, str1, 10);
strxfrm(buf2, str2, 10);
printf("%d\n", strcmp(str1, str2));   //  1 ("е" больше "ё")
printf("%d\n", strcoll(str1, str2)); // -1 ("е" меньше "ё")
printf("%d\n", strcmp(buf1, buf2));   // -1 ("е" меньше "е")
```

- `_stricmp()` и `_stricmp_l()` — **сравнивают С-строки без учета регистра символов. Функции учитывают настройки локали для категории** `LC_CTYPE`. **Для русских букв необходимо настроить локаль. Прототипы функций:**

```
#include <string.h>
int _stricmp(const char *str1, const char *str2);
int _stricmp_l(const char *str1, const char *str2, _locale_t locale);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str1[] = "абв", str2[] = "АБВ";
printf("%d\n", _stricmp(str1, str2)); // 0
```

- `_strnicmp()` и `_strnicmp_l()` — **сравнивают** `maxCount` **первых символов в С-строках** `str1` **и** `str2` **без учета регистра символов. Функции учитывают настройки локали для категории** `LC_CTYPE`. **Для русских букв необходимо настроить локаль. Прототипы функций:**

```
#include <string.h>
int _strnicmp(const char *str1, const char *str2, size_t maxCount);
int _strnicmp_l(const char *str1, const char *str2,
                size_t maxCount, _locale_t locale);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char  str1[] = "абве", str2[] = "АБВЖ";
printf("%d\n", _strnicmp(str1, str2, 3)); // 0
printf("%d\n", _strnicmp(str1, str2, 4)); // -1
```

- `_stricoll()` и `_stricoll_l()` — **сравнивают С-строки без учета регистра символов. Функции учитывают настройки локали для категорий** `LC_CTYPE` **и** `LC_COLLATE`. **Для русских букв необходимо настроить локаль. Прототипы функций:**

```
#include <string.h>
int _stricoll(const char *str1, const char *str2);
int _stricoll_l(const char *str1, const char *str2, _locale_t locale);
```

Пример:

```
char str1[] = "абв", str2[] = "АБВ";
printf("%d\n", _stricoll(str1, str2));          // 32
_locale_t locale = _create_locale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", _stricoll_l(str1, str2, locale)); // 0
_free_locale(locale);
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%d\n", _stricoll(str1, str2));          // 0
```

- `_strnicoll()` и `_strnicoll_l()` — **сравнивают** `maxCount` **первых символов в С-строках** `str1` **и** `str2` **без учета регистра символов. Функции учитывают настройки локали для категорий** `LC_CTYPE` **и** `LC_COLLATE`. **Для русских букв необходимо настроить локаль. Прототипы функций:**

```
#include <string.h>
int _strnicoll(const char *str1, const char *str2, size_t maxCount);
int _strnicoll_l(const char *str1, const char *str2,
                 size_t maxCount, _locale_t locale);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str1[] = "абвг", str2[] = "АБВД";
printf("%d\n", _strnicoll(str1, str2, 3)); // 0
printf("%d\n", _strnicoll(str1, str2, 4)); // -1
```

Для сравнения строк можно также использовать функцию `memcmp()`. **Функция** `memcmp()` **сравнивает первые** `size` **байтов массивов** `buf1` **и** `buf2`. **В качестве значения функция возвращает:**

- **отрицательное число — если** `buf1` **меньше** `buf2`;
- `0` **— если массивы равны;**
- **положительное число — если** `buf1` **больше** `buf2`.

Прототип функции:

```
#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t size);
```

Пример:

```
char str1[] = "abc", str2[] = "abc";
int x = memcmp(str1, str2, sizeof str2);
printf("%d\n", x);                          // 0
str1[2] = 'b';
x = memcmp(str1, str2, sizeof str2);
printf("%d\n", x);                          // -1
str1[2] = 'd';
x = memcmp(str1, str2, sizeof str2);
printf("%d\n", x);                          // 1
```

Функция `memcmp()` **производит сравнение с учетом регистра символов. Если необходимо произвести сравнение без учета символов, то можно воспользоваться функциями** `_memicmp()` **и** `_memicmp_l()`. **Для сравнения русских букв следует настроить локаль. Прототипы функций:**

```
#include <string.h>
int _memicmp(const void *buf1, const void *buf2, size_t size);
int _memicmp_l(const void *buf1, const void *buf2, size_t size,
               _locale_t locale);
```

Предназначение параметров и возвращаемое значение такое же, как у функции `memcmp()`. **Функция** `_memicmp_l()` **позволяет дополнительно задать локаль. Пример использования функций:**

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str1[] = "абв", str2[] = "АБВ";
int x = _memicmp(str1, str2, sizeof str2);
printf("%d\n", x);                  // 0
x = memcmp(str1, str2, sizeof str2);
printf("%d\n", x);                  // 1
_locale_t locale = _create_locale(LC_ALL, "Russian_Russia.1251");
x = _memicmp_l(str1, str2, sizeof str2, locale);
printf("%d\n", x);                  // 0
_free_locale(locale);
```

7.12. Форматирование С-строк

Выполнить форматирование С-строки, а также преобразовать значения элементарных типов в С-строку можно с помощью функции `sprintf()`. Существует также функция `_sprintf_l()`, которая позволяет дополнительно задать локаль. Прототипы функций:

```
#include <stdio.h>
int sprintf(char *buf, const char *format, ...);
int _sprintf_l(char *buf, const char *format, _locale_t locale, ...);
```

В параметре `format` указывается строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа `%`. Спецификаторы формата совпадают со спецификаторами, используемыми в функции `printf()` *(см. разд. 2.8)*. Вместо спецификаторов формата подставляются значения, указанные в качестве параметров. Количество спецификаторов должно совпадать с количеством переданных параметров. Результат записывается в буфер, адрес которого передается в первом параметре (`buf`). В качестве значения функция возвращает количество символов, записанных в символьный массив. Пример:

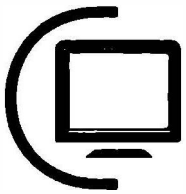
```
setlocale(LC_ALL, "Russian_Russia.1251");
char buf[50] = "";
int x = 100, count = 0;
count = sprintf(buf, "x = %d", x);
printf("%s\n", buf);                // x = 100
printf("%d\n", count);              // 7
```

Функция `sprintf()` не производит никакой проверки размера буфера, поэтому возможно переполнение буфера. Вместо функции `sprintf()` следует использовать функцию `sprintf_s()` или `_sprintf_s_l()`. Прототипы функций:

```
#include <stdio.h>
int sprintf_s(char *buf, size_t sizeInBytes, const char *format, ...);
int _sprintf_s_l(char *buf, size_t sizeInBytes, const char *format,
                 _locale_t locale, ...);
```

Параметры `buf` **и** `format` **аналогичны параметрам функции** `sprintf()`. **В параметре** `sizeInBytes` **указывается размер буфера. В качестве значения функции возвращают количество символов, записанных в символьный массив. Пример:**

```
setlocale(LC_ALL, "Russian_Russia.1251");
char buf[50] = "";
int count = 0;
double pi = 3.14159265359;
count = sprintf_s(buf, 50, "pi = %.2f", pi);
printf("%s\n", buf);                // pi = 3,14
printf("%d\n", count);              // 9
```



ГЛАВА 8

Широкие символы и L-строки

Помимо обычных символов, имеющих тип `char`, язык C поддерживает *широкие символы*, имеющие тип `wchar_t`. Строки, состоящие из широких символов, мы будем называть *L-строками*.

Прежде чем мы начнем рассматривать широкие символы и L-строки, попробуйте запустить этот код и сравните результат:

```
// #include <stdio.h>
// #include <string.h>
wchar_t str[] = L"строка string";
for (int i = 0, len = (int)wcslen(str); i < len; ++i) {
   printf("%u ", str[i]);
}
printf("\n");
// 1089 1090 1088 1086 1082 1072 32 115 116 114 105 110 103
```

Если получили точно такие же числа, то можно начать изучение. Если же числа отличаются, то следует проверить кодировку файла с программой — она должна быть windows-1251. Хотя на самом деле кодировка файлов по умолчанию в MinGW должна быть UTF-8, но в этом случае вы не сможете работать с обычными строками, т. к. русские буквы в кодировке UTF-8 кодируются двумя байтами, а не одним. Для того чтобы избежать проблем с русскими буквами, для файлов мы используем кодировку windows-1251, а не UTF-8.

Если при компиляции получили эту ошибку:

```
error: converting to execution character set: Illegal byte sequence
```

то следует явным образом указать кодировку файла с программой и кодировку символов в C-строке. Для этого в MinGW предназначены следующие флаги:

- `-finput-charset` — задает кодировку файла с программой;
- `-fexec-charset` — определяет кодировку символов в C-строке;
- `-fwide-exec-charset` — задает кодировку символов в L-строке.

Команда компиляции в командной строке должна выглядеть так:

```
C:\book>gcc -Wall -Wconversion -O3 -finput-charset=cp1251
            -fexec-charset=cp1251 -o test.exe test.c

C:\book>test.exe
1089 1090 1088 1086 1082 1072 32 115 116 114 105 110 103
```

Если получили следующую ошибку:

```
cc1.exe: error: no iconv implementation, cannot convert from cp1251 to
UTF-8
```

то нужно либо сменить компилятор, т. к. он собран без поддержки библиотеки iconv, либо использовать для файлов кодировку UTF-8 и не указывать флаги `-finput-charset` и `-fexec-charset` при компиляции.

Для того чтобы задать флаги в Eclipse, в свойствах проекта из списка слева выбираем пункт **C/C++ Build | Settings**. На отобразившейся вкладке из списка **Configuration** выбираем пункт **All configurations**, а затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Miscellaneous** (см. рис. 1.35). В поле **Other flags** через пробел к имеющемуся значению добавляем следующие инструкции:

```
-finput-charset=cp1251 -fexec-charset=cp1251
```

Содержимое поля **Other flags** после изменения должно быть таким:

```
-c -fmessage-length=0 -finput-charset=cp1251 -fexec-charset=cp1251
```

Эти флаги должны быть в окне **Console** при компиляции в Eclipse:

```
07:22:28 **** Incremental Build of configuration Debug for project
Test64c ****
Info: Internal Builder is used for build
gcc -O0 -g3 -Wall -Wconversion -c -fmessage-length=0
-finput-charset=cp1251 -fexec-charset=cp1251 -o "src\\Test64c.o"
"..\\src\\Test64c.c"
gcc -o Test64c.exe "src\\Test64c.o"

07:22:29 Build Finished. 0 errors, 0 warnings. (took 266ms)
```

8.1. Объявление и инициализация широкого символа

Широкие символы объявляются с помощью типа `wchar_t`:

```
typedef unsigned short wchar_t;
```

Диапазон допустимых значений содержится в макросах `WCHAR_MIN` и `WCHAR_MAX`:

```
// #include <stdint.h> или #include <wchar.h>
printf("%u\n", WCHAR_MIN);              // 0
printf("%u\n", WCHAR_MAX);              // 65535
```

Обратите внимание: не следует рассчитывать, что при использовании другого компилятора тип `wchar_t` будет объявлен точно так же. Для определения размера типа `wchar_t` всегда используйте оператор `sizeof`:

```
printf("%d\n", (int) sizeof(wchar_t)); // 2
```

Кроме того, существует тип `wint_t`:

```
typedef unsigned short wint_t;
```

Диапазон допустимых значений содержится в макросах `WINT_MIN` и `WINT_MAX`:

```
printf("%d\n", (int) sizeof(wint_t)); // 2
// #include <stdint.h>
printf("%u\n", WINT_MIN);             // 0
printf("%u\n", WINT_MAX);             // 65535
```

Переменной, имеющей тип `wchar_t`, можно присвоить числовое значение (код символа) или указать символ внутри апострофов, перед которыми добавлена буква `L`. Внутри апострофов можно указать специальные символы, например, `\n`, `\t` и др. Пример объявления и инициализации широкого символа:

```
wchar_t ch1 = 1087, ch2 = L'п', ch3 = L'\u043F';
printf("%u\n", ch1); // 1087
printf("%u\n", ch2); // 1087
printf("%u\n", ch3); // 1087
```

Коды первых 128 символов совпадают с кодами символов в кодировке ASCII (см. табл. 7.1). Коды русских букв приведены в табл. 8.1. Обратите внимание на то, что коды букв "ё" и "Ё" выпадают из последовательности кодов.

***Таблица 8.1.** Коды русских букв при использовании широких символов*

Буква	Код	Unicode	Буква	Код	Unicode
А	1040	\u0410	а	1072	\u0430
Б	1041	\u0411	б	1073	\u0431
В	1042	\u0412	в	1074	\u0432
Г	1043	\u0413	г	1075	\u0433
Д	1044	\u0414	д	1076	\u0434
Е	1045	\u0415	е	1077	\u0435
Ё	1025	\u0401	ё	1105	\u0451
Ж	1046	\u0416	ж	1078	\u0436
З	1047	\u0417	з	1079	\u0437
И	1048	\u0418	и	1080	\u0438
Й	1049	\u0419	й	1081	\u0439
К	1050	\u041A	к	1082	\u043A
Л	1051	\u041B	л	1083	\u043B

Таблица 8.1 (окончание)

Буква	Код	Unicode	Буква	Код	Unicode
М	1052	\u041C	м	1084	\u043C
Н	1053	\u041D	н	1085	\u043D
О	1054	\u041E	о	1086	\u043E
П	1055	\u041F	п	1087	\u043F
Р	1056	\u0420	р	1088	\u0440
С	1057	\u0421	с	1089	\u0441
Т	1058	\u0422	т	1090	\u0442
У	1059	\u0423	у	1091	\u0443
Ф	1060	\u0424	ф	1092	\u0444
Х	1061	\u0425	х	1093	\u0445
Ц	1062	\u0426	ц	1094	\u0446
Ч	1063	\u0427	ч	1095	\u0447
Ш	1064	\u0428	ш	1096	\u0448
Щ	1065	\u0429	щ	1097	\u0449
Ъ	1066	\u042A	ъ	1098	\u044A
Ы	1067	\u042B	ы	1099	\u044B
Ь	1068	\u042C	ь	1100	\u044C
Э	1069	\u042D	э	1101	\u044D
Ю	1070	\u042E	ю	1102	\u044E
Я	1071	\u042F	я	1103	\u044F

8.2. Вывод и ввод широких символов

Прежде чем выводить широкие символы, нужно настроить локаль с помощью функции `setlocale()`:

```
// #include <locale.h>
setlocale(LC_ALL, "Russian_Russia.1251");
```

или с помощью функции `_wsetlocale()`:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
```

Для вывода широких символов нужно использовать функцию `putwchar()`. Прототип функции:

```
#include <wchar.h> /* или #include <stdio.h> */
wint_t putwchar(wchar_t ch);
```

Пример вывода символа:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t ch = L'п';
putwchar(ch);          // п
```

Можно также воспользоваться спецификатором `%c` **в строке формата функции** `wprintf()` **или спецификаторами** `%lc` **и** `%C` **в функции** `printf()`:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t ch = L'п';
wprintf(L"%c\n", ch); // п
printf("%lc\n", ch);  // п
printf("%C\n", ch);   // п
```

Для ввода широкого символа предназначена функция `getwchar()`. **Прототип функции:**

```
#include <wchar.h> /* или #include <stdio.h> */
wint_t getwchar(void);
```

Пример:

```
_wsystem(L"chcp 1251");    // Смена кодировки консоли
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t ch;
printf("ch = ");           // Вывод подсказки
fflush(stdout);            // Сброс буфера
ch = getwchar();           // Получение символа
printf("%u\n", ch);        // Вывод кода
wprintf(L"%c\n", ch);      // Вывод символа
```

Функция `getwchar()` **позволяет получить символ только после нажатия клавиши <Enter>. Если необходимо получить символ сразу после нажатия клавиши на клавиатуре, то можно воспользоваться функциями** `_getwche()` **и** `_getwch()`. **Прототипы функций:**

```
#include <wchar.h> /* или #include <conio.h> */
wint_t _getwche(void);
wint_t _getwch(void);
```

Функция `_getwche()` **возвращает код символа и выводит его на экран. При нажатии клавиши на клавиатуре функция** `_getwch()` **возвращает код символа, но сам символ на экран не выводится. Это обстоятельство позволяет использовать функцию** `_getwch()` **для получения конфиденциальных данных (например, пароля). Следует учитывать, что код символа возвращается и при нажатии некоторых служебных клавиш, например <Home>, <End> и др. Пример:**

```
_wsystem(L"chcp 1251"); // Смена кодировки консоли
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t ch;
wprintf(L"Введите символ: ");
fflush(stdout);
```

```
ch = _getwch();
wprintf(L"код символа: %u\n", ch);
```

Для ввода широких символов и других данных предназначена функция `wscanf()`. **Прототип функции:**

```
#include <wchar.h> /* или #include <stdio.h> */
int wscanf(const wchar_t *format, ...);
```

В первом параметре указывается строка специального формата, внутри которой задаются спецификаторы, аналогичные применяемым в функции `printf()`, **а также некоторые дополнительные спецификаторы. В последующих параметрах передаются адреса переменных. Функция возвращает количество произведенных присваиваний. Пример получения символа:**

```
_wsystem(L"chcp 1251");          // Смена кодировки консоли
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t ch;
wprintf(L"Введите символ: ");
fflush(stdout);                  // Сброс буфера вывода
fflush(stdin);                   // Очистка буфера ввода
int status = wscanf(L"%c", &ch); // Символ & обязателен!!!
if (status == 1) {
   wprintf(L"Вы ввели: %c\n", ch);
   wprintf(L"Код: %u\n", ch);
}
else {
   wprintf(L"Ошибка при вводе символа\n");
}
wprintf(L"status = %d\n", status);
```

8.3. Изменение регистра символов

Для изменения регистра символов предназначены следующие функции.

- `towupper()` — **возвращает код символа в верхнем регистре. Если преобразования регистра не было, то код символа возвращается без изменений. Прототип функции:**

  ```
  #include <wchar.h> /* или #include <ctype.h> */
  wint_t towupper(wint_t ch);
  ```

  Пример:

  ```
  _wsetlocale(LC_ALL, L"Russian_Russia.1251");
  wprintf(L"%c\n", towupper(L'w'));          // W
  wprintf(L"%c\n", towupper(L'б'));          // Б
  wchar_t str[] = L"абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
  for (int i = 0, len = (int)wcslen(str); i < len; ++i) {
     str[i] = towupper(str[i]);
  }
  ```

```
wprintf(L"%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

- `towlower()` — возвращает код символа в нижнем регистре. Если преобразования регистра не было, то код символа возвращается без изменений. Прототип функции:

```
#include <wchar.h> /* или #include <ctype.h> */
wint_t towlower(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%c\n", towlower(L'W'));          // w
wprintf(L"%c\n", towlower(L'Б'));          // б
wchar_t str[] = L"АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
for (int i = 0, len = (int)wcslen(str); i < len; ++i) {
   str[i] = towlower(str[i]);
}
wprintf(L"%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

- `_wcsupr()` — заменяет все буквы в L-строке `str` соответствующими прописными буквами. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *_wcsupr(wchar_t *str);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
_wcsupr(str);
wprintf(L"%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

Вместо функции `_wcsupr()` лучше использовать функцию `_wcsupr_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
errno_t _wcsupr_s(wchar_t *str, size_t sizeInWords);
```

В параметре `sizeInWords` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[40] = L"абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
_wcsupr_s(str, 40);
wprintf(L"%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

- `_wcslwr()` — заменяет все буквы в L-строке `str` соответствующими строчными буквами. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *_wcslwr(wchar_t *str);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
_wcslwr(str);
wprintf(L"%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

Вместо функции `_wcslwr()` лучше использовать функцию `_wcslwr_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
errno_t _wcslwr_s(wchar_t *str, size_t sizeInWords);
```

В параметре `sizeInWords` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[40] = L"АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
_wcslwr_s(str, 40);
wprintf(L"%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

8.4. Проверка типа содержимого широкого символа

Для проверки типа содержимого широкого символа предназначены следующие функции.

- `iswdigit()` — возвращает ненулевое значение, если символ является десятичной цифрой, в противном случае — `0`. Прототип функции:

```
#include <wchar.h> /* или #include <ctype.h> */
int iswdigit(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswdigit(L'w'));          // 0
wprintf(L"%d\n", iswdigit(L'2'));          // 4
wprintf(L"%d\n", iswdigit(L'б'));          // 0
```

- `iswxdigit()` — возвращает ненулевое значение, если символ является шестнадцатеричной цифрой (число от `0` до `9` или буква от `A` до `F`; регистр не имеет значения), в противном случае — `0`. Прототип функции:

```
#include <wchar.h> /* или #include <ctype.h> */
int iswxdigit(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswxdigit(L'8')); // 128
wprintf(L"%d\n", iswxdigit(L'a')); // 128
wprintf(L"%d\n", iswxdigit(L'F')); // 128
wprintf(L"%d\n", iswxdigit(L'g')); // 0
```

- `iswalpha()` — **возвращает ненулевое значение, если символ является буквой, в противном случае —** `0`. **Прототип функции:**

```
#include <wchar.h> /* или #include <ctype.h> */
int iswalpha(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswalpha(L'w'));              // 258
wprintf(L"%d\n", iswalpha(L'2'));              // 0
wprintf(L"%d\n", iswalpha(L'б'));              // 258
wprintf(L"%d\n", iswalpha(L'Б'));              // 257
```

- `iswspace()` — **возвращает ненулевое значение, если символ является пробельным символом (пробелом, табуляцией, переводом строки или возвратом каретки), в противном случае —** `0`. **Прототип функции:**

```
#include <wchar.h> /* или #include <ctype.h> */
int iswspace(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswspace(L'w'));              // 0
wprintf(L"%d\n", iswspace(L'\n'));             // 8
wprintf(L"%d\n", iswspace(L'\t'));             // 8
wprintf(L"%d\n", iswspace(L'б'));              // 0
```

- `iswalnum()` — **возвращает ненулевое значение, если символ является буквой или цифрой, в противном случае —** `0`. **Прототип функции:**

```
#include <wchar.h> /* или #include <ctype.h> */
int iswalnum(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswalnum(L'w'));              // 258
wprintf(L"%d\n", iswalnum(L'8'));              // 4
wprintf(L"%d\n", iswalnum(L'\t'));             // 0
wprintf(L"%d\n", iswalnum(L'б'));              // 258
```

- `iswlower()` — **возвращает ненулевое значение, если символ является буквой в нижнем регистре, в противном случае —** `0`. **Прототип функции:**

```
#include <wchar.h> /* или #include <ctype.h> */
int iswlower(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswlower(L'w'));              // 2
wprintf(L"%d\n", iswlower(L'8'));              // 0
wprintf(L"%d\n", iswlower(L'\t'));             // 0
wprintf(L"%d\n", iswlower(L'б'));              // 2
wprintf(L"%d\n", iswlower(L'Б'));              // 0
```

- `iswupper()` — **возвращает ненулевое значение, если символ является буквой в верхнем регистре, в противном случае** — `0`. **Прототип функции:**

```
#include <wchar.h> /* или #include <ctype.h> */
int iswupper(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswupper(L'W'));              // 1
wprintf(L"%d\n", iswupper(L'8'));              // 0
wprintf(L"%d\n", iswupper(L'\t'));             // 0
wprintf(L"%d\n", iswupper(L'б'));              // 0
wprintf(L"%d\n", iswupper(L'Б'));              // 1
```

- `iswpunct()` — **возвращает ненулевое значение, если символ является символом пунктуации, в противном случае** — `0`. **Прототип функции:**

```
#include <wchar.h> /* или #include <ctype.h> */
int iswpunct(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswpunct(L'8')); // 0
wprintf(L"%d\n", iswpunct(L'f')); // 0
wprintf(L"%d\n", iswpunct(L',')); // 16
wprintf(L"%d\n", iswpunct(L'.')); // 16
wprintf(L"%d\n", iswpunct(L' ')); // 0
```

- `iswprint()` — **возвращает ненулевое значение, если символ является печатаемым (включая пробел), в противном случае** — `0`. **Прототип функции:**

```
#include <wchar.h> /* или #include <ctype.h> */
int iswprint(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswprint(L'8'));              // 4
wprintf(L"%d\n", iswprint(L'\x5'));            // 0
wprintf(L"%d\n", iswprint(L' '));              // 64
wprintf(L"%d\n", iswprint(L'б'));              // 258
```

- `iswgraph()` — **возвращает ненулевое значение, если символ является печатаемым (пробел печатаемым не считается), в противном случае** — `0`. **Прототип функции:**

```
#include <wchar.h> /* или #include <ctype.h> */
int iswgraph(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswgraph(L'8'));                // 4
wprintf(L"%d\n", iswgraph(L'\x5'));              // 0
wprintf(L"%d\n", iswgraph(L' '));                // 0
wprintf(L"%d\n", iswgraph(L'б'));                // 258
```

- `iswcntrl()` — возвращает ненулевое значение, если символ является непечатаемым, в противном случае — `0`. Прототип функции:

```
#include <wchar.h> /* или #include <ctype.h> */
int iswcntrl(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswcntrl(L'8'));   // 0
wprintf(L"%d\n", iswcntrl(L'\x5')); // 32
wprintf(L"%d\n", iswcntrl(L' '));   // 0
```

- `iswblank()` — возвращает ненулевое значение, если символ является пробельным (пробел, горизонтальная табуляция и др.), в противном случае — `0`. Прототип функции:

```
#include <wchar.h> /* или #include <ctype.h> */
int iswblank(wint_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", iswblank(L'8'));   // 0
wprintf(L"%d\n", iswblank(L'\t'));  // 1
wprintf(L"%d\n", iswblank(L' '));   // 1
```

В качестве примера проверим регистр символа и выведем соответствующее сообщение, а затем преобразуем символ к верхнему регистру и выведем его в окно консоли (листинг 8.1).

Листинг 8.1. Пример работы с расширенными символами

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

int main(void) {
   _wsetlocale(LC_ALL, L"Russian_Russia.1251");
   wchar_t ch = L'п';
   if (iswlower(ch)) {
      wprintf(L"Строчная\n");
   }
```

```
    else {
        wprintf(L"Прописная\n");
    }                       // Результат: Строчная
    ch = towupper(ch);
    wprintf(L"%c\n", ch); // П
    return 0;
}
```

8.5. Преобразование широких символов в обычные и наоборот

Для преобразования широких символов (тип `wchar_t`) в обычные (тип `char`) и наоборот предназначены две функции.

- `btowc()` — **преобразует обычный символ в широкий. В случае ошибки функция возвращает значение макроопределения `WEOF` (`65 535`). Прототип функции:**

```
#include <wchar.h>
wint_t btowc(int ch);
#define WEOF (wint_t)(0xFFFF)
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char ch = 'п';
wchar_t wch = btowc((unsigned char)ch);
wprintf(L"%c\n", wch); // п
```

- `wctob()` — **преобразует широкий символ в обычный. В случае ошибки функция возвращает значение `-1`. Обратите внимание: значение `-1` также соответствует букве "я" в кодировке windows-1251. В Visual C дополнительно устанавливается `errno` соответствующим значению `EILSEQ` (равно `42`). В MinGW `errno` не устанавливается. Прототип функции:**

```
#include <wchar.h>
int wctob(wint_t ch);
#include <errno.h>
#define EILSEQ 42
```

Пример:

```
// #include <stdlib.h>
setlocale(LC_ALL, "Russian_Russia.1251");
wchar_t wch = L'п';
char ch = (char)wctob(wch);
printf("%c\n", ch);          // п
printf("%d\n", errno);       // 0
ch = (char)wctob(L'\u01CB');
printf("%d\n", ch);          // -1
printf("%d\n", errno);       // В MinGW: 0; в Visual C: 42
```

8.6. Объявление и инициализация L-строки

L-строки являются массивами, содержащими символы типа `wchar_t`. Последним символом L-строки является нулевой символ. Объявляется L-строка так же, как и массив элементов типа `wchar_t`:

```
wchar_t str[7];
```

При инициализации L-строки можно перечислить широкие символы внутри фигурных скобок или указать строку внутри двойных кавычек, перед которыми добавлена буква `L`:

```
wchar_t str1[7] = { L'т', L'е', L'к', L'с', L'т', L'\0' };
wchar_t str2[] = { L'\u0442', L'\u0435', L'\u043A',
                   L'\u0441', L'\u0442', L'\0' };
wchar_t str3[] = { 1090, 1077, 1082, 1089, 1090, 0 };
wchar_t str4[] = L"\u0442\u0435\u043A\u0441\u0442";
wchar_t str5[] = L"текст";
```

При использовании двойных кавычек следует учитывать, что длина строки на один символ больше, т. к. в конец будет автоматически вставлен нулевой символ. Если это не предусмотреть и объявить массив из шести элементов вместо семи, то это приведет к ошибке.

Обратите внимание на то, что присваивать строку в двойных кавычках можно только при инициализации. Попытка присвоить строку позже приведет к ошибке:

```
wchar_t str[7];
str = L"String"; // Ошибка!!!
```

В этом случае нужно воспользоваться функцией `wcscpy()` или `wcscpy_s()`:

```
wchar_t str[7];
wcscpy_s(str, 7, L"String");
wprintf(L"%s\n", str); // String
```

Внутри строки в двойных кавычках можно указывать специальные символы (например, `\n`, `\r` и др.), которые мы уже рассматривали в *разд. 7.1*. Если внутри строки встречается кавычка, то ее необходимо экранировать с помощью обратного слэша:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"Группа \"Кино\"\n";
wprintf(L"%s\n", str); // Группа "Кино"
```

Если внутри строки встречается обратный слэш, то его необходимо экранировать. Это обязательно следует учитывать при указании пути к файлу:

```
wchar_t str1[] = L"C:\\temp\\new\\file.txt"; // Правильно
wchar_t str2[] = L"C:\temp\new\file.txt";    // Неправильно!!!
wprintf(L"%s\n", str1); // C:\temp\new\file.txt
wprintf(L"%s\n", str2); // Строка с ошибками!!!
```

Обратите внимание на вторую строку. В этом пути присутствуют сразу три специальных символа: \t, \n и \f. После преобразования специальных символов путь будет выглядеть следующим образом:

```
C:<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```

Разместить L-строку при инициализации на нескольких строках нельзя. Переход на новую строку вызовет синтаксическую ошибку:

```
wchar_t str[] = L"string1
string2"; // Ошибка!!!
```

Для того чтобы разместить L-строку на нескольких строках, следует перед символом перевода строки указать символ \:

```
wchar_t str[] = L"string1 \
string2 \
string3"; // После символа \ не должно быть никаких символов
wprintf(L"%s\n", str); // string1 string2 string3
```

Кроме того, можно воспользоваться неявной *конкатенацией*. Если строки расположены подряд внутри одной инструкции, то они объединяются в одну большую строку. Пример:

```
wchar_t str[] = L"string1"
L"string2" L"string3";
wprintf(L"%s\n", str); //string1string2string3
```

8.7. Доступ к символам внутри L-строки

После определения L-строки в переменной сохраняется адрес первого символа. Иными словами, название переменной является указателем, который ссылается на первый символ строки. Поэтому доступ к символу в строке может осуществляться как по индексу (нумерация начинается с нуля), указанному внутри квадратных скобок, так и с использованием адресной арифметики. Например, следующие две инструкции вывода являются эквивалентными:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
wprintf(L"%c\n", str[1]);     // т
wprintf(L"%c\n", *(str + 1)); // т
```

Символ можно не только получить таким образом, но и изменить:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
str[0] = L'C';          // Изменение с помощью индекса
*(str + 1) = L'T';      // Изменение с помощью указателя
wprintf(L"%s\n", str); // СТрока
```

Обратите внимание на то, что отдельный символ указывается внутри апострофов, а не внутри кавычек. Если указать символ внутри кавычек, то вместо одного символа будет два: собственно сам символ плюс нулевой символ.

Объявить указатель и присвоить ему адрес строки можно следующим образом:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
wchar_t *p = NULL;
p = str;
*p = L'C';
++p;      // Перемещаем указатель на второй символ
*p = L'T';
wprintf(L"%s\n", str); // СТрока
```

Обратите внимание на то, что перед названием строки не указывается оператор &, т. к. название переменной содержит адрес первого символа. Если использовать оператор &, то необходимо дополнительно указать индекс внутри квадратных скобок:

```
p = &str[0]; // Эквивалентно: p = str;
```

При инициализации указателя ему можно присвоить строку. Такие строки нельзя изменять, поэтому обычно перед типом указывают ключевое слово `const`. Пример:

```
const wchar_t *str = L"String";
wprintf(L"%s\n", str); // String
```

8.8. Определение длины L-строки

Получить длину L-строки можно с помощью функции `wcslen()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
size_t wcslen(const wchar_t *str);
```

Функция `wcslen()` возвращает количество символов без учета нулевого символа. Тип `size_t` объявлен как беззнаковое целое:

```
typedef unsigned __int64 size_t;
#define __int64 long long
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
int len = (int)wcslen(str);
wprintf(L"%d\n", len);                          // 6
wprintf(L"%d\n", (int) sizeof(str));            // 14
wprintf(L"%d\n",
       (int) (sizeof(str) / sizeof(wchar_t))); // 7
wchar_t *p = str;
wprintf(L"%d\n", (int) sizeof(p));
// Значение в проекте Test64c: 8
wprintf(L"%s\n", str); // строка
```

Обратите внимание: длина L-строки и длина символьного массива — это разные вещи. Длина L-строки — это количество символов с начала массива до первого нулевого символа. Длина символьного массива — это общее количество символов, доступное под строку. Кроме того, следует учитывать, что один символ в L-строках занимает два байта, поэтому, чтобы получить количество символов в символьном массиве, нужно разделить размер массива на размер типа `wchar_t`. Не нужно свято верить, что размер типа `wchar_t` всегда равен двум байтам. Для определения размера обязательно используйте оператор `sizeof`.

8.9. Перебор символов L-строки

Для перебора символов удобно использовать цикл `for`. В первом параметре переменной-счетчику присваивается значение `0` (символы в строке нумеруются с нуля), условием продолжения является значение переменной-счетчика меньше количества символов в строке. В третьем параметре цикла `for` указывается приращение на единицу на каждой итерации цикла. Выведем все символы строки в прямом и обратном порядке:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
int len = (int)wcslen(str);
// Выводим символы в прямом порядке
for (int i = 0; i < len; ++i) {
   wprintf(L"%c\n", str[i]);
}
_putws(L"------------------");
// Выводим символы в обратном порядке
for (int i = len - 1; i >= 0; --i) {
   wprintf(L"%c\n", str[i]);
}
```

В этом примере количество символов сохраняется в переменной `len` вне цикла. Если функцию `wcslen()` указать внутри условия, то вычисление количества символов будет выполняться на каждой итерации цикла. Поэтому количество символов лучше получать вне цикла или присваивать значение переменной в первом параметре цикла `for`. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
for (int i = 0, len = (int)wcslen(str); i < len; ++i) {
   wprintf(L"%c\n", str[i]);
}
```

Перебор символов L-строки с помощью указателя и цикла `for` выполняется так:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
```

```
for (wchar_t *p = str; *p; ++p) {
   wprintf(L"%c\n", *p);
}
```

В этом примере начальным значением является адрес первого символа. Условием продолжения цикла является значение, на которое ссылается указатель. Любой символ трактуется как Истина, кроме нулевого символа, который имеет код 0. Так как L-строка завершается нулевым символом, то этот символ вернет значение Ложь и цикл завершится. В третьем параметре цикла for указывается приращение указателя на единицу на каждой итерации цикла. В этом случае используются правила адресной арифметики.

Вместо цикла for всегда можно использовать цикл while:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
wchar_t *p = str;
while (*p) {
   wprintf(L"%c\n", *p++);
}
```

Вначале объявляются строка и указатель, которому присваивается адрес первого символа. Цикл while выполняется до тех пор, пока значение, на которое ссылается указатель, не равно нулевому символу. Внутри цикла while выводится символ, на который ссылается указатель, а затем указатель перемещается на следующий символ (*p++). Выражение p++ возвращает текущий адрес, а затем увеличивает его. Символ * позволяет получить доступ к символу по указанному адресу. Последовательность выполнения соответствует следующей расстановке скобок: *(p++) или двум инструкциям: *p и p = p + 1.

8.10. Вывод и ввод L-строк

Прежде чем выводить L-строки, нужно настроить локаль с помощью функции setlocale():

```
setlocale(LC_ALL, "Russian_Russia.1251");
```

или с помощью функции _wsetlocale():

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
```

Для вывода L-строк используется функция _putws(). Прототип функции:

```
#include <wchar.h> /* или #include <stdio.h> */
int _putws(const wchar_t *str);
```

Функция выводит строку str и вставляет символ перевода строки. Пример:

```
_putws(L"String1");
_putws(L"String2");
```

Результат выполнения:

```
String1
String2
```

Для форматированного вывода L-строк и других данных используется функция `wprintf()`. **Прототип функции:**

```
#include <wchar.h> /* или #include <stdio.h> */
int wprintf(const wchar_t *format, ...);
```

В параметре `format` **указывается L-строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа** `%`. **Вместо спецификаторов формата подставляются значения, указанные в качестве параметров. Количество спецификаторов должно совпадать с количеством переданных параметров. Спецификаторы мы уже рассматривали в** *разд. 2.8* **применительно к функции** `printf()`. **В качестве значения функция возвращает количество выведенных символов. Пример вывода L-строки и числа:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"строка\n");              // строка
wprintf(L"Count %d\n", 10);        // Count 10
wprintf(L"%s %d\n", L"Count", 10); // Count 10
```

Для вывода L-строки можно также воспользоваться спецификаторами `%ls` **и** `%S` **в строке формата функции** `printf()`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("%ls\n", L"строка"); // строка
printf("%S\n", L"строка");  // строка
```

Если спецификатор `%S` **указать в строке формата функции** `wprintf()`, **то можно будет вывести С-строку:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%S\n", "строка"); // строка
```

Для ввода L-строки предназначена функция `_getws()`, **однако применять ее в программе не следует, т. к. функция не производит никакой проверки длины строки, что может привести к переполнению буфера. Прототип функции** `_getws()`:

```
#include <wchar.h> /* или #include <stdio.h> */
wchar_t *_getws(wchar_t *buffer);
```

Лучше получать строку посимвольно с помощью функции `getwchar()` **или воспользоваться функцией** `fgetws()`. **Прототип функции** `fgetws()`:

```
#include <wchar.h> /* или #include <stdio.h> */
wchar_t *fgetws(wchar_t *buffer, int maxCount, FILE *stream);
```

В качестве параметра `stream` **указывается стандартный поток ввода** `stdin`. **Считывание производится до первого символа перевода строки или до конца потока или пока не будет прочитано** `maxCount-1` **символов. Содержимое строки** `buffer` **включает символ перевода строки. Исключением является последняя строка. Если она не за-**

вершается символом перевода строки, то символ добавлен не будет. Если произошла ошибка или достигнут конец потока, то функция возвращает нулевой указатель. Пример получения строки приведен в листинге 8.2.

Листинг 8.2. Получение строки с помощью функции `fgetws()`

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

int main(void) {
   _wsystem(L"chcp 1251"); // Смена кодировки консоли
   _wsetlocale(LC_ALL, L"Russian_Russia.1251");
   wchar_t buf[256] = {0}, *p = NULL;
   wprintf(L"Введите строку: ");
   fflush(stdout);
   fflush(stdin);
   p = fgetws(buf, 256, stdin);
   if (p) {
      // Удаляем символ перевода строки
      size_t len = wcslen(buf);
      if (len != 0 && buf[len - 1] == '\n') {
         buf[len - 1] = L'\0';
      }
      // Выводим результат
      wprintf(L"Вы ввели: %s\n", buf);
   }
   else _putws(L"Возникла ошибка");
   return 0;
}
```

Для получения и автоматического преобразования данных в конкретный тип (например, в целое число) предназначена функция `wscanf()`. При вводе строки функция не производит никакой проверки длины строки, что может привести к переполнению буфера. Обязательно указывайте ширину при использовании спецификатора `%s` (например, `%255s`). Прототип функции:

```
#include <wchar.h> /* или #include <stdio.h> */
int wscanf(const wchar_t *format, ...);
```

В первом параметре указывается строка специального формата, внутри которой задаются спецификаторы, аналогичные применяемым в функции `printf()`, а также некоторые дополнительные спецификаторы. В последующих параметрах передаются адреса переменных. Функция возвращает количество произведенных присваиваний. Пример получения целого числа:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
int x = 0;
```

```
wprintf(L"Введите число: ");
fflush(stdout);                  // Сброс буфера вывода
fflush(stdin);                   // Очистка буфера ввода
int status = wscanf(L"%d", &x); // Символ & обязателен!!!
if (status == 1) {
   wprintf(L"Вы ввели: %d\n", x);
}
else {
   _putws(L"Ошибка при вводе числа");
}
wprintf(L"status = %d\n", status);
```

Пример ввода строки:

```
_wsystem(L"chcp 1251"); // Смена кодировки консоли
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[256] = {0};
wprintf(L"Введите строку: ");
fflush(stdout);
fflush(stdin);
int status = wscanf(L"%255s", str); // Символ & не указывается!!!
if (status == 1) {
   wprintf(L"Вы ввели: %s\n", str);
}
else {
   _putws(L"Ошибка при вводе строки");
}
```

Считывание символов производится до первого символа-разделителя, например пробела, табуляции или символа перевода строки. Поэтому после ввода строки `"Hello, world!"` **переменная** `str` **будет содержать лишь фрагмент** `"Hello,"`**, а не всю строку.**

8.11. Преобразование C-строки в L-строку и наоборот

Для преобразования C-строки в L-строку и наоборот предназначены две функции.

- `mbstowcs()` **— преобразует C-строку** `source` **в L-строку** `dest`**. Кодировка символов C-строки указывается с помощью локали. Прототип функции:**

  ```
  #include <stdlib.h>
  size_t mbstowcs(wchar_t *dest, const char *source,  size_t maxCount);
  ```

  В параметре `maxCount` **задается максимальное количество символов строки** `dest`**. Для того чтобы узнать требуемое количество символов, нужно передать в параметре** `dest` **нулевой указатель. Функция возвращает количество записанных символов или код ошибки** `(size_t)(-1)`**. Пример:**

  ```
  _wsetlocale(LC_ALL, L"Russian_Russia.1251");
  wchar_t wstr[256] = {0};
  ```

```
char str[] = "строка";
size_t count;
// Узнаем требуемый размер wstr
count = mbstowcs(NULL, str, 0);
wprintf(L"%u + 1\n", (unsigned int)count);      // 6 + 1
count = mbstowcs(wstr, str, 255);
wprintf(L"count = %u\n", (unsigned int)count); // count = 6
wprintf(L"%s\n", wstr);                         // строка
```

- `wcstombs()` — **преобразует L-строку** `source` **в C-строку** `dest`. **Требуемая кодировка C-строки указывается с помощью локали. Прототип функции:**

```
#include <stdlib.h>
size_t wcstombs(char *dest, const wchar_t *source, size_t maxCount);
```

В параметре `maxCount` **задается максимальное количество символов строки** `dest`. **Для того чтобы узнать требуемое количество символов, нужно передать в параметре** `dest` **нулевой указатель. Функция возвращает количество записанных байтов или код ошибки** `(size_t)(-1)`. **Пример:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t wstr[] = L"строка";
char str[256] = {0};
size_t count;
// Узнаем требуемый размер str
count = wcstombs(NULL, wstr, 0);
wprintf(L"%u + 1\n", (unsigned int)count);      // 6 + 1
count = wcstombs(str, wstr, 255);
wprintf(L"count = %u\n", (unsigned int)count); // count = 6
printf("%s\n", str);                            // строка
```

Функции `mbstowcs()` **и** `wcstombs()` **в Visual C выводят предупреждающее сообщение** `warning C4996`. **Для того чтобы предупреждения не выводились, можно воспользоваться функциями** `mbstowcs_s()` **и** `wcstombs_s()` **или в самое начало программы добавить инструкцию:**

```
#define _CRT_SECURE_NO_WARNINGS
```

Прототипы функций `mbstowcs_s()` **и** `wcstombs_s()`:

```
#include <stdlib.h>
errno_t mbstowcs_s(size_t *returnValue, wchar_t *dest,
        size_t sizeInWords, const char *source, size_t maxCount);
errno_t wcstombs_s(size_t *returnValue, char *dest,
        size_t destSizeInBytes, const wchar_t *source,
        size_t maxCount);
```

На заметку

В Windows для преобразования C-строки в различных кодировках в L-строку и наоборот можно воспользоваться функциями из WinAPI `MultiByteToWideChar()` **и** `WideCharToMultiByte()`. **Описание функций приведено в** ***разд. 8.12***.

8.12. Преобразование кодировок

При использовании L-строк все символы кодируются двумя байтами. Не следует рассматривать L-строку, как строку в какой-то кодировке, например в UTF-16. Думайте об L-строке как о строке в абстрактной кодировке, позволяющей закодировать более 65 тыс. символов. Когда мы говорим о строке в какой-либо кодировке, мы всегда подразумеваем С-строку. По умолчанию во всех проектах мы настроили для С-строк кодировку windows-1251.

Используя функции `mbstowcs()` и `wcstombs()`, которые мы рассмотрели в предыдущем разделе, можно преобразовать строку из одной однобайтовой кодировки в другую. Кодировка символов С-строки указывается с помощью локали. В этом случае L-строка используется как промежуточная стадия. Преобразуем строку из кодировки windows-1251 в кодировку windows-866 (листинг 8.3).

Листинг 8.3. Преобразование строки из кодировки windows-1251 в кодировку windows-866

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <wchar.h>
#include <stdlib.h>

int main(void) {
   size_t count;
   char cp866[256] = {0};
   wchar_t wstr[256] = {0};
   char cp1251[] = "строка"; // Исходная строка (windows-1251)
   for (int i = 0, len = (int)strlen(cp1251); i < len; ++i) {
      printf("%d ", cp1251[i]);
   } // -15 -14 -16 -18 -22 -32
   printf("\n");

   // Преобразование С-строки в кодировке windows-1251 в L-строку
   _wsetlocale(LC_ALL, L"Russian_Russia.1251");
   count = mbstowcs(wstr, cp1251, 255);
   if (count == (size_t)(-1)) {
      _putws(L"Error 1");
      exit(1);
   }
   for (int i = 0, len = (int)wcslen(wstr); i < len; ++i) {
      wprintf(L"%u ", wstr[i]);
   } // 1089 1090 1088 1086 1082 1072
   printf("\n");
```

```
    // Преобразование L-строки в C-строку в кодировке windows-866
    _wsetlocale(LC_ALL, L"Russian_Russia.866");
    count = wcstombs(cp866, wstr, 255);
    if (count == (size_t)(-1)) {
       _putws(L"Error 2");
       exit(1);
    }
    for (int i = 0, len = (int)strlen(cp866); i < len; ++i) {
       printf("%d ", cp866[i]);
    } // -31 -30 -32 -82 -86 -96
    printf("\n");
    return 0;
}
```

Для преобразования кодировок в Windows можно воспользоваться следующими функциями из WinAPI (кодировки указаны для русской версии Windows):

- `CharToOemA()` — преобразует строку `source` в кодировке windows-1251 в строку `dest` в кодировке windows-866. Прототип функции:

```
#include <windows.h> /* User32.lib */
WINBOOL CharToOemA(LPCSTR source, LPSTR dest);
```

- `CharToOemBuffA()` — преобразует строку `source` в кодировке windows-1251 в строку `dest` в кодировке windows-866. Количество символов указывается в параметре `length`. Прототип функции:

```
#include <windows.h> /* User32.lib */
WINBOOL CharToOemBuffA(LPCSTR source, LPSTR dest, DWORD length);
```

- `OemToCharA()` — преобразует строку `source` в кодировке windows-866 в строку `dest` в кодировке windows-1251. Прототип функции:

```
#include <windows.h> /* User32.lib */
WINBOOL OemToCharA(LPCSTR source, LPSTR dest);
```

- `OemToCharBuffA()` — преобразует строку `source` в кодировке windows-866 в строку `dest` в кодировке windows-1251. Количество символов указывается в параметре `length`. Прототип функции:

```
#include <windows.h> /* User32.lib */
WINBOOL OemToCharBuffA(LPCSTR source, LPSTR dest, DWORD length);
```

Пример использования функций `CharToOemA()` и `OemToCharA()` приведен в листинге 8.4.

Листинг 8.4. Функции `CharToOemA()` и `OemToCharA()`

```
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <windows.h>
```

```
int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char cp866[256] = {0};
   char cp1251[256] = {0};
   char str[] = "строка"; // Исходная строка в кодировке windows-1251

   // Преобразование windows-1251 в windows-866
   CharToOemA(str, cp866);
   for (int i = 0, len = (int)strlen(cp866); i < len; ++i) {
      printf("%d ", cp866[i]);
   } // -31 -30 -32 -82 -86 -96
   printf("\n");

   // Преобразование windows-866 в windows-1251
   OemToCharA(cp866, cp1251);
   for (int i = 0, len = (int)strlen(cp1251); i < len; ++i) {
      printf("%d ", cp1251[i]);
   } // -15 -14 -16 -18 -22 -32
   printf("\n");
   return 0;
}
```

Если вы используете компилятор Visual C в Eclipse и получили сообщение об отсутствии файла windows.h, то нужно в свойствах проекта указать путь к каталогу Include из Microsoft SDK. Если Microsoft SDK отсутствует на компьютере, то его нужно дополнительно установить.

Если в Visual C в Eclipse вы получили ошибку `error LNK2019`, то в свойствах проекта на вкладке **C Compiler | Preprocessor** нужно указать путь к каталогу Include в Microsoft SDK, а также добавить путь к каталогу Lib и библиотеку `User32.lib` на вкладке **Linker | Libraries** (рис. 8.1) или вставить в самое начало программы инструкцию:

```
#pragma comment(lib, "User32.lib")
```

Для преобразования кодировок в Windows можно также воспользоваться следующими функциями из WinAPI.

- `MultiByteToWideChar()` — преобразует C-строку `lpMultiByteStr` в L-строку `lpWideCharStr`. Прототип функции:

  ```
  #include <windows.h> /* Kernel32.lib */
  int MultiByteToWideChar(UINT codePage, DWORD dwFlags,
     LPCCH lpMultiByteStr, int cbMultiByte,
     LPWSTR lpWideCharStr, int cchWideChar);
  ```

  Параметр `codePage` задает кодировку C-строки. Примеры указания кодировки: `866` (для кодировки windows-866), `1251` (для windows-1251), `65001` или `CP_UTF8` (для UTF-8), `1200` (для UTF-16LE), `1201` (для UTF-16BE), `12000` (для UTF-32LE) и `12001` (для UTF-32BE). Полный список кодировок см. в документации.

В параметре `dwFlags` можно указать различные флаги (`MB_PRECOMPOSED` (1), `MB_ERR_INVALID_CHARS` (8), `MB_COMPOSITE` (2), `MB_USEGLYPHCHARS` (4)), задающие поведение функции при конверсии, или значение `0`. Полный список и описание флагов см. в документации.

Параметр `cbMultiByte` задает количество байтов в C-строке `lpMultiByteStr`, подлежащих конверсии. Если C-строка завершается нулевым символом, то лучше указать значение `-1`. В этом случае размер строки вычисляется автоматически и в L-строку вставляется нулевой символ. Если указано конкретное число, то нулевой символ автоматически в L-строку не вставляется!

Параметр `cchWideChar` задает максимальный размер L-строки `lpWideCharStr` в символах. Если указать значение `0`, то функция вернет требуемое количество символов. Если указано конкретное значение, то функция вернет количество записанных символов или значение `0` в случае ошибки.

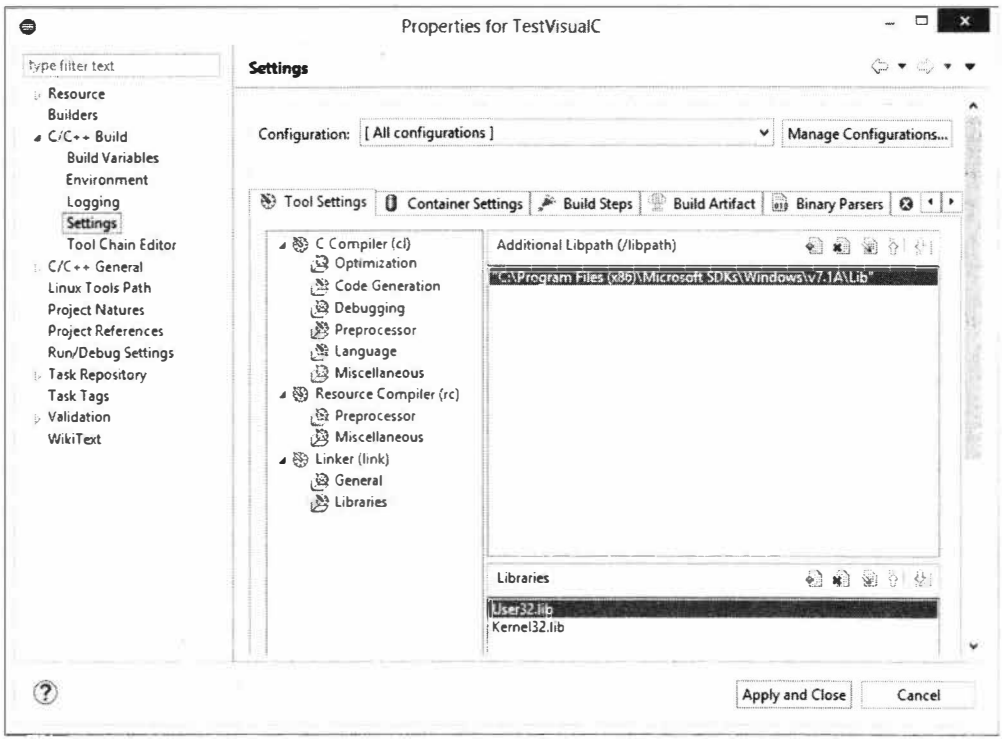


Рис. 8.1. Указание пути к каталогу Lib и добавление библиотек

- `WideCharToMultiByte()` — преобразует L-строку `lpWideCharStr` в C-строку `lpMultiByteStr`. Требуемая кодировка C-строки указывается в параметре `codePage`. Прототип функции:

```
#include <windows.h> /* Kernel32.lib */
int WideCharToMultiByte(UINT codePage, DWORD dwFlags,
   LPCWCH lpWideCharStr, int cchWideChar,
```

```
    LPSTR lpMultiByteStr, int cbMultiByte,
    LPCCH lpDefaultChar, LPBOOL lpUsedDefaultChar);
```

В параметре `dwFlags` **можно указать различные флаги (**`WC_COMPOSITECHECK` **(**`512`**),** `WC_DEFAULTCHAR` **(**`64`**),** `WC_DISCARDNS` **(**`16`**),** `WC_SEPCHARS` **(**`32`**),** `WC_NO_BEST_FIT_CHARS` **(**`1024`**)), задающие поведение функции при конверсии, или значение** `0`**. Полный список и описание флагов см. в документации.**

Параметр `cchWideChar` **задает количество символов в L-строке** `lpWideCharStr`**, подлежащих конверсии. Если L-строка завершается нулевым символом, то лучше указать значение** `-1`**. В этом случае размер строки вычисляется автоматически и в С-строку вставляется нулевой символ. Если указано конкретное число, то нулевой символ автоматически в С-строку не вставляется!**

Параметр `cbMultiByte` **задает максимальный размер С-строки** `lpMultiByteStr` **в байтах. Если указать значение** `0`**, то функция вернет требуемое количество байтов. Если указано конкретное значение, то функция вернет количество записанных байтов или значение** `0` **в случае ошибки.**

В параметре `lpDefaultChar` **можно задать адрес символа, который будет использоваться, если символ в L-строке не может быть преобразован (символ отсутствует в кодировке** `codePage`**). Если задан нулевой указатель, то будет использоваться символ по умолчанию. При использовании кодировки UTF-8 параметр** `lpDefaultChar` **должен иметь значение** `NULL`**.**

Параметр `lpUsedDefaultChar` **позволяет задать адрес логической переменной. Если параметр не равен значению** `NULL`**, то в переменную будет записано логическое значение** `Истина`**, если какого-либо символа нет в кодировке** `codePage`**, и** `0` **— в противном случае. При использовании кодировки UTF-8 параметр** `lpUsedDefaultChar` **должен иметь значение** `NULL`**.**

Пример использования функций `MultiByteToWideChar()` **и** `WideCharToMultiByte()` **приведен в листинге 8.5.**

Листинг 8.5. Функции `MultiByteToWideChar()` и `WideCharToMultiByte()`

```
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <wchar.h>
#include <windows.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int count = 0;
   char cp866[256] = {0};
   char utf8[256] = {0};
   wchar_t wstr[256] = {0};
   wchar_t wstr2[256] = {0};
   char str[] = "строка str"; // Исходная строка (windows-1251)
```

```
for (int i = 0, len = (int)strlen(str); i < len; ++i) {
   printf("%d ", str[i]);
} // -15 -14 -16 -18 -22 -32 32 115 116 114
printf("\n");

// Узнаем требуемый размер wstr
count = MultiByteToWideChar(1251, 0, str, -1, wstr, 0);
printf("требуется %d\n", count); // требуется 11
// Преобразование С-строки в кодировке windows-1251 в L-строку
count = MultiByteToWideChar(1251, 0, str, -1, wstr, 255);
printf("записано %d\n", count);  // записано 11
for (int i = 0, len = (int)wcslen(wstr); i < len; ++i) {
   wprintf(L"%u ", wstr[i]);
} // 1089 1090 1088 1086 1082 1072 32 115 116 114
printf("\n");

// Узнаем требуемый размер utf8
count = WideCharToMultiByte(CP_UTF8, 0, wstr, -1, utf8, 0,
                            NULL, NULL);
printf("требуется %d\n", count); // требуется 17
// Преобразование L-строки в С-строку в кодировке UTF-8
count = WideCharToMultiByte(CP_UTF8, 0, wstr, -1, utf8, 255,
                            NULL, NULL);
printf("записано %d\n", count);  // записано 17
for (int i = 0, len = (int)strlen(utf8); i < len; ++i) {
   printf("%d ", utf8[i]);
} // -47 -127 -47 -126 -47 -128 -48 -66 -48 -70 -48 -80 32 115 116 114
printf("\n");

// Узнаем требуемый размер wstr2
count = MultiByteToWideChar(CP_UTF8, 0, utf8, -1, wstr2, 0);
printf("требуется %d\n", count); // требуется 11
// Преобразование С-строки в кодировке UTF-8 в L-строку
count = MultiByteToWideChar(CP_UTF8, 0, utf8, -1, wstr2, 255);
printf("записано %d\n", count);  // записано 11
for (int i = 0, len = (int)wcslen(wstr2); i < len; ++i) {
   wprintf(L"%u ", wstr2[i]);
} // 1089 1090 1088 1086 1082 1072 32 115 116 114
printf("\n");

// Узнаем требуемый размер cp866
char ch = '?';
int flag = 1;
// wstr[1] = L'\u0263'; // символа нет в windows-866
count = WideCharToMultiByte(866, 0, wstr, -1, cp866, 0, &ch, &flag);
printf("требуется %d\n", count); // требуется 11
```

```
    // Преобразование L-строки в С-строку в кодировке windows-866
    count = WideCharToMultiByte(866, 0, wstr, -1, cp866, 255, &ch, &flag);
    printf("записано %d\n", count);  // записано 11
    for (int i = 0, len = (int)strlen(cp866); i < len; ++i) {
       printf("%d ", cp866[i]);
    } // -31 -30 -32 -82 -86 -96 32 115 116 114
    printf("\n");
    printf("%d\n", flag); // 0
    return 0;
}
```

8.13. Основные функции для работы с L-строками

Перечислим основные функции для работы с L-строками.

- `wcslen()` — возвращает количество символов в L-строке без учета нулевого символа. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
size_t wcslen(const wchar_t *str);
```

Тип `size_t` объявлен как беззнаковое целое число:

```
typedef unsigned __int64 size_t;
#define __int64 long long
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
int len = (int)wcslen(str);
wprintf(L"%d\n", len);        // 6, а не 7
```

Определить общий размер массива можно с помощью оператора `sizeof`. Оператор возвращает размер в байтах. Для того чтобы получить длину в символах, полученное значение нужно разделить на размер типа `wchar_t`. Пример:

```
wchar_t str[20] = L"строка";
wprintf(L"%d\n", (int) sizeof(str));           // 40
wprintf(L"%d\n",
       (int) (sizeof(str) / sizeof(wchar_t))); // 20
```

- `wcsnlen()` — возвращает количество символов в L-строке без учета нулевого символа. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
size_t wcsnlen(const wchar_t *str, size_t maxCount);
```

Во втором параметре указывается размер символьного массива. Если нулевой символ не встретился в числе `maxCount` символов, то возвращается значение `maxCount`. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[7] = L"строка";
int len = (int)wcsnlen(str, 7);
wprintf(L"%d\n", len);                    // 6
```

- `wcscpy()` — **копирует символы из L-строки** `source` **в L-строку** `dest` **и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку** `dest`. **Если строка** `source` **длиннее строки** `dest`, **то произойдет переполнение буфера. Прототип функции:**

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcscpy(wchar_t *dest, const wchar_t *source);
```

Пример использования функции:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[7];
wcscpy(str, L"String");
wprintf(L"%s\n", str); // String
```

Вместо функции `wcscpy()` **лучше использовать функцию** `wcscpy_s()`. **Прототип функции:**

```
#include <wchar.h> /* или #include <string.h> */
errno_t wcscpy_s(wchar_t *dest, rsize_t destSize, const wchar_t *source);
```

Функция `wcscpy_s()` **копирует символы из строки** `source` **в строку** `dest` **и вставляет нулевой символ. В параметре** `destSize` **указывается максимальное количество элементов массива** `dest`. **Пример:**

```
#define SIZE 7
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[SIZE];
wcscpy_s(str, SIZE, L"String");
wprintf(L"%s\n", str); // String
```

Вместо функций `wcscpy()` **и** `wcscpy_s()` **можно воспользоваться функциями** `wmemcpy()` **и** `wmemcpy_s()`, **которые копируют** `count` **первых символов из массива** `source` **в массив** `dest`, **но не вставляют нулевой символ. В параметре** `numberOfElements` **указывается количество элементов массива** `dest`. **Прототипы функций:**

```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t *dest, const wchar_t *source, size_t count);
errno_t wmemcpy_s(wchar_t *dest, size_t numberOfElements,
                  const wchar_t *source, size_t count);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[8] = {0};
str[6] = L'1';
wmemcpy(str, L"String", 6);
wprintf(L"%s\n", str);
```

```
// String1 (нулевой символ не вставляется!!!)
wmemcpy_s(str, 8, L"string2", 7);
wprintf(L"%s\n", str); // string2
```

Функции `wmemcpy()` **и** `wmemcpy_s()` **при пересечении указателей работают некорректно. Для того чтобы копирование при пересечении выполнялось правильно, нужно воспользоваться функциями** `wmemmove()` **и** `wmemmove_s()`. **Функции копируют** `count` **первых символов из массива** `source` **в массив** `dest`. **Нулевой символ функции не вставляют. В параметре** `numberOfElements` **указывается количество элементов массива** `dest`. **Прототипы функций:**

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *dest, const wchar_t *source, size_t count);
errno_t wmemmove_s(wchar_t *dest, size_t numberOfElements,
                   const wchar_t *source, size_t count);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[8] = {0};
str[6] = L'1';
wmemmove(str, L"String", 6);
wprintf(L"%s\n", str);
// String1 (нулевой символ не вставляется!!!)
wmemmove_s(str, 8, L"string2", 7);
wprintf(L"%s\n", str); // string2
```

- `wcsncpy()` — **копирует первые** `count` **символов из L-строки** `source` **в L-строку** `dest`. **В качестве значения функция возвращает указатель на строку** `dest`. **Если строка** `source` **длиннее строки** `dest`, **то произойдет переполнение буфера. Прототип функции:**

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcsncpy(wchar_t *dest, const wchar_t *source, size_t count);
```

Если количество символов в строке `source` **меньше числа** `count`, **то строка** `dest` **будет дополнена нулевыми символами, а если больше или равно, то копируются только** `count` **символов, при этом нулевой символ автоматически не вставляется. Пример копирования шести символов:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[7];
wcsncpy(str, L"String", 6);
str[6] = L'\0'; // Нулевой символ автоматически не вставляется
wprintf(L"%s\n", str); // String
```

Вместо функции `wcsncpy()` **лучше использовать функцию** `wcsncpy_s()`. **Прототип функции:**

```
#include <wchar.h> /* или #include <string.h> */
errno_t wcsncpy_s(wchar_t *dest, size_t destSizeInChars,
                  const wchar_t *source, size_t maxCount);
```

Функция `wcsncpy_s()` копирует первые `maxCount` символов из строки `source` в строку `dest` и вставляет нулевой символ. В параметре `destSizeInChars` указывается максимальное количество элементов массива `dest`. Пример:

```
#define SIZE 7
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[SIZE];
wcsncpy_s(str, SIZE, L"String", 5);
wprintf(L"%s\n", str); // Strin
```

- `wcscat()` — копирует символы из L-строки `source` в конец L-строки `dest` и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку `dest`. Если строка `dest` имеет недостаточный размер, то произойдет переполнение буфера. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcscat(wchar_t *dest, const wchar_t *source);
```

Пример использования функции:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[20] = L"ст";
wcscat(str, L"ро");
wcscat(str, L"ка");
wprintf(L"%s\n", str); // строка
```

Обратите внимание на то, что перед копированием в строке `dest` обязательно должен быть нулевой символ. Локальные переменные автоматически не инициализируются, поэтому этот код приведет к ошибке:

```
wchar_t str[20];          // Локальная переменная
wcscat(str, L"строка");   // Ошибка, нет нулевого символа!
```

Для того чтобы избавиться от ошибки, нужно произвести инициализацию строки нулями следующим образом:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[20] = {0};    // Инициализация нулями
wcscat(str, L"строка");   // Все нормально
wprintf(L"%s\n", str);    // строка
```

Вместо функции `wcscat()` лучше использовать функцию `wcscat_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
errno_t wcscat_s(wchar_t *dest, rsize_t destSize,
                 const wchar_t *source);
```

Функция `wcscat_s()` копирует символы из строки `source` в конец строки `dest`. В параметре `destSize` указывается максимальное количество элементов массива `dest`. Пример:

```
#define SIZE 20
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
```

```
wchar_t str[SIZE] = L"ст";
wcscat_s(str, SIZE, L"ро");
wcscat_s(str, SIZE, L"ка");
wprintf(L"%s\n", str); // строка
```

- `wcsncat()` — копирует первые `count` символов из L-строки `source` в конец строки `dest` и вставляет нулевой символ. В качестве значения функция возвращает указатель на строку `dest`. Обратите внимание на то, что перед копированием в строке `dest` обязательно должен быть нулевой символ. Если строка `dest` имеет недостаточный размер, то произойдет переполнение буфера. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcsncat(wchar_t *dest, const wchar_t *source, size_t count);
```

Пример использования функции:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[20] = L"ст";
wcsncat(str, L"ро", 2);
wcsncat(str, L"какао", 2);
wprintf(L"%s\n", str); // строка
```

Вместо функции `wcsncat()` лучше использовать функцию `wcsncat_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
errno_t wcsncat_s(wchar_t *dest, size_t destSizeInChars,
                  const wchar_t *source, size_t maxCount);
```

Функция `wcsncat_s()` копирует первые `maxCount` символов из строки `source` в конец строки `dest`. В параметре `destSizeInChars` указывается максимальное количество элементов массива `dest`. Пример:

```
#define SIZE 20
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[SIZE] = L"ст";
wcsncat_s(str, SIZE, L"ро", 2);
wcsncat_s(str, SIZE, L"какао", 2);
wprintf(L"%s\n", str); // строка
```

- `_wcsdup()` — создает копию L-строки в динамической памяти и возвращает указатель на нее. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *_wcsdup(const wchar_t *str);
```

Память выделяется с помощью функции `malloc()`, поэтому после использования строки следует освободить память с помощью функции `free()`. Пример:

```
// #include <stdlib.h>
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка", *p = NULL;
p = _wcsdup(str);
```

```
if (p) {
   wprintf(L"%s\n", p); // строка
   free(p);
}
```

- `wcstok()` — разделяет L-строку `str` на подстроки, используя в качестве разделителей символы из L-строки `delim`. При первом вызове указываются оба параметра. В качестве значения возвращается указатель на первую подстроку или нулевой указатель, если символы-разделители не найдены в L-строке `str`. Для того чтобы получить последующие подстроки, необходимо каждый раз вызывать функцию `wcstok()`, указывая в первом параметре нулевой указатель, а во втором параметре те же самые символы-разделители. Обратите внимание: функция изменяет исходную строку `str`, вставляя вместо символов-разделителей нулевой символ. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcstok(wchar_t *str, const wchar_t *delim);
```

Пример использования функции:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"a,b.c=d", *p = NULL;
p = wcstok(str, L",.=");
while (p) {
   wprintf(L"%s-", p);
   p = wcstok(NULL, L",.=");
} // Результат: a-b-c-d-
wprintf(L"\n");
// Функция изменяет исходную строку!!!
int len = (int) (sizeof(str) / sizeof(wchar_t));
for (int i = 0; i < len; ++i) {
   wprintf(L"%d ", str[i]);
} // 97 0 98 0 99 0 100 0
wprintf(L"\n");
```

Вместо функции `wcstok()` лучше использовать функцию `wcstok_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcstok_s(wchar_t *str, const wchar_t *delim, wchar_t **context);
```

Первые два параметра аналогичны параметрам функции `wcstok()`. В параметре `context` передается адрес указателя. Обратите внимание: функция изменяет исходную строку `str`, вставляя вместо символов-разделителей нулевой символ. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"a b c,d", *p = NULL, *context = NULL;
p = wcstok_s(str, L" ,", &context);
while (p) {
   wprintf(L"%s-", p);
```

```
    p = wcstok_s(NULL, L" ,", &context);
} // Результат: a-b-c-d-
wprintf(L"\n");
// Функция изменяет исходную строку!!!
int len = (int) (sizeof(str) / sizeof(wchar_t));
for (int i = 0; i < len; ++i) {
    wprintf(L"%d ", str[i]);
} // 97 0 98 0 99 0 100 0
wprintf(L"\n");
```

Практически все функции без суффикса _s, которые мы рассмотрели в этом разделе, в Visual C выводят предупреждающее сообщение warning C4996, т. к. по умолчанию функции не производят никакой проверки корректности данных. В этом случае возможно переполнение буфера. Забота о корректности данных полностью лежит на плечах программиста. Если вы уверены в своих действиях, то можно подавить вывод предупреждающих сообщений. Сделать это можно двумя способами:

- определить макрос с названием _CRT_SECURE_NO_WARNINGS в самом начале программы, перед подключением заголовочных файлов. Пример:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <wchar.h>
```

- вставить прагму warning. Пример:

```
#pragma warning( disable : 4996 )
```

> ***ОБРАТИТЕ ВНИМАНИЕ***
>
> После директив #define и #pragma точка с запятой не указывается.

8.14. Поиск и замена в L-строке

Для поиска и замены в L-строке предназначены следующие функции.

- wcschr() — ищет в L-строке str первое вхождение символа ch. В качестве значения функция возвращает указатель на первый найденный символ в L-строке str или нулевой указатель, если символ не найден. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcschr(const wchar_t *str, wchar_t ch);
```

Пример использования функции:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка строка", *p = NULL;
p = wcschr(str, L'к');
if (p) {
    wprintf(L"Индекс: %d\n", (int)(p - str));
} // Индекс: 4
```

- `wcsrchr()` — ищет в L-строке `str` последнее вхождение символа `ch`. В качестве значения функция возвращает указатель на символ или нулевой указатель, если символ не найден. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcsrchr(const wchar_t *str, wchar_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка строка", *p = NULL;
p = wcsrchr(str, L'к');
if (p) {
   wprintf(L"Индекс: %d\n", (int)(p - str));
} // Индекс: 11
```

- `wmemchr()` — ищет в L-строке `str` первое вхождение символа `ch`. Максимально просматривается `maxCount` символов. В качестве значения функция возвращает указатель на первый найденный символ в строке `str` или нулевой указатель, если символ не найден. Прототип функции:

```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *str, wchar_t ch, size_t maxCount);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка строка", *p = NULL;
p = wmemchr(str, L'к', wcslen(str));
if (p) {
   wprintf(L"Индекс: %d\n", (int)(p - str));
} // Индекс: 4
```

- `wcspbrk()` — ищет в L-строке `str` первое вхождение одного из символов (нулевой символ не учитывается), входящих в L-строку `control`. В качестве значения функция возвращает указатель на первый найденный символ или нулевой указатель, если символы не найдены. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcspbrk(const wchar_t *str, const wchar_t *control);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка строка", *p = NULL;
p = wcspbrk(str, L"кр");
if (p) {
   wprintf(L"Индекс: %d\n", (int)(p - str));
} // Индекс: 2 (индекс первой буквы "р")
```

- `wcscspn()` — возвращает индекс первого символа в L-строке `str`, который совпадает с одним из символов, входящих в L-строку `control`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
size_t wcscspn(const wchar_t *str, const wchar_t *control);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка строка";
int index = (int)wcscspn(str, L"кр");
wprintf(L"Индекс: %d\n", index);
// Индекс: 2 (индекс первой буквы "р")
index = (int)wcscspn(str, L"ф");
wprintf(L"Индекс: %d\n", index);
// Индекс: 13 (длина строки, если ничего не найдено)
```

- `wcsspn()` — возвращает индекс первого символа в L-строке `str`, который не совпадает ни с одним из символов, входящих в L-строку `control`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
size_t wcsspn(const wchar_t *str, const wchar_t *control);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка строка";
int index = (int)wcsspn(str, L"окстр");
wprintf(L"Индекс: %d\n", index);
// Индекс: 5 ("а" не входит в "окстр")
```

- `wcsstr()` — ищет в L-строке `str` первое вхождение целого фрагмента из L-строки `subStr`. В качестве значения функция возвращает указатель на первое вхождение фрагмента в строку или нулевой указатель — в противном случае. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *wcsstr(const wchar_t *str, const wchar_t *subStr);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка строка", *p = NULL;
p = wcsstr(str, L"ока");
if (p) {
   wprintf(L"Индекс: %d\n", (int)(p - str));
} // Индекс: 3
```

- `_wcsset()` — заменяет все символы в L-строке `str` указанным символом `ch`. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *_wcsset(wchar_t *str, wchar_t ch);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
_wcsset(str, L'*');
wprintf(L"'%s'\n", str); // '******'
```

Вместо функции `_wcsset()` лучше использовать функцию `_wcsset_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
errno_t _wcsset_s(wchar_t *str, size_t sizeInWords, wchar_t ch);
```

В параметре `sizeInWords` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[10] = L"строка";
_wcsset_s(str, 10, L'*');
wprintf(L"'%s'\n", str); // '******'
```

- `_wcsnset()` — заменяет первые `count` символов в L-строке `str` указанным символом `ch`. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *_wcsnset(wchar_t *str, wchar_t ch, size_t count);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
_wcsnset(str, L'*', 4);
wprintf(L"'%s'\n", str); // '****ка'
```

Вместо функции `_wcsnset()` лучше использовать функцию `_wcsnset_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
errno_t _wcsnset_s(wchar_t *str, size_t strSizeInWords,
                   wchar_t ch, size_t count);
```

В параметре `strSizeInWords` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[10] = L"строка";
_wcsnset_s(str, 10, L'*', 4);
wprintf(L"'%s'\n", str); // '****ка'
```

- `wmemset()` — заменяет первые `count` элементов массива `dest` символом `ch`. В качестве значения функция возвращает указатель на массив `dest`. Прототип функции:

```
#include <wchar.h>
wchar_t *wmemset(wchar_t *dest, wchar_t ch, size_t count);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"String";
wmemset(str, L'*', 4);
wprintf(L"'%s'\n", str); // '****ng'
```

- `_wcsrev()` — меняет порядок следования символов внутри L-строки на противоположный, как бы переворачивает строку. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *_wcsrev(wchar_t *str);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"строка";
_wcsrev(str);
wprintf(L"%s\n", str); // акортс
```

- `_wcsupr()` — заменяет все буквы в L-строке `str` соответствующими прописными буквами. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *_wcsupr(wchar_t *str);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
_wcsupr(str);
wprintf(L"%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

Вместо функции `_wcsupr()` лучше использовать функцию `_wcsupr_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
errno_t _wcsupr_s(wchar_t *str, size_t sizeInWords);
```

В параметре `sizeInWords` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[40] = L"абвгдеёжзийклмнопрстуфхцчшщъыьэюя";
_wcsupr_s(str, 40);
wprintf(L"%s\n", str);
// АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
```

- `_wcslwr()` — заменяет все буквы в L-строке `str` соответствующими строчными буквами. Функция возвращает указатель на строку `str`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
wchar_t *_wcslwr(wchar_t *str);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
_wcslwr(str);
wprintf(L"%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

Вместо функции `_wcslwr()` лучше использовать функцию `_wcslwr_s()`. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
errno_t _wcslwr_s(wchar_t *str, size_t sizeInWords);
```

В параметре `sizeInWords` указывается размер строки. Функция возвращает значение `0`, если операция успешно выполнена, или код ошибки. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[40] = L"АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
_wcslwr_s(str, 40);
wprintf(L"%s\n", str);
// абвгдеёжзийклмнопрстуфхцчшщъыьэюя
```

8.15. Сравнение L-строк

Для сравнения L-строк предназначены следующие функции.

- `wcscmp()` — сравнивает L-строку `str1` с L-строкой `str2` без учета настроек локали и возвращает одно из значений:
  - отрицательное число — если `str1` меньше `str2`;
  - `0` — если строки равны;
  - положительное число — если `str1` больше `str2`.

  Сравнение производится с учетом регистра символов. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
int wcscmp(const wchar_t *str1, const wchar_t *str2);
```

  Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str1[] = L"абв", str2[] = L"абв", str3[] = L"АБВ";
wprintf(L"%d\n", wcscmp(str1, str2)); // 0
wprintf(L"%d\n", wcscmp(str1, str3)); // 1
str1[2] = L'б'; // str1[] = L"абб", str2[] = L"абв";
wprintf(L"%d\n", wcscmp(str1, str2)); // -1
str1[2] = L'г'; // str1[] = L"абг", str2[] = L"абв";
wprintf(L"%d\n", wcscmp(str1, str2)); // 1
```

- `wcsncmp()` — сравнивает `count` первых символов в L-строках `str1` и `str2`. Если нулевой символ встретится раньше, то значение параметра `count` игнорируется. Функция возвращает одно из значений:
  - отрицательное число — если `str1` меньше `str2`;
  - `0` — если строки равны;
  - положительное число — если `str1` больше `str2`.

Сравнение производится с учетом регистра символов. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
int wcsncmp(const wchar_t *str1, const wchar_t *str2, size_t count);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str1[] = L"абв", str2[] = L"абг";
wprintf(L"%d\n", wcsncmp(str1, str2, 2)); // 0
wprintf(L"%d\n", wcsncmp(str1, str2, 3)); // -1
```

- `wcscoll()` — функция аналогична функции `wcscmp()`, но сравнение производится с учетом значения `LC_COLLATE` в текущей локали. Например, буква "Ё" в диапазон между буквами "Е" и "Ж" не попадает, т. к. буква "Ё" имеет код `1025`, буква "Е" — `1045`, а буква "Ж" — `1046`. Если сравнение производится с помощью функции `wcscmp()`, то буква "Е" будет больше буквы "Ё" (`1045 > 1025`). При использовании функции `wcscoll()` с настройкой локали буква "Ё" попадет в диапазон между буквами "Е" и "Ж", т. к. используются локальные настройки по алфавиту. Если локаль не настроена, то эта функция эквивалентна функции `wcscmp()`. Сравнение производится с учетом регистра символов. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
int wcscoll(const wchar_t *str1, const wchar_t *str2);
```

Пример:

```
wchar_t str1[] = L"Е", str2[] = L"Ё";
wprintf(L"%d\n", str1[0]);              // 1045 (Е)
wprintf(L"%d\n", str2[0]);              // 1025 (Ё)
wprintf(L"%d\n", wcscmp(str1, str2));  // 1
wprintf(L"%d\n", wcscoll(str1, str2)); // 1
// Без настройки локали: Е (код 1045) больше Ё (код 1025)
// Настройка локали
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", wcscmp(str1, str2));  // 1
wprintf(L"%d\n", wcscoll(str1, str2)); // -1
// После настройки локали: Е меньше Ё
```

- `_wcsncoll()` — сравнивает `maxCount` первых символов в L-строках `str1` и `str2`. Сравнение производится с учетом регистра символов. Функция аналогична функции `wcsncmp()`, но сравнение производится с учетом значения `LC_COLLATE` в текущей локали. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
int _wcsncoll(const wchar_t *str1, const wchar_t *str2, size_t maxCount);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str1[] = L"абв", str2[] = L"абг";
wprintf(L"%d\n", _wcsncoll(str1, str2, 2)); // 0
wprintf(L"%d\n", _wcsncoll(str1, str2, 3)); // -1
```

- `wcsxfrm()` — преобразует L-строку `source` в строку специального формата и записывает ее в `dest`. В конец вставляется нулевой символ. Записывается не более `maxCount` символов. Если количество символов `maxCount` меньше необходимого количества символов после преобразования, то содержимое `dest` не определено. В качестве значения функция возвращает число необходимых символов. Для того чтобы просто получить количество необходимых символов (без учета нулевого символа), в параметре `maxCount` указывается число `0`, а в параметре `dest` передается нулевой указатель. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
size_t wcsxfrm(wchar_t *dest, const wchar_t *source, size_t maxCount);
```

Функция `wcscoll()`, прежде чем произвести сравнение, неявно выполняет преобразование переданных строк в строки специального формата. Функция `wcsxfrm` позволяет выполнить такое преобразование явным образом. Преобразование строки производится с учетом значения `LC_COLLATE` в текущей локали. Если локаль не настроена, то просто выполняется копирование. В дальнейшем строки специального формата можно сравнивать с помощью функции `wcscmp()`. Результат сравнения будет соответствовать результату сравнения с помощью функции `wcscoll()`. Функцию `wcsxfrm()` следует использовать, если сравнение строк производится многократно. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
int x = 0;
wchar_t str1[] = L"E", str2[] = L"Ё";
wchar_t buf1[10] = {0}, buf2[10] = {0};
x = (int)wcsxfrm(NULL, str1, 0);
wprintf(L"%d\n", x);                    //  7 (нужен буфер 7 + 1)
wcsxfrm(buf1, str1, 10);
wcsxfrm(buf2, str2, 10);
wprintf(L"%d\n", wcscmp(str1, str2));  //  1 ("E" больше "Ё")
wprintf(L"%d\n", wcscoll(str1, str2)); // -1 ("E" меньше "Ё")
wprintf(L"%d\n", wcscmp(buf1, buf2));  // -1 ("E" меньше "Ё")
```

- `_wcsicmp()` — сравнивает L-строки без учета регистра символов. Функция учитывает настройки локали для категории `LC_CTYPE`. Для русских букв необходимо настроить локаль. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
int _wcsicmp(const wchar_t *str1, const wchar_t *str2);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str1[] = L"абв", str2[] = L"АБВ";
wprintf(L"%d\n", _wcsicmp(str1, str2)); // 0
```

- `_wcsnicmp()` — сравнивает `maxCount` первых символов в L-строках `str1` и `str2` без учета регистра символов. Функция учитывает настройки локали для категории `LC_CTYPE`. Для русских букв необходимо настроить локаль. Прототип функции:

```
#include <wchar.h> /* или #include <string.h> */
int _wcsnicmp(const wchar_t *str1, const wchar_t *str2,
              size_t maxCount);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str1[] = L"абве", str2[] = L"АБВЖ";
wprintf(L"%d\n", _wcsnicmp(str1, str2, 3)); // 0
wprintf(L"%d\n", _wcsnicmp(str1, str2, 4)); // -1
```

- `_wcsicoll()` — **сравнивает L-строки без учета регистра символов. Функция учитывает настройки локали для категорий** `LC_CTYPE` **и** `LC_COLLATE`. **Для русских букв необходимо настроить локаль. Прототип функции:**

```
#include <wchar.h> /* или #include <string.h> */
int _wcsicoll(const wchar_t *str1, const wchar_t *str2);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str1[] = L"абв", str2[] = L"АБВ";
wprintf(L"%d\n", _wcsicoll(str1, str2)); // 0
```

- `_wcsnicoll()` — **сравнивает** `maxCount` **первых символов в L-строках** `str1` **и** `str2` **без учета регистра символов. Функция учитывает настройки локали для категорий** `LC_CTYPE` **и** `LC_COLLATE`. **Для русских букв необходимо настроить локаль. Прототип функции:**

```
#include <wchar.h> /* или #include <string.h> */
int _wcsnicoll(const wchar_t *str1, const wchar_t *str2,
               size_t maxCount);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str1[] = L"абве", str2[] = L"АБВЖ";
wprintf(L"%d\n", _wcsnicoll(str1, str2, 3)); // 0
wprintf(L"%d\n", _wcsnicoll(str1, str2, 4)); // -1
```

Для сравнения строк можно также использовать функцию `wmemcmp()`. **Она сравнивает первые** `count` **символов массивов** `buf1` **и** `buf2`. **В качестве значения функция возвращает:**

- **отрицательное число — если** `buf1` **меньше** `buf2`;
- `0` **— если массивы равны;**
- **положительное число — если** `buf1` **больше** `buf2`.

Прототип функции:

```
#include <wchar.h>
int wmemcmp(const wchar_t *str1, const wchar_t *str2, size_t count);
```

Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str1[] = L"abc", str2[] = L"abc";
wprintf(L"%d\n", wmemcmp(str1, str2, 3));   // 0
str1[2] = L'b';
wprintf(L"%d\n", wmemcmp(str1, str2, 3));   // -1
str1[2] = L'd';
wprintf(L"%d\n", wmemcmp(str1, str2, 3));   // 1
```

8.16. Преобразование L-строки в число

Для преобразования L-строки в число используются следующие функции.

- `_wtoi()` — преобразует L-строку в число, имеющее тип `int`. Прототип функции:

```
#include <stdlib.h> /* или #include <wchar.h> */
int _wtoi(const wchar_t *str);
```

Считывание символов производится, пока они соответствуют цифрам. Иными словами, в строке могут содержаться не только цифры. Пробельные символы в начале строки игнорируются. Пример преобразования:

```
wprintf(L"%d\n", _wtoi(L"25"));         // 25
wprintf(L"%d\n", _wtoi(L"2.5"));        // 2
wprintf(L"%d\n", _wtoi(L"5str"));       // 5
wprintf(L"%d\n", _wtoi(L"5s10"));       // 5
wprintf(L"%d\n", _wtoi(L" \t\n 25"));   // 25
wprintf(L"%d\n", _wtoi(L"-25"));        // -25
wprintf(L"%d\n", _wtoi(L"str"));        // 0
```

Если значение выходит за диапазон значений типа `int`, то в Visual C возвращается максимальное или минимальное значение типа `int` и переменной `errno` присваивается значение `ERANGE`:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%d\n", _wtoi(L"2147483649"));  // 2147483647
wprintf(L"%d\n", errno);                 // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   _putws(L"Выход за диапазон значений");
}
```

В MinGW будет усечение значения:

```
wprintf(L"%d\n", _wtoi(L"2147483649"));  // -2147483647
wprintf(L"%d\n", errno);                 // 0
```

- `_wtol()` — преобразует L-строку в число, имеющее тип `long`. Прототип функции:

```
#include <stdlib.h> /* или #include <wchar.h> */
long _wtol(const wchar_t *str);
```

Пример преобразования:

```
wprintf(L"%ld\n", _wtol(L"25"));        // 25
wprintf(L"%ld\n", _wtol(L" \n -25"));   // -25
wprintf(L"%ld\n", _wtol(L"2.5"));       // 2
wprintf(L"%ld\n", _wtol(L"5str"));      // 5
wprintf(L"%ld\n", _wtol(L"5s10"));      // 5
wprintf(L"%ld\n", _wtol(L"str"));       // 0
```

Если значение выходит за диапазон значений типа `long`, то в Visual C возвращается максимальное или минимальное значение типа `long` и переменной `errno` присваивается значение `ERANGE`:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%ld\n", _wtol(L"2147483649")); // 2147483647
wprintf(L"%d\n", errno);                 // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   _putws(L"Выход за диапазон значений");
}
```

В MinGW будет усечение значения:

```
wprintf(L"%ld\n", _wtol(L"2147483649")); // -2147483647
wprintf(L"%d\n", errno);                 // 0
```

- `_wtoll()` — преобразует L-строку в число, имеющее тип `long long`. Функция доступна только в Visual C. Прототип функции:

```
#include <stdlib.h> /* или #include <wchar.h> */
long long _wtoll(const wchar_t *str);
```

Пример преобразования:

```
wprintf(L"%I64d\n", _wtoll(L"92233720368547758O7"));
// 9223372036854775807
```

Если значение выходит за диапазон значений типа `long long`, то в Visual C возвращается максимальное или минимальное значение типа `long long` и переменной `errno` присваивается значение `ERANGE`:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%I64d\n", _wtoll(L"9223372036854775809"));
// 9223372036854775807
wprintf(L"%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   _putws(L"Выход за диапазон значений");
}
```

В MinGW можно воспользоваться функцией `wtoll()`. Прототип функции:

```
#include <stdlib.h>
long long wtoll(const wchar_t *str);
```

Пример преобразования:

```
wprintf(L"%I64d\n", wtoll(L"9223372036854775807"));
// 9223372036854775807
wprintf(L"%I64d\n", wtoll(L"9223372036854775809"));
// 9223372036854775807
wprintf(L"%d\n", errno); // 34
```

- `_wtoi64()` — **преобразует L-строку в число, имеющее тип** `__int64`. **Прототип функции:**

```
#include <stdlib.h> /* или #include <wchar.h> */
__int64 _wtoi64(const wchar_t *str);
```

Пример преобразования:

```
wprintf(L"%I64d\n", _wtoi64(L"9223372036854775807"));
// 9223372036854775807
```

Если значение выходит за диапазон значений типа `__int64`, **то возвращается максимальное или минимальное значение типа** `__int64` **и переменной** `errno` **присваивается значение** `ERANGE`:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%I64d\n", _wtoi64(L"9223372036854775809"));
// 9223372036854775807
wprintf(L"%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   _putws(L"Выход за диапазон значений");
}
```

- `wcstol()` **и** `wcstoul()` — **преобразуют L-строку в число, имеющее типы** `long` **и** `unsigned long` **соответственно. Прототипы функций:**

```
#include <stdlib.h> /* или #include <wchar.h> */
long wcstol(const wchar_t *str, wchar_t **endPtr, int radix);
unsigned long wcstoul(const wchar_t *str, wchar_t **endPtr, int radix);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью числа. Указатель на этот символ доступен через параметр `endPtr`, **если он не равен** `NULL`. **В параметре** `radix` **можно указать систему счисления (число от** `2` **до** `36`). **Если в параметре** `radix` **задано число** `0`, **то система счисления определяется автоматически. Пример преобразования:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t *p = NULL;
wprintf(L"%ld\n", wcstol(L"25", NULL, 0));    // 25
wprintf(L"%ld\n", wcstol(L"025", NULL, 0));   // 21
wprintf(L"%ld\n", wcstol(L"0x25", NULL, 0));  // 37
wprintf(L"%ld\n", wcstol(L"111", NULL, 2));   // 7
```

```
wprintf(L"%ld\n", wcstol(L"025", NULL, 8));   // 21
wprintf(L"%ld\n", wcstol(L"0x25", NULL, 16)); // 37
wprintf(L"%ld\n", wcstol(L"5sl0", &p, 0));    // 5
wprintf(L"%s\n", p);                          // sl0
```

Если в строке нет числа, то возвращается число 0. **Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному (**LONG_MIN **или** 0**) или максимальному (**LONG_MAX **или** ULONG_MAX**) значению для типа, а переменной** errno **присваивается значение** ERANGE. **Пример:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%ld\n", wcstol(L"str", NULL, 0));        // 0
wprintf(L"%ld\n", wcstol(L"2147483649", NULL, 0));
// 2147483647 (соответствует LONG_MAX)
wprintf(L"%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   _putws(L"Выход за диапазон значений");
}
```

- wcstoll() **и** wcstoull() **— преобразуют L-строку в число, имеющее типы** long long **и** unsigned long long **соответственно. Прототипы функций:**

```
#include <wchar.h>

long long wcstoll(const wchar_t *str, wchar_t **endPtr, int radix);
unsigned long long wcstoull(const wchar_t *str, wchar_t **endPtr,
                            int radix);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью числа. Указатель на этот символ доступен через параметр endPtr, **если он не равен** NULL. **В параметре** radix **можно указать систему счисления (число от** 2 **до** 36**). Если в параметре** radix **задано число** 0, **то система счисления определяется автоматически. Пример преобразования:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t *p = NULL;
wprintf(L"%I64d\n", wcstoll(L"25", NULL, 0));    // 25
wprintf(L"%I64d\n", wcstoll(L"025", NULL, 0));   // 21
wprintf(L"%I64d\n", wcstoll(L"0x25", NULL, 0));  // 37
wprintf(L"%I64d\n", wcstoll(L"111", NULL, 2));   // 7
wprintf(L"%I64d\n", wcstoll(L"025", NULL, 8));   // 21
wprintf(L"%I64d\n", wcstoll(L"0x25", NULL, 16)); // 37
wprintf(L"%I64d\n", wcstoll(L"5s10", &p, 0));    // 5
wprintf(L"%s\n", p);                             // s10
```

Если в строке нет числа, то возвращается число 0. **Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному (**LLONG_MIN **или** 0**) или максимальному (**LLONG_MAX **или** ULLONG_MAX**) значению для типа, а переменной** errno **присваивается значение** ERANGE. **Пример:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%I64d\n", wcstoll(L"str", NULL, 0));   // 0
wprintf(L"%I64d\n", wcstoll(L"9223372036854775809", NULL, 0));
// 9223372036854775807 (соответствует LLONG_MAX)
wprintf(L"%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   _putws(L"Выход за диапазон значений");
}
```

- `_wcstoi64()` и `_wcstoui64()` — преобразуют L-строку в число, имеющее типы `__int64` и `unsigned __int64` соответственно. Прототипы функций:

```
#include <stdlib.h> /* или #include <wchar.h> */
__int64 _wcstoi64(const wchar_t *str, wchar_t **endPtr, int radix);
unsigned __int64 _wcstoui64(const wchar_t *str, wchar_t **endPtr,
                            int radix);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью числа. Указатель на этот символ доступен через параметр `endPtr`, если он не равен `NULL`. В параметре `radix` можно указать систему счисления (число от `2` до `36`). Если в параметре `radix` задано число `0`, то система счисления определяется автоматически. Пример преобразования:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t *p = NULL;
wprintf(L"%I64d\n", _wcstoi64(L"25", NULL, 0));    // 25
wprintf(L"%I64u\n", _wcstoui64(L"25", NULL, 0));   // 25
wprintf(L"%I64d\n", _wcstoi64(L"025", NULL, 0));   // 21
wprintf(L"%I64d\n", _wcstoi64(L"0x25", NULL, 0));  // 37
wprintf(L"%I64d\n", _wcstoi64(L"111", NULL, 2));   // 7
wprintf(L"%I64d\n", _wcstoi64(L"025", NULL, 8));   // 21
wprintf(L"%I64d\n", _wcstoi64(L"0x25", NULL, 16)); // 37
wprintf(L"%I64d\n", _wcstoi64(L"5s10", &p, 0));    // 5
wprintf(L"%s\n", p);                               // s10
```

Если в строке нет числа, то возвращается число `0`. Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному или максимальному значению для типа, а переменной `errno` присваивается значение `ERANGE`. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%I64d\n", _wcstoi64(L"str", NULL, 0));   // 0
wprintf(L"%I64d\n", _wcstoi64(L"9223372036854775809", NULL, 0));
// 9223372036854775807
wprintf(L"%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   _putws(L"Выход за диапазон значений");
}
```

- `wcstoimax()` и `wcstoumax()` — преобразуют L-строку в число, имеющее типы `intmax_t` и `uintmax_t` соответственно. Прототипы функций:

```
#include <inttypes.h>
intmax_t wcstoimax(const wchar_t *str, wchar_t **endPtr, int radix);
typedef long long intmax_t;
uintmax_t wcstoumax(const wchar_t *str, wchar_t **endPtr, int radix);
typedef unsigned long long uintmax_t;
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью числа. Указатель на этот символ доступен через параметр `endPtr`, если он не равен `NULL`. В параметре `radix` можно указать систему счисления (число от `2` до `36`). Если в параметре `radix` задано число `0`, то система счисления определяется автоматически. Пример преобразования:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t *p = NULL;
wprintf(L"%I64u\n", wcstoumax(L"25", NULL, 0));    // 25
wprintf(L"%I64d\n", wcstoimax(L"25", NULL, 0));    // 25
wprintf(L"%I64d\n", wcstoimax(L"025", NULL, 0));   // 21
wprintf(L"%I64d\n", wcstoimax(L"0x25", NULL, 0));  // 37
wprintf(L"%I64d\n", wcstoimax(L"111", NULL, 2));   // 7
wprintf(L"%I64d\n", wcstoimax(L"025", NULL, 8));   // 21
wprintf(L"%I64d\n", wcstoimax(L"0x25", NULL, 16)); // 37
wprintf(L"%I64d\n", wcstoimax(L"5s10", &p, 0));    // 5
wprintf(L"%s\n", p);                               // s10
```

Если в строке нет числа, то возвращается число `0`. Если число выходит за диапазон значений для типа, то значение будет соответствовать минимальному (`INTMAX_MIN` или `0`) или максимальному (`INTMAX_MAX` или `UINTMAX_MAX`) значению для типа, а переменной `errno` присваивается значение `ERANGE`. Пример:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
// #include <stdint.h>
wprintf(L"%I64d\n", INTMAX_MIN);   // -9223372036854775808
wprintf(L"%I64d\n", INTMAX_MAX);   // 9223372036854775807
wprintf(L"%I64u\n", UINTMAX_MAX);  // 18446744073709551615
wprintf(L"%I64d\n", wcstoimax(L"str", NULL, 0));   // 0
wprintf(L"%I64d\n", wcstoimax(L"9223372036854775809", NULL, 0));
// 9223372036854775807 (соответствует INTMAX_MAX)
wprintf(L"%d\n", errno); // 34
// #include <math.h> или #include <errno.h>
if (errno == ERANGE) {
   _putws(L"Выход за диапазон значений");
}
```

- `_wtof()` — преобразует L-строку в вещественное число, имеющее тип `double`. Прототип функции:

```
#include <stdlib.h> /* или #include <wchar.h> */
double _wtof(const wchar_t *str);
```

Пример преобразования (обратите внимание: от настроек локали зависит десятичный разделитель вещественных чисел):

```
_wsetlocale(LC_ALL, L"C");
wprintf(L"%.2f\n", _wtof(L"25"));            // 25.00
wprintf(L"%.2f\n", _wtof(L"2.5"));           // 2.50
wprintf(L"%.2f\n", _wtof(L"2,5"));           // 2.00
wprintf(L"%.2f\n", _wtof(L"5.1str"));        // 5.10
wprintf(L"%.2f\n", _wtof(L"5s10"));          // 5.00
wprintf(L"%.2f\n", _wtof(L"str"));           // 0.00
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%.2f\n", _wtof(L"2.5"));           // 2,00
wprintf(L"%.2f\n", _wtof(L"2,5"));           // 2,50
```

- `wcstod()` — **преобразует L-строку в вещественное число, имеющее тип** `double`. **Прототип функции:**

```
#include <stdlib.h> /* или #include <wchar.h> */
double wcstod(const wchar_t *str, wchar_t **endPtr);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью вещественного числа. Указатель на этот символ доступен через параметр `endPtr`, **если он не равен** `NULL`. **Пример преобразования (обратите внимание: от настроек локали зависит десятичный разделитель вещественных чисел):**

```
_wsetlocale(LC_ALL, L"C");
wchar_t *p = NULL;
wprintf(L"%.2f\n", wcstod(L" \t\n 25", NULL)); // 25.00
wprintf(L"%.2f\n", wcstod(L"2.5", NULL));      // 2.50
wprintf(L"%.2f\n", wcstod(L"2,5", NULL));      // 2.00
wprintf(L"%.2f\n", wcstod(L"5.1str", NULL));   // 5.10
wprintf(L"%.2f\n", wcstod(L"str", NULL));      // 0.00
wprintf(L"%e\n", wcstod(L"14.5e5s10", &p));    // 1.450000e+006
wprintf(L"%s\n", p);                           // s10
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%.2f\n", wcstod(L"2.5", NULL));      // 2,00
wprintf(L"%.2f\n", wcstod(L"2,5", NULL));      // 2,50
```

Если в строке нет числа, то возвращается число `0`. **Если число выходит за диапазон значений для типа, то возвращается значение** `-HUGE_VAL` **или** `HUGE_VAL`, **а переменной** `errno` **присваивается значение** `ERANGE`;

- `wcstof()` — **преобразует L-строку в вещественное число, имеющее тип** `float`. **Прототип функции:**

```
#include <stdlib.h> /* или #include <wchar.h> */
float wcstof(const wchar_t *str, wchar_t **endPtr);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью вещественного числа. Указатель на этот символ доступен через параметр `endPtr`**, если он не равен** `NULL`**. Пример преобразования (обратите внимание: от настроек локали зависит десятичный разделитель вещественных чисел):**

```
_wsetlocale(LC_ALL, L"C");
wchar_t *p = NULL;
wprintf(L"%.2f\n", wcstof(L" \t\n 25", NULL)); // 25.00
wprintf(L"%.2f\n", wcstof(L"2.5", NULL));      // 2.50
wprintf(L"%.2f\n", wcstof(L"5.1str", NULL));   // 5.10
wprintf(L"%e\n", wcstof(L"14.5e5s10", &p));    // 1.450000e+006
wprintf(L"%s\n", p);                           // s10
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wprintf(L"%.2f\n", wcstof(L"2.5", NULL));      // 2,00
wprintf(L"%.2f\n", wcstof(L"2,5", NULL));      // 2,50
```

Если в строке нет числа, то возвращается число `0`**. Если число выходит за диапазон значений для типа, то возвращается значение** `-HUGE_VALF` **или** `HUGE_VALF`**, а переменной** `errno` **присваивается значение** `ERANGE`**;**

- `wcstold()` **— преобразует L-строку в вещественное число, имеющее тип** `long double`**. Прототип функции:**

```
#include <stdlib.h> /* или #include <wchar.h> */
long double wcstold(const wchar_t *str, wchar_t **endPtr);
```

Пробельные символы в начале строки игнорируются. Считывание символов заканчивается на символе, не являющемся записью вещественного числа. Указатель на этот символ доступен через параметр `endPtr`**, если он не равен** `NULL`**. Пример преобразования (обратите внимание: от настроек локали зависит десятичный разделитель вещественных чисел):**

```
_wsetlocale(LC_ALL, L"C");
wchar_t *p = NULL;
__mingw_printf("%.2Lf\n", wcstold(L" \t\n 25", NULL)); // 25.00
__mingw_printf("%.2Lf\n", wcstold(L"2.5", NULL));      // 2.50
__mingw_printf("%.2Lf\n", wcstold(L"5.1str", NULL));   // 5.10
__mingw_printf("%Le\n", wcstold(L"14.5e5s10", &p));
// 1.450000e+006
wprintf(L"%s\n", p);                                   // s10
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
__mingw_printf("%.2Lf\n", wcstold(L"2.5", NULL));      // 2,00
__mingw_printf("%.2Lf\n", wcstold(L"2,5", NULL));      // 2,50
```

Если в строке нет числа, то возвращается число `0`**. Если число выходит за диапазон значений для типа, то возвращается значение** `-HUGE_VALL` **или** `HUGE_VALL`**, а переменной** `errno` **присваивается значение** `ERANGE`**.**

Для преобразования L-строки в число можно также воспользоваться функциями `swscanf()` **и** `_snwscanf()`**. Прототипы функций:**

```
#include <stdio.h> /* или #include <wchar.h> */
int swscanf(const wchar_t *str, const wchar_t *format, ...);
int _snwscanf(const wchar_t *str, size_t maxCount,
              const wchar_t *format, ...);
```

В параметре `str` указывается L-строка, содержащая число, в параметре `maxCount` — максимальное количество просматриваемых символов в строке, а в параметре `format` — строка специального формата, внутри которой задаются спецификаторы, аналогичные применяемым в функции `printf()` *(см. разд. 2.8)*, а также некоторые дополнительные спецификаторы. В последующих параметрах передаются адреса переменных. Функции возвращают количество произведенных присваиваний. Пример получения двух целых чисел:

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t str[] = L"20 30 str";
int n = 0, k = 0;
int count = swscanf(str, L"%d%d", &n, &k);
// int count = _snwscanf(str, wcslen(str), L"%d%d", &n, &k);
if (count == 2) {
   wprintf(L"n = %d\n", n); // n = 20
   wprintf(L"k = %d\n", k); // k = 30
}
else _putws(L"Не удалось преобразовать");
```

При вводе строки функция `swscanf()` не производит никакой проверки длины строки, что может привести к переполнению буфера. Обязательно указывайте ширину при использовании спецификатора `%s` (например, `%255s`). Кроме того, следует учитывать, что функция не проверяет возможность выхода значения за диапазон значений для типа, что может привести к неправильному результату. Функция никак об этом вас не предупредит, поэтому лучше использовать функции, рассмотренные в начале этого раздела.

8.17. Преобразование числа в L-строку

Преобразовать целое число в L-строку позволяют следующие функции.

- `_itow()` и `_itow_s()` — преобразуют число типа `int` в L-строку. Прототипы функций:

  ```
  #include <stdlib.h> /* или #include <wchar.h> */
  wchar_t *_itow(int val, wchar_t *dest, int radix);
  errno_t _itow_s(int val, wchar_t *dest, size_t sizeInWords, int radix);
  ```

  В параметре `val` указывается число, в параметре `dest` — указатель на символьный массив, в который будет записан результат, в параметре `sizeInWords` — максимальный размер символьного массива `dest`, а в параметре `radix` — система счисления (`2`, `8`, `10` или `16`). Пример использования функции `_itow_s()`:

```
wchar_t str[100] = {0};
int x = 255;
_itow_s(x, str, 100, 10);
wprintf(L"%s\n", str);                    // 255
```

Пример вывода двоичного, восьмеричного и шестнадцатеричного значений:

```
wchar_t str[100] = {0};
// Двоичное значение
wprintf(L"%s\n", _itow(100, str, 2));  // 1100100
// Восьмеричное значение
wprintf(L"%s\n", _itow(10, str, 8));   // 12
// Шестнадцатеричное значение
wprintf(L"%s\n", _itow(255, str, 16)); // ff
```

- `_ltow()` и `_ltow_s()` — **преобразуют число типа** `long` **в L-строку. Прототипы функций:**

```
#include <stdlib.h> /* или #include <wchar.h> */
wchar_t *_ltow(long val, wchar_t *dest, int radix);
errno_t _ltow_s(long val, wchar_t *dest, size_t sizeInWords, int radix);
```

Пример:

```
wchar_t str[100] = {0};
long x = 255;
_ltow_s(x, str, 100, 10);
wprintf(L"%s\n", str);                    // 255
wprintf(L"%s\n", _ltow(x, str, 16));      // ff
```

- `_ultow()` и `_ultow_s()` — **преобразуют число типа** `unsigned long` **в L-строку. Прототипы функций:**

```
#include <stdlib.h> /* или #include <wchar.h> */
wchar_t *_ultow(unsigned long val, wchar_t *dest, int radix);
errno_t _ultow_s(unsigned long val, wchar_t *dest,
                 size_t sizeInWords, int radix);
```

Пример:

```
wchar_t str[100] = {0};
unsigned long x = 255;
_ultow_s(x, str, 100, 10);
wprintf(L"%s\n", str);                    // 255
wprintf(L"%s\n", _ultow(x, str, 16));     // ff
```

- `_i64tow()` и `_i64tow_s()` — **преобразуют число типа** `__int64` **в L-строку. Прототипы функций:**

```
#include <stdlib.h> /* или #include <wchar.h> */
wchar_t *_i64tow(__int64 val, wchar_t *dest, int radix);
errno_t _i64tow_s(__int64 val, wchar_t *dest, size_t sizeInWords,
                  int radix);
```

Пример:

```
wchar_t str[100] = {0};
__int64 x = 255;
_i64tow_s(x, str, 100, 10);
wprintf(L"%s\n", str);                  // 255
wprintf(L"%s\n", _i64tow(x, str, 16)); // ff
```

- **_ui64tow() и _ui64tow_s() — преобразуют число типа** `unsigned __int64` **в L-строку. Прототипы функций:**

```
#include <stdlib.h> /* или #include <wchar.h> */
wchar_t *_ui64tow(unsigned __int64 val, wchar_t *dest, int radix);
errno_t _ui64tow_s(unsigned __int64 val, wchar_t *dest,
                   size_t sizeInWords, int radix);
```

Пример:

```
wchar_t str[100] = {0};
unsigned __int64 x = 255;
_ui64tow_s(x, str, 100, 10);
wprintf(L"%s\n", str);                   // 255
wprintf(L"%s\n", _ui64tow(x, str, 16)); // ff
```

Выполнить форматирование L-строки, а также преобразовать значения элементарных типов в L-строку можно с помощью функции `swprintf_s()`. **Прототип функции:**

```
#include <stdio.h> /* или #include <wchar.h> */
int swprintf_s(wchar_t *buf, size_t sizeInWords,
               const wchar_t *format, ...);
```

В параметре `format` **указывается строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа** `%`. **Спецификаторы формата совпадают со спецификаторами, используемыми в функции** `printf()` *(см. разд. 2.8)*. **Вместо спецификаторов формата подставляются значения, указанные в качестве параметров. Количество спецификаторов должно совпадать с количеством переданных параметров. Результат записывается в буфер, адрес которого передается в первом параметре (**`buf`**). В параметре** `sizeInWords` **указывается размер буфера. Функция возвращает количество символов, записанных в символьный массив. Пример:**

```
_wsetlocale(LC_ALL, L"Russian_Russia.1251");
wchar_t buf[50] = {0};
int x = 100, count = 0;
count = swprintf_s(buf, 50, L"x = %d", x);
wprintf(L"%s\n", buf);                  // x = 100
wprintf(L"%d\n", count);                // 7
```

8.18. Типы *char16_t* и *char32_t*

Стандарт C11 ввел в язык C два новых типа — `char16_t` и `char32_t`. Тип `char16_t` описывает символ в кодировке UTF-16. Объявление:

```
#include <uchar.h>
typedef uint_least16_t char16_t;
#include <stdint.h>
typedef unsigned short  uint_least16_t;
```

При инициализации символа или строки перед литералом указывается строчная буква `u`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char16_t ch = u'я';
printf("%u\n", ch);                  // 1103
char16_t str[] = u"строка";
size_t size = sizeof(char16_t);
printf("%d\n", (int)size);           // 2
size_t size_str = sizeof(str);
printf("%d\n", (int)size_str);       // 14
int len = (int) (size_str / size);
printf("%d\n", (int)len);            // 7
for (int i = 0; i < len; ++i) {
   printf("%u ", str[i]);
} // 1089 1090 1088 1086 1082 1072 0
```

Тип `char32_t` описывает символ в кодировке UTF-32. Объявление:

```
#include <uchar.h>
typedef uint_least32_t char32_t;
#include <stdint.h>
typedef unsigned  uint_least32_t;
```

При инициализации символа или строки перед литералом указывается прописная буква `U`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char32_t ch = U'я';
printf("%u\n", ch);                  // 1103
char32_t str[] = U"строка";
size_t size = sizeof(char32_t);
printf("%d\n", (int)size);           // 4
size_t size_str = sizeof(str);
printf("%d\n", (int)size_str);       // 28
int len = (int) (size_str / size);
printf("%d\n", (int)len);            // 7
for (int i = 0; i < len; ++i) {
   printf("%u ", str[i]);
} // 1089 1090 1088 1086 1082 1072 0
```

Для преобразования типов предназначены следующие функции.

- `c16rtomb()` — преобразует символ `c16` из кодировки UTF-16 в кодировку, указанную с помощью локали, и записывает результат в символьный массив `buf`. Нулевой символ в массив не добавляется. Функция возвращает число байтов, записанных в массив, или значение `(size_t)(-1)` в случае ошибки. Прототип функции:

```
#include <uchar.h>
size_t c16rtomb(char *buf, char16_t c16, mbstate_t *state);
typedef int mbstate_t;
```

Пример преобразования символа в кодировку windows-1251:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char buf[10] = {0};
char16_t ch = u'я';
mbstate_t state = 0;
size_t count = c16rtomb(buf, ch, &state);
printf("%I64u\n", count); // 1
printf("%d\n", state);    // 0
printf("%c\n", buf[0]);   // я
printf("%d\n", buf[0]);   // -1
```

- `mbrtoc16()` — преобразует многобайтовый символ, записанный в строку `str`, в кодировку UTF-16 и сохраняет его в переменной `pc16`. Кодировка указывается с помощью локали. Максимально просматривается `n` символов строки `str`. Функция возвращает число байтов, представляющих символ, `0` или значения `(size_t)(-1)`, `(size_t)(-2)` или `(size_t)(-3)` в случае ошибки. Прототип функции:

```
#include <uchar.h>
size_t mbrtoc16(char16_t *pc16, const char *str, size_t n,
                mbstate_t *state);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char16_t pc16 = 0;
char str[10] = "я";
mbstate_t state = 0;
size_t count = mbrtoc16(&pc16, str, 10, &state);
printf("%I64u\n", count); // 1
printf("%d\n", state);    // 0
printf("%u\n", pc16);     // 1103
```

- `c32rtomb()` — преобразует символ `c32` и записывает результат в символьный массив `buf`. Нулевой символ в массив не добавляется. Функция возвращает число байтов, записанных в массив, или значение `(size_t)(-1)` в случае ошибки. Прототип функции:

```
#include <uchar.h>
size_t c32rtomb(char *buf, char32_t c32, mbstate_t *state);
```

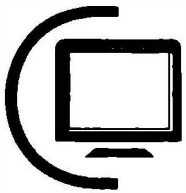
- `mbrtoc32()` — преобразует многобайтовый символ, записанный в строку `str`, в символ в кодировке UTF-32 и сохраняет его в переменной `pc32`. Прототип функции:

```
#include <uchar.h>
size_t mbrtoc32(char32_t *pc32, const char *str, size_t n,
                mbstate_t *state);
```

В Windows функции `c32rtomb()` и `mbrtoc32()` работают с многобайтовыми символами в кодировке UTF-8, а не используют настройки кодировки из локали:

```
char buf[10] = {0};
char32_t ch = U'я';
mbstate_t state = 0;
size_t count = c32rtomb(buf, ch, &state);
printf("%I64u\n", count); // 2
printf("%d\n", state);    // 0
printf("%d\n", buf[0]);   // -47
printf("%d\n", buf[1]);   // -113
char32_t ch32 = 0;
count = mbrtoc32(&ch32, buf, 10, &state);
printf("%I64u\n", count); // 2
printf("%d\n", state);    // 0
printf("%u\n", ch32);     // 1103
```

ГЛАВА 9



Работа с датой и временем

Основные функции для работы с датой и временем в языке C объявлены в заголовочном файле time.h. В этом файле объявлены также следующие типы данных:

- `clock_t` — возвращается функцией `clock()`. Объявление типа:

```
typedef long clock_t;
```

- `time_t` — используется для представления времени в виде целочисленного значения. Размер типа зависит от настроек компилятора. Объявление типа:

```
typedef __time32_t time_t; // В проекте Test32c
typedef __time64_t time_t; // В проекте Test64c
```

Объявления типов `__time32_t` и `__time64_t` выглядят так:

```
typedef long __time32_t;
typedef __int64 __time64_t;
#define __int64 long long
```

Некоторые функции в качестве значения возвращают указатель на структуру `tm`. Объявление структуры `tm` выглядит следующим образом:

```
struct tm {
   int tm_sec;   // Секунды (число от 0 до 59, изредка до 61)
   int tm_min;   // Минуты (число от 0 до 59)
   int tm_hour;  // Час (число от 0 до 23)
   int tm_mday;  // День месяца (число от 1 до 31)
   int tm_mon;   // Месяц (число от 0 (январь) до 11 (декабрь))
   int tm_year;  // Год, начиная с 1900 года
   int tm_wday;  // День недели (число от 0 до 6 (0 - это воскресенье,
                 // 1 - это понедельник, ..., 6 - это суббота))
   int tm_yday;  // День в году (число от 0 до 365)
   int tm_isdst; // Если больше 0, значит, действует летнее время
                 // Если 0, значит, летнее время не установлено
                 // Если меньше 0, то информации нет
};
```

9.1. Получение текущей даты и времени

Получить текущую дату и время позволяют следующие функции.

- `time()` — возвращает количество секунд, прошедших с начала эпохи (с 1 января 1970 г.). Прототип функции:

```
#include <time.h>
time_t time(time_t *Time);
```

Функцию можно вызвать двумя способами, передавая в качестве параметра нулевой указатель или адрес переменной, в которую будет записано возвращаемое значение. Пример:

```
time_t t1 = time(NULL);      // Передаем нулевой указатель
printf("%ld\n", (long)t1);   // 1548383508
time_t t2;
time(&t2);                   // Передаем адрес переменной
printf("%ld\n", (long)t2);   // 1548383508
```

Вместо функции `time()` можно использовать функцию `_time64()`. Число перед открывающей круглой скобкой означает количество битов. Прототип функции:

```
__time64_t _time64(__time64_t *Time);
```

Пример использования:

```
__time64_t t1 = _time64(NULL);
printf("%I64d\n", t1);       // 1548383659
```

- `gmtime()` — возвращает указатель на структуру `tm` с универсальным временем (UTC) или нулевой указатель в случае ошибки. В качестве параметра указывается адрес переменной, которая содержит количество секунд, прошедших с начала эпохи. Для того чтобы получить текущую дату, в качестве параметра следует передать результат выполнения функции `time()`. Прототип функции:

```
#include <time.h>
struct tm *gmtime(const time_t *Time);
```

Пример использования функции:

```
struct tm *ptm = NULL;
time_t t = time(NULL);
ptm = gmtime(&t);
if (!ptm) {
   printf("Error\n");
   exit(1);
}
printf("%d\n", ptm->tm_mday);  // 25
printf("%d\n", ptm->tm_mon);   // 0 (январь)
printf("%d\n", ptm->tm_year);  // 119 (1900 + 119 = 2019 г.)
printf("%d\n", ptm->tm_hour);  // 2
printf("%d\n", ptm->tm_min);   // 36
printf("%d\n", ptm->tm_sec);   // 10
```

```
printf("%d\n", ptm->tm_wday);  // 5 (пятница)
printf("%d\n", ptm->tm_yday);  // 24
printf("%d\n", ptm->tm_isdst); // 0
```

Вместо функции `gmtime()` можно использовать функцию `_gmtime64()`. Число перед открывающей круглой скобкой означает количество битов. Прототип функции:

```
struct tm *_gmtime64(const __time64_t *Time);
```

Вместо функции `gmtime()` лучше использовать функцию `gmtime_s` (`_gmtime64_s()` вместо `_gmtime64()`). Прототипы функций:

```
#include <time.h>
errno_t gmtime_s(struct tm *Tm, const time_t *Time);
errno_t _gmtime64_s(struct tm *Tm, const __time64_t *Time);
```

Если ошибок нет, то функции возвращают значение `0`. При наличии ошибки возвращается значение макроса `EINVAL` (значение равно `22`) и переменная `errno` устанавливается равной `EINVAL`. Пример использования функции `gmtime_s()`:

```
struct tm ptm;
time_t t = time(NULL);
errno_t err = gmtime_s(&ptm, &t);
if (err) {
   printf("Error\n");
   exit(1);
}
printf("%d\n", ptm.tm_mday);  // 25
printf("%d\n", ptm.tm_mon);   // 0 (январь)
printf("%d\n", ptm.tm_year);  // 119 (1900 + 119 = 2019 г.)
printf("%d\n", ptm.tm_hour);  // 2
printf("%d\n", ptm.tm_min);   // 41
printf("%d\n", ptm.tm_sec);   // 10
printf("%d\n", ptm.tm_wday);  // 5 (пятница)
printf("%d\n", ptm.tm_yday);  // 24
printf("%d\n", ptm.tm_isdst); // 0
```

- `localtime()` — возвращает указатель на структуру `tm` с локальным временем или нулевой указатель в случае ошибки. В качестве параметра указывается адрес переменной, которая содержит количество секунд, прошедших с начала эпохи. Для того чтобы получить текущую дату, в качестве параметра следует передать результат выполнения функции `time()`. Прототип функции:

```
#include <time.h>
struct tm *localtime(const time_t *Time);
```

Пример использования функции:

```
struct tm *ptm = NULL;
time_t t = time(NULL);
ptm = localtime(&t);
```

```
if (!ptm) {
   printf("Error\n");
   exit(1);
}
printf("%d\n", ptm->tm_mday);  // 25
printf("%d\n", ptm->tm_mon);   // 0 (январь)
printf("%d\n", ptm->tm_year);  // 119 (1900 + 119 = 2019 г.)
printf("%d\n", ptm->tm_hour);  // 5
printf("%d\n", ptm->tm_min);   // 45
printf("%d\n", ptm->tm_sec);   // 49
printf("%d\n", ptm->tm_wday);  // 5 (пятница)
printf("%d\n", ptm->tm_yday);  // 24
printf("%d\n", ptm->tm_isdst); // 0
```

Вместо функции `localtime()` **можно использовать функцию** `_localtime64()`. **Число перед открывающей круглой скобкой означает количество битов. Прототип функции:**

```
struct tm *_localtime64(const __time64_t *Time);
```

Вместо функции `localtime()` **лучше использовать функцию** `localtime_s()` (`_localtime64_s()` **вместо** `_localtime64()`). **Прототипы функций:**

```
#include <time.h>

errno_t localtime_s(struct tm *Tm, const time_t *Time);
errno_t _localtime64_s(struct tm *Tm, const __time64_t *Time);
```

Если ошибок нет, то функции возвращают значение `0`. **При наличии ошибки возвращается значение макроса** `EINVAL` **(значение равно** `22`**) и переменная** `errno` **устанавливается равной** `EINVAL`. **Пример использования функции** `localtime_s()`:

```
struct tm ptm;
time_t t = time(NULL);
errno_t err = localtime_s(&ptm, &t);
if (err) {
   printf("Error\n");
   exit(1);
}
printf("%d\n", ptm.tm_mday);  // 25
printf("%d\n", ptm.tm_mon);   // 0 (январь)
printf("%d\n", ptm.tm_year);  // 119 (1900 + 119 = 2019 г.)
printf("%d\n", ptm.tm_hour);  // 5
printf("%d\n", ptm.tm_min);   // 50
printf("%d\n", ptm.tm_sec);   // 26
printf("%d\n", ptm.tm_wday);  // 5 (пятница)
printf("%d\n", ptm.tm_yday);  // 24
printf("%d\n", ptm.tm_isdst); // 0
```

- `mktime()` **— возвращает количество секунд, прошедших с начала эпохи. В качестве параметра передается указатель на структуру** `tm` **с локальной датой и временем. В случае ошибки возвращается значение** `-1`. **Прототип функции:**

```
#include <time.h>
time_t mktime(struct tm *Tm);
```

Пример использования функции:

```
struct tm ptm;
time_t t1 = time(NULL), t2 = 0;
errno_t err = localtime_s(&ptm, &t1);
if (err) {
   printf("Error\n");
   exit(1);
}
t2 = mktime(&ptm);
printf("%I64d\n", t1);  // 1548384762
printf("%I64d\n", t2);  // 1548384762
```

Вместо функции `mktime()` **можно использовать функцию** `_mktime64()`. **Число перед открывающей круглой скобкой означает количество битов. Прототип функции:**

```
__time64_t _mktime64(struct tm *Tm);
```

- `difftime()` — **возвращает разность между двумя датами (**`Time1 - Time2`**). В случае ошибки возвращается значение** `0` **и переменная** `errno` **устанавливается равной** `EINVAL`. **Прототип функции:**

```
#include <time.h>
double difftime(time_t Time1, time_t Time2);
```

Пример:

```
time_t t1 = time(NULL), t2 = 1527298420LL;
double result = difftime(t1, t2);
printf("%.1f\n", result);
```

Вместо функции `difftime()` **можно использовать функцию** `_difftime64()`. **Число перед открывающей круглой скобкой означает количество битов. Прототип функции:**

```
double _difftime64(__time64_t Time1, __time64_t Time2);
```

Выведем текущую дату и время таким образом, чтобы день недели и месяц были написаны по-русски (листинг 9.1).

Листинг 9.1. Вывод текущей даты и времени

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <time.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
```

```
    struct tm *ptm = NULL;
    time_t t = time(NULL);
    char d[][25] = {"воскресенье", "понедельник", "вторник",
                    "среда", "четверг", "пятница", "суббота"};
    char m[][25] = {"января", "февраля", "марта", "апреля", "мая",
                    "июня", "июля", "августа", "сентября", "октября",
                    "ноября", "декабря"};
    ptm = localtime(&t); // Получаем текущее время
    if (!ptm) {
       printf("Error\n");
       exit(1);
    }
    printf("Сегодня:\n%s %d %s %d", d[ptm->tm_wday], ptm->tm_mday,
                                    m[ptm->tm_mon], ptm->tm_year + 1900);
    printf(" %02d:%02d:%02d\n", ptm->tm_hour, ptm->tm_min, ptm->tm_sec);
    printf("%02d.%02d.%d\n", ptm->tm_mday, ptm->tm_mon + 1,
                             ptm->tm_year + 1900);
    return 0;
}
```

Результат выполнения:

```
Сегодня:
пятница 25 января 2019 15:58:28
25.01.2019
```

9.2. Форматирование даты и времени

Получить форматированный вывод даты и времени позволяют следующие функции.

- `asctime()` и `_wasctime()` — **возвращают указатель на строку специального формата или нулевой указатель в случае ошибки. В конец строки вставляется символ перевода строки (`\n`) и нулевой символ (`\0`). Прототипы функций:**

```
#include <time.h>
char *asctime(const struct tm *Tm);
#include <time.h> /* или #include <wchar.h> */
wchar_t *_wasctime(const struct tm *Tm);
```

Пример использования функции `asctime()`:

```
struct tm *ptm = NULL;
char *p = NULL;
time_t t = time(NULL);
ptm = localtime(&t);
if (!ptm) {
   printf("Error\n");
   exit(1);
}
```

```
p = asctime(ptm);
if (!p) {
   printf("Error\n");
   exit(1);
}
printf("%s", p); // Fri Jan 25 06:01:48 2019\n
```

Вместо функции `asctime()` лучше использовать функцию `asctime_s()`, а вместо функции `_wasctime()` — `_wasctime_s()`. Прототипы функций:

```
#include <time.h>
errno_t asctime_s(char *buf, size_t sizeInBytes, const struct tm *Tm);
#include <time.h> /* или #include <wchar.h> */
errno_t _wasctime_s(wchar_t *buf, size_t sizeInWords,
                    const struct tm *Tm);
```

В первом параметре передается указатель на строку, во втором параметре — максимальный размер строки, а в третьем параметре — указатель на структуру `tm`. Если ошибок нет, то функции возвращают значение `0`. При наличии ошибки возвращается значение макроса `EINVAL` (значение равно `22`). Пример использования функции `asctime_s()`:

```
struct tm ptm;
char str[80] = {0};
time_t t = time(NULL);
errno_t err = localtime_s(&ptm, &t);
if (err) {
   printf("Error\n");
   exit(1);
}
err = asctime_s(str, 80, &ptm);
if (err) {
   printf("Error\n");
   exit(1);
}
printf("%s", str); // Fri Jan 25 06:05:04 2019\n
```

- `ctime()`, `_ctime64()`, `_wctime()` и `_wctime64()` — функции аналогичны `asctime`, но в качестве параметра принимают количество секунд, прошедших с начала эпохи. Прототипы функций:

```
#include <time.h>
char *ctime(const time_t *Time);
char *_ctime64(const __time64_t *Time);
#include <time.h> /* или #include <wchar.h> */
wchar_t *_wctime(const time_t *Time);
wchar_t *_wctime64(const __time64_t *Time);
```

Пример использования функции `ctime()`:

```
char *p = NULL;
time_t t = time(NULL);
```

```
p = ctime(&t);
if (!p) {
   printf("Error\n");
   exit(1);
}
printf("%s", p); // Fri Jan 25 06:07:37 2019\n
```

Вместо этих функций лучше использовать функции `ctime_s()`, `_ctime64_s()`, `_wctime_s()` и `_wctime64_s()`. Прототипы функций:

```
#include <time.h>
errno_t ctime_s(char *buf, size_t sizeInBytes,
                const time_t *Time);
errno_t _ctime64_s(char *buf, size_t sizeInBytes,
                   const __time64_t *Time);
#include <time.h> /* или #include <wchar.h> */
errno_t _wctime_s(wchar_t *buf, size_t sizeInWords,
                  const time_t *Time);
errno_t _wctime64_s(wchar_t *buf, size_t sizeInWords,
                    const __time64_t *Time);
```

Если ошибок нет, то функции возвращают значение `0`. При наличии ошибки возвращается значение макроса `EINVAL`. Пример использования функции `ctime_s()`:

```
char str[80] = {0};
time_t t = time(NULL);
errno_t err = ctime_s(str, 80, &t);
if (err) {
   printf("Error\n");
   exit(1);
}
printf("%s", str); // Fri Jan 25 06:10:19 2019\n
```

- `strftime()` и `wcsftime()` — записывают строковое представление даты `Tm` в соответствии со строкой формата `format` в строку `buf`. Прототипы функций:

```
#include <time.h>
size_t strftime(char *buf, size_t sizeInBytes,
                const char *format, const struct tm *Tm);
#include <time.h> /* или #include <wchar.h> */
size_t wcsftime(wchar_t *buf, size_t sizeInWords,
                const wchar_t *format, const struct tm *Tm);
```

В первом параметре передается указатель на символьный массив, в который будет записан результат выполнения функции. Во втором параметре задается максимальный размер символьного массива. В параметре `format` указывается строка специального формата, а в последнем параметре передается указатель на структуру `tm` с представлением даты. Функции возвращают количество записанных символов. В случае ошибки возвращается значение `0` и переменная `errno` устанавливается равной `EINVAL`. Функции зависят от настроек локали.

В параметре `format` в функциях `strftime()` и `wcsftime()` помимо обычных символов могут быть указаны следующие комбинации специальных символов:

- `%y` — год из двух цифр (от `00` до `99`);
- `%Y` — год из четырех цифр (например, `2019`);
- `%m` — номер месяца с предваряющим нулем (от `01` до `12`);
- `%b` — аббревиатура месяца в зависимости от настроек локали (например, `Сен` в локали `Russian_Russia.1251` или `Sep` в локали `C` для сентября);
- `%B` — название месяца в зависимости от настроек локали (например, `Сентябрь` или `September`);
- `%d` — номер дня в месяце с предваряющим нулем (от `01` до `31`);
- `%j` — день с начала года (от `001` до `366`);
- `%U` — номер недели в году (от `00` до `53`), неделя начинается с воскресенья;
- `%W` — номер недели в году (от `00` до `53`), неделя начинается с понедельника;
- `%w` — номер дня недели (`0` — для воскресенья, `6` — для субботы);
- `%a` — аббревиатура дня недели в зависимости от настроек локали (например, `Ср` или `Wed` для среды);
- `%A` — название дня недели в зависимости от настроек локали (например, `среда` или `Wednesday`);
- `%H` — часы в 24-часовом формате (от `00` до `23`);
- `%I` — часы в 12-часовом формате (от `01` до `12`);
- `%M` — минуты (от `00` до `59`);
- `%S` — секунды (от `00` до `59`, изредка до `61`);
- `%p` — эквивалент значениям `AM` и `PM` в текущей локали;
- `%c` — представление даты и времени в текущей локали (например, `25.01.2019 7:24:01` или `01/25/19 07:24:01`);
- `%#c` — расширенное представление даты и времени в текущей локали (например, `25 Январь 2019 г. 7:26:01` или `Friday, January 25, 2019 07:26:01`);
- `%x` — представление даты в текущей локали (например, `25.01.2019` или `01/25/19`);
- `%#x` — расширенное представление даты в текущей локали (например, `25 Январь 2019 г.` или `Friday, January 25, 2019`);
- `%X` — представление времени в текущей локали (например, `07:29:44`);
- `%Z` — название часового пояса или пустая строка (например, `RTZ 2 (зима)`);
- `%%` — символ `%`.

Если после символа `%` указан символ `#` в комбинациях `%#d`, `%#H`, `%#I`, `%#j`, `%#m`, `%#M`, `%#S`, `%#U`, `%#W`, `%#y` и `%#Y`, то предваряющие нули выводиться не будут. В качестве примера использования функции `strftime()` выведем текущую дату и время (листинг 9.2).

Листинг 9.2. Форматирование даты и времени

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <time.h>

#define SIZE 100

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   struct tm *ptm = NULL;
   char str[SIZE] = {0};
   time_t t = time(NULL);
   ptm = localtime(&t);
   if (!ptm) {
      printf("Error\n");
      exit(1);
   }
   size_t err2 = strftime(str, SIZE,
                "Сегодня:\n%A %d %b %Y %H:%M:%S\n%d.%m.%Y", ptm);
   if (!err2) {
      printf("Error\n");
      exit(1);
   }
   printf("%s\n", str);
   return 0;
}
```

Результат выполнения:

```
Сегодня:
пятница 25 янв 2019 16:17:15
25.01.2019
```

9.3. «Засыпание» программы

В MinGW прерывать выполнение программы на указанное время (по истечении срока программа продолжит работу) позволяют следующие функции.

- `sleep()` — **прерывает выполнение текущего потока на указанное количество секунд. Прототип функции:**

  ```
  #include <unistd.h>
  unsigned int sleep(unsigned int seconds)
  ```

  Пример указания одной секунды:

  ```
  sleep(1);          // Засыпаем на секунду
  ```

- `usleep()` — прерывает выполнение текущего потока на указанное количество микросекунд. Прототип функции:

```
#include <unistd.h>
int usleep(useconds_t usec);
typedef unsigned int useconds_t;
```

Пример указания одной секунды:

```
usleep(1000000); // Засыпаем на секунду
```

- `nanosleep()` — прерывает выполнение текущего потока на указанное количество секунд и наносекунд. Прототип функции:

```
#include <time.h>
int nanosleep(const struct timespec *request,
              struct timespec *remain);
struct timespec {
  time_t  tv_sec;   /* Секунды */
  long    tv_nsec;  /* Наносекунды */
};
```

В первом параметре указывается адрес структуры со временем задержки, а во втором — адрес структуры, в которую будет записан результат (можно указать значение `NULL`, если результат не важен). Пример указания одной секунды:

```
struct timespec tw = {1, 0}, tr;
nanosleep(&tw, &tr); // Засыпаем на секунду
```

В Windows можно воспользоваться функцией из WinAPI `Sleep()`. Прототип функции:

```
#include <windows.h>
VOID Sleep(DWORD dwMilliseconds);
```

В параметре `dwMilliseconds` указывается количество миллисекунд, на которое прерывается выполнение текущего потока. Тип данных `DWORD` объявлен так:

```
typedef unsigned long DWORD;
```

Для примера выведем числа от `1` до `10` (листинг 9.3). Между выводом чисел "заснем" на одну секунду.

Листинг 9.3. "Засыпание" программы

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

int main(void) {
   printf("Start\n");
   for (int i = 1; i <= 10; ++i) {
      Sleep(1000); // Засыпаем на секунду
      printf("i = %d\n", i);
```

```
        fflush(stdout);
    }
    printf("End\n");
    return 0;
}
```

9.4. Измерение времени выполнения фрагментов кода

В некоторых случаях необходимо измерить время выполнения фрагментов кода, например, с целью оптимизации программы. Измерить время выполнения позволяет функция `clock()`. Прототип функции:

```
#include <time.h>
clock_t clock(void);
typedef long clock_t;
```

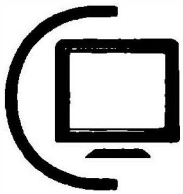
Функция возвращает приблизительное время выполнения программы до вызова функции. Если время получить не удалось, функция возвращает значение `(clock_t)(-1)`. Для того чтобы измерить время выполнения фрагмента, следует вызвать функцию перед фрагментом кода и сохранить результат. Затем вызвать функцию после фрагмента и вычислить разность между двумя значениями. Для того чтобы получить значение в секундах, необходимо разделить результат на значение макроса `CLOCKS_PER_SEC`. Определение макроса выглядит следующим образом:

```
#define CLOCKS_PER_SEC 1000
```

Для примера имитируем фрагмент кода с помощью функции `Sleep()` из WinAPI и произведем измерение времени выполнения (листинг 9.4).

Листинг 9.4. Измерение времени выполнения

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

int main(void) {
    clock_t t1, t2, t3;
    t1 = clock();                          // Метка 1
    printf("t1 = %ld\n", t1);
    fflush(stdout);
    Sleep(3000);                           // Имитация фрагмента кода
    t2 = clock();                          // Метка 2
    printf("t2 = %ld\n", t2);
    t3 = t2 - t1;                          // Разница между метками
    printf("t3 = %ld\n", t3);
    printf("%.2f sec.\n", (double)t3 / CLOCKS_PER_SEC);
    return 0;
}
```



ГЛАВА 10

Пользовательские функции

Функция — это фрагмент кода, который можно неоднократно вызвать из любого места программы. В предыдущих главах мы уже не один раз использовали встроенные функции, входящие в состав стандартной библиотеки. Например, с помощью функции `strlen()` получали количество символов в С-строке. В этой главе мы рассмотрим создание пользовательских функций, которые позволят уменьшить избыточность программного кода и повысить его структурированность.

10.1. Создание функции и ее вызов

Описание функции состоит из двух частей: *объявления* и *определения*. Объявление функции (называемое также *прототипом функции*) содержит информацию о типе. Используя эту информацию, компилятор может найти несоответствие типа и количества параметров. Формат прототипа функции:

```
<Тип результата> <Название функции>([<Тип> [<Название параметра 1>]
                                [, ..., <Тип> [<Название параметра N>]]]);
```

Параметр `<Тип результата>` задает тип значения, которое возвращает функция с помощью оператора `return`. Если функция не возвращает никакого значения, то вместо типа указывается ключевое слово `void`. Название функции должно быть допустимым идентификатором, к которому предъявляются такие же требования, как и к названиям переменных. После названия функции, внутри круглых скобок, указывается тип и названия параметров через запятую. Названия параметров в прототипе функции можно не задавать вообще, т. к. компилятор интересует только тип данных и количество параметров. Если функция не принимает параметров, то указываются круглые скобки, внутри которых задается ключевое слово `void`. После объявления функции должна стоять точка с запятой. Пример объявления функций:

```
int sum(int x, int y);          // или int sum(int, int);
void print(const char *str);    // или void print(const char *);
void print_ok(void);
```

Определение функции содержит описание типа и названия параметров, а также реализацию. Определение функции имеет следующий формат:

```
<Тип результата> <Название функции>([<Тип> <Название параметра 1>
                                [, ..., <Тип> <Название параметра N>]])
{
   <Тело функции>
   [return[ <Возвращаемое значение>];]
}
```

В отличие от прототипа в определении функции после типа обязательно должно быть указано название параметра, которое является локальной переменной. Эта переменная создается при вызове функции, а после выхода из функции она удаляется. Таким образом, локальная переменная видна только внутри функции. Если название локальной переменной совпадает с названием глобальной переменной, то все операции будут производиться с локальной переменной, а значение глобальной не изменится. Пример:

```
int sum(int x, int y) {
   int z = x + y; // Обращение к локальным переменным x и y
   return z;
}
```

После описания параметров, внутри фигурных скобок, размещаются инструкции, которые будут выполняться при каждом вызове функции. Фигурные скобки указываются в любом случае, даже если тело функции состоит только из одной инструкции. Точка с запятой после закрывающей фигурной скобки не указывается.

Вернуть значение из функции позволяет оператор `return`. После исполнения этого оператора выполнение функции останавливается, и управление передается обратно в точку вызова функции. Это означает, что инструкции после оператора `return` никогда не будут выполнены. При использовании оператора `return` не должно быть неоднозначных ситуаций. Например, в этом случае возвращаемое значение зависит от условия:

```
int sum(int x, int y) {
   if (x > 0) {
      return x + y;
   }
}
```

Если переменная `x` имеет значение больше нуля, то все будет нормально, но если переменная равна нулю или имеет отрицательное значение, то возвращаемое значение не определено и функция вернет произвольное значение, так называемый "мусор". В этом случае при компиляции выводится предупреждающее сообщение: `warning: control reaches end of non-void function`. Для того чтобы избежать подобной ситуации, следует в конце тела функции вставить оператор `return` со значением по умолчанию:

```
int sum(int x, int y) {
   if (x > 0) {
      return x + y;
   }
```

```
    return 0;
}
```

Если перед названием функции указано ключевое слово `void`, то оператора `return` может не быть. Однако если необходимо досрочно прервать выполнение функции, то оператор `return` указывается без возвращаемого значения. Пример:

```
void print_ok(void) {
    puts("OK");
    return; // Преждевременное завершение функции
    puts("Эта инструкция никогда не будет выполнена!!!");
}
```

При вызове функции из программы указывается название функции, после которого внутри круглых скобок передаются значения. Если функция не принимает параметров, то указываются только круглые скобки. Если функция возвращает значение, то его можно присвоить переменной или просто проигнорировать. Необходимо заметить, что количество и тип параметров в объявлении функции должны совпадать с количеством и типом параметров при вызове. Пример вызова трех функций:

```
print("Message");     // Функция выведет сообщение Message
print_ok();           // Вызываем функцию без параметров
int z = sum(10, 20); // Переменной z будет присвоено значение 30
```

Переданные значения присваиваются переменным, расположенным в той же позиции в определении функции. Так, при использовании функции `sum()` переменной `x` будет присвоено значение `10`, а переменной `y` — значение `20`. Результат выполнения функции присваивается переменной `z`.

В качестве примера создадим три разные функции и вызовем их (листинг 10.1).

Листинг 10.1. Создание функций и их вызов

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

// Объявления функций
int sum(int x, int y);          // или int sum(int, int);
void print(const char *str);    // или void print(const char *);
void print_ok(void);

int main(void) {
    setlocale(LC_ALL, "Russian_Russia.1251");
    // Вызов функций
    print("Сообщение");          // Сообщение
    print_ok();                 // OK
    int z = sum(10, 20);
    printf("%d\n", z);          // 30
    return 0;
}
```

```
// Определения функций
int sum(int x, int y) {          // Два параметра
   return x + y;
}
void print(const char *str) {    // Один параметр
   printf("%s\n", str);
}
void print_ok(void) {            // Без параметров
   puts("OK");
}
```

10.2. Расположение объявлений и определений функций

Все инструкции в программе выполняются последовательно сверху вниз. Это означает, что прежде чем использовать функцию в программе, ее необходимо предварительно объявить. Поэтому объявление функции должно быть расположено перед вызовом функции. Обратите внимание на то, что размещать определение одной функции внутри другой нельзя. Таким образом, название функции всегда является глобальным идентификатором.

В небольших программах допускается объявление функции не указывать, при условии, что определение функции расположено перед функцией `main()` (листинг 10.2). Кстати, функция `main()` также не требует объявления, т. к. она вызывается первой.

Листинг 10.2. Расположение определения функции перед функцией `main()`

```
#include <stdio.h>

int sum_i(int x, int y) {
   return x + y;
}
int main(void) {
   int z = sum_i(10, 20);
   printf("z = %d\n", z); // z = 30
   return 0;
}
```

При увеличении количества функций возникает ситуация, когда внутри функции вызывается вторая функция, а внутри второй — третья и т. д. В результате приходится решать вопрос: что было раньше — курица или яйцо? Для того чтобы избежать такой ситуации, объявления функций следует размещать в начале программы перед функцией `main()`, а определения — после функции `main()` (листинг 10.3). В этом случае порядок следования определений функций не имеет значения.

Листинг 10.3. Расположение объявлений и определений функций

```
#include <stdio.h>

int sum_i(int, int);          // Объявление

int main(void) {
   int z = sum_i(10, 20);
   printf("z = %d\n", z);     // z = 30
   return 0;
}
int sum_i(int x, int y) {     // Определение
   return x + y;
}
```

При увеличении размера программы объявлений и определений функций становится все больше и больше. В этом случае программу разделяют на несколько отдельных файлов. Объявления функций выносят в заголовочный файл с расширением h, а определения функций размещают в одноименном файле с расширением c. Все файлы располагают в одном каталоге с основным файлом, содержащим функцию `main()`. В дальнейшем с помощью директивы `#include` подключают заголовочный файл во всех остальных файлах. Если в директиве `#include` название заголовочного файла указывается внутри угловых скобок, то поиск файла осуществляется в путях поиска заголовочных файлов. Если название указано внутри кавычек, то поиск вначале производится в каталоге с основным файлом, а затем в путях поиска заголовочных файлов.

В качестве примера вынесем функцию `sum_i()` в отдельный файл (например, с названием my_module.c), а затем подключим его к основному файлу. Для создания файла my_module.c в окне **Project Explorer** щелкаем правой кнопкой мыши на названии каталога src и из контекстного меню выбираем пункт **New | Source File** (см. рис. 2.2). В открывшемся окне (см. рис. 2.4) в поле **Source folder** должно быть уже вставлено значение, например, `Test64c/src`. В поле **Source file** вводим `my_module.c`, а из списка **Template** выбираем пункт **<None>**. Нажимаем кнопку **Finish**.

В результате файл будет добавлен в каталог проекта, и его название отобразится в окне **Project Explorer**, а сам файл будет открыт на отдельной вкладке. Вставляем в этот файл код из листинга 10.4.

Листинг 10.4. Файл my_module.c

```
#include "my_module.h"

int sum_i(int x, int y) { // Определение
   return x + y;
}
```

Для создания файла my_module.h в окне **Project Explorer** щелкаем правой кнопкой мыши на названии каталога src и из контекстного меню выбираем пункт **New | Header File** (см. рис. 2.2). В открывшемся окне (см. рис. 2.6) в поле **Source folder** должно быть уже вставлено значение, например, `Test64c/src`. В поле **Header file** вводим `my_module.h`, а из списка **Template** выбираем пункт **Default C header template**. Нажимаем кнопку **Finish**.

В результате файл будет добавлен в каталог проекта, и его название отобразится в окне **Project Explorer**, а сам файл будет открыт на отдельной вкладке. Вставляем в этот файл код из листинга 10.5.

Листинг 10.5. Файл my_module.h

```
#ifndef MY_MODULE_H_
#define MY_MODULE_H_

#include <stdio.h>
int sum_i(int, int);          // Объявление функции

#endif /* MY_MODULE_H_ */
```

Все содержимое файла my_module.h расположено внутри условия, которое проверяется перед компиляцией. Условие выглядит следующим образом:

```
#ifndef MY_MODULE_H_
// Инструкции
#endif /* MY_MODULE_H_ */
```

Это условие следует читать так: если не существует макроопределение MY_MODULE_H_, то вставить инструкции в то место, где подключается файл. Название макроопределения обычно совпадает с названием заголовочного файла. Только все буквы указываются в верхнем регистре и точка заменяется символом подчеркивания. Условие начинается с директивы `#ifndef` и заканчивается директивой `#endif`. Все это необходимо, чтобы объявления идентификаторов не вставлялись дважды. Для этого в первой инструкции внутри условия определяется макрос `MY_MODULE_H_` с помощью директивы `#define`. В этом случае повторная проверка условия вернет ложное значение.

Вместо этих директив можно указать в самом начале заголовочного файла директиву препроцессора `#pragma` со значением `once`, которая также препятствует повторному включению файла (в старых компиляторах директива может не поддерживаться):

```
#pragma once
// Объявление функций и др.
```

Теперь, чтобы использовать функцию `sum_i()`, достаточно подключить файл my_module.h к основной программе. Пример подключения приведен в листинге 10.6.

Листинг 10.6. Основная программа

```
#include "my_module.h"

int main(void) {
   int z = sum_i(10, 20);
   printf("z = %d\n", z); // z = 30
   return 0;
}
```

В директиве `#include` допускается указывать не только название файла, но и абсолютный или относительный путь к нему. Это позволяет размещать файлы по различным каталогам. Примеры указания абсолютного и относительного пути:

```
#include "C:\\cpp\\projects\\Test64c\\src\\my_module.h"
#include "C:/cpp/projects/Test64c/src/my_module.h"
#include "/cpp/projects/Test64c/src/my_module.h"
#include "folder/my_module.h"
```

При использовании больших программ создают статическую (файлы с расширением lib в Visual C или с расширением a в MinGW) или динамическую (файлы с расширением dll в Windows) библиотеку. Статические библиотеки становятся частью программы при компиляции, а динамические библиотеки подгружаются при запуске программы. Процесс создания библиотек мы будем изучать в *главе 16*.

10.3. Способы передачи параметров в функцию

Как вы уже знаете, после названия функции, внутри круглых скобок, указываются тип и названия параметров через запятую. Если функция не принимает параметров, то внутри круглых скобок задается ключевое слово `void`. В определении функции после типа обязательно должно быть указано название параметра, которое является локальной переменной. Эта переменная создается при вызове функции, а после выхода из функции она удаляется. Таким образом, локальная переменная видна только внутри функции, и ее значение между вызовами не сохраняется (исключением являются статические переменные). Если название локальной переменной совпадает с названием глобальной переменной, то все операции будут производиться с локальной переменной, а значение глобальной не изменится.

При вызове функции указывается название функции, после которого внутри круглых скобок передаются значения. Количество и тип параметров в объявлении функции должны совпадать с количеством и типом параметров при вызове. Переданные значения присваиваются переменным, расположенным в той же позиции в определении функции. Так, при вызове функции `sum(10, 20)` (прототип `int sum(int x, int y)`) переменной `x` будет присвоено значение `10`, а переменной `y` —

значение 20. Если функция не принимает параметров, то при вызове указываются только круглые скобки.

По умолчанию в функцию передается копия значения переменной. Таким образом, изменение значения внутри функции не затронет значение исходной переменной. Пример передачи параметра по значению приведен в листинге 10.7.

Листинг 10.7. Передача параметра по значению

```
#include <stdio.h>

void func(int x);

int main(void) {
   int x = 10;
   func(x);
   printf("%d\n", x); // 10
   return 0;
}
void func(int x) {
   x = x + 20;        // Значение нигде не сохраняется!
}
```

Передача копии значения для чисел является хорошим решением, но при использовании массивов, а также при необходимости изменить значение исходной переменной, лучше применять передачу указателя. Для этого при вызове функции перед названием переменной указывается оператор & (взятие адреса), а в прототипе функции объявляется указатель. В этом случае в функцию передается не значение переменной, а ее адрес. Внутри функции вместо переменной используется указатель. Пример передачи указателя приведен в листинге 10.8.

Листинг 10.8. Передача указателя в функцию

```
#include <stdio.h>

void func(int *px);

int main(void) {
   int x = 10;
   func(&x);          // Передаем адрес
   printf("%d\n", x); // 30, а не 10
   return 0;
}
void func(int *px) {
   *px = *px + 20;    // Изменяется значение переменной x
}
```

10.4. Передача массивов и строк в функцию

Передача одномерных массивов и строк осуществляется с помощью указателей. Обратите внимание на то, что при вызове функции перед названием переменной не нужно добавлять оператор &, т. к. название переменной содержит адрес первого элемента массива. Пример передачи С-строки приведен в листинге 10.9.

Листинг 10.9. Передача С-строки в функцию

```
#include <stdio.h>

void func1(char *s);
void func2(char s[]);

int main(void) {
   char str[] = "String";
   printf("%d\n", (int) sizeof(str)); // 7
   func1(str);          // Оператор & перед str не указывается!!!
   func2(str);
   printf("%s\n", str); // stringG
   return 0;
}
void func1(char *s) {
   s[0] = 's';          // Изменяется значение элемента массива str
   printf("%d\n", (int) sizeof(s));
   // 8 (размер указателя в проекте Test64c), а не 7!!!
}
void func2(char s[]) {
   s[5] = 'G'; // Изменяется значение элемента массива str
}
```

Объявление `char *s` эквивалентно объявлению `char s[]`. И в том и в другом случае объявляется указатель на тип `char`. Чаще используется первый способ. Обратите внимание на значения, возвращаемые оператором `sizeof` вне и внутри функции. Вне функции оператор возвращает размер всего символьного массива, в то время как внутри функции оператор `sizeof` возвращает только размер указателя. Это происходит потому, что внутри функции переменная `s` является указателем, а не массивом. Поэтому если внутри функции необходимо знать размер массива, то количество элементов (или размер в байтах) следует передавать в дополнительном параметре.

При вызове функции в качестве параметра можно передать *составной литерал*. Литерал состоит из круглых скобок, внутри которых указываются тип данных и количество элементов внутри квадратных скобок. После круглых скобок указываются фигурные скобки, внутри которых через запятую перечисляются значения:

```
const long *p = (long [3]) {1, 2, 3};
printf("%ld ", *p);          // 1
printf("%ld ", *(p + 1));    // 2
printf("%ld ", *(p + 2));    // 3
```

Количество элементов внутри квадратных скобок можно не указывать. Размер массива будет соответствовать количеству элементов внутри фигурных скобок:

```
const long *p = (long []) {1, 2, 3};
```

При передаче многомерного массива необходимо явным образом указывать все размеры (например, `int a[2][4]`). Самый первый размер допускается не указывать (например, `int a[][4]`). Пример передачи двумерного массива в функцию приведен в листинге 10.10.

Листинг 10.10. Передача двумерного массива

```
#include <stdio.h>

#define ARR_ROWS 2
#define ARR_COLS 4

void func(int a[][4]);

int main(void) {
   int i, j;
   int arr[ARR_ROWS][ARR_COLS] = {
      {1, 2, 3, 4},
      {5, 6, 7, 8}
   };
   func(arr);
   // Выводим значения
   for (i = 0; i < ARR_ROWS; ++i) {
      for (j = 0; j < ARR_COLS; ++j) {
         printf("%2d ", arr[i][j]);
      }
      printf("\n");
   }
   return 0;
}
void func(int a[][4]) { // или void func(int a[2][4])
   a[0][0] = 80;
}
```

Такой способ передачи многомерного массива нельзя назвать универсальным, т. к. существует жесткая привязка к размеру. Одним из способов решения проблемы является создание дополнительного массива указателей. В этом случае в функцию

передается адрес первого элемента массива указателей, а объявление параметра в функции выглядит так:

```
int *a[]
```

или так:

```
int **a
```

Пример передачи массива указателей приведен в листинге 10.11.

Листинг 10.11. Передача массива указателей

```
#include <stdio.h>

#define ARR_ROWS 2
#define ARR_COLS 4

void func(int *a[], int rows, int cols);

int main(void) {
   int arr[ARR_ROWS][ARR_COLS] = {
      {1, 2, 3, 4},
      {5, 6, 7, 8}
   };
   // Создаем массив указателей
   int *parr[] = {arr[0], arr[1]};
   // Передаем адрес массива указателей
   func(parr, ARR_ROWS, ARR_COLS);
   return 0;
}
void func(int *a[], int rows, int cols) {
   // или void func(int **a, int rows, int cols)
   // Выводим значения
   for (int i = 0, j; i < rows; ++i) {
      for (j = 0; j < cols; ++j) {
         printf("%2d ", a[i][j]);
      }
      printf("\n");
   }
}
```

Однако и этот способ имеет недостаток, т. к. нужно создавать дополнительный массив указателей. Кроме того, в Visual C выводится предупреждение (warning C4221), т. к. мы сохраняем адреса локальной области видимости. Наиболее приемлемым способом является передача многомерного массива как одномерного. В этом случае в функцию передается адрес первого элемента массива, а в параметре объявляется указатель. Так как все элементы многомерного массива располага-

ются в памяти последовательно, зная количество элементов или размеры, можно вычислить положение текущего элемента, используя адресную арифметику. Пример передачи двумерного массива как одномерного показан в листинге 10.12.

Листинг 10.12. Передача двумерного массива как одномерного

```
#include <stdio.h>

#define ARR_ROWS 2
#define ARR_COLS 4

void func(int *pa, int rows, int cols);

int main(void) {
   int arr[ARR_ROWS][ARR_COLS] = {
      {1, 2, 3, 4},
      {5, 6, 7, 8}
   };
   // Передаем в функцию адрес первого элемента массива
   func(arr[0], ARR_ROWS, ARR_COLS);
   return 0;
}
void func(int *pa, int rows, int cols) {
   // Выводим значения
   for (int i = 0, j; i < rows; ++i) {
      for (j = 0; j < cols; ++j) {
         // Вычисляем положение элемента
         printf("%2d ", *(pa + i * cols + j));
         //printf("%2d ", pa[i * cols + j]);
      }
      printf("\n");
   }
}
```

Передача в функцию массива С-строк осуществляется так же, как передача массива указателей. Для того чтобы в функцию не передавать количество элементов, можно при инициализации массива С-строк последнему элементу присвоить нулевой указатель. Этот элемент будет служить ориентиром конца массива. В качестве примера выведем все строки внутри функции (листинг 10.13).

Листинг 10.13. Передача массива С-строк

```
#include <stdio.h>

void func(char *s[]);
```

```
int main(void) {
   char *str[] = {"String1", "String2", "String3",
      NULL // Вставляем нулевой указатель, чтобы был ориентир
   };
   func(str);
   return 0;
}
void func(char *s[]) { // или void func(char **s)
   while (*s) {        // Выводим все строки
      printf("%s\n", *s);
      ++s;
   }
}
```

10.5. Переменное количество параметров

Количество параметров в функции может быть произвольным при условии, что существует один обязательный параметр. В объявлении и определении функции переменное число параметров обозначается тремя точками. Например, прототип функции `printf()` выглядит так:

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

Получить доступ к этим параметрам внутри функции можно с помощью специальных макросов `va_start()`, `va_arg()` и `va_end()`:

```
#include <stdarg.h>
void va_start(va_list <Указатель>, <Последний параметр>)
<Тип> va_arg(va_list <Указатель>, <Тип данных>)
void va_end(va_list <Указатель>)
```

Вначале объявляется указатель типа `va_list`. Далее должна производиться инициализация указателя с помощью макроса `va_start()`. В первом параметре передается указатель, а во втором — название последнего обязательного параметра. Доступ к параметрам осуществляется с помощью макроса `va_arg()`, который возвращает значение текущего параметра и перемещает указатель на следующий параметр. В первом параметре макроса `va_arg()` передается указатель, а во втором — название типа данных. Макрос `va_end()` сообщает об окончании перебора параметров. Количество параметров обычно указывается в обязательном параметре в виде числа или определяется другим способом (например, количеством спецификаторов внутри строки формата функции `printf()`).

Если ожидается произвольное количество значений, то их может не быть и вовсе:

```
printf("10 20");                    // 10 20
```

В качестве примера напишем функцию суммирования произвольного количества целых чисел (листинг 10.14).

Листинг 10.14. Суммирование произвольного количества целых чисел

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

int sum(int n, ...);

int main(void) {
   printf("%d\n", sum(2, 20, 30));      // 50
   printf("%d\n", sum(3, 1, 2, 3));     // 6
   printf("%d\n", sum(4, 1, 2, 3, 4)); // 10
   return 0;
}
int sum(int n, ...) {
   int result = 0;
   va_list p;
   va_start(p, n);
   for(int i = 0; i < n; ++i) {
      result += va_arg(p, int);
   }
   va_end(p);
   return result;
}
```

На заметку

При передаче в функцию значения типов `char` **и** `short` **автоматически расширяются до типа** `int`**, а значение типа** `float` **— до типа** `double`.

10.6. Константные параметры

Если внутри функции значение параметра не изменяется, то такой параметр следует объявить константным. Для этого при объявлении перед параметром указывается ключевое слово `const`**. Например, функция** `sum()`**, предназначенная для суммирования чисел, не производит изменение значений параметров, поэтому параметры можно объявить константными:**

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

При использовании указателей важно учитывать местоположение ключевого слова `const`**. Например, следующие объявления не эквивалентны:**

```
void print(const char *s);
void print(char const *s);
void print(char * const s);
void print(const char * const s);
```

Первые два объявления являются эквивалентными. В этом случае изменить значение, на которое ссылается указатель, нельзя, но указателю можно присвоить другой адрес:

```
void print(const char *s) {
   char s2[] = "New";
   s = s2;                              // Нормально
   s[0] = 's';                          // Ошибка
   printf("%s", s);
}
```

При третьем объявлении изменить значение, на которое ссылается указатель, можно, но указателю нельзя присвоить другой адрес:

```
void print(char * const s) {
   char s2[] = "New";
   s = s2;                              // Ошибка
   s[0] = 's';                          // Нормально
   printf("%s", s);
}
```

Четвертое объявление запрещает изменение значения, на которое ссылается указатель, и присвоение другого адреса:

```
void print(const char * const s) {
   char s2[] = "New";
   s = s2;                              // Ошибка
   s[0] = 's';                          // Ошибка
   printf("%s", s);
}
```

10.7. Статические переменные и функции

Переменные, указанные в параметрах, а также переменные, объявленные внутри функции, являются локальными переменными. Эти переменные создаются при вызове функции, а после выхода из функции они удаляются. Таким образом, локальная переменная видна только внутри функции. Если внутри функции при объявлении локальной переменной не было присвоено начальное значение, то переменная будет содержать произвольное значение, так называемый "мусор". Если название локальной переменной совпадает с названием глобальной переменной, то все операции будут производиться с локальной переменной, а значение глобальной не изменится.

Пример сохранения промежуточного значения между вызовами функции в глобальной переменной приведен в листинге 10.15.

Листинг 10.15. Сохранение промежуточного значения в глобальной переменной

```
#include <stdio.h>

int x = 0;
void sum(int _x);
```

```
int main(void) {
   sum(10);
   sum(20);
   printf("%d\n", x); // 30
   return 0;
}
void sum(int _x) {
   x += _x;
}
```

Статические переменные позволяют отказаться от использования глобальных переменных, для сохранения промежуточных значений между вызовами функции. Инициализация статической переменной производится только при первом вызове функции. Если при объявлении статической переменной не присвоено начальное значение, то переменная автоматически инициализируется нулевым значением. После завершения работы функции статическая переменная сохраняет свое значение, которое доступно при следующем вызове функции. При объявлении статической переменной перед типом данных указывается ключевое слово `static`. В качестве примера переделаем предыдущий пример и используем статическую переменную вместо глобальной (листинг 10.16).

Листинг 10.16. Использование статических переменных

```
#include <stdio.h>

int sum(int _x);

int main(void) {
   printf("%d\n", sum(10)); // 10
   printf("%d\n", sum(20)); // 30
   printf("%d\n", sum(55)); // 85
   return 0;
}
int sum(int _x) {
   static int x = 0; // Статическая переменная
   x += _x;
   return x;
}
```

Ключевое слово `static` можно также указать для глобальной переменной или функции. В этом случае область видимости будет ограничена текущим файлом. Пример определения статической функции:

```
static void print_ok(void) {
   puts("OK");
}
```

10.8. Способы возврата значения из функции

Вернуть значение из функции позволяет оператор `return`. После исполнения этого оператора выполнение функции останавливается, и управление передается обратно в точку вызова функции. Это означает, что инструкции после оператора `return` никогда не будут выполнены. Если внутри функции нет оператора `return`, то по достижении закрывающей фигурной скобки управление будет передано в точку вызова функции. В этом случае возвращаемое значение не определено.

Если функция не возвращает никакого значения, то вместо типа данных в объявлении и определении функции указывается ключевое слово `void`. Внутри такой функции оператора `return` может не быть, однако его можно использовать без указания значения для преждевременного выхода из функции. Вызов функции, не возвращающей никакого значения, нельзя размещать внутри какой-либо инструкции. Только в отдельной инструкции. Пример функции, которая не возвращает никакого значения:

```
void print(const char *s) {
   printf("%s\n", s);
}
```

В остальных случаях в объявлении и определении функции перед названием функции задается возвращаемый тип данных. Значение этого типа должно быть указано в операторе `return`. В этом случае возвращается копия значения. Функция может вернуть значение любого типа, кроме массива. Работать с массивом необходимо через параметры функции, передавая указатель на него или возвращая указатель на конкретный элемент. Вызов функции, возвращающей какое-либо значение, можно разместить внутри выражения с правой стороны от оператора `=`. Возвращаемое значение можно присвоить переменной или просто проигнорировать. Пример функции, возвращающей копию значения:

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

При использовании оператора `return` не должно быть неоднозначных ситуаций. Например, в этом случае возвращаемое значение зависит от условия:

```
int sum(int x, int y) {
   if (x > 0) {
      return x + y;
   }
}
```

Если переменная `x` имеет значение больше нуля, то все будет нормально, но если переменная равна нулю или имеет отрицательное значение, то возвращаемое значение не определено и функция вернет произвольное значение, так называемый

"мусор". В этом случае при компиляции выводится предупреждающее сообщение: `warning: control reaches end of non-void function`. Для того чтобы избежать подобной ситуации, следует в конце тела функции вставить оператор `return` со значением по умолчанию:

```
int sum(int x, int y) {
   if (x > 0) {
      return x + y;
   }
   return 0;
}
```

Функция может возвращать указатель. В этом случае в объявлении и определении функции указывается соответствующий тип указателя. В качестве примера вернем указатель на последний символ строки или нулевой указатель, если строка пустая или параметр имеет значение `NULL` (листинг 10.17).

Листинг 10.17. Возврат указателя

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

char *func(char *s);

int main(void) {
   char *p = NULL, str[] = "String";
   p = func(str);
   if (p) {                    // Если ненулевой указатель
      printf("%c\n", *p);      // g
   }
   else puts("NULL");
   return 0;
}
char *func(char *s) {
   if (!s) return NULL;
   size_t len = strlen(s);     // Получаем длину строки
   if (!len) return NULL;      // Нулевой указатель, если пусто
   else return &s[len - 1];    // Указатель на последний символ
}
```

Обратите внимание

Нельзя возвращать указатель на переменные или массивы, объявленные внутри функции, т. к. при выходе из функции локальные переменные будут удалены. Работать с массивом необходимо через параметры функции, передавая указатель на него или возвращая указатель на конкретный элемент.

10.9. Указатели на функции

Функции так же, как и переменные, имеют адрес, который можно сохранить в указателе. В дальнейшем через указатель можно вызвать эту функцию. Кроме того, допускается передавать указатель на функцию в качестве параметра другой функции. Объявление указателя на функцию выглядит так:

```
<Тип> (*<Название указателя>)([<Тип 1>[, ..., <Тип N>]]);
```

Для того чтобы присвоить указателю адрес функции, необходимо указать название функции без параметров и круглых скобок справа от оператора =. Типы параметров указателя и функции должны совпадать. Пример объявления указателя и присваивания адреса функции:

```
int (*pfunc)(int, int) = NULL;
pfunc = sum;
```

Вызвать функцию через указатель можно так:

```
int x = pfunc(30, 10); // Аналогично x = sum(10, 20);
```

Или так:

```
int y = (*pfunc)(30, 10);
```

Пример объявления указателя, вызова функции через указатель и передачи указателя в качестве параметра функции приведен в листинге 10.18.

Листинг 10.18. Указатели на функции

```
#include <stdio.h>

void print_ok(void);
int add(int x, int y);
int sub(int x, int y);
int func(int x, int y, int (*pfunc)(int, int));

int main(void) {
   // Объявление указателей на функции
   void (*pf1)(void) = NULL;
   int (*pf2)(int, int) = NULL;
   // Присваивание адреса функции
   pf1 = print_ok;
   pf2 = add;
   // Вызов функций через указатели
   pf1();                              // OK
   printf("%d\n", pf2(30, 10));        // 40
   // Передача указателя на функцию
   // в качестве параметра
   printf("%d\n", func(10, 5, pf2)); // 15
   printf("%d\n", func(10, 5, add)); // 15
```

```
    printf("%d\n", func(10, 5, sub)); //  5
    return 0;
}
void print_ok(void) {
    puts("OK");
}
int add(int x, int y) {
    return x + y;
}
int sub(int x, int y) {
    return x - y;
}
int func(int x, int y, int (*pfunc)(int, int)) {
    return pfunc(x, y);
}
```

10.10. Передача в функцию и возврат данных произвольного типа

Если в функцию нужно передать данные произвольного типа или вернуть данные произвольного типа, то нужно использовать указатели с типом `void *`. Пример объявления указателя:

```
void *p = NULL;
```

Присвоить адрес указателю можно, как обычно:

```
int x = 10;
p = &x;
```

Для того чтобы получить значение, нужно привести указатель к определенному типу, а затем выполнить разыменование указателя:

```
printf("%d\n", *((int *)p));        // 10
```

Нетипизированный указатель можно присвоить типизированному. В этом случае в языке C приведение типа выполняется автоматически:

```
int *px = p;                        // int *px = (int *)p;
printf("%d\n", *px);                // 10
```

Мы можем сохранить в указателе данные любого типа, но, чтобы получить данные, нужно знать, к какому типу выполнить приведение. Например, сохраним вещественное число, а затем получим его и выведем на консоль:

```
double y = 20.2123;
p = &y;
printf("%.1f\n", *((double *)p)); // 20.2
```

В качестве примера напишем функцию, позволяющую вывести на консоль данные типов `int`, `long`, `long long` и `double` (листинг 10.19).

Листинг 10.19. Передача в функцию данных произвольного типа

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

enum TypeParam {
   TINT, TLONG, TLONGLONG, TDOUBLE
};

void println(const char *s, void *p, enum TypeParam t);

int main(void) {
   int x = INT_MAX;
   println("x = ", &x, TINT);      // x = 2147483647
   long y = LONG_MAX;
   println("y = ", &y, TLONG);     // y = 2147483647
   long long z = LLONG_MAX;
   println("", &z, TLONGLONG);     // 9223372036854775807
   double k = 20.2123;
   println("k = ", &k, TDOUBLE);   // k = 20.21
   return 0;
}
void println(const char *s, void *p, enum TypeParam t) {
   if (!p || !s) return;
   switch (t) {
      case TINT:
         printf("%s%d\n", s, *((int *)p));
         break;
      case TLONG:
         printf("%s%ld\n", s, *((long *)p));
         break;
      case TLONGLONG:
         printf("%s%I64d\n", s, *((long long *)p));
         break;
      case TDOUBLE:
         printf("%s%.2f\n", s, *((double *)p));
         break;
   }
}
```

10.11. Рекурсия

Рекурсия — это возможность функции вызывать саму себя. При каждом вызове функции создается новый набор локальных переменных. Рекурсию удобно использовать для перебора объекта, имеющего заранее неизвестную структуру, или выполнения неопределенного количества операций. Типичным применением рекурсии является вычисление факториала числа (листинг 10.20).

Листинг 10.20. Вычисление факториала

```
#include <stdio.h>

unsigned long long factorial(unsigned long n);

int main(void) {
   for (unsigned i = 3; i < 11; ++i) {
      printf("%d! = %I64u\n", i, factorial(i));
   }
   return 0;
}
unsigned long long factorial(unsigned long n) {
   if (n <= 1) return 1;
   else return n * factorial(n - 1);
}
```

Результат выполнения:

```
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

10.12. Встраиваемые функции

Передача управления в функцию сопряжена с потерей скорости выполнения программы, т. к. вызов функции, передача в нее параметров и возврат значений требуют дополнительных затрат времени. Если размер функции небольшой, то имеет смысл объявить такую функцию *встраиваемой*. В этом случае при компиляции содержимое функции будет вставлено в место вызова функции. Тем не менее следует помнить, что при этом происходит увеличение размера исполняемого файла. Поэтому если большая функция вызывается много раз, то лучше ее оставить обычной функцией, в то время как функция, состоящая из одной инструкции, является первым кандидатом для встраивания.

Для объявления функции встраиваемой следует перед функцией добавить ключевое слово `__inline` или `inline`. Для того чтобы функция не была видна в других файлах, лучше дополнительно добавить ключевое слово `static`. Следует учитывать, что ключевое слово `__inline` является лишь рекомендацией компилятору и может быть проигнорировано. В качестве примера объявим функцию `sum()` встраиваемой (листинг 10.21).

Листинг 10.21. Встраиваемые функции

```
#include <stdio.h>

static __inline int sum(int x, int y) {
   return x + y;
}
int main(void) {
   printf("%d\n", sum(10, 20));
   return 0;
}
```

Создать встраиваемую функцию можно также с помощью директивы `#define`. Значение, указанное в этой директиве, подставляется в место вызова функции до компиляции. Название, указанное в директиве `#define`, принято называть *макроопределением* или *макросом*. Директива имеет следующий формат:

```
#define <Название функции>(<Параметры>) <Инструкция>
```

Пример использования директивы `#define` приведен в листинге 10.22.

Листинг 10.22. Использование директивы `#define`

```
#include <stdio.h>

#define SUM(x, y) ((x) + (y))

int main(void) {
   printf("%d\n", SUM(10, 20));
   return 0;
}
```

Обратите внимание на то, что в конце инструкции точка с запятой отсутствует. Концом инструкции является конец строки. Если точку с запятой указать, то значение вместе с ней будет вставлено в выражение. Например, если определить макрос так:

```
#define SUM(x, y) (x + y);
```

то после подстановки значения инструкция

```
printf("%d\n", SUM(10, 20));
```

будет выглядеть следующим образом:

```
printf("%d\n", (10 + 20););
```

Точка с запятой после закрывающей круглой скобки является концом инструкции, поэтому этот код вызовет ошибку при компиляции. Однако в следующем примере ошибки не будет, но результат окажется совершенно другим. Инструкция

```
int z = SUM(10, 20) + 40;
```

после подстановки будет выглядеть следующим образом:

```
int z = (10 + 20); + 40;
```

Подобная ситуация приводит к ошибкам, которые бывает трудно найти.

Обратите также внимание на то, что выражение внутри тела макроса расположено внутри круглых скобок. Если скобки не указать, то это может привести к недоразумениям, т. к. никакого вычисления выражения не производится. Все выражение внутри тела макроса после подстановки формальных параметров целиком вставляется в место вызова макроса. Например, если определить макрос следующим образом:

```
#define SUM(x, y) x + y
```

то после подстановки значения инструкция

```
int z = SUM(10, 20) * 40;
```

будет выглядеть так:

```
int z = 10 + 20 * 40;
```

Приоритет оператора умножения выше приоритета оператора сложения, поэтому число `20` будет умножено на `40`, а затем к результату прибавлено число `10`. Таким образом, результат будет `810`, а не `1200`.

При указании длинного выражения внутри тела функции следует учитывать, что определение макроса должно быть расположено на одной строке. Если нужно разместить выражение на нескольких строках, то в конце строки необходимо добавить обратную косую черту. После косой черты не должно быть никаких символов, в том числе и комментариев. Пример:

```
#define SUM(x, y) \
((x) + (y))
```

ГЛАВА 11

Обработка ошибок

Если вы когда-нибудь учились водить автомобиль, то наверняка вспомните, что при первой посадке на водительское сиденье все внимание было приковано к трем деталям: рулю, педалям и рычагу переключения передач. Происходящее вне автомобиля уходило на второй план, т. к. вначале нужно было стронуться с места. По мере практики навыки вождения улучшались, и эти три детали постепенно уходили на задний план. Как ни странно, но руль и рычаг переключения передач всегда оказывались там, куда вы не смотря протягивали руки, а ноги сами находили педали. Теперь все внимание стало занимать происходящее на дороге. Иными словами, вы стали опытным водителем.

В программировании все абсолютно так же. Начинающие программисты больше обращают внимание на первые попавшиеся на глаза операторы, функции и другие элементы языка, а сам алгоритм уходит на задний план. Если программа скомпилировалась без ошибок, то это уже большое счастье, хотя это еще не означает, что программа работает правильно. По мере практики мышление программиста меняется, он начинает обращать внимание на мелочи, на форматирование программы, использует более эффективные алгоритмы и в результате всего этого допускает меньше ошибок. Подводя итоги, можно сказать, что начинающий программист просто пишет программу, а опытный программист пытается найти оптимальный алгоритм и предусмотреть поведение программы в различных ситуациях. Однако от ошибок никто не застрахован, поэтому очень важно знать, как быстро найти ошибку.

11.1. Типы ошибок

Существуют три типа ошибок в программе.

- *Синтаксические* — это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычки и т. д., т. е. ошибки в синтаксисе языка. Как правило, компилятор предупредит о наличии ошибки, а программа не будет выполняться совсем. Пример синтаксической ошибки:

```
printf("%s", "Нет завершающей кавычки!);
```

При компиляции будет выведено следующее сообщение об ошибке:

```
..\src\Test64c.c:4:17: error: missing terminating " character
```

Первое число после названия компилируемого файла и двоеточия обозначает номер строки, в которой обнаружена ошибка, а второе число — номер символа внутри строки.

Если после попытки скомпилировать программу в окне **Console** редактора Eclipse (открыть окно можно из меню **Window**, выбрав пункт **Show View | Console**) выполнить щелчок мышью на строке с описанием ошибки, то инструкция с ошибкой станет активной. Кроме того, строка с синтаксической ошибкой подсвечивается редактором желтой волнистой линией еще до компиляции программы. Так что обращайте внимание на различные подчеркивания кода. При ошибках подчеркивание будет красного цвета, а при предупреждениях — желтого. При предупреждениях слева от инструкции дополнительно отображается значок в виде желтого квадрата с вопросительным знаком. При наведении указателя мыши на подчеркнутый фрагмент или на значок, расположенный слева, отобразится текст описания проблемы.

Все ошибки и предупреждения (например, о том, что переменная не используется) можно посмотреть в окне **Problems**. Если окно не отображается, то в меню **Window** выбираем пункт **Show View | Problems**.

- *Логические* — это ошибки в логике работы программы, которые можно выявить только по результатам работы программы. Как правило, компилятор не предупреждает о наличии ошибки, а программа будет выполняться, т. к. не содержит синтаксических ошибок. Основные ошибки в языке C связаны с указателями и массивами, поскольку компилятор не производит никакой проверки корректности указателя и не контролирует выход за границы массива. Весь контроль полностью лежит на плечах программиста. Логические ошибки достаточно трудно выявить, и их поиск часто заканчивается бессонными ночами.
- *Ошибки времени выполнения* — это ошибки, которые возникают во время работы программы. В одних случаях ошибки времени выполнения являются следствием логических ошибок, а в других случаях причиной служат внешние события, например нехватка оперативной памяти, отсутствие прав для записи в файл и др.

11.2. Предупреждающие сообщения при компиляции

Для того чтобы избежать множества проблем, на этапе отладки следует включить вывод предупреждающих сообщений при компиляции программы. Как минимум нужно указать флаг `-Wall`. Если дополнительно указать флаг `-Wconversion`, то компилятор предупредит о несоответствии типов. Пример указания флагов:

```
gcc -Wall -Wconversion -O3 -o test.exe test.c
```

В редакторе Eclipse установка флагов выполняется в свойствах проекта. Слева из списка выбираем пункт **C/C++ Build | Settings**, а затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Warnings** (рис. 11.1). В списке **Configuration** находим режим компиляции, устанавливаем нужные флажки и сохраняем настройки проекта, нажимая кнопку **Apply and Close**.

При использовании компилятора Visual C можно в начало программы вставить следующую инструкцию:

```
#pragma warning( push, 4)
```

Для отключения вывода каких-либо предупреждающих сообщений в Visual C используется следующая инструкция:

```
#pragma warning( disable : <Код предупреждения> )
```

Пример:

```
#pragma warning( disable : 4996 )
```

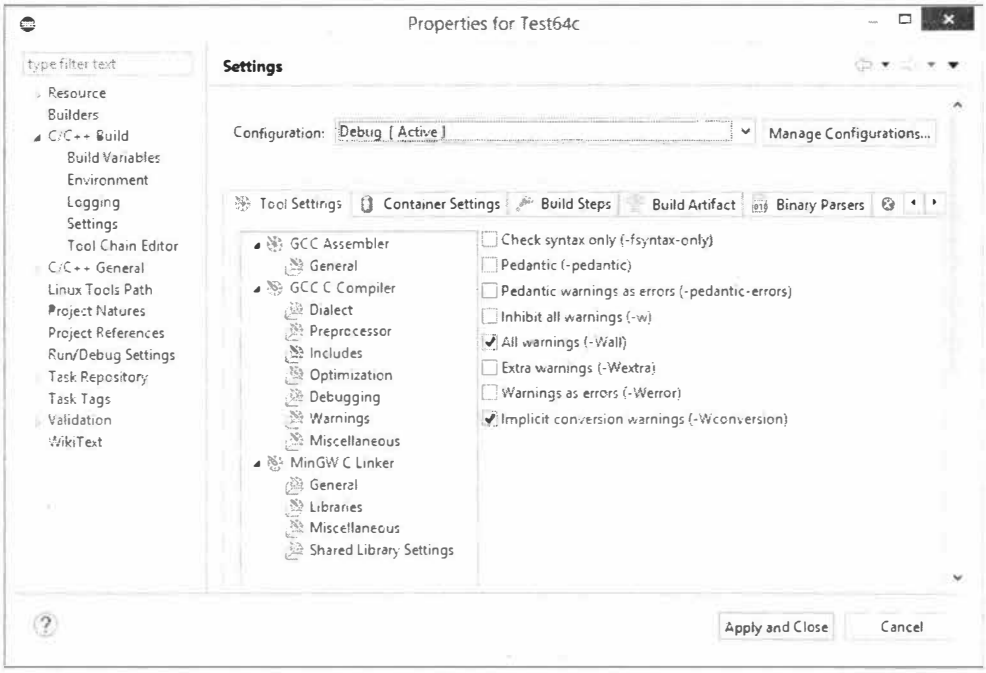


Рис. 11.1. Управление выводом предупреждающих сообщений

11.3. Переменная *errno* и вывод сообщения об ошибке

Отдельные функции из стандартной библиотеки при ошибке возвращают некоторое значение и присваивают переменной `errno` номер ошибки. Например, функция `strtol()`, предназначенная для преобразования С-строки в число типа `long`, в случае

выхода за диапазон значений для типа, присваивает переменной errno значение макроса ERANGE:

```
// #include <stdlib.h>
// #include <errno.h>
printf("%ld\n", strtol("2147483649", NULL, 0)); // 2147483647
printf("%d\n", errno);  // 34
printf("%d\n", ERANGE); // 34
```

Макрос ERANGE и остальные макросы с номерами ошибок определены в заголовочном файле errno.h:

```
#define ERANGE 34
```

В Visual C при использовании некоторых функций, например strtol(), переменная errno не становится равной 0 при успешном преобразовании, что может привести к ошибкам при следующем преобразовании. Перед использованием функций лучше сбросить флаг ошибки явным образом, присвоив переменной errno значение 0:

```
printf("%d\n", errno);                          // 0
printf("%ld\n", strtol("2147483649", NULL, 0)); // 2147483647
printf("%d\n", errno);                          // 34
printf("%ld\n", strtol("50", NULL, 0));         // 50
printf("%d\n", errno);  // В Visual C 2013 значение 34, а не 0 !!!
errno = 0;              // Сбрасываем флаг ошибки
printf("%ld\n", strtol("50", NULL, 0));         // 50
printf("%d\n", errno);                          // 0
```

Получить текстовое описание ошибки по ее коду позволяют функции strerror() и _wcserror(). Прототипы функций:

```
#include <string.h>
char *strerror(int errNum);
wchar_t *_wcserror(int errNum);
```

Пример вывода сообщения:

```
printf("%s\n", strerror(ERANGE));    // Result too large
printf("%s\n", strerror(22));        // Invalid argument
wprintf(L"%s\n", _wcserror(EILSEQ)); // Illegal byte sequence
```

Вместо функций strerror() и _wcserror() лучше воспользоваться функциями strerror_s() и _wcserror_s(). Прототипы функций:

```
#include <string.h>
errno_t strerror_s(char *buf, size_t sizeInBytes, int errNum);
errno_t _wcserror_s(wchar_t *buf, size_t sizeInWords, int errNum);
```

В первом параметре функции принимают указатель на строку, а во втором параметре — максимальный размер этой строки. Номер ошибки задается в третьем параметре. Пример использования функции strerror_s() приведен в листинге 11.1.

Листинг 11.1. Использование функции strerror_s()

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

int main(void) {
   char err[256] = {0};
   strerror_s(err, 255, EILSEQ);
   printf("%s\n", err);            // Illegal byte sequence
   strerror_s(err, 255, ERANGE);
   printf("%s\n", err);            // Result too large
   return 0;
}
```

Получить описание последней ошибки позволяют функции _strerror() и __wcserror(). Прототипы функций:

```
#include <string.h>
char *_strerror(const char *errMsg);
wchar_t *__wcserror(const wchar_t *errMsg);
```

Если в параметре указано значение NULL, то строка будет содержать только описание последней ошибки. Если передать строку, то она будет добавлена перед описанием ошибки. В строку вставляется символ перевода строки. Пример:

```
// #include <stdlib.h>
printf("%ld\n", strtol("2147483649", NULL, 0)); // 2147483647
printf("%s", _strerror(NULL));        // Result too large
printf("%s", _strerror("Error"));     // Error: Result too large
wprintf(L"%s", __wcserror(NULL));     // Result too large
wprintf(L"%s", __wcserror(L"Error")); // Error: Result too large
```

Вместо функций _strerror() и __wcserror() лучше воспользоваться функциями _strerror_s() и __wcserror_s(). Прототипы функций:

```
#include <string.h>
errno_t _strerror_s(char *buf, size_t sizeInBytes, const char *errMsg);
errno_t __wcserror_s(wchar_t *buf, size_t sizeInWords,
                     const wchar_t *errMsg);
```

В первом параметре функции принимают указатель на строку, а во втором параметре — максимальный размер этой строки. Если в третьем параметре указано значение NULL, то строка будет содержать только описание последней ошибки. Если передать строку, то она будет добавлена перед описанием ошибки. Пример:

```
// #include <stdlib.h>
printf("%ld\n", strtol("2147483649", NULL, 0)); // 2147483647
char err[256] = {0};
```

```
_strerror_s(err, 255, NULL);
printf("%s\n", err);                    // Result too large
_strerror_s(err, 255, "Error");
printf("%s\n", err);                    // Error: Result too large
```

Функции `perror()` **и** `_wperror()` **выводят текст сообщения об ошибке с кодом** `errno` **в стандартный поток вывода сообщений об ошибках** `stderr`. **Перед этим сообщением добавляется текст, переданный в параметре** `errMsg`, **и символ двоеточия. Прототипы функций:**

```
#include <stdio.h> /* или #include <stdlib.h> */
void perror(const char *errMsg);
#include <stdio.h> /* или #include <stdlib.h> или #include <wchar.h> */
void _wperror(const wchar_t *errMsg);
```

Пример использования функций `perror()` **и** `_wperror()`:

```
errno = ERANGE;
perror("Error");    // Error: Result too large
_wperror(L"Error"); // Error: Result too large
```

11.4. Способы поиска ошибок в программе

Наибольшее количество времени программист затрачивает на поиск логических ошибок. В этом случае программа компилируется без ошибок, но результат выполнения программы не соответствует ожидаемому результату. Ситуация еще более осложняется, когда неверный результат проявляется лишь периодически, а не постоянно. Инсценировать такую же ситуацию, чтобы получить этот же неверный результат, бывает крайне сложно и занимает очень много времени. В этом разделе мы рассмотрим лишь "дедовские" (но по-прежнему актуальные) способы поиска ошибок, а также дадим несколько советов по оформлению кода, что будет способствовать быстрому поиску ошибок.

Первое, на что следует обратить внимание, — это форматирование кода. Начинающие программисты обычно не обращают на форматирование никакого внимания, считая этот процесс лишним. А на самом деле зря! Компилятору абсолютно все равно, разместите вы все инструкции на одной строке или выполните форматирование кода. Однако при поиске ошибок форматирование кода позволит найти ошибку гораздо быстрее.

Перед всеми инструкциями внутри блока должно быть расположено одинаковое количество пробелов. Обычно используется три или четыре пробела. От применения символов табуляции лучше отказаться. Если все же их используете, то не следует в одном файле совмещать и пробелы, и табуляцию. Для вложенных блоков количество пробелов умножают на уровень вложенности. Если для блока первого уровня вложенности использовалось три пробела, то для блока второго уровня вложенности поставьте шесть пробелов, для третьего уровня — девять пробелов и т. д. Пример форматирования вложенных блоков приведен в листинге 11.2.

Листинг 11.2. Пример форматирования вложенных блоков

```
#define ARR_ROWS 2
#define ARR_COLS 4
int arr[ARR_ROWS][ARR_COLS] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8}
};
int i, j;
for (i = 0; i < ARR_ROWS; ++i) {
   for (j = 0; j < ARR_COLS; ++j) {
      printf("%3d", arr[i][j]);
   }
   printf("\n");
}
```

Открывающая фигурная скобка может быть расположена как на одной строке с оператором, так и на следующей строке. Какой способ использовать, зависит от предпочтений программиста или от требований по оформлению кода, принятых внутри фирмы. Пример размещения открывающей фигурной скобки на отдельной строке:

```
for (i = 0; i < ARR_ROWS; ++i)
{
   for (j = 0; j < ARR_COLS; ++j)
   {
      printf("%3d", arr[i][j]);
   }
   printf("\n");
}
```

Одна строка кода не должна содержать более 80 символов. Если количество символов больше, то следует выполнить переход на новую строку. При этом продолжение смещается относительно основной инструкции на величину отступа или выравнивается по какому-либо элементу. Иначе приходится пользоваться горизонтальной полосой прокрутки, а это очень неудобно при поиске ошибок.

Если программа слишком большая, то следует задуматься о разделении программы на отдельные функции, которые выполняют логически законченные действия. Помните, что отлаживать отдельную функцию гораздо легче, чем "спагетти"-код. Причем, прежде чем вставить функцию в основную программу, ее следует протестировать в отдельном проекте, передавая функции различные значения и проверяя результат ее выполнения.

Обратите внимание на то, что форматирование кода должно выполняться при написании кода, а не во время поиска ошибок. Этим вы сократите время поиска ошибки и, скорее всего, заметите ошибку еще на этапе написания. Если все же ошибка возникла, то вначале следует инсценировать ситуацию, при которой ошибка проявляется. После этого можно начать поиск ошибки.

Причиной периодических ошибок чаще всего являются внешние данные. Например, если числа получаются от пользователя, а затем производится деление чисел, то вполне возможна ситуация, при которой пользователь введет число 0. Деление на ноль приведет к ошибке. Следовательно, все данные, которые поступают от пользователей, должны проверяться на соответствие допустимым значениям. Если данные не соответствуют, то нужно вывести сообщение об ошибке, а затем повторно запросить новое число или прервать выполнение всей программы. Нужно также обработать возможность того, что пользователь может ввести вовсе не число, а строку.

Также следует учитывать, что при использовании функции `scanf()` возможно переполнение, если пользователь введет значение вне допустимого диапазона для указанного типа. Функция при этом никак не проинформирует вас об ошибке. Переменная `errno` будет иметь значение 0. В результате положительное число может стать отрицательным, что приведет уже к логической ошибке.

Пример получения числа от пользователя с проверкой корректности данных показан в листинге 11.3.

Листинг 11.3. Пример получения числа от пользователя с проверкой данных

```
#include <stdio.h>
#include <errno.h>
#include <locale.h>
#include <stdlib.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   long x = 100, y = 0;
   char buf[256] = {0}, *p = NULL;
   for ( ; ; ) {                        // Бесконечный цикл
      printf("y = ");                   // Выводим подсказку
      fflush(stdout);                   // Сбрасываем буфер вывода
      fflush(stdin);                    // Очищаем буфер ввода
      p = fgets(buf, 256, stdin);       // Получаем значение
      if (!p) {                         // Проверяем успешность операции
         puts("Ошибка при вводе");
         exit(1);
      }
      p = NULL;
      errno = 0;                        // Сбрасываем флаг ошибки
      y = strtol(buf, &p, 0);           // Преобразуем строку в число
      if (!p || p == buf) {             // Проверяем успешность операции
         puts("Вы ввели не число!");
         continue;
      }
      if (errno != 0) {                 // Проверяем диапазон
         puts("Выход за диапазон значений");
```

```
        continue;
      }
      if (y == 0) {          // Проверяем допустимость значения
        puts("Значение 0 недопустимо");
        continue;
      }
      break;                 // Если ошибок нет, то выходим из цикла
    }
    printf("y = %ld\n", y);  // Выводим результат
    printf("x / y = %.2f\n", (double)x / y);
    return 0;
}
```

Функцию `printf()` удобно использовать для вывода промежуточных значений. В этом случае значения переменных вначале выводятся в самом начале программы (внутри функции `main()`), и производится проверка соответствия значений. Если значения соответствуют, то инструкция с функцией `printf()` перемещается на следующую строку программы, и опять производится проверка и т. д. Если значения не совпали, то ошибка возникает в инструкции, расположенной перед инструкцией с функцией `printf()`. Если это пользовательская функция, то проверку значений производят внутри функции, каждый раз перемещая инструкцию с выводом значений. На одном из этих многочисленных этапов ошибка обычно обнаруживается. В больших программах можно логически догадаться о примерном расположении инструкции с ошибкой и начать поиск ошибки оттуда, а не с самого начала программы.

Инструкции для вывода промежуточных значений можно расставить уже при написании программы, не дожидаясь возникновения ошибки. В этом случае в начале программы определяется макрос, а внутри программы производится проверка наличия макроса. Если макрос определен, то выводятся значения. В противном случае инструкция просто игнорируется. Создать макрос позволяет директива `#define`:

```
#define MY_DEBUG 1
```

Проверить существование макроса позволяет следующая конструкция:

```
#ifdef MY_DEBUG
   // Здесь размещаем инструкции вывода значений
   printf("Макрос определен\n");
#endif
```

Проверить существование макроса и его значение можно так:

```
#if defined(MY_DEBUG) && MY_DEBUG == 1
   printf("Макрос имеет значение 1\n");
#endif
```

Пример использования макросов приведен в листинге 11.4.

Листинг 11.4. Использование макросов

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

#define MY_DEBUG 1

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   int x = 10;
#ifdef MY_DEBUG
   // Эта инструкция будет выполнена только при отладке
   printf("x = %d\n", x);
#endif
#if defined(MY_DEBUG) && MY_DEBUG == 1
   // Эта инструкция будет выполнена только если макрос
   // MY_DEBUG равен 1
   printf("x = %d\n", x);
#endif
   printf("Этот текст выводится в любом случае. x = %d\n", x);
   return 0;
}
```

Результат в окне консоли:

```
x = 10
x = 10
Этот текст выводится в любом случае. x = 10
```

Если закомментировать инструкцию:

```
#define MY_DEBUG 1
```

то вывод будет таким:

```
Этот текст выводится в любом случае. x = 10
```

11.5. Отладка программы в редакторе Eclipse

Сделать поиск ошибок более эффективным позволяет отладчик gdb.exe, который можно запустить в редакторе Eclipse. С его помощью можно выполнять программу по шагам, при этом контролируя значения переменных на каждом шаге. Отладчик позволяет также проверить, соответствует ли порядок выполнения инструкций разработанному ранее алгоритму. В качестве примера мы будем отлаживать программу из листинга 11.5, которая содержит две наиболее часто встречающиеся ошибки: выход за пределы массива и нумерация элементов массива с `1`, а не с `0`.

Листинг 11.5. Отладка программы

```
#include <stdio.h>

#define ARR_ROWS 2
#define ARR_COLS 4

void echo(int x);

int main(void) {
   int arr[ARR_ROWS][ARR_COLS] = {                    // 9
      {1, 2, 3, 4},
      {5, 6, 7, 8}
   };
   int i = 0, j = 0;                                  // 13
   for (i = 0; i <= ARR_ROWS; ++i) {                  // 14
      for (j = 1; j < ARR_COLS; ++j) {                // 15
         echo(arr[i][j]);                             // 16
      }
      printf("\n");                                   // 18
   }
   return 0;
}
void echo(int x) {                                    // 22
   printf("  ");                                      // 23
   printf("%d", x);                                   // 24
}
```

Прежде чем начать отладку, необходимо пометить строки внутри программы с помощью точек останова. Для добавления точки останова делаем строку активной, а затем в меню **Run** выбираем пункт **Toggle Breakpoint** или нажимаем комбинацию клавиш <Shift>+<Ctrl>+<B>, — слева от строки появится кружок, обозначающий точку останова. Повторное действие убирает точку останова. Добавить точку останова можно еще быстрее. Для этого достаточно выполнить двойной щелчок левой кнопкой мыши слева от строки. Повторный двойной щелчок позволяет удалить точку останова. Для того чтобы изменить свойства точки, временно отключить или удалить ее, следует щелкнуть на ней правой кнопкой мыши и в открывшемся контекстном меню выбрать пункт **Toggle Breakpoint**, **Disable Breakpoint** или **Enable Breakpoint**. Давайте установим в нашей программе две точки останова: напротив строк 13 и 16.

Когда точки останова расставлены, можно начать отладку. Для этого в меню **Run** выбираем пункт **Debug** или нажимаем клавишу <F11>. После запуска отладки интерфейс редактора Eclipse может поменяться кардинальным образом (рис. 11.2). После запуска отладки выполнение программы прерывается, и отладчик ожидает дальнейших действий программиста. Инструкция, которая будет выполняться на следующем шаге, помечается стрелкой слева от строки. Для того чтобы прервать

отладку, в меню **Run** выбираем пункт **Terminate** или нажимаем комбинацию клавиш <Ctrl>+<F2>.

> ***ОБРАТИТЕ ВНИМАНИЕ***
>
> **Если на компьютере установлена антивирусная программа или включен брандмауэр Windows, то они могут заблокировать запуск отладчика. Необходимо разрешить запуск отладчика при запросе действия или добавить программу в список исключения брандмауэра, если отладчик не запустился.**

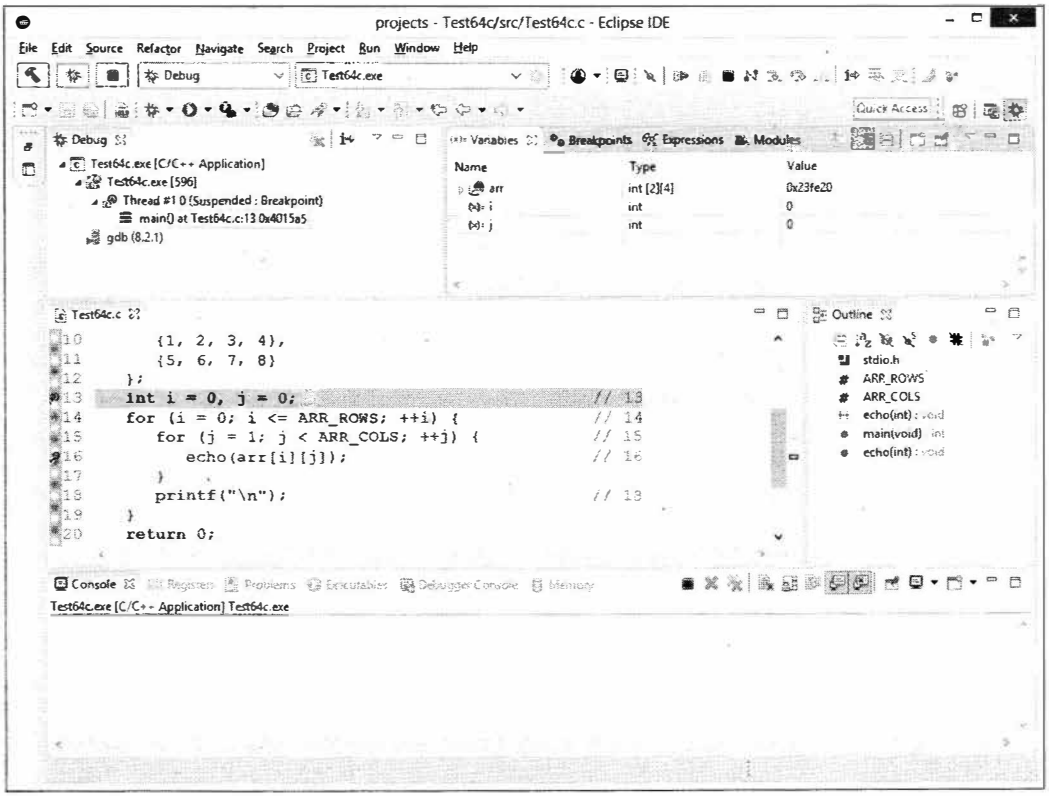


Рис. 11.2. Окно редактора Eclipse после запуска отладки

После запуска отладчика выполнение программы прервется на строке 9. Для того чтобы выполнить инструкции до первой точки останова (строка 13), нажимаем клавишу <F8>. В режиме прерывания можно посмотреть значения различных переменных. Для этого достаточно навести указатель мыши на название переменной. Попробуйте навести указатель на название массива `arr` — в нижней части всплывающего окна отобразятся значения всех элементов массива. Если навести указатель мыши на переменные `i` и `j`, то никакого значения мы не получим, т. к. поток управления еще не дошел до этих переменных. Их значение мы увидим только после следующего шага.

Посмотреть значения сразу всех переменных можно в окне **Variables** (рис. 11.3). Если окно не отображается, то отобразить его можно, выбрав в меню **Window**

пункт **Show View | Variables**. При отладке можно контролировать значения отдельных переменных, а не всех сразу. Для этого следует добавить название переменной в окне **Expressions** (рис. 11.4), щелкнув левой кнопкой мыши на надписи **Add new expression**. Для того чтобы отобразить это окно, в меню **Window** выбираем пункт **Show View | Expressions**. Можно также выделить название переменной и из контекстного меню выбрать пункт **Add Watch Expression**. Давайте добавим в это окне переменные `i` и `j`. Сейчас значения этих переменных не определены, поэтому в поле **Value** отображаются произвольные значения.

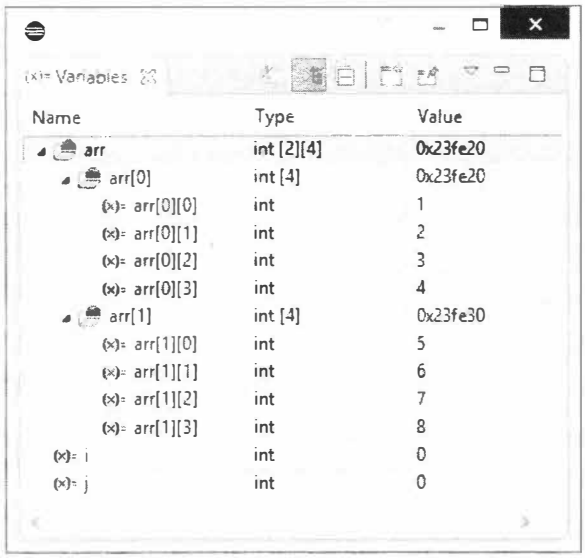


Рис. 11.3. Окно **Variables**

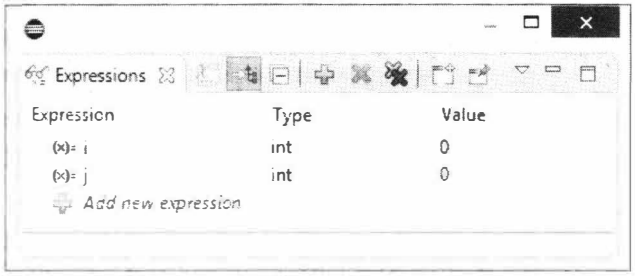


Рис. 11.4. Окно **Expressions**

Для пошагового выполнения программы предназначены следующие пункты в меню **Run** или соответствующие кнопки на панели инструментов.

- **Step Into** (клавиша <F5>) — выполняет одну инструкцию. Давайте сделаем один шаг и посмотрим на значения переменных `i` и `j` в окне **Expressions** — обе переменные стали видны и получили значение `0`. Делаем еще два шага и останавливаемся на строке 16, которая у нас помечена точкой останова. Если мы сделаем еще один шаг, то попадем внутрь функции `echo()`. Обратите внимание на значе-

ния переменных `i` и `j` в окне **Expressions** — теперь они не определены, т. к. мы находимся в области видимости функции `echo()`.

- **Step Return** (клавиша <F7>) — позволяет сразу выйти из функции. Если мы не находимся внутри функции, то команда будет недоступна. Давайте нажмем клавишу <F7> и выйдем из функции `echo()`. В результате текущей станет строка 15.
- **Step Over** (клавиша <F6>) — выполняет одну инструкцию. Если в этой инструкции производится вызов функции, то функция выполняется, и отладчик переходит в режим ожидания после выхода из функции. Давайте сделаем два шага и опять окажемся на строке 15, не входя в функцию `echo()`.
- **Resume** (клавиша <F8>) — выполняет инструкции до следующей точки останова или до конца программы при отсутствии точек. Нажимая клавишу <F8>, мы каждый раз будем останавливаться на строке 16 и сможем наблюдать за текущими значениями переменных `i` и `j` в окне **Expressions**, а также за выводом результатов работы программы в окне **Console**. На одном из шагов можно обнаружить, что значение переменной `i` стало равно `2`, а строк в массиве всего две и нумеруются они с нуля. Смотрим внимательно на инструкции и обнаруживаем неправильное условие окончания цикла в строке 14 (`i <= ARR_ROWS`). Исправляем ошибку, и выражение примет вид `i < ARR_ROWS`. Опять запускаем отладку, чтобы убедиться в правильности внесенных изменений.
- **Run to Line** (комбинация клавиш <Ctrl>+<R>) — выполняет инструкции до текущей строки при условии, что между инструкциями нет точки останова. Если точка есть, то выполнение будет идти до этой точки. Если мы попробуем перейти сразу к строке 18, то это не получится, т. к. существует точка останова на строке 16. А нам бы хотелось увидеть вывод одной строки массива сразу. Для этого можно временно отключить точку останова, щелкнув на ней правой кнопкой мыши и выбрав в контекстном меню пункт **Disable Breakpoint** (в отключенном состоянии отображается лишь контур кружка без заливки, для повторной активации выбираем пункт **Enable Breakpoint**). Если у нас точек будет много, то каждый раз выбирать пункт из контекстного меню станет утомительно. Можно временно отключить сразу все точки останова, выбрав в меню **Run** пункт **Skip All Breakpoints** (в отключенном состоянии точки отображаются перечеркнутыми, повторное действие сделает все точки останова снова активными). После этого действия можно сразу перейти к строке 18, предварительно сделав ее текущей и нажав комбинацию клавиш <Ctrl>+<R>. После этого действия мы заметим, что вместо четырех значений выводятся только три. Для того чтобы понять причину, опять выполняем программу по шагам и наблюдаем за значениями переменных в окне **Expressions**. В один прекрасный момент понимаем, что индексы нумеруются с `0`, а не с `1`, и исправляем ошибку в строке 15 (`j = 0` вместо `j = 1`).
- **Terminate** (комбинация клавиш <Ctrl>+<F2>) — останавливает отладку.

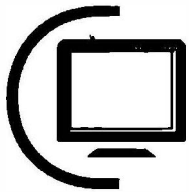
Управлять отдельными точками останова или всеми сразу можно в окне **Breakpoints** (рис. 11.5). Для того чтобы отобразить это окно, в меню **Window** следует выбрать пункт **Show View | Breakpoints**.

Рис. 11.5. Окно **Breakpoints**

Применение отладки — самый эффективный способ нахождения ошибок, не требующий вставки никаких инструкций вывода промежуточных значений в текст программы. На каждом шаге мы и так можем наблюдать за значениями всех или только избранных переменных. После отладки не надо удалять или временно отключать какие-либо инструкции, даже точки останова убирать не нужно. При обычном выполнении точки останова ни на что не влияют. Пользуйтесь отладкой при возникновении любой ошибки, и вы очень быстро ее найдете и исправите.

На заметку

Для того чтобы из режима **Debug** вернуться в обычный режим, в меню **Window** выбираем пункт **Perspective | Open Perspective | C/C++** или нажимаем соответствующую кнопку на панели инструментов.



ГЛАВА 12

Чтение и запись файлов

В *разд. 2.8* и *2.9* мы уже рассматривали ввод/вывод данных в окно консоли. В этой главе мы научимся читать данные из файла и записывать их в файл. Все функции, используемые в этой главе, объявлены в заголовочном файле stdio.h. Не забудьте в начало программы добавить следующую инструкцию:

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

12.1. Открытие и закрытие файла

Для открытия файла предназначены функции `fopen()` и `_wfopen()`. Прототипы функций:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
#include <stdio.h> /* или #include <wchar.h> */
FILE *_wfopen(const wchar_t *filename, const wchar_t *mode);
```

В первом параметре функции принимают путь к файлу, а во втором — режим открытия файла. Функции возвращают указатель на структуру `FILE` или нулевой указатель в случае ошибки. Пример использования функции `fopen()`:

```
// #include <locale.h>
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = NULL;
// Открываем файл на запись в текстовом режиме
fp = fopen("C:\\book\\file1.txt", "w");
if (fp) {                    // Проверяем успешность
   fputs("строка", fp);      // Записываем строку
   fclose(fp);               // Закрываем файл
}
else puts("Не удалось открыть файл");
```

Пример использования функции `_wfopen()`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = NULL;
fp = _wfopen(L"C:\\book\\file2.txt", L"w");
```

```
if (fp) {
   fputws(L"строка", fp);
   fclose(fp);
}
else _putws(L"Не удалось открыть файл");
```

Количество одновременно открытых файлов ограничено значением макроса FOPEN_MAX. Определение макроса выглядит так:

```
#define FOPEN_MAX 20
```

В качестве результата функции возвращают указатель на структуру FILE. Этот указатель следует передавать функциям, которые предназначены для работы с уже открытым файлом. Если открыть файл не удалось, функции возвращают нулевой указатель и присваивают глобальной переменной errno код ошибки. Структура FILE объявлена следующим образом:

```
struct _iobuf {
  char *_ptr;
  int _cnt;
  char *_base;
  int _flag;
  int _file;
  int _charbuf;
  int _bufsiz;
  char *_tmpfname;
};
typedef struct _iobuf FILE;
```

Вместо функций fopen() и _wfopen() лучше использовать функции fopen_s() и _wfopen_s(). Прототипы функций:

```
#include <stdio.h>
errno_t fopen_s(FILE **File, const char *filename, const char *mode);
#include <stdio.h> /* или #include <wchar.h> */
errno_t _wfopen_s(FILE **File, const wchar_t *filename,
                  const wchar_t *mode);
```

В первом параметре функция принимает адрес указателя на файловую структуру. Остальные параметры аналогичны параметрам функции fopen(). В качестве результата функции возвращают 0 при отсутствии ошибки или код ошибки.

Пример использования функции fopen_s():

```
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = NULL;
errno_t err;
err = fopen_s(&fp, "C:\\book\\file1.txt", "w");
if (!err) {
   fputs("строка", fp);
   fclose(fp);
}
else puts("Не удалось открыть файл");
```

Пример использования функции `_wfopen_s()`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = NULL;
errno_t err;
err = _wfopen_s(&fp, L"C:\\book\\file2.txt", L"w");
if (!err) {
   fputws(L"строка", fp);
   fclose(fp);
}
else puts("Не удалось открыть файл");
```

Открыть файл в режиме совместного доступа позволяют функции `_fsopen()` **и** `_wfsopen()`. **Прототипы функций:**

```
#include <stdio.h>
FILE *_fsopen(const char *filename, const char *mode, int shareFlag);
#include <stdio.h> /* или #include <wchar.h> */
FILE *_wfsopen(const wchar_t *filename, const wchar_t *mode,
               int shareFlag);
```

В первом параметре функции принимают путь к файлу, а во втором — режим открытия файла. В параметре `shareFlag` **можно указать один из следующих макросов:**

```
#include <share.h>
```

- `_SH_DENYRW` — **запрещает доступ на чтение и запись. Определение макроса:**
  `#define _SH_DENYRW 0x10`
- `_SH_DENYWR` — **запрещает доступ на запись. Определение макроса:**
  `#define _SH_DENYWR 0x20`
- `_SH_DENYRD` — **запрещает доступ на чтение. Определение макроса:**
  `#define _SH_DENYRD 0x30`
- `_SH_DENYNO` — **разрешает доступ на чтение и запись. Определение макроса:**
  `#define _SH_DENYNO 0x40`

Функции возвращают указатель на структуру `FILE` **или нулевой указатель в случае ошибки. Пример использования функции** `_fsopen()`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = NULL;
fp = _fsopen("C:\\book\\file1.txt", "w", _SH_DENYWR);
if (fp) {
   fputs("строка", fp);
   fclose(fp);
}
else puts("Не удалось открыть файл");
```

Закрыть файл позволяет функция `fclose()`. **Прототип функции:**

```
#include <stdio.h>
int fclose(FILE *File);
```

В единственном параметре передается указатель на открытый ранее файл. В качестве значения функция возвращает:

- 0 — если файл успешно закрыт;
- EOF — если при закрытии файла произошла ошибка. Макрос EOF определен следующим образом:

```
#define EOF (-1)
```

Закрыть все открытые ранее файлы позволяет функция _fcloseall(). Стандартные потоки stdin, stdout и stderr функция не закрывает. Прототип функции:

```
#include <stdio.h>
int _fcloseall(void);
```

12.2. Указание пути к файлу

В параметре filename в функциях fopen() и _wfopen() указывается путь к файлу. Длина пути ограничена значением макроса FILENAME_MAX. Определение макроса:

```
#include <stdio.h>
#define FILENAME_MAX 260
```

Существует также макрос _MAX_PATH, содержащий максимальную длину пути:

```
#include <stdlib.h>
#define _MAX_PATH 260
```

Путь может быть *абсолютным* или *относительным*. При указании абсолютного пути следует учитывать, что слэш является специальным символом. По этой причине его необходимо удваивать:

```
"C:\\temp\\new\\file.txt"
```

Если вместо двух слэшей использовать только один, то в пути будут присутствовать сразу три специальных символа: \t, \n и \f. После преобразования специальных символов путь будет выглядеть следующим образом:

```
C:<Табуляция>emp<Перевод строки>ew<Перевод формата>ile.txt
```

Вместо абсолютного пути к файлу можно указать относительный путь. В этом случае путь определяется с учетом местоположения *текущего рабочего каталога*. Обратите внимание на то, что текущий рабочий каталог не всегда совпадает с каталогом, в котором находится исполняемый файл. Если файл запускается из командной строки, то текущим рабочим каталогом будет каталог, из которого запускается файл. Получить строковое представление текущего рабочего каталога позволяют функции _getcwd() и _wgetcwd(). Прототипы функций:

```
#include <direct.h>
char *_getcwd(char *buf, int sizeInBytes);
#include <wchar.h> /* или #include <direct.h> */
wchar_t *_wgetcwd(wchar_t *buf, int sizeInWords);
```

В первом параметре функции принимают указатель на буфер, а во втором параметре — максимальный размер буфера.

Пример использования функции `_getcwd()`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char buf[260] = {0}, *p = NULL;
p = _getcwd(buf, 260);
if (p) {
   printf("%s\n", buf); // C:\cpp\projects\Test64c
}
```

Если в качестве параметров заданы нулевой указатель и нулевое значение, то строка создается динамически с помощью функции `malloc()`. В этом случае функции возвращают указатель на эту строку или нулевой указатель в случае ошибки. После окончания работы со строкой следует освободить память с помощью функции `free()`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char *pbuf = NULL;
pbuf = _getcwd(NULL, 0);
if (pbuf) {
   printf("%s\n", pbuf); // C:\cpp\projects\Test64c
   free(pbuf);
   pbuf = NULL;
}
```

Возможны следующие варианты указания относительного пути в Windows:

- если открываемый файл находится в текущем рабочем каталоге, то можно указать только название файла или перед названием файла добавить одну точку и прямой или обратный слэш:

```
"file.txt"
"./file.txt"
".\\file.txt"
```

- если открываемый файл расположен во вложенном каталоге, то перед названием файла перечисляются названия вложенных каталогов через прямой или обратный слэш:

```
"folder1\\file.txt"
"folder1/file.txt"
"folder1\\folder2\\file.txt"
"folder1/folder2/file.txt"
```

- если каталог с файлом расположен выше уровнем, то перед названием файла указываются две точки и прямой или обратный слэш:

```
"..\\file.txt"
"../file.txt"
"..\\..\\file.txt"
"../../file.txt"
```

- если в начале пути расположен слэш, то путь отсчитывается от корня текущего диска. В этом случае местоположение текущего рабочего каталога не имеет значения. Пример:

```
"\\book\\test\\file.txt"
"/book/test/file.txt"
```

12.3. Режимы открытия файла

Параметр `mode` в функциях `fopen()` и `_wfopen()` может принимать следующие значения внутри строки.

- `r` — только чтение. После открытия файла курсор устанавливается на начало файла. Если файл не существует, то функции возвращают нулевой указатель.
- `r+` — чтение и запись. После открытия файла курсор устанавливается на начало файла. Если файл не существует, то функции возвращают нулевой указатель.
- `w` — запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла курсор устанавливается на начало файла.
- `w+` — чтение и запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла курсор устанавливается на начало файла.
- `a` — запись. Если файл не существует, то он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется.
- `a+` — чтение и запись. Если файл не существует, то он будет создан. Чтение производится с начала файла. Запись осуществляется в конец файла. Содержимое файла не удаляется.

После режима может следовать модификатор:

- `b` — файл будет открыт в бинарном режиме;
- `t` — файл будет открыт в текстовом режиме (значение по умолчанию в Windows). В этом режиме при чтении символ `\r` будет удален, а при записи наоборот, добавлен.

> ***Примечание***
>
> **В Visual C имеются дополнительные модификаторы: `c`, `n`, `N`, `S`, `R`, `T` и `D`. За информацией обращайтесь к документации.**

После режима и модификатора через запятую можно указать параметр `ccs`. Параметр принимает значения `UNICODE`, `UTF-8` или `UTF-16LE`. Если указано значение `UTF-8` или `UTF-16LE`, то файл создается в соответствующей кодировке и в начало файла вставляются служебные байты, сокращенно называемые *BOM* (Byte Order Mark — метка порядка байтов). Если открываемый файл содержит BOM, то значение, указанное в параметре `ccs`, игнорируется. При отсутствии параметра `ccs` автоматиче-

ское определение кодировки файла по BOM не производится. Пример записи в файл в кодировке UTF-8 (содержимое файла включает метку порядка байтов):

```
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = NULL;
fp = _wfopen(L"C:\\book\\file.txt", L"w,ccs=UTF-8");
if (fp) {
   fputws(L"строка", fp);
   fclose(fp);
}
else _putws(L"Не удалось открыть файл");
```

Откроем файл на запись в текстовом режиме и запишем в него две строки, разделенные символом `\n` (листинг 12.1). Затем откроем тот же файл на чтение в бинарном режиме и считаем первую строку. Для того чтобы продемонстрировать факт добавления символа `\r` при записи в текстовом режиме, выведем коды символов, которые расположены в конце считанной строки.

Листинг 12.1. Работа с файлами в текстовом и бинарном режимах

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char buf[200] = {0};
   FILE *fp = NULL;
   // Открываем файл на запись в текстовом режиме
   fp = fopen("C:\\book\\file.txt", "w");
   if (!fp) {                          // Проверяем успешность
      perror("Error");                 // Выводим сообщение об ошибке
      exit(1);                         // Выходим при ошибке
   }
   fputs("String1\nString2", fp);      // Записываем строку
   fflush(fp);                         // Сбрасываем буфер
   fclose(fp);                         // Закрываем файл

   // Открываем файл на чтение в бинарном режиме
   fp = fopen("C:\\book\\file.txt", "rb");
   if (!fp) {                          // Проверяем успешность
      perror("Error");                 // Выводим сообщение об ошибке
      exit(1);                         // Выходим при ошибке
   }
   fgets(buf, 30, fp);                 // Читаем одну строку
   printf("%d %d\n", (int)buf[7],
                     (int)buf[8]); // 13 10 == \r \n
```

```
    fclose(fp);                    // Закрываем файл
    return 0;
}
```

Попробуйте во втором случае изменить режим с `rb` на `r`. В результате вывод будет `10 0`, а не `13 10`. Это доказывает, что при чтении в текстовом режиме символ `^Z` удаляется.

Функция `perror()` выводит текст сообщения об ошибке. Перед этим сообщением добавляется текст, переданный в параметре `errMsg`, и символ двоеточия. Например, если в первом случае файл доступен только для чтения, то получим следующее сообщение в окне консоли: `Error: Permission denied`. Если во втором случае файл не существует, то сообщение будет другим: `Error: No such file or directory`. Прототип функции:

```
#include <stdio.h> /* или #include <stdlib.h> */
void perror(const char *errMsg);
```

Вместо функции `perror()` можно воспользоваться функцией `strerror()`, которой в параметре передается значение переменной `errno`. Вывести сообщение об ошибке в поток `stderr` позволяет функция `fprintf()`:

```
fprintf(stderr, "%s", strerror(errno));
```

Функция `fputs()` записывает С-строку в файл. Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только в момент закрытия файла. Сбросить буфер в файл явным образом можно с помощью функции `fflush()`. Прототип функции:

```
#include <stdio.h>
int fflush(FILE *File);
```

Прочитать одну строку из файла позволяет функция `fgets()`. Считывание производится, пока не встретится символ перевода строки или из файла не будет прочитано указанное количество символов минус один символ (используется для вставки нулевого символа).

12.4. Запись в файл

Для записи в файл предназначены следующие функции.

- `fputc()` и `putc()` — записывают символ в файл. Функции возвращают код записанного символа или значение макроса `EOF` в случае ошибки. Прототипы функций:

```
#include <stdio.h>
int fputc(int ch, FILE *File);
int putc(int ch, FILE *File);
#define EOF (-1)
```

Пример:

```
fputc('T', fp);
putc('S', fp);
```

- `fputwc()` и `putwc()` — записывают широкий символ в файл. Функции возвращают код записанного символа или значение макроса `WEOF` в случае ошибки. Прототипы функций:

```
#include <stdio.h> /* или #include <wchar.h> */
wint_t fputwc(wchar_t ch, FILE *File);
#define putwc(_c,_stm) fputwc(_c,_stm)
#define WEOF (wint_t)(0xFFFF)
```

Пример:

```
// setlocale(LC_ALL, "Russian_Russia.1251");
fputwc(L'T', fp);
putwc(L'S', fp);
```

- `fputs()` — записывает С-строку в файл. Функция возвращает неотрицательное число, если ошибок нет, и значение макроса `EOF` при наличии ошибки. Прототип функции:

```
#include <stdio.h>
int fputs(const char *str, FILE *File);
```

Пример:

```
fputs("String1\nString2", fp);
```

- `fputws()` — записывает L-строку в файл. Функция возвращает неотрицательное число, если ошибок нет, и значение макроса `WEOF` при наличии ошибки. Прототип функции:

```
#include <stdio.h> /* или #include <wchar.h> */
int fputws(const wchar_t *str, FILE *file);
```

Пример:

```
// setlocale(LC_ALL, "Russian_Russia.1251");
fputws(L"String1\nString2", fp);
```

- `fprintf()` — производит форматированный вывод в файл. При наличии ошибки функция возвращает значение макроса `EOF`. Прототип функции:

```
#include <stdio.h>
int fprintf(FILE *File, const char *format, ...);
```

В параметре `format` указывается строка специального формата. Внутри этой строки можно указать обычные символы и спецификаторы формата, начинающиеся с символа `%`. Эти спецификаторы мы рассматривали при изучении функции `printf()`. Пример использования функции `fprintf()`:

```
int a = 10, b = 20;
fprintf(fp, "%d %d", a, b);
```

Можно также воспользоваться функцией `fprintf_s()`:

```
#include <stdio.h>
int fprintf_s(FILE *File, const char *format, ...);
```

Пример:

```
int a = 10, b = 20;
fprintf_s(fp, "%d %d", a, b);
```

- `fwprintf()` — производит форматированный вывод в файл. При наличии ошибки функция возвращает значение макроса `WEOF`. Прототип функции:

```
#include <stdio.h> /* или #include <wchar.h> */
int fwprintf(FILE *File, const wchar_t *format, ...);
```

Пример:

```
// setlocale(LC_ALL, "Russian_Russia.1251");
int a = 10, b = 20;
fwprintf(fp, L"%d %d", a, b);
```

Можно также воспользоваться функцией `fwprintf_s()`:

```
#include <stdio.h> /* или #include <wchar.h> */
int fwprintf_s(FILE *File, const wchar_t *format, ...);
```

Пример:

```
// setlocale(LC_ALL, "Russian_Russia.1251");
fwprintf_s(fp, L"%d %d", 10, 20);
```

12.5. Чтение из файла

Перечислим функции, предназначенные для чтения файла.

- `fgetc()` и `getc()` — считывают один символ из файла. В качестве значений функции возвращают код прочитанного символа. Если в файле нет больше символов или произошла ошибка, то функции возвращают значение макроса `EOF`. Прототипы функций:

```
#include <stdio.h>
int fgetc(FILE *File);
int getc(FILE *File);
```

Пример использования функций:

```
int ch1 = fgetc(fp);
printf("%d\n", ch1);
printf("%c\n", (char)ch1);
int ch2 = getc(fp);
printf("%d\n", ch2);
printf("%c\n", (char)ch2);
```

- `fgetwc()` и `getwc()` — считывают широкий символ из файла. В качестве значений функции возвращают код прочитанного символа. Если в файле нет больше сим-

волов или произошла ошибка, то функции возвращают значение макроса WEOF. Прототипы функций:

```
#include <stdio.h> /* или #include <wchar.h> */
wint_t fgetwc(FILE *File);
#define getwc(_stm) fgetwc(_stm)
```

Пример использования функций:

```
// setlocale(LC_ALL, "Russian_Russia.1251");
wint_t ch1 = fgetwc(fp);
wprintf(L"%u\n", ch1);
wprintf(L"%c\n", (wchar_t)ch1);
wint_t ch2 = getwc(fp);
wprintf(L"%u\n", ch2);
wprintf(L"%c\n", (wchar_t)ch2);
```

- `fgets()` — читает одну строку из файла. Прототип функции:

```
#include <stdio.h>
char *fgets(char *buf, int maxCount, FILE *File);
```

Считывание производится до первого символа перевода строки или до конца файла, или пока не будет прочитано `maxCount-1` символов. Если не достигнут предел буфера, то возвращаемая строка включает символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ добавлен не будет. Если произошла ошибка или достигнут конец файла, то функция возвращает нулевой указатель. Пример считывания одной строки из файла:

```
char buf[200] = {0}, *p = NULL;
p = fgets(buf, 200, fp);
if (p) printf("%s", buf);
```

- `fgetws()` — читает одну строку из файла. Прототип функции:

```
#include <stdio.h> /* или #include <wchar.h> */
wchar_t *fgetws(wchar_t *buf, int sizeInWords, FILE *File);
```

Функция `fgetws()` аналогична функции `fgets()`, только работает с L-строками. Пример использования функции:

```
// setlocale(LC_ALL, "Russian_Russia.1251");
wchar_t buf[200] = {0}, *p = NULL;
p = fgetws(buf, 200, fp);
if (p) wprintf(L"%s", buf);
```

- `fscanf()` — позволяет считать данные из файла и автоматически преобразовать в конкретный тип, например, в целое число. Прототип функции:

```
#include <stdio.h>
int fscanf(FILE *File, const char *format, ...);
```

В параметре `format` указывается строка специального формата, внутри которой задаются спецификаторы, аналогичные применяемым в функции `scanf()`. В по-

следующих параметрах передаются адреса переменных, в которых будут сохранены значения. Функция возвращает количество преобразованных данных. Пример считывания из файла целого числа:

```
int x = 0, count = 0;
count = fscanf(fp, "%d", &x); // Символ & обязателен!!!
printf("%d\n", count);
printf("%d\n", x);
```

- `fwscanf()` — аналогична функции `fscanf()`, только работает с L-строками. Прототип функции:

```
#include <stdio.h> /* или #include <wchar.h> */
int fwscanf(FILE *File, const wchar_t *format, ...);
```

Пример:

```
int x = 0, count = 0;
count = fwscanf(fp, L"%d", &x); // Символ & обязателен!!!
printf("%d\n", count);
printf("%d\n", x);
```

В качестве примера запишем строку в файл, а затем произведем посимвольное считывание (листинг 12.2).

Листинг 12.2. Посимвольное считывание из файла

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <string.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   FILE *fp = NULL;
   int ch = 0;
   fp = fopen("C:\\book\\file.txt", "w+"); // Чтение и запись
   if (!fp) {
      perror("Error");
      exit(1);
   }
   fputs("Строка1\nСтрока2", fp); // Записываем строку
   fflush(fp);                   // Сбрасываем буфер в файл
   rewind(fp);                   // Перемещаем курсор в начало файла
   ch = fgetc(fp);               // Считываем один символ
   while (!feof(fp)) {
      if (ferror(fp)) {          // Проверяем наличие ошибок
         printf("Error: %s\n", strerror(ferror(fp)));
         exit(1);
      }
```

```
        printf("%c", (char)ch);
        ch = fgetc(fp);              // Считываем один символ
    }
    fclose(fp);
    return 0;
}
```

В этом примере были использованы три новые функции:

- `rewind()` — устанавливает курсор на начало файла. Прототип функции:

```
#include <stdio.h>
void rewind(FILE *File);
```

- `feof()` — возвращает ненулевое значение, если достигнут конец файла. В противном случае возвращается значение `0`. Прототип функции:

```
#include <stdio.h>
int feof(FILE *File);
```

- `ferror()` — возвращает ненулевое значение, если при работе с файлом возникла ошибка. В противном случае возвращается значение `0`. Функцию следует вызывать сразу после выполнения операции. Прототип функции:

```
#include <stdio.h>
int ferror(FILE *File);
```

Удалить все ошибки, связанные с потоком, позволяет функция `clearerr()`. Прототип функции:

```
#include <stdio.h>
void clearerr(FILE *File);
```

Вместо функции `clearerr()` можно использовать функцию `clearerr_s()`. Если операция успешно произведена, то функция возвращает значение `0`. Прототип функции:

```
#include <stdio.h>
errno_t clearerr_s(FILE *File);
```

12.6. Чтение и запись двоичных файлов

Для чтения и записи бинарных файлов можно воспользоваться функциями, работающими с отдельными символами, например, `fgetc()` и `fputc()`, а также следующими функциями.

- `fwrite()` — записывает объекты произвольного типа в файл. Обратите внимание: файл должен быть открыт в бинарном режиме. Функция возвращает количество записанных объектов. Прототип функции:

```
#include <stdio.h>
size_t fwrite(const void *buf, size_t size, size_t count, FILE *File);
```

В первом параметре функция принимает указатель на объект. Во втором параметре указывается размер объекта в байтах, а в третьем параметре — количество записываемых объектов. Размер объекта следует определять с помощью оператора `sizeof`.

- `fread()` — считывает из файла объект произвольного типа. Обратите внимание: файл должен быть открыт в бинарном режиме. Прототип функции:

```
#include <stdio.h>
size_t fread(void *buf, size_t elementSize, size_t count, FILE *File);
```

В первом параметре функция принимает указатель на объект. Во втором параметре указывается размер объекта в байтах, а в третьем параметре — количество считываемых объектов. Размер объекта следует определять с помощью оператора `sizeof`. Функция возвращает количество считанных объектов.

Запишем структуру в файл, открытый на запись в бинарном режиме, а затем считаем структуру из файла и выведем значения (листинг 12.3).

Листинг 12.3. Сохранение структуры в файл

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

struct Point {
   int x, y;
} point1, point2;

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   FILE *fp = NULL;
   // Запись в файл
   fp = fopen("C:\\book\\file.txt", "wb"); // Бинарный режим!
   if (!fp) {
      perror("Error");
      exit(1);
   }
   point1.x = 10;
   point1.y = 20;
   size_t n = fwrite(&point1, sizeof(struct Point), 1, fp);
   printf("%d\n", (int)n); // 1
   fflush(fp);
   fclose(fp);
   // Чтение из файла
   fp = fopen("C:\\book\\file.txt", "rb"); // Бинарный режим!
   if (!fp) {
      perror("Error");
      exit(1);
   }
```

```
    n = fread(&point2, sizeof(struct Point), 1, fp);
    printf("%d\n", point2.x); // 10
    printf("%d\n", point2.y); // 20
    printf("%d\n", (int)n);   // 1
    printf("%d\n", (int) sizeof(struct Point));
    fclose(fp);
    return 0;
}
```

Обратите внимание: если структура содержит указатели, то в файл будут записаны адреса, на которые указывает указатель, а не данные. После восстановления структуры из файла эти адреса могут быть уже не актуальными. Поэтому при использовании указателей нужно предусмотреть дополнительные действия для записи этих данных в файл и при чтении их из файла.

12.7. Файлы произвольного доступа

Файл можно читать не только с самого начала, но и с произвольной позиции. Как правило, в этом случае файл открывают в бинарном режиме. Записывать в файл также можно с произвольной позиции. Для работы с файлами произвольного доступа предназначены следующие функции.

- `fgetpos()` — записывает позицию курсора относительно начала файла в переменную, адрес которой передается во втором параметре. В качестве результата функция возвращает значение `0`, если ошибок не возникло, и ненулевое значение в противном случае. Прототип функции:

  ```
  #include <stdio.h>
  int fgetpos(FILE *File, fpos_t *pos);
  ```

  Тип `fpos_t` объявлен как целочисленный тип:

  ```
  typedef __int64 fpos_t;
  #define __int64 long long
  ```

  Пример использования функции:

  ```
  fpos_t pos = 0;
  fgetpos(fp, &pos);
  printf("%I64d\n", pos);
  ```

- `ftell()` — возвращает позицию курсора относительно начала файла. При ошибке возвращается значение `-1L`. Прототип функции:

  ```
  #include <stdio.h>
  long ftell(FILE *File);
  ```

  Пример использования функции:

  ```
  long pos = 0;
  pos = ftell(fp);
  printf("%ld\n", pos);
  ```

Вместо функции `ftell()` можно воспользоваться функцией `_ftelli64()`. Прототип функции:

```
#include <stdio.h>
__int64 _ftelli64(FILE *File);
```

Пример:

```
__int64 pos = 0;
pos = _ftelli64(fp);
printf("%I64d\n", pos);
```

- `rewind()` — устанавливает курсор на начало файла. Функция дополнительно сбрасывает флаг ошибки для потока и флаг конца файла. Прототип функции:

```
#include <stdio.h>
void rewind(FILE *File);
```

- `fsetpos()` — перемещает курсор относительно начала файла. Адрес переменной, в которой содержится значение, передается во втором параметре. В качестве результата функция возвращает значение `0`, если ошибок не возникло, и ненулевое значение в противном случае. Прототип функции:

```
#include <stdio.h>
int fsetpos(FILE *File, const fpos_t *pos);
```

Пример:

```
fpos_t pos = 5;
fsetpos(fp, &pos);
```

- `fseek()` — устанавливает курсор в позицию, имеющую смещение `offset` относительно позиции `origin`. В качестве результата функция возвращает значение `0`, если ошибок не возникло, и ненулевое значение в противном случае. Прототип функции:

```
#include <stdio.h>
int fseek(FILE *File, long offset, int origin);
```

В параметре `origin` могут быть указаны следующие макросы:

- `SEEK_SET` — начало файла;
- `SEEK_CUR` — текущая позиция курсора;
- `SEEK_END` — конец файла.

Определения макросов выглядят следующим образом:

```
#include <stdio.h>
#define SEEK_SET    0
#define SEEK_CUR    1
#define SEEK_END    2
```

Вместо функции `fseek()` можно воспользоваться функцией `_fseeki64()`. Прототип функции:

```
#include <stdio.h>
int _fseeki64(FILE *File, __int64 offset, int origin);
```

В качестве примера использования файлов произвольного доступа запишем массив в файл, а затем считаем из файла значения первого и предпоследнего элементов массива (листинг 12.4).

Листинг 12.4. Файлы произвольного доступа

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   FILE *fp = NULL;
   int arr[] = { 1, 2, 3, 4, 5 }, x = 0;
   long pos = 0;
   fp = fopen("C:\\book\\file.txt", "w+b"); // Бинарный режим!
   if (!fp) {
      perror("Error");
      exit(1);
   }
   size_t size = sizeof(int);
   fwrite(arr, size, 5, fp); // Записываем массив
   fflush(fp);
   fseek(fp, 0, SEEK_SET);    // Перемещаем курсор в начало файла
   pos = ftell(fp);
   printf("%ld\n", pos);      // 0
   fread(&x, size, 1, fp);
   printf("%d\n", x);         // 1
   // Перемещаем курсор на предпоследний элемент
   fseek(fp, -2 * (long)size, SEEK_END);
   pos = ftell(fp);
   printf("%ld\n", pos);      // 12
   fread(&x, size, 1, fp);
   printf("%d\n", x);         // 4
   fclose(fp);
   return 0;
}
```

12.8. Создание временных файлов

Для создания временного файла предназначена функция `tmpfile()`. Она возвращает файловый указатель на поток, открытый в режиме `w+b`, или нулевой указатель в случае ошибки. После закрытия временный файл автоматически удаляется. Обра-

тите внимание на то, что в некоторых операционных системах для создания временного файла могут потребоваться права администратора. Прототип функции:

```
#include <stdio.h>
FILE *tmpfile(void);
```

В Visual C при использовании функции `tmpfile()` выводится предупреждающее сообщение `warning C4996`. Для того чтобы избежать этого сообщения, следует применять функцию `tmpfile_s()`. При успешном выполнении операции функция возвращает значение `0`, а в противном случае — код ошибки. В проектах `Test32c` и `Test64c` функция не работает. Прототип функции:

```
#include <stdio.h>
errno_t tmpfile_s(FILE **File);
```

Вместо создания временного файла можно сгенерировать уникальное название с помощью функции `tmpnam()`, а затем создать файл обычным образом. Прототип функции:

```
#include <stdio.h>
char *tmpnam(char *buf);
```

Если произошла ошибка, то функция возвращает нулевой указатель. Сгенерированное название записывается в символьный массив `buf`. Если в параметре передан нулевой указатель, то сгенерированное название размещается во внутреннем статическом буфере, а ссылка на него возвращается в качестве значения. Эти данные будут перезаписаны при следующем вызове функции `tmpnam()`. Пример использования функции:

```
char *p = 0;
p = tmpnam(NULL);
if (p) printf("%s", p); // Например, \s2mg.
```

Вместо функции `tmpnam()` можно воспользоваться функцией `_wtmpnam()`, которая возвращает название временного файла в виде L-строки. Прототип функции:

```
#include <stdio.h> /* или #include <wchar.h> */
wchar_t *_wtmpnam(wchar_t *buf);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
wchar_t *p = 0;
p = _wtmpnam(NULL);
if (p) wprintf(L"%s", p); // Например, \s74g.
```

Вместо функций `tmpnam()` и `_wtmpnam()` можно использовать функции `tmpnam_s()` и `_wtmpnam_s()`. Прототипы функций:

```
#include <stdio.h>
errno_t tmpnam_s(char *buf, rsize_t size);
#include <stdio.h> /* или #include <wchar.h> */
errno_t _wtmpnam_s(wchar_t *buf, size_t size);
```

При успешном выполнении операции функции возвращают значение 0, а в противном случае — код ошибки. Сгенерированное название сохраняется в символьном массиве buf, максимальный размер которого указывается в параметре size. Пример:

```
char buf[100] = {0};
if (tmpnam_s(buf, 100) == 0) {
   printf("%s\n", buf);        // Например, \sarg.
}
```

Кроме перечисленных функций, для генерации названия временного файла можно воспользоваться функциями _tempnam() и _wtempnam(). Прототипы функций:

```
#include <stdio.h>
char *_tempnam(const char *dirName, const char *filePrefix);
#include <stdio.h> /* или #include <wchar.h> */
wchar_t *_wtempnam(const wchar_t *dirName, const wchar_t *filePrefix);
```

В первом параметре указывается путь к каталогу, который будет использоваться, если переменная окружения TMP не определена или содержит некорректный путь. Если переменная TMP корректна, то будет использоваться ее значение. Во втором параметре указывается префикс. Функции динамически выделяют под название необходимый объем памяти с помощью функции malloc() и возвращают указатель на строку. Обратите внимание: ответственность за освобождение динамической памяти лежит на плечах программиста, поэтому после использования памяти следует обязательно вызвать функцию free() и передать ей указатель. В случае ошибки функции возвращают нулевой указатель. Пример использования функции _tempnam():

```
char *p = _tempnam("C:\\book", "tmp_");
if (p) {
   printf("%s\n", p);
   // Например, C:\Users\Unicross\AppData\Local\Temp\tmp_2
   free(p); // Освобождаем память!!!
}
```

12.9. Перенаправление ввода/вывода

Потоки ввода/вывода на консоль аналогичны потокам ввода/вывода в файл. При запуске программы автоматически открываются три потока:

- stdin — стандартный ввод;
- stdout — стандартный вывод;
- stderr — стандартный вывод сообщений об ошибках.

Все эти идентификаторы являются файловыми указателями, связанными по умолчанию с окном консоли. Следовательно, их можно использовать вместо обычных файловых указателей в функциях, предназначенных для работы с файлами. Например, вывести строку в окно консоли можно так:

```
fputs("String1\nString2", stdout);
fflush(stdout);
```

Пример вывода сообщения об ошибке в поток `stderr`:

```
fputs("Сообщение об ошибке", stderr);
fflush(stderr);
```

Ввод символа выполняется следующим образом:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("Введите символ: ");
fflush(stdout);
int ch = fgetc(stdin);
printf("%c\n", (char)ch);
```

Стандартные потоки можно перенаправить таким образом, чтобы данные записывались в файл или считывались из файла. Для этого существуют три способа.

- Перенаправить из командной строки. В этом случае ничего в программе менять не нужно. Для того чтобы выполнить перенаправление вывода в файл, следует в командной строке выполнить одну из команд:

  ```
  test.exe > file.txt
  test.exe >> file.txt
  ```

  Первая строка записывает результат выполнения программы test.exe в файл file.txt. Если файл не существует, то он будет создан, а если существует, то он будет перезаписан. Вторая строка производит дозапись в конец файла.

  Предыдущий пример сохранял в файл только данные из потока `stdout`. Для того чтобы сохранить данные из потока `stderr`, нужно дополнительно указать дескриптор потока. Поток `stdout` имеет дескриптор `1`, а поток `stderr` — дескриптор `2`:

  ```
  test.exe 1> путь_к_файлу_для_out 2> путь_к_файлу_для_err
  test.exe 1>> путь_к_файлу_для_out 2>> путь_к_файлу_для_err
  ```

  Пример:

  ```
  test.exe 1> out.txt 2> err.txt
  ```

  Для того чтобы ввести в программу данные из файла, следует выполнить команду:

  ```
  test.exe < file.txt
  ```

- Перенаправить с помощью функции `freopen()` или `_wfreopen()`. Прототипы функций:

  ```
  #include <stdio.h>
  FILE *freopen(const char *filename, const char *mode,
                FILE *oldFile);
  #include <stdio.h> /* или #include <wchar.h> */
  FILE *_wfreopen(const wchar_t *filename, const wchar_t *mode,
                  FILE *oldFile);
  ```

Первые два параметра аналогичны параметрам функции `fopen()`. Пример перенаправления стандартного вывода в файл:

```
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = NULL;
fp = freopen("C:\\book\\file.txt", "w", stdout);
if (!fp) exit(1);
printf("%s\n", "Пишем в файл!");
```

Вместо функций `freopen()` и `_wfreopen()` можно использовать функции `freopen_s()` и `_wfreopen_s()`. Прототипы функций:

```
#include <stdio.h>
errno_t freopen_s(FILE **File, const char *filename,
                  const char *mode, FILE *oldFile);
#include <stdio.h> /* или #include <wchar.h> */
errno_t _wfreopen_s(FILE **File, const wchar_t *filename,
                    const wchar_t *mode, FILE *oldFile);
```

Первые три параметра аналогичны параметрам функции `fopen_s()`. Пример использования функции `freopen_s()`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = NULL;
if (freopen_s(&fp, "C:\\book\\file.txt", "w", stdout) != 0) {
   exit(1);
}
printf("%s\n", "Пишем в файл!");
```

- Перенаправить с помощью функции `_dup2()`. В качестве параметров функция принимает дескрипторы файлов, которые можно получить с помощью функции `_fileno()`. Прототипы функций:

```
#include <io.h>
int _dup2(int fileHandleSrc, int fileHandleDst);
#include <stdio.h>
int _fileno(FILE *File);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
FILE *fp = fopen("C:\\book\\file.txt", "w");
if (!fp) exit(1);
if (_dup2(_fileno(fp), _fileno(stdout)) == -1) {
   perror("Не удалось перенаправить\n");
   exit(1);
}
printf("%s\n", "Пишем в файл!");
```

Если после перенаправления нужно восстановить вывод на консоль, то можно воспользоваться кодом из листинга 13.5.

12.10. Работа с буфером ввода и вывода

Для ускорения работы производится буферизация записываемых данных. Информация из буфера записывается в файл полностью только при заполнении буфера или в момент закрытия файла. Сбросить буфер в файл явным образом можно с помощью функции `fflush()`. Прототип функции:

```
#include <stdio.h>
int fflush(FILE *File);
```

Функция возвращает значение `0`, если буфер успешно сброшен, или значение макроса `EOF` в случае ошибки. Пример:

```
printf("%s\n", "строка");
fflush(stdout);
fprintf(stderr, "%s\n", "сообщение об ошибке");
fflush(stderr);
```

Функцию `fflush()` можно также использовать в некоторых версиях Windows для очистки буфера ввода `stdin`. Очистить буфер ввода особенно важно, например, при использовании функции `scanf()`, когда предыдущий ввод закончился ошибкой:

```
int x = 0;
fflush(stdin);
int status = scanf("%d", &x);
if (status != 1) {
   puts("Вы ввели не число");
   // Здесь при ошибке нужно обязательно очистить буфер
   fflush(stdin);
}
else printf("x = %d\n", x);
```

Проблема в том, что это нестандартный способ сброса буфера, который не работает даже в некоторых версиях Windows, а в других операционных системах вообще не работает. Тем не менее это единственный способ, позволяющий сбросить буфер при отсутствии ошибок. Способы, рассматриваемые далее, при пустом буфере будут ожидать ввода пользователя.

Более универсальным способом является использование функции `getchar()`, но только при наличии ошибки. Как уже говорилось, при пустом буфере функция будет ждать ввода пользователя, который нигде не сохранится! В этом примере мы дополнительно проверим статус потока и при ошибке сбросим флаг ошибки:

```
int x = 0;
fflush(stdin); // Здесь использовать функцию getchar() нельзя!
int status = scanf("%d", &x);
if (status != 1) {
   puts("Вы ввели не число");
   // Сбрасываем флаг ошибки и очищаем буфер
   if (feof(stdin) || ferror(stdin)) clearerr(stdin);
   int ch = 0;
```

```
   while ((ch = getchar()) != '\n' && ch != EOF);
}
else printf("x = %d\n", x);
```

Зачем очищать буфер ввода перед функцией `scanf()`? Если пользователь, например, введет `3str`, то число `3` будет считано функцией `scanf()`, а вот фрагмент `str` останется в буфере и приведет к считыванию без ожидания ввода пользователя, что в свою очередь приведет к ошибке при вводе числа.

Следующий способ очистки буфера заключается в применении функции `scanf()`:

```
int x = 0;
fflush(stdin);
int status = scanf("%d", &x);
if (status != 1) {
   puts("Вы ввели не число");
   // Сбрасываем флаг ошибки и очищаем буфер
   if (feof(stdin) || ferror(stdin)) clearerr(stdin);
   scanf("%*[^\n]"); // Считываем все символы до \n
   scanf("%*c");     // Считываем символ \n
}
else printf("x = %d\n", x);
```

На самом деле советую вообще отказаться от использования функции `scanf()`. Во-первых, она оставляет часть данных в буфере, что может стать причиной ошибки при следующей операции ввода. Во-вторых, при вводе строки, если в составе спецификатора не указать ширину, возможно переполнение буфера. В-третьих, при вводе чисел функция никак не проверяет диапазон значений, и при переполнении вы никак об этом не узнаете. Лучше получать от пользователя строку с данными, а затем уже выполнять преобразования. Пример можно посмотреть в листинге 11.3.

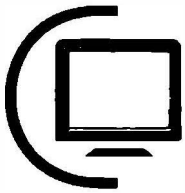
Функция `_flushall()` позволяет сбросить данные сразу всех потоков ввода и вывода. Она возвращает количество открытых потоков. Прототип функции:

```
#include <stdio.h>
int _flushall(void);
```

Пример:

```
printf("%d\n", _flushall());
```

ГЛАВА 13



Низкоуровневые потоки ввода и вывода

В этой главе мы рассмотрим функции, предназначенные для выполнения низкоуровневых операций ввода и вывода. Такие функции не поддерживают буферизацию и позволяют читать и писать только байты.

13.1. Открытие и закрытие файла

Для открытия файла предназначены функции `_open()` и `_wopen()`. Прототипы функций:

```
#include <io.h>
int _open(const char *filename, int openFlag[, int pmode]);
#include <io.h> /* или #include <wchar.h> */
int _wopen(const wchar_t *filename, int openFlag[, int pmode]);
```

В первом параметре функции принимают путь к файлу, во втором — режим открытия файла, а в третьем — режим разрешений. Функции открывают файл и возвращают целочисленный дескриптор, с помощью которого производится дальнейшая работа с файлом. Если файл открыть не удалось, то возвращается значение `-1` и устанавливается переменная `errno`.

В параметре `openFlag` могут быть указаны следующие флаги (или их комбинация через оператор `|`):

```
#include <fcntl.h>
```

- `_O_RDONLY` — только чтение. Определение макроса:

  ```
  #define _O_RDONLY 0x0000
  ```

- `_O_WRONLY` — только запись. Определение макроса:

  ```
  #define _O_WRONLY 0x0001
  ```

- `_O_RDWR` — чтение и запись. Определение макроса:

  ```
  #define _O_RDWR 0x0002
  ```

- `_O_APPEND` — добавление в конец файла. Определение макроса:

```
#define _O_APPEND 0x0008
```

- `_O_CREAT` — создать файл, если он не существует, и открыть его для записи. Если флаг указан, то параметр `pmode` является обязательным. Определение макроса:

```
#define _O_CREAT 0x0100
```

- `_O_CREAT | _O_EXCL` — создать файл. Если файл уже существует, то вернуть код ошибки. Определение макроса `_O_EXCL`:

```
#define _O_EXCL 0x0400
```

- `_O_CREAT | _O_TEMPORARY` — создать временный файл. При закрытии файла он будет удален. Определение макроса `_O_TEMPORARY`:

```
#define _O_TEMPORARY 0x0040
```

- `_O_TRUNC` — очистить содержимое файла. Определение макроса:

```
#define _O_TRUNC 0x0200
```

- `_O_BINARY` — файл будет открыт в бинарном режиме. Определение макроса:

```
#define _O_BINARY 0x8000
```

- `_O_TEXT` — файл будет открыт в текстовом режиме. В Windows файлы по умолчанию открываются в текстовом режиме. Определение макроса:

```
#define _O_TEXT 0x4000
```

- `_O_WTEXT` — открывает файл в режиме `UNICODE`. Определение макроса:

```
#define _O_WTEXT 0x10000
```

- `_O_U8TEXT` — открывает файл в режиме `UTF-8`. Определение макроса:

```
#define _O_U8TEXT 0x40000
```

- `_O_U16TEXT` — открывает файл в режиме `UTF-16LE`. Определение макроса:

```
#define _O_U16TEXT 0x20000
```

Существуют также следующие флаги (за подробностями обращайтесь к документации):

```
#define _O_NOINHERIT 0x0080
#define _O_SHORT_LIVED 0x1000
#define _O_SEQUENTIAL 0x0020
#define _O_RANDOM 0x0010
```

Если флаг `_O_CREAT` указан, то в параметре `pmode` нужно обязательно задать режим разрешений с помощью следующих флагов (или их комбинацию через оператор `|`):

```
#include <sys/stat.h>
```

- `_S_IREAD` — разрешено только чтение. Определение макроса:

```
#define _S_IREAD 0x0100
```

- _S_IWRITE — разрешена запись. В Windows флаг одновременно разрешает и чтение и запись, т. е. флаг эквивалентен комбинации флагов _S_IREAD | _S_IWRITE. Определение макроса:

```
#define _S_IWRITE 0x0080
```

Для открытия файла можно также воспользоваться функциями _sopen() и _wsopen(), которые дополнительно позволяют управлять совместным доступом к файлу. Прототипы функций:

```
#include <io.h>
int _sopen(const char *filename, int openFlag, int shareFlag[,
           int pmode]);
#include <io.h> /* или #include <wchar.h> */
int _wsopen(const wchar_t *filename, int openFlag, int shareFlag[,
            int pmode]);
```

Параметр pmode является обязательным только при использовании флага _O_CREAT. В параметре shareFlag можно указать один из следующих макросов:

```
#include <share.h>
```

- _SH_DENYRW — запрещает доступ на чтение и запись. Определение макроса:

```
#define _SH_DENYRW 0x10
```

- _SH_DENYWR — запрещает доступ на запись. Определение макроса:

```
#define _SH_DENYWR 0x20
```

- _SH_DENYRD — запрещает доступ на чтение. Определение макроса:

```
#define _SH_DENYRD 0x30
```

- _SH_DENYNO — разрешает доступ на чтение и запись. Определение макроса:

```
#define _SH_DENYNO 0x40
```

Вместо функций _open(), _wopen(), _sopen() и _wsopen() можно использовать функции _sopen_s() и _wsopen_s(). Прототипы функций.

```
#include <io.h>
errno_t _sopen_s(int *fileHandle, const char *filename, int openFlag,
                 int shareFlag, int pmode);
#include <io.h> /* или #include <wchar.h> */
errno_t _wsopen_s(int *fileHandle, const wchar_t *_Filename,
                  int openFlag, int shareFlag, int pmode);
```

Параметр pmode обязателен, но используется только при наличии флага _O_CREAT. Функции возвращают ненулевое значение в случае ошибки и 0 — в противном случае. Пример открытия файла на чтение в текстовом режиме:

```
int openFlag = _O_RDONLY | _O_TEXT;
int fd = 0;
if (_wsopen_s(&fd, L"C:\\book\\file.txt", openFlag, _SH_DENYWR, 0)) {
   perror("Не удалось открыть файл");
```

```
    exit(1);
}
// Читаем данные из файла
_close(fd); // Закрываем файл
```

Закрыть файл позволяет функция `_close()`. В качестве параметра указывается дескриптор файла. Функция возвращает значение `0`, если файл успешно закрыт, или значение `-1` — в случае ошибки. Прототип функции:

```
#include <io.h>
int _close(int fileHandle);
```

13.2. Чтение из файла и запись в файл

Для чтения из файла и записи в файл предназначены следующие функции.

- `_read()` — читает байты из файла и записывает их в буфер. Прототип функции:

  ```
  #include <io.h>
  int _read(int fileHandle, void *buf, unsigned int maxCharCount);
  ```

  В первом параметре задается дескриптор файла, во втором — указатель на буфер, а в третьем — максимальное число байтов. Функция возвращает количество прочтенных байтов, значение `0` — если достигнут конец файла или значение `-1` — в случае ошибки (дополнительно устанавливает переменную `errno`). Пример чтения строки:

  ```
  char buf[256] = {0};
  int count = _read(fd, buf, 255);
  printf("count = %d\n", count);
  if (count > 0 && count < 256) {
     buf[count] = '\0';
     printf("%s\n", buf);
  }
  else puts("Не удалось прочитать");
  ```

- `_write()` — записывает последовательность байтов в файл. Прототип функции:

  ```
  #include <io.h>
  int _write(int fileHandle, const void *buf,  unsigned int maxCharCount);
  ```

  В первом параметре указывается дескриптор файла, во втором — записываемые данные, а в третьем — число байтов. Функция возвращает количество записанных байтов или значение `-1` — в случае ошибки (дополнительно устанавливает переменную `errno`). Пример:

  ```
  char str[] = "Строка1\nСтрока2";
  unsigned int str_len = (unsigned int) strlen(str);
  int count = _write(fd, str, str_len);
  printf("count = %d\n", count);
  ```

Для явного сброса файла на диск предназначена функция `_commit()`. Функция возвращает значение `0`, если операция успешно выполнена, или значение `-1` — в случае ошибки. Прототип функции:

```
#include <io.h>
int _commit(int fileHandle);
```

Рассмотрим несколько примеров. Откроем файл на запись и запишем в него строку (листинг 13.1). Если файл не существует, то создадим его. Если файл существует, то очистим его (режим `w` в функции `fopen()`).

Листинг 13.1. Запись строки в файл в текстовом режиме

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char str[] = "Строка1\nСтрока2";
   unsigned int str_len = (unsigned int) strlen(str);
   int openFlag = _O_WRONLY | _O_CREAT | _O_TRUNC | _O_TEXT;
   int pmode = _S_IREAD | _S_IWRITE;
   int fd = _open("C:\\book\\file.txt", openFlag, pmode);
   if (fd != -1) {                          // Проверяем успешность
      int count = _write(fd, str, str_len); // Записываем строку
      printf("count = %d\n", count);
      if (count == -1) {
         perror("Не удалось записать строку");
      }
      else puts("Строка записана");
      _close(fd);                           // Закрываем файл
   }
   else perror("Не удалось открыть файл");
   return 0;
}
```

Добавим еще одну строку в конец файла (листинг 13.2). Если файл не существует, то создадим его. Если файл существует, то установим курсор на конец файла (режим `a` в функции `fopen()`).

Листинг 13.2. Запись строки в конец файла в текстовом режиме

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

```
#include <locale.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char str[] = "\nСтрока3";
   unsigned int str_len = (unsigned int) strlen(str);
   int openFlag = _O_WRONLY | _O_CREAT | _O_APPEND | _O_TEXT;
   int pmode = _S_IREAD | _S_IWRITE;
   int fd = _open("C:\\book\\file.txt", openFlag, pmode);
   if (fd != -1) {                          // Проверяем успешность
      int count = _write(fd, str, str_len); // Записываем строку
      printf("count = %d\n", count);
      if (count == -1) {
         perror("Не удалось записать строку");
      }
      else puts("Строка записана");
      _close(fd);                           // Закрываем файл
   }
   else perror("Не удалось открыть файл");
   return 0;
}
```

Прочитаем содержимое файла в текстовом режиме и выведем результат в окно консоли (листинг 13.3). Если файл не существует, то выведем сообщение об ошибке (режим `r` в функции `fopen()`).

Листинг 13.3. Чтение файла в текстовом режиме

```
#include <stdio.h>
#include <locale.h>
#include <io.h>
#include <fcntl.h>

#define BUF_LEN 10

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char buf[BUF_LEN] = {0};
   int openFlag = _O_RDONLY | _O_TEXT;
   int fd = _open("C:\\book\\file.txt", openFlag);
   if (fd != -1) {
      int count = 0;
      while ( !_eof(fd) ) { // Пока не достигнут конец файла
         count = _read(fd, buf, BUF_LEN - 1);
```

```
        if (count == -1) {
           perror("Не удалось прочитать");
           break;
        }
        if (count > 0 && count < BUF_LEN) {
           buf[count] = '\0';
           printf("%s", buf);
        }
     }
     _close(fd);
  }
  else perror("Не удалось открыть файл");
  return 0;
}
```

В этом примере, для проверки достижения конца файла, мы воспользовались функцией `_eof()`. Функция возвращает значение `1`, если был достигнут конец файла, и `0` — если это не так. При ошибке возвращает значение `-1`. Прототип функции:

```
#include <io.h>
int _eof(int fileHandle);
```

13.3. Файлы произвольного доступа

Файл можно читать не только с самого начала, но и с произвольной позиции. Как правило, в этом случае файл открывают в бинарном режиме. Записывать в файл также можно с произвольной позиции. Для работы с файлами произвольного доступа предназначены следующие функции.

- `_tell()` — возвращает позицию курсора относительно начала файла. При ошибке возвращается значение `-1L`. Прототип функции:

  ```
  #include <io.h>
  long _tell(int fileHandle);
  ```

  Пример:

  ```
  long pos = _tell(fd);
  printf("%ld\n", pos);
  ```

  Вместо функции `_tell()` можно воспользоваться функцией `_telli64()`. Прототип функции:

  ```
  #include <io.h>
  __int64 _telli64(int fileHandle);
  ```

  Пример:

  ```
  __int64 pos = _telli64(fd);
  printf("%I64d\n", pos);
  ```

- `_lseek()` — устанавливает курсор в позицию, имеющую смещение `offset` относительно позиции `origin`. В качестве результата функция возвращает новое смещение в байтах от начала файла, если ошибок не возникло, или значение `-1L` — в противном случае. Прототип функции:

```
#include <io.h>
long _lseek(int fileHandle, long offset, int origin);
```

В параметре `origin` могут быть указаны следующие макросы:

- `SEEK_SET` — начало файла;
- `SEEK_CUR` — текущая позиция курсора;
- `SEEK_END` — конец файла.

Определения макросов выглядят следующим образом:

```
#include <stdio.h>
#define SEEK_SET    0
#define SEEK_CUR    1
#define SEEK_END    2
```

Пример:

```
// Относительно начала файла
printf("%ld\n", _lseek(fd, 5L, SEEK_SET)); // 5
// Относительно указателя
printf("%ld\n", _lseek(fd, 2L, SEEK_CUR)); // 7
// Перемещение в конец файла
printf("%ld\n", _lseek(fd, 0L, SEEK_END));
// Перемещение в начало файла
printf("%ld\n", _lseek(fd, 0L, SEEK_SET)); // 0
```

Вместо функции `_lseek()` можно воспользоваться функцией `_lseeki64()`. Прототип функции:

```
#include <io.h>
__int64 _lseeki64(int fileHandle, __int64 offset, int origin);
```

Пример:

```
// Относительно начала файла
printf("%I64d\n", _lseeki64(fd, 5LL, SEEK_SET)); // 5
```

13.4. Создание временных файлов

Для создания временного файла нужно при открытии файла указать комбинацию флагов: `_O_CREAT | _O_TEMPORARY`. Файл будет автоматически удален при закрытии файла. Определение макросов:

```
#include <fcntl.h>
#define _O_CREAT 0x0100
#define _O_TEMPORARY 0x0040
```

Пример создания временного файла приведен в листинге 13.4. Понаблюдайте за каталогом C:\book в течение пяти секунд. Файл будет создан, а затем удален.

Листинг 13.4. Создание временного файла

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <windows.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char str[] = "строка";
   unsigned int str_len = (unsigned int) strlen(str);
   int openFlag = _O_RDWR | _O_CREAT | _O_TEMPORARY | _O_BINARY;
   int pmode = _S_IREAD | _S_IWRITE;
   int fd = _open("C:\\book\\tmpfile.txt", openFlag, pmode);
   if (fd != -1) {
      _write(fd, str, str_len); // Записываем строку
      puts("Понаблюдайте за каталогом C:\\book в течение пяти секунд");
      fflush(stdout);
      Sleep(5000); // Имитация работы с файлом
      puts("Закрываем файл");
      _close(fd);
   }
   else perror("Не удалось открыть файл");
   return 0;
}
```

13.5. Дескрипторы потоков ввода/вывода

Низкоуровневые операции ввода и вывода можно применять и к стандартным потокам. В этом случае указываются следующие макросы или просто их значения:

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

- `STDIN_FILENO` — стандартный ввод `stdin`. Определение макроса:

  ```
  #define STDIN_FILENO 0
  ```

- `STDOUT_FILENO` — стандартный вывод `stdout`. Определение макроса:

  ```
  #define STDOUT_FILENO 1
  ```

- `STDERR_FILENO` — стандартный вывод сообщений об ошибках `stderr`. Определение макроса:

  ```
  #define STDERR_FILENO 2
  ```

Например, вывести строки в окно консоли можно так:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "строка\n";
unsigned int str_len = (unsigned int) strlen(str);
_write(STDOUT_FILENO, str, str_len);
_write(1, str, str_len);
fflush(stdout);
```

Пример вывода сообщения об ошибке в поток `stderr`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "Сообщение об ошибке\n";
unsigned int str_len = (unsigned int) strlen(str);
_write(2, str, str_len);
fflush(stderr);
```

Получить дескриптор открытого потока позволяет функция `_fileno()`. Она возвращает дескриптор потока или значение `-1` в случае ошибки. Если стандартный поток не связан с окном консоли, то функция вернет значение `-2`. Прототип функции:

```
#include <stdio.h>
int _fileno(FILE *File);
```

Пример вывода сообщения об ошибке в поток `stderr`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char str[] = "Сообщение об ошибке\n";
unsigned int str_len = (unsigned int) strlen(str);
int fd = _fileno(stderr);
if (fd > 0) _write(fd, str, str_len);
```

13.6. Преобразование низкоуровневого потока в обычный

Функции `_fdopen()` и `_wfdopen()` преобразуют низкоуровневый поток в обычный. Прототипы функций:

```
#include <stdio.h>
FILE *_fdopen(int fileHandle, const char *mode);
#include <stdio.h> /* или #include <wchar.h> */
FILE *_wfdopen(int fileHandle, const wchar_t *mode);
```

В первом параметре указывается дескриптор низкоуровневого потока, а во втором — режим открытия файла *(см. разд. 12.3)*. Функции возвращают указатель на открытый поток или нулевой указатель в случае ошибки. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
int openFlag = _O_WRONLY | _O_CREAT | _O_TRUNC;
int pmode = _S_IREAD | _S_IWRITE;
int fd = _open("C:\\book\\file.txt", openFlag, pmode);
```

```
if (fd == -1) exit(1);
FILE *fp = _fdopen(fd, "w");
if (!fp) exit(1);
fputs("Пишем строку в файл", fp);
fflush(fp);
fclose(fp); // Закрываем fp и fd
```

13.7. Создание копии потока

Функция `_dup()` создает копию потока и возвращает дескриптор. При возникновении ошибки функция вернет значение `-1`. Прототип функции:

```
#include <io.h>
int _dup(int fileHandle);
```

Пример:

```
int dup_stdout = _dup(_fileno(stdout));
printf("%d\n", dup_stdout);       // 3
printf("%d\n", _fileno(stdout));  // 1
```

13.8. Перенаправление потоков

Функция `_dup2()` связывает дескриптор потока `fileHandleSrc` с дескриптором потока `fileHandleDst`. Если операция успешно выполнена, то функция возвращает значение `0`. При ошибке возвращает значение `-1`. Прототип функции:

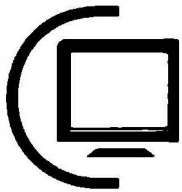
```
#include <io.h>
int _dup2(int fileHandleSrc, int fileHandleDst);
```

В качестве примера использования функций `_dup()` и `_dup2()` создадим копию потока `stdout`, откроем файл и перенаправим поток `stdout` в файл, а затем восстановим вывод на консоль (листинг 13.5).

Листинг 13.5. Временное перенаправление потока `stdout` в файл

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <io.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   // Сохраняем поток stdout
   int dup_stdout = _dup(_fileno(stdout));
   // Открываем файл
   FILE *fp = fopen("C:\\book\\file.txt", "w");
```

```
    if (!fp) exit(1);
    // Перенаправляем поток в файл
    if (_dup2(_fileno(fp), 1) == -1) {
       perror("Не удалось перенаправить\n");
       exit(1);
    }
    printf("%s\n", "Пишем строку в файл");
    fflush(stdout);
    fclose(fp);
    // Восстанавливаем вывод на консоль
    _dup2(dup_stdout, 1);
    printf("%s\n", "Выводим строку на консоль");
    return 0;
}
```



ГЛАВА 14

Работа с файловой системой

В предыдущих двух главах были описаны способы работы с файлами в языке C. Теперь мы переходим к рассмотрению вопросов переименования и удаления файлов, получения информации о файлах, а также созданию, удалению и чтению каталогов. Следует сразу заметить, что реализация функций, описанных в этой главе, зависит от операционной системы и компилятора. Рассматриваемые функции применимы для операционной системы Windows.

14.1. Преобразование пути к файлу или каталогу

Преобразовать путь к файлу или каталогу позволяют следующие функции.

- `_fullpath()` и `_wfullpath()` — преобразуют относительный путь в абсолютный путь, учитывая местоположение текущего рабочего каталога. Прототипы функций:

```
#include <stdlib.h>
ohar *_fullpath(char *fullPath, const char *path,
                size_t sizeInBytes);
#include <stdlib.h> /* или #include <wchar.h> */
wchar_t *_wfullpath(wchar_t *fullPath, const wchar_t *path,
                    size_t sizeInWords);
```

  В первом параметре передается указатель на буфер, в который будет записан абсолютный путь. Если в первом параметре передать нулевой указатель, то память под абсолютный путь будет выделена динамически с помощью функции `malloc()`. В этом случае после использования памяти ее следует освободить с помощью функции `free()`. Во втором параметре указывается относительный путь, а в третьем параметре — максимальный размер буфера. В качестве максимального размера буфера можно указать макрос `_MAX_PATH`. Определение макроса выглядит так:

```
#include <stdlib.h>
#define _MAX_PATH   260
```

Пример преобразования относительного пути в абсолютный путь:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char full_path[_MAX_PATH] = {0}, *p = NULL;
p = _fullpath(full_path, "test.txt", _MAX_PATH);
if (p) {
   printf("%s\n", full_path);
} // Результат: C:\cpp\projects\Test64c\test.txt
p = _fullpath(NULL, "..\\..\\test.txt", _MAX_PATH);
if (p) {
   printf("%s\n", p); // C:\cpp\test.txt
   free(p);
}
```

- `_splitpath_s()` и `_wsplitpath_s()` — разбивают абсолютный путь на составляющие: имя диска, путь, имя файла и расширение. Прототипы функций:

```
#include <stdlib.h>
errno_t _splitpath_s(const char *fullPath, char *drive,
        size_t driveSize, char *dir, size_t dirSize,
        char *filename, size_t filenameSize,
        char *ext, size_t extSize);
#include <stdlib.h> /* или #include <wchar.h> */
errno_t _wsplitpath_s(const wchar_t *fullPath, wchar_t *drive,
        size_t driveSize, wchar_t *dir, size_t dirSize,
        wchar_t *filename, size_t filenameSize,
        wchar_t *ext, size_t extSize);
```

В параметре `fullPath` задается абсолютный путь. В параметре `drive` передается указатель на буфер, в котором будет сохранено имя диска и двоеточие, а в параметре `driveSize` — максимальный размер этого буфера. В параметре `dir` передается указатель на буфер, в котором будет сохранен путь, а в параметре `dirSize` — максимальный размер этого буфера. В параметре `filename` передается указатель на буфер, в котором будет сохранено имя файла без расширения, а в параметре `filenameSize` — максимальный размер этого буфера. В параметре `ext` передается указатель на буфер, в котором будет сохранено расширение файла с предваряющей точкой, а в параметре `extSize` — максимальный размер этого буфера. Функции возвращают значение `0`, если ошибок нет, и код ошибки при неудачном выполнении.

В качестве размеров обычно указываются следующие макросы:

```
#include <stdlib.h>
#define _MAX_DRIVE  3
#define _MAX_DIR    256
#define _MAX_FNAME  256
#define _MAX_EXT    256
```

Пример разбиения абсолютного пути на составляющие:

```
setlocale(LC_ALL, "Russian_Russia.1251");
errno_t err = 0;
```

```
char full_path[] = "C:\\book\\test.txt";
char drive[_MAX_DRIVE] = {0}, dir[_MAX_DIR] = {0};
char name[_MAX_FNAME] = {0}, ext[_MAX_EXT] = {0};
err = _splitpath_s(full_path, drive, _MAX_DRIVE,
        dir, _MAX_DIR, name, _MAX_FNAME, ext, _MAX_EXT);
if (err == 0) {
   printf("drive: '%s'\n", drive); // drive: 'C:'
   printf("dir: '%s'\n", dir);     // dir: '\book\'
   printf("name: '%s'\n", name);   // name: 'test'
   printf("ext: '%s'\n", ext);     // ext: '.txt'
}
```

Если какая-либо из составляющих не нужна, то в соответствующем параметре следует передать нулевой указатель и в качестве размера указать значение 0. Пример получения только имени файла:

```
setlocale(LC_ALL, "Russian_Russia.1251");
errno_t err = 0;
char full_path[] = "C:\\book\\test.txt";
char name[_MAX_FNAME] = {0};
err = _splitpath_s(full_path, NULL, 0,
                   NULL, 0, name, _MAX_FNAME, NULL, 0);
if (err == 0) {
   printf("name: %s\n", name); // name: test
}
```

Вместо функций `_splitpath_s()` и `_wsplitpath_s()` можно воспользоваться функциями `_splitpath()` и `_wsplitpath()`, но они не проверяют размеры буферов. Прототипы функций:

```
#include <stdlib.h>
void _splitpath(const char *fullPath, char *drive,
                char *dir, char *filename, char *ext);
#include <stdlib.h> /* или #include <wchar.h> */
void _wsplitpath(const wchar_t *fullPath, wchar_t *drive,
            wchar_t *dir, wchar_t *filename, wchar_t *ext);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char full_path[] = "C:\\book\\test.txt";
char drive[_MAX_DRIVE] = {0}, dir[_MAX_DIR] = {0};
char name[_MAX_FNAME] = {0}, ext[_MAX_EXT] = {0};
_splitpath(full_path, drive, dir, name, ext);
printf("drive: '%s'\n", drive); // drive: 'C:'
printf("dir: '%s'\n", dir);     // dir: '\book\'
printf("name: '%s'\n", name);   // name: 'test'
printf("ext: '%s'\n", ext);     // ext: '.txt'
```

- `_makepath_s()` и `_wmakepath_s()` — соединяют элементы пути (имя диска, путь, имя файла и расширение) в абсолютный путь. Прототипы функций:

```
#include <stdlib.h>
errno_t _makepath_s(char *pathResult, size_t sizeInWords,
                const char *drive, const char *dir,
                const char *filename, const char *ext);
#include <stdlib.h> /* или #include <wchar.h> */
errno_t _wmakepath_s(wchar_t *pathResult, size_t sizeInWords,
                const wchar_t *drive, const wchar_t *dir,
                const wchar_t *filename, const wchar_t *ext);
```

В первом параметре передается указатель на буфер, в который будет записан абсолютный путь, а во втором параметре указывается максимальный размер буфера. В параметре `drive` задается имя диска, в параметре `dir` — путь, в параметре `filename` — имя файла, а в параметре `ext` — расширение файла. Некоторые из этих четырех параметров можно не указывать, в этом случае следует передать нулевой указатель или пустую строку. Функции возвращают значение `0`, если ошибок не было, и код ошибки при неудачном выполнении. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
errno_t err = 0;
char full_path[_MAX_PATH] = {0};
err = _makepath_s(full_path, _MAX_PATH, "C", "\\book\\",
                  "test", "txt");
if (err == 0) {
   printf("%s\n", full_path); // C:\book\test.txt
}
```

Вместо функций `_makepath_s()` и `_wmakepath_s()` можно воспользоваться функциями `_makepath()` и `_wmakepath()`, но они не проверяют размер буфера. Прототипы функций:

```
#include <stdlib.h>
void _makepath(char *pathResult, const char *drive,
                const char *dir, const char *filename,
                const char *ext);
#include <stdlib.h> /* или #include <wchar.h> */
void _wmakepath(wchar_t *pathResult, const wchar_t *drive,
                 const wchar_t *dir, const wchar_t *filename,
                 const wchar_t *ext);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
wchar_t full_path[_MAX_PATH] = {0};
_wmakepath(full_path, L"C", L"\\book\\", L"test", L"txt");
wprintf(L"%s\n", full_path); // C:\book\test.txt
```

14.2. Переименование, перемещение и удаление файла

Для переименования, перемещения и удаления файла предназначены следующие функции.

- `rename()` и `_wrename()` — переименовывают файл или каталог, а также позволяют переместить файл в другой каталог. Прототипы функций:

```
#include <stdio.h> /* или #include <io.h> */
int rename(const char *oldFilename, const char *newFilename);
#include <io.h> /* или #include <wchar.h> */
int _wrename(const wchar_t *oldFilename, const wchar_t *newFilename);
```

  В первом параметре задается старое название файла, а во втором параметре — новое название. Перед названием файлов можно указать путь. Если новый путь не совпадает со старым, то файл будет перемещен. Каталоги можно только переименовать, перемещать каталоги нельзя. Если операция успешно произведена, то функции возвращают значение `0`. В противном случае функции возвращают ненулевое значение и переменной `errno` присваивается код ошибки. Пример переименования и перемещения файла (каталог C:\book\folder1 должен существовать):

```
setlocale(LC_ALL, "Russian_Russia.1251");
int r = rename("C:\\book\\file1.txt", "C:\\book\\file3.txt");
if (r != 0) {
   printf("%s\n", "Не удалось переименовать файл");
}
else {
   printf("%s\n", "Файл успешно переименован");
}
r = _wrename(L"C:\\book\\file2.txt",
             L"C:\\book\\folder1\\file2.txt");
if (r == 0) {
   printf("%s\n", "Файл успешно перемещен в каталог folder1");
}
```

- `remove()`, `_unlink()`, `_wremove()` и `_wunlink()` — удаляют файл, название которого передано в параметре. Если операция успешно произведена, то функции возвращают значение `0`. В противном случае функции возвращают значение `-1` и переменной `errno` присваивается код ошибки. Прототипы функций:

```
#include <stdio.h> /* или #include <io.h> */
int remove(const char *filename);
int _unlink(const char *filename);
#include <stdio.h> /* или #include <wchar.h> */
int _wremove(const wchar_t *filename);
#include <io.h> /* или #include <wchar.h> */
int _wunlink(const wchar_t *filename);
```

Пример удаления файла:

```
setlocale(LC_ALL, "Russian_Russia.1251");
if (remove("C:\\book\\folder1\\file2.txt") != 0) {
   printf("%s\n", "Не удалось удалить файл");
}
else {
   printf("%s\n", "Файл удален");
}
```

14.3. Проверка прав доступа к файлу и каталогу

Для проверки существования файла и каталога, а также возможности чтения и записи файла предназначены функции `_access()` и `_waccess()`. Прототипы функций:

```
#include <io.h>
int _access(const char *filename, int accessMode);
#include <io.h> /* или #include <wchar.h> */
int _waccess(const wchar_t *filename, int accessMode);
```

В первом параметре указывается путь к файлу или каталогу, а во втором параметре задается одно из следующих значений:

- `0` — проверка существования файла или каталога;
- `2` — проверка возможности записи в файл;
- `4` — проверка возможности чтения файла;
- `6` — проверка возможности чтения и записи файла.

В MinGW вместо чисел можно указать следующие макросы:

```
#include <io.h>
#define F_OK  0
#define W_OK  2
#define R_OK  4
```

В случае успешности проверки функции возвращают значение `0`. В противном случае функции возвращают значение `-1` и переменной `errno` присваивается код ошибки. Пример проверки доступа к файлу приведен в листинге 14.1.

Листинг 14.1. Проверка доступа к файлу

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
```

```
  char path[] = "C:\\book\\file.txt";
  if (_access(path, 0) != -1) {
     printf("%s\n", "Файл существует");
     if (_access(path, 2) != -1) {
        printf("%s\n", "Файл доступен для записи");
     }
     else perror("");
     if (_access(path, 4) != -1) {
        printf("%s\n", "Файл доступен для чтения");
     }
     else perror("");
     if (_access(path, 6) != -1) {
        printf("%s\n", "Файл доступен для чтения и записи");
     }
     else perror("");
  }
  else perror("");
  return 0;
}
```

Вместо функций `_access()` и `_waccess()` можно воспользоваться функциями `_access_s()` и `_waccess_s()`. Прототипы функций:

```
#include <io.h>
errno_t _access_s(const char *filename, int accessMode);
#include <io.h> /* или #include <wchar.h> */
errno_t _waccess_s(const wchar_t *filename, int accessMode);
```

В случае успешности проверки функции возвращают значение `0`, в противном случае — код ошибки. Пример проверки:

```
setlocale(LC_ALL, "Russian_Russia.1251");
wchar_t path[] = L"C:\\book\\file.txt";
if (_waccess_s(path, 0) == 0) {
   printf("%s\n", "Файл существует");
   if (_waccess_s(path, 2) == 0) {
      printf("%s\n", "Файл доступен для записи");
   }
}
```

Обратите внимание

Успешная проверка на чтение или запись еще не гарантирует, что можно прочитать или записать файл. Например, файл может быть заблокирован другим процессом.

14.4. Изменение прав доступа к файлу

Для изменения прав доступа к файлу предназначены функции `_chmod()` и `_wchmod()`. Прототипы функций:

```
#include <io.h>
int _chmod(const char *filename, int mode);
#include <io.h> /* или #include <wchar.h> */
int _wchmod(const wchar_t *filename, int mode);
```

В первом параметре указывается путь к файлу, а во втором параметре задается режим разрешений с помощью следующих флагов (или их комбинация через оператор `|`):

```
#include <sys/stat.h>
```

- `_S_IREAD` — разрешено только чтение. Определение макроса:

  ```
  #define _S_IREAD 0x0100
  ```

- `_S_IWRITE` — разрешена запись. В Windows флаг одновременно разрешает и чтение, и запись, т. е. флаг эквивалентен комбинации флагов `_S_IREAD | _S_IWRITE`. Определение макроса:

  ```
  #define _S_IWRITE 0x0080
  ```

В случае успешности установки прав доступа функции возвращают значение `0`. В противном случае функции возвращают значение `-1` и глобальной переменной `errno` присваивается код ошибки. Пример установки прав доступа только для чтения приведен в листинге 14.2.

Листинг 14.2. Установка прав доступа только для чтения

```
#include <stdio.h>
#include <locale.h>
#include <io.h>
#include <sys/stat.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char path[] = "C:\\book\\file3.txt";
   if (_chmod(path, _S_IREAD) != -1) {
      printf("%s\n", "OK");
   }
   else perror("Error");
   return 0;
}
```

14.5. Делаем файл скрытым

Для того чтобы в Windows сделать файл скрытым, нужно воспользоваться макросами `GetFileAttributes()` и `SetFileAttributes()` из WinAPI. В зависимости от установки режима `UNICODE` макросы вызывают следующие функции:

```
#include <windows.h>
DWORD GetFileAttributesA(LPCSTR lpFileName);
DWORD GetFileAttributesW(LPCWSTR lpFileName);
WINBOOL SetFileAttributesA(LPCSTR lpFileName, DWORD dwFileAttributes);
WINBOOL SetFileAttributesW(LPCWSTR lpFileName, DWORD dwFileAttributes);
```

Две первые функции возвращают атрибуты файла `lpFileName` или значение `(DWORD)(-1)` — в случае ошибки. Две последние функции устанавливают атрибуты `dwFileAttributes` для файла `lpFileName`. Для того чтобы сделать файл скрытым, вначале нужно получить атрибуты файла, а затем установить флаг `FILE_ATTRIBUTE_HIDDEN` (листинг 14.3).

Листинг 14.3. Делаем файл скрытым

```
#include <stdio.h>
#include <locale.h>
#include <io.h>
#include <windows.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   char path[] = "C:\\book\\file3.txt";
   if (_access(path, 0) == -1) {
      puts("Файл не существует");
      return 1;
   }
   DWORD attr = GetFileAttributes(path);
   if (attr != (DWORD)(-1)) {
      if (attr & FILE_ATTRIBUTE_HIDDEN) {
         puts("Файл уже скрытый");
         // Сбросить флаг можно так
         //SetFileAttributes(path, attr ^ FILE_ATTRIBUTE_HIDDEN);
      }
      else {
         if (SetFileAttributes(path, attr | FILE_ATTRIBUTE_HIDDEN)) {
            printf("%s\n", "OK");
         }
         else puts("Error");
      }
   }
```

```
   else puts("Error");
   return 0;
}
```

14.6. Получение информации о файле

Получить размер файла и время создания, изменения и доступа к файлу, а также значения других метаданных позволяют функции `_stat()`, `_stat32()`, `_stat64()`, `_stati64()`, `_stat32i64()`, `_stat64i32()`, `_wstat()`, `_wstat32()`, `_wstat64()`, `_wstati64()`, `_wstat32i64()` и `_wstat64i32()`.

По умолчанию в проекте `Test64c` функция `_stat()` эквивалентна функции `_stat64i32()`. Число `64` в этой функции означает количество битов, выделяемых под дату, а число `32` — количество битов, выделяемых под размер файла. Прототип функции `_stat64i32()`:

```
#include <sys/types.h>
#include <sys/stat.h>
#define _stat _stat64i32
int _stat64i32(const char *name, struct _stat64i32 *stat);
```

В первом параметре указывается путь к файлу. Результат записывается в структуру `stat`. Функция возвращает значение `0`, если ошибок не было, и значение `-1` в случае ошибки. Код ошибки сохраняется в переменной `errno`.

По умолчанию в проекте `Test64c` функция `_wstat()` эквивалентна функции `_wstat64i32()`. Прототип функции:

```
#include <sys/types.h>
#include <sys/stat.h> /* или #include <wchar.h> */
#define _wstat _wstat64i32
int _wstat64i32(const wchar_t *name, struct _stat64i32 *stat);
```

Структура `_stat64i32` объявлена следующим образом:

```
struct _stat64i32 {
   _dev_t          st_dev;     // Номер диска
   _ino_t          st_ino;
   unsigned short  st_mode;    // Маска прав доступа
   short           st_nlink;
   short           st_uid;
   short           st_gid;
   _dev_t          st_rdev;
   _off_t          st_size;    // Размер в байтах
   __time64_t      st_atime;   // Дата последнего доступа
   __time64_t      st_mtime;   // Дата изменения
   __time64_t      st_ctime;   // Дата создания
};
```

Поле `st_mode` содержит маску типа устройства и прав доступа. Возможна комбинация следующих флагов:

```
#include <sys/stat.h>
#define _S_IFMT         0xF000   /* Маска для типа файла */
#define _S_IFDIR        0x4000   /* Каталог */
#define _S_IFCHR        0x2000   /* Символьное устройство */
#define _S_IFIFO        0x1000   /* Канал */
#define _S_IFREG        0x8000   /* Обычный файл */
#define _S_IREAD        0x0100   /* Разрешение на чтение */
#define _S_IWRITE       0x0080   /* Разрешение на запись */
#define _S_IEXEC        0x0040   /* Разрешение на выполнение/поиск */
```

При использовании функций `_stat64()` **и** `_wstat64()` **64 бита выделяется и под дату, и под размер файла. Прототипы функций:**

```
#include <sys/types.h>
#include <sys/stat.h>
int _stat64(const char *name, struct _stat64 *stat);
#include <sys/stat.h> /* или #include <wchar.h> */
int _wstat64(const wchar_t *name, struct _stat64 *stat);
```

Объявление структуры `_stat64`:

```
struct _stat64 {
   _dev_t st_dev;
   _ino_t st_ino;
   unsigned short st_mode;
   short st_nlink;
   short st_uid;
   short st_gid;
   _dev_t st_rdev;
   __int64 st_size;
   __time64_t st_atime;
   __time64_t st_mtime;
   __time64_t st_ctime;
};
```

Пример использования функции `_stat()` **приведен в листинге** 14.4.

Листинг 14.4. Пример использования функции _stat()

```
#include <stdio.h>
#include <locale.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>

void print_time(__time64_t t);

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
```

```
   struct _stat finfo;
   unsigned int ch = 0;
   int err = _stat("C:\\book\\file.txt", &finfo);
   if (err == 0) {
      printf("%ld\n", finfo.st_size); // Размер файла
      ch = finfo.st_dev + 'A';
      printf("%c\n", (char)ch);       // Имя диска
      print_time(finfo.st_ctime);     // Создание файла
      print_time(finfo.st_atime);     // Последний доступ
      print_time(finfo.st_mtime);     // Изменение файла
   }
   else perror("Error");
   return 0;
}
void print_time(__time64_t t) {
   struct tm *ptm = NULL;
   ptm = _localtime64(&t);
   if (ptm) {
      char str[100] = {0};
      size_t count = strftime(str, 100, "%d.%m.%Y %H:%M:%S", ptm);
      if (count) {
         printf("%s\n", str);
      }
   }
}
```

Получить информацию об уже открытом файле позволяют функции `_fstat()`, `_fstat32()`, `_fstat64()`, `_fstati64()`, `_fstat64i32()` **и** `_fstat32i64()`. **По умолчанию в проекте** `Test64c` **функция** `_fstat()` **эквивалентна функции** `_fstat64i32()`. **Прототип функции** `_fstat64i32()`:

```
#include <sys/types.h>
#include <sys/stat.h>
#define _fstat _fstat64i32
int _fstat64i32(int fileHandle, struct _stat64i32 *stat);
```

В первом параметре указывается дескриптор открытого файла. Результат записывается в структуру `stat`. **Функция возвращает значение** `0`, **если ошибок не было, и значение** `-1` **в случае ошибки. Код ошибки сохраняется в переменной** `errno`.

Определить размер открытого файла позволяют функции `_filelength()` **и** `_filelengthi64()`. **Прототипы функций:**

```
#include <io.h>
long _filelength(int fileHandle);
__int64 _filelengthi64(int fileHandle);
```

В качестве параметра указывается дескриптор открытого файла. Пример:

```
printf("%ld\n", _filelength(fd));
printf("%I64d\n", _filelengthi64(fd));
```

Обновить время последнего доступа и время изменения файла позволяют функции `_utime()`, `_utime32()`, `_utime64()`, `_wutime()`, `_wutime32()` и `_wutime64()`. Прототип функции `_utime()`:

```
#include <sys/utime.h>
int _utime(const char *filename, struct _utimbuf *utimbuf);
```

В первом параметре функция принимает путь к файлу. Во втором параметре передается адрес структуры `_utimbuf`. Объявление структуры выглядит так:

```
struct _utimbuf {
   time_t actime;          /* Дата последнего доступа */
   time_t modtime;         /* Дата изменения */
};
```

Если во втором параметре передать нулевой указатель, то дата будет текущей. Функция возвращает значение `0`, если ошибок не было, и значение `-1` в противном случае. Код ошибки сохраняется в переменной `errno`.

Пример использования функции `_utime()` приведен в листинге 14.5.

Листинг 14.5. Пример использования функции `_utime()`

```
#include <stdio.h>
#include <locale.h>
#include <time.h>
#include <sys/utime.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   struct _utimbuf new_time;
   new_time.actime = time(NULL) - 3600;
   new_time.modtime = time(NULL) - 7200;
   int err = _utime("C:\\book\\file.txt", &new_time);
   if (err == 0) {
      printf("%s\n", "OK");
   }
   else perror("Error");
   // Текущее время
   err = _utime("C:\\book\\file3.txt", NULL);
   if (err == 0) {
      printf("%s\n", "OK");
   }
   else perror("Error");
   return 0;
}
```

Обновить время последнего доступа для открытого файла позволяют функции `_futime()`, `_futime32()` и `_futime64()`. Прототип функции `_futime()`:

```
#include <sys/utime.h>
int _futime(int fileHandle, struct _utimbuf *utimbuf);
```

В первом параметре указывается дескриптор открытого файла. Во втором параметре передается адрес структуры `utimbuf`. Функция возвращает значение `0`, если ошибок не было, и значение `-1` в случае ошибки. Код ошибки сохраняется в переменной `errno`.

14.7. Функции для работы с дисками

Для работы с дисками используются следующие функции.

- `_getdrives()` — возвращает информацию о доступных дисках в виде набора битов. Прототип функции:

```
#include <direct.h>
unsigned long _getdrives(void);
```

Пример:

```
unsigned long mask = _getdrives();
int ch = 'A';
while (mask) {
   if (mask & 1) printf("%c\n", (char)ch);
   ++ch;
   mask >>= 1;
}
```

- `_getdrive()` — возвращает номер текущего диска (`1` — диск `A`, `2` — диск `B` и т. д.). Прототип функции:

```
#include <direct.h>
int _getdrive(void);
```

Пример:

```
int drive = _getdrive();
int ch = drive - 1 + 'A';
printf("%c\n", (char)ch); // C
```

- `_chdrive()` — делает указанный диск текущим. Прототип функции:

```
#include <direct.h>
int _chdrive(int drive);
```

В параметре `drive` указывается номер диска (`1` — диск `A`, `2` — диск `B` и т. д.). Функция возвращает значение `0`, если ошибок нет, и значение `-1` в противном случае. Код ошибки сохраняется в переменной `errno`. Пример:

```
if (_chdrive(4) == 0) {
   printf("%d\n", _getdrive()); // 4
}
```

Для того чтобы получить информацию о типе диска в Windows, можно воспользоваться макросом `GetDriveType()` **из WinAPI, который в зависимости от определения макроса** `UNICODE` **вызывает одну из следующих функций:**

```
#include <windows.h>
UINT GetDriveTypeA(LPCSTR lpRootPathName);
UINT GetDriveTypeW(LPCWSTR lpRootPathName);
```

Пример:

```
UINT t = GetDriveType("C:\\");
if (t == DRIVE_REMOVABLE) puts("DRIVE_REMOVABLE");
else if (t == DRIVE_FIXED) puts("DRIVE_FIXED");
else if (t == DRIVE_REMOTE) puts("DRIVE_REMOTE");
else if (t == DRIVE_CDROM) puts("DRIVE_CDROM");
else if (t == DRIVE_RAMDISK) puts("DRIVE_RAMDISK");
else puts("Не удалось определить тип диска");
```

Для того чтобы получить информацию о размерах диска в Windows, можно воспользоваться макросом `GetDiskFreeSpaceEx()` **из WinAPI, который в зависимости от определения макроса** `UNICODE` **вызывает одну из следующих функций:**

```
#include <windows.h>
BOOL GetDiskFreeSpaceExA(LPCSTR lpDirectoryName,
        PULARGE_INTEGER lpFreeBytesAvailableToCaller,
        PULARGE_INTEGER lpTotalNumberOfBytes,
        PULARGE_INTEGER lpTotalNumberOfFreeBytes);
BOOL GetDiskFreeSpaceExW(LPCWSTR lpDirectoryName,
        PULARGE_INTEGER lpFreeBytesAvailableToCaller,
        PULARGE_INTEGER lpTotalNumberOfBytes,
        PULARGE_INTEGER lpTotalNumberOfFreeBytes);
```

Пример:

```
unsigned __int64 free_bytes;
unsigned __int64 total_bytes;
unsigned __int64 total_free_bytes;
BOOL st = GetDiskFreeSpaceExA("C:\\",
               (PULARGE_INTEGER) &free_bytes,
               (PULARGE_INTEGER) &total_bytes,
               (PULARGE_INTEGER) &total_free_bytes);
if (st) {
   printf("free_bytes = %I64u\n", free_bytes);
   printf("total_bytes = %I64u\n", total_bytes);
   printf("total_free_bytes = %I64u\n", total_free_bytes);
}
else puts("Не удалось определить размеры диска");
```

14.8. Функции для работы с каталогами

Для работы с каталогами используются следующие функции.

- `_getcwd()` и `_wgetcwd()` — позволяют получить строковое представление текущего рабочего каталога. От этого значения зависит преобразование относительного пути в абсолютный. Кроме того, важно помнить, что текущим рабочим каталогом будет каталог, из которого запускается файл, а не каталог с исполняемым файлом. Прототипы функций:

```
#include <direct.h>
char *_getcwd(char *buf, int sizeInBytes);
#include <wchar.h> /* или #include <direct.h> */
wchar_t *_wgetcwd(wchar_t *buf, int sizeInWords);
```

  В первом параметре функции принимают указатель на буфер, а во втором параметре — максимальный размер буфера. В качестве максимального размера буфера можно указать макрос `_MAX_PATH`. В случае ошибки функции возвращают нулевой указатель. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char buf[_MAX_PATH] = {0}, *p = NULL;
p = _getcwd(buf, _MAX_PATH);
if (p) {
   printf("%s\n", buf); // C:\cpp\projects\Test64c
}
```

  Если в качестве параметров заданы нулевой указатель и нулевое значение, то строка создается динамически с помощью функции `malloc()`. В этом случае функции возвращают указатель на эту строку или нулевой указатель в случае ошибки. После окончания работы со строкой следует освободить память с помощью функции `free()`. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char *pbuf = NULL;
pbuf = _getcwd(NULL, 0);
if (pbuf) {
   printf("%s\n", pbuf); // C:\cpp\projects\Test64c
   free(pbuf);
   pbuf = NULL;
}
```

- `_getdcwd()` и `_wgetdcwd()` — функции аналогичны функциям `_getcwd()` и `_wgetcwd()`, но позволяют получить строковое представление текущего рабочего каталога не для текущего диска, а для указанного в первом параметре. Прототипы функций:

```
#include <direct.h>
char *_getdcwd(int drive, char *buf, int sizeInBytes);
#include <direct.h> /* или #include <wchar.h> */
wchar_t *_wgetdcwd(int drive, wchar_t *buf, int sizeInWords);
```

В первом параметре указывается номер диска (0 — диск по умолчанию, 1 — диск A, 2 — диск B и т. д.). Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char buf[_MAX_PATH] = {0}, *p = NULL;
p = _getdcwd(3, buf, _MAX_PATH);
if (p) {
   printf("%s\n", buf); // C:\cpp\projects\Test64c
}
```

- _chdir() и _wchdir() — делают указанный каталог текущим. Функции возвращают значение 0, если ошибок не было, и значение -1 в противном случае. Код ошибки сохраняется в переменной errno. Прототипы функций:

```
#include <direct.h>
int _chdir(const char *path);
#include <direct.h> /* или #include <wchar.h> */
int _wchdir(const wchar_t *path);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char buf[_MAX_PATH] = {0};
if (_chdir("C:\\book\\") == 0) {
   _getcwd(buf, _MAX_PATH);
   printf("%s\n", buf); // C:\book
}
```

- _mkdir() и _wmkdir() — создают новый каталог. Функции возвращают значение 0, если каталог создан, и значение -1 в противном случае. Код ошибки сохраняется в переменной errno. Прототипы функций:

```
#include <direct.h>
int _mkdir(const char *path);
#include <direct.h> /* или #include <wchar.h> */
int _wmkdir(const wchar_t *path);
```

Пример:

```
if (_mkdir("C:\\book\\folderA") == 0) {
   printf("%s\n", "OK");
}
if (_wmkdir(L"C:\\book\\folderB") == 0) {
   printf("%s\n", "OK");
}
```

- _rmdir() и _wrmdir() — удаляют каталог. Обратите внимание: удалить можно только пустой каталог. Функции возвращают значение 0, если каталог удален, и значение -1 в противном случае. Код ошибки сохраняется в переменной errno. Прототипы функций:

```
#include <direct.h>
int _rmdir(const char *path);
```

```
#include <direct.h> /* или #include <wchar.h> */
int _wrmdir(const wchar_t *path);
```

Пример удаления пустых каталогов:

```
if (_rmdir("C:\\book\\folderA") == 0) {
   printf("%s\n", "OK");
}
if (_wrmdir(L"C:\\book\\folderB") == 0) {
   printf("%s\n", "OK");
}
```

14.9. Перебор объектов, расположенных в каталоге

Перебрать все (или только некоторые) объекты в указанном каталоге позволяют функции `_findfirst()`, `_findnext()` и `_findclose()`. Поиск осуществляется следующим образом:

1. Вызывается функция `_findfirst()`. Функция возвращает дескриптор, с помощью которого производится дальнейший поиск.
2. В цикле вызывается функция `_findnext()`. Этой функции необходимо передать дескриптор, возвращаемый функцией `_findfirst()`. Если объектов больше нет, то функция возвращает значение `-1`.
3. С помощью функции `_findclose()` завершается поиск.

По умолчанию в проекте `Test64c` функция `_findfirst()` соответствует функции `_findfirst64i32()`, а функция `_findnext()` — функции `_findnext64i32()`. Число `64` в этих функциях означает количество битов, выделяемых под дату, а число `32` — количество битов, выделяемых под размер файла. Прототипы функций:

```
#include <io.h>
#define _findfirst _findfirst64i32
intptr_t _findfirst64i32(const char *filename,
                  struct _finddata64i32_t *findData);
#define _findnext _findnext64i32
int _findnext64i32(intptr_t findHandle,
                  struct _finddata64i32_t *findData);
int _findclose(intptr_t findHandle);
```

В первом параметре функции `_findfirst64i32()` указываются путь к каталогу и маска поиска. Если путь не указан, то поиск производится в текущем рабочем каталоге. В маске можно использовать следующие специальные символы:

- `?` — любой одиночный символ;
- `*` — любое количество символов.

Рассмотрим несколько примеров масок:

- `*` — все файлы и подкаталоги;
- `*.txt` — все файлы с расширением txt;
- `f???.t??` — файлы с именами из четырех букв, названия которых начинаются с буквы `f`, имеющие расширение из трех букв, начинающееся с буквы `t`;
- `f*` — файлы и подкаталоги, названия которых начинаются с буквы `f`.

В параметре `findData` передается адрес структуры `_finddata64i32_t`, в которую будут записаны атрибуты найденного объекта. Структура объявлена так:

```
struct _finddata64i32_t {
   unsigned    attrib;        // Флаги
   __time64_t  time_create;   // Дата создания
   __time64_t  time_access;   // Дата последнего доступа
   __time64_t  time_write;    // Дата изменения
   _fsize_t    size;          // Размер файла
   char        name[260];     // Название объекта
};
```

Поле `attrib` может содержать комбинацию следующих флагов:

```
#define _A_NORMAL       0x00    /* Обычный файл */
#define _A_RDONLY       0x01    /* Файл только для чтения */
#define _A_HIDDEN       0x02    /* Скрытый файл */
#define _A_SYSTEM       0x04    /* Системный файл */
#define _A_SUBDIR       0x10    /* Подкаталог */
#define _A_ARCH         0x20    /* Архивный файл */
```

Если объект, соответствующий параметру `filename`, найден, то функция `_findfirst64i32()` записывает атрибуты объекта в структуру `_finddata64i32_t` и возвращает дескриптор. Если поиск окончился неудачей, то функция возвращает значение `-1` и присваивает глобальной переменной `errno` код ошибки.

Поиск остальных объектов производится в цикле с помощью функции `_findnext64i32()`. В первом параметре функция принимает дескриптор, который был получен с помощью функции `_findfirst64i32()`, а во втором — адрес структуры `_finddata64i32_t`. Если объект найден, то его атрибуты записываются в структуру и возвращается значение `0`. Если поиск окончился неудачей, то функция возвращает значение `-1` и присваивает глобальной переменной `errno` код ошибки.

Завершение поиска осуществляется с помощью функции `_findclose()`, которая в качестве параметра принимает дескриптор. Если ошибок не произошло, то функция возвращает значение `0`. В противном случае функция возвращает значение `-1` и присваивает переменной `errno` код ошибки.

Пример поиска всех файлов и подкаталогов показан в листинге 14.6.

Листинг 14.6. Перебор всех объектов, расположенных в каталоге

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

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   struct _finddata_t info;
   intptr_t h = _findfirst("C:\\book\\*", &info);
   if (h == -1) perror("Error");
   else {
      do {
         printf("%-30s", info.name);
         if (info.attrib & _A_SUBDIR)
            printf("%s", " _A_SUBDIR");
         if (info.attrib & _A_RDONLY)
            printf("%s", " _A_RDONLY");
         if (info.attrib & _A_HIDDEN)
            printf("%s", " _A_HIDDEN");
         if (info.attrib & _A_SYSTEM)
            printf("%s", " _A_SYSTEM");
         if (info.attrib & _A_ARCH)
            printf("%s", " _A_ARCH");
         printf("\n");
      } while(_findnext(h, &info) == 0);
      _findclose(h);
   }
   return 0;
}
```

Примерный результат выполнения:

```
.                              _A_SUBDIR
..                             _A_SUBDIR
file.txt                       _A_ARCH
file3.txt                      _A_RDONLY _A_HIDDEN _A_ARCH
folder1                        _A_SUBDIR
helloworld.c                   _A_ARCH
helloworld.exe                 _A_ARCH
script.bat                     _A_ARCH
test.c                         _A_ARCH
test.exe                       _A_ARCH
```

Обратите внимание на первые две строки. Одна точка обозначает текущий каталог. Две точки обозначают каталог выше уровнем, например, если поиск осуществляется в каталоге C:\book, то две точки указывают на корень диска C:. Если поиск производится в корневом каталоге диска, то этих двух строк не будет.

Помимо рассмотренных функций для поиска можно воспользоваться следующими функциями:

```
#include <io.h>
intptr_t _findfirst32(const char *filename,
                      struct _finddata32_t *findData);
int _findnext32(intptr_t findHandle, struct _finddata32_t *findData);
intptr_t _findfirst32i64(const char *filename,
                         struct _finddata32i64_t *findData);
int _findnext32i64(intptr_t findHandle,
                   struct _finddata32i64_t *findData);
intptr_t _findfirst64(const char *filename,
                      struct __finddata64_t *findData);
int _findnext64(intptr_t findHandle, struct __finddata64_t *findData);

#include <io.h> /* или #include <wchar.h> */
#define _wfindfirst _wfindfirst64i32
#define _wfindnext _wfindnext64i32
intptr_t _wfindfirst64i32(const wchar_t *filename,
                          struct _wfinddata64i32_t *findData);
int _wfindnext64i32(intptr_t findHandle,
                    struct _wfinddata64i32_t *findData);
intptr_t _wfindfirst32i64(const wchar_t *filename,
                          struct _wfinddata32i64_t *findData);
int _wfindnext32i64(intptr_t findHandle,
                    struct _wfinddata32i64_t *findData);
intptr_t _wfindfirst32(const wchar_t *filename,
                       struct _wfinddata32_t *findData);
int _wfindnext32(intptr_t findHandle,
                 struct _wfinddata32_t *findData);
intptr_t _wfindfirst64(const wchar_t *filename,
                       struct _wfinddata64_t *findData);
int _wfindnext64(intptr_t findHandle,
                 struct _wfinddata64_t *findData);
```

Структура `_wfinddata64i32_t` объявлена так:

```
struct _wfinddata64i32_t {
   unsigned attrib;
   __time64_t time_create;
   __time64_t time_access;
   __time64_t time_write;
   _fsize_t size;
   wchar_t name[260];
};
```

ГЛАВА 15

Потоки и процессы

В предыдущих главах мы познакомились с потоками ввода/вывода. В этой главе мы рассмотрим еще один вид потоков — *потоки управления*. Благодаря потокам управления можно выполнять различные задачи параллельно. Если процессор является одноядерным, то будет происходить переключение между потоками, имитируя параллельное выполнение.

Когда следует использовать потоки управления? Во-первых, при выполнении длительной операции. Например, получение данных из Интернета может блокировать основной поток, если сервер не отвечает. Во-вторых, когда необходимо ускорить выполнение операции. В этом случае операция разбивается на отдельные задачи, и эти задачи раздаются потокам. В-третьих, при использовании оконных приложений — операция не должна выполняться в потоке диспетчера обработки событий, иначе приложение не сможет выполнить перерисовку компонентов и перестанет реагировать на действия пользователя.

Потоки выполняются в рамках процесса и имеют общий доступ к его ресурсам, например к глобальным переменным. Процесс содержит минимум один поток, который создается при запуске приложения. В этом основном потоке выполняются инструкции, расположенные внутри тела функции `main()`. Помимо запуска задачи в отдельном потоке можно запустить задачу в отдельном процессе, который будет выполняться параллельно или заменит текущий процесс.

15.1. Потоки в WinAPI

В Windows нам доступны три способа создания потоков управления:

- с помощью функций из WinAPI;
- с помощью функций, объявленных в заголовочном файле process.h;
- с помощью функций, объявленных в заголовочном файле pthread.h.

15.1.1. Создание и завершение потока

Для создания потока управления в WinAPI предназначена функция `CreateThread()`. Прототип функции:

```
#include <windows.h>
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
                    SIZE_T dwStackSize,
                    LPTHREAD_START_ROUTINE lpStartAddress,
                    LPVOID lpParameter,
                    DWORD dwCreationFlags,
                    LPDWORD lpThreadId);
typedef void *HANDLE;
typedef void *LPVOID;
typedef unsigned __LONG32 DWORD;
#define __LONG32 long
typedef DWORD *LPDWORD;
```

Рассмотрим параметры функции `CreateThread()`:

- `lpThreadAttributes` — **указатель на структуру** `_SECURITY_ATTRIBUTES` **или значение** `NULL`, **если дескриптор не может быть унаследован. Объявление структуры:**

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength;
   LPVOID lpSecurityDescriptor;
   WINBOOL bInheritHandle;
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES,
  *LPSECURITY_ATTRIBUTES;
```

- `dwStackSize` — **размер стека или значение** `0`, **для размера по умолчанию;**
- `lpStartAddress` — **указатель на пользовательскую функцию, инструкции внутри которой будут выполнены в отдельном потоке:**

```
typedef DWORD (WINAPI *PTHREAD_START_ROUTINE) (
                                LPVOID lpThreadParameter);
typedef PTHREAD_START_ROUTINE LPTHREAD_START_ROUTINE;
```

Пример функции:

```
DWORD WINAPI thread_func(LPVOID param) {
   // Инструкции, выполняемые в отдельном потоке
   return 0;
}
```

- `lpParameter` — **задает данные, указатель на которые будет доступен через параметр** `param` **функции** `thread_func()`. **Если данные не передаются, то указывается значение** `NULL`;
- `dwCreationFlags` — **может принимать следующие основные значения (полный список см. в документации):** `0` — **поток запускается сразу после создания и** `CREATE_SUSPENDED` — **после создания поток приостанавливается до вызова функции** `ResumeThread()`:

```
#define CREATE_SUSPENDED 0x4
DWORD ResumeThread(HANDLE hThread);
```

- lpThreadId — указывается адрес переменной, в которой будет сохранен идентификатор созданного потока, или значение NULL.

Если поток успешно создан, то функция CreateThread() возвращает дескриптор созданного потока, а при ошибке — значение NULL. Получить информацию об ошибке можно с помощью функции GetLastError(). Прототип функции:

```
#include <windows.h>
DWORD GetLastError(VOID);
```

Поток завершает работу в следующих случаях.

- Функция thread_func() завершила свою работу (например, поток управления достиг оператора return). С помощью оператора return указывается статус завершения: значение 0 при успешном завершении и любое другое — при ошибке. Получить статус завершения потока позволяет функция GetExitCodeThread(). Прототип функции:

```
#include <windows.h>
WINBOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);
```

- Внутри функции thread_func() вызывается функция ExitThread(). В параметре указывается статус завершения. Прототип функции:

```
#include <windows.h>
VOID ExitThread(DWORD dwExitCode);
```

- В другом потоке вызвана функция TerminateThread(). В первом параметре функции указывается дескриптор потока, а во втором — статус завершения. Прототип функции:

```
#include <windows.h>
WINBOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

- Завершается процесс, например, поток управления достиг конца функции main().

После запуска потоков мы должны дождаться завершения работы потоков в основном потоке, иначе они прекратят работу на любой стадии выполнения при завершении процесса. Для этого предназначены следующие функции.

- WaitForSingleObject() — ожидает до тех пор, пока поток с дескриптором hHandle не будет завершен или не истечет время ожидания dwMilliseconds. Если время ожидания не ограничено, то в параметре dwMilliseconds задается значение макроса INFINITE. Прототип функции:

```
#include <windows.h>
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
#define INFINITE 0xffffffff
```

- WaitForMultipleObjects() — ожидает до тех пор, пока потоки из массива lpHandles не будут завершены или не истечет время ожидания dwMilliseconds. Если время ожидания не ограничено, то в параметре dwMilliseconds задается значение макроса INFINITE. Количество дескрипторов потоков в массиве

lpHandles указывается в параметре nCount. Если в параметре bWaitAll задано значение TRUE, то функция будет ожидать завершения всех потоков, а если FALSE — то одного. Прототип функции:

```
#include <windows.h>
DWORD WaitForMultipleObjects(DWORD nCount,
            CONST HANDLE *lpHandles, WINBOOL bWaitAll,
            DWORD dwMilliseconds);
```

После завершения работы потока нужно освободить ресурсы, вызвав функцию CloseHandle(). Прототип функции:

```
#include <windows.h>
WINBOOL CloseHandle(HANDLE hObject);
```

При работе с потоками могут быть полезными следующие функции:

- GetCurrentThreadId() — возвращает идентификатор текущего потока. Прототип функции:

  ```
  #include <windows.h>
  DWORD GetCurrentThreadId(VOID);
  ```

  Пример:

  ```
  DWORD thread_id = GetCurrentThreadId();
  printf("%lu\n", thread_id);
  ```

- GetCurrentThread() — возвращает дескриптор текущего потока. Прототип функции:

  ```
  #include <windows.h>
  HANDLE GetCurrentThread(VOID);
  ```

- Sleep() — прерывает выполнение текущего потока на указанное в параметре dwMilliseconds количество миллисекунд. Прототип функции:

  ```
  #include <windows.h>
  VOID Sleep(DWORD dwMilliseconds);
  ```

  Пример:

  ```
  Sleep(1000); // Засыпаем на секунду
  ```

В качестве примера запустим три потока, внутри которых просто будем выводить данные в окно консоли (листинг 15.1). Программу следует запускать в командной строке, иначе в окне **Console** редактора Eclipse отобразятся сообщения только о создании потоков и завершении программы.

Листинг 15.1. Создание потока

```
#include <stdio.h>
#include <locale.h>
#include <windows.h>
#include <strsafe.h>
```

```
#define NUM_THREADS 3
#define MSG_SIZE 256

DWORD WINAPI thread_func(LPVOID param) {
   // Инструкции, выполняемые в отдельном потоке
   DWORD thread_id = GetCurrentThreadId();
   char msg[MSG_SIZE] = {0};
   size_t msg_length;
   DWORD chars;

   // Получаем дескриптор потока вывода
   HANDLE hstdout = GetStdHandle(STD_OUTPUT_HANDLE);
   if (hstdout == INVALID_HANDLE_VALUE) return 1;

   for (int i = 0; i < 10; ++i) {
      Sleep(1000); // Имитация выполнения задачи
      // Выводим сообщение в консоль
      StringCchPrintfA(msg, MSG_SIZE, "%d %lu\n", i, thread_id);
      StringCchLengthA(msg, MSG_SIZE, &msg_length);
      WriteConsoleA(hstdout, msg, (DWORD)msg_length, &chars, NULL);
   }
   StringCchPrintfA(msg, MSG_SIZE, "Exit: %lu\n", thread_id);
   StringCchLengthA(msg, MSG_SIZE, &msg_length);
   WriteConsoleA(hstdout, msg, (DWORD)msg_length, &chars, NULL);
   return 0;
}

int main(void) {
   HANDLE arr_threads[NUM_THREADS];    // Массив с дескрипторами
   DWORD arr_threads_id[NUM_THREADS]; // Массив с идентификаторами

   setlocale(LC_ALL, "Russian_Russia.1251");
   DWORD main_id = GetCurrentThreadId();
   printf("Вход поток main: %lu\n", main_id);

   // Запускаем потоки
   for (int i = 0; i < NUM_THREADS; ++i) {
      arr_threads[i] = CreateThread(NULL, 0, thread_func, NULL, 0,
                                    &arr_threads_id[i]);
      if (!arr_threads[i]) {
         DWORD err = GetLastError();
         printf("Не удалось запустить поток. Error: %lu\n", err);
         return 1;
      }
      else {
         printf("Создан поток: %lu\n", arr_threads_id[i]);
         fflush(stdout);
      }
   }
```

```
   // Ожидаем завершения всех потоков
   WaitForMultipleObjects(NUM_THREADS, arr_threads, TRUE, INFINITE);

   // Закрываем все дескрипторы потоков
   for (int i = 0; i < NUM_THREADS; ++i) {
      CloseHandle(arr_threads[i]);
   }

   printf("Выход поток main: %lu\n", main_id);
   return 0;
}
```

ОБРАТИТЕ ВНИМАНИЕ

Многие функции из стандартной библиотеки языка C не являются потокобезопасными. Поэтому внутри функции `thread_func()` мы пользуемся потокобезопасными функциями, объявленными в заголовочном файле strsafe.h, а также функциями из WinAPI. При использовании внутри потоков функций из стандартной библиотеки языка C поведение программы может стать непредсказуемым, поэтому вместо функции `CreateThread()` лучше использовать функцию `_beginthreadex()` *(см. разд. 15.2)*.

15.1.2. Синхронизация потоков

Если несколько потоков пытаются получить доступ к одному ресурсу, то результат этого действия может стать непредсказуемым. Для того чтобы не допустить проблем и избежать состояния гонок, необходимо выполнять *синхронизацию* критичных секций (секций, к которым имеют доступ несколько потоков). При доступе к синхронизированной секции поток запрашивает блокировку. Если блокировка получена, то поток изменяет что-то внутри синхронизированного блока. Если не удалось получить блокировку, то поток блокируется до момента получения разрешения. Таким образом, код внутри критичной секции в один момент времени выполняется только одним потоком как атомарная операция.

Например, если мы из разных потоков попробуем вызвать функцию `print()`, содержащую следующий код:

```
void print(DWORD id) {
   printf("1. %lu\n", id);
   printf("2. %lu\n", id);
   printf("3. %lu\n", id);
   fflush(stdout);
}
```

то можем получить несколько проблем. Во-первых, мы используем функцию `printf()` из стандартной библиотеки языка C, что, как уже говорилось, может привести к непредсказуемым результатам. Во-вторых, тело пользовательской функции `print()` состоит из четырех инструкций. При выполнении этой функции первый поток может быть прерван на любой из этих инструкций, второй поток получит доступ, но опять его прервет третий поток, и т. д. В итоге никакой атомарности при

выполнении функции print() не будет, и результат ее выполнения становится непредсказуемым. А если внутри этой функции мы меняем значения глобальных переменных, то данные будут испорчены. Если попробуем выполнить функцию print() из разных потоков, то получим вот такую "кашу" в окне консоли:

```
1. 11172
1. 4436
2. 4436
3. 4436
2. 11172
3. 11172
```

Как видно из результата, поток с идентификатором 11172 был прерван после выполнения первой инструкции внутри функции print().

Для синхронизации критичных секций можно воспользоваться *мьютексами*. Создать объект мьютекса позволяют функции CreateMutexA() и CreateMutexW(). Прототипы функций:

```
#include <windows.h>
HANDLE CreateMutexA(LPSECURITY_ATTRIBUTES lpMutexAttributes,
                    WINBOOL bInitialOwner, LPCSTR lpName);
HANDLE CreateMutexW(LPSECURITY_ATTRIBUTES lpMutexAttributes,
                    WINBOOL bInitialOwner, LPCWSTR lpName);
```

Первый параметр принимает указатель на структуру _SECURITY_ATTRIBUTES или значение NULL, если дескриптор не может быть унаследован. Если во втором параметре указано значение TRUE, то текущий поток получит право владения мьютексом. В третьем параметре можно задать имя мьютекса или передать значение NULL. Функции возвращают дескриптор или значение NULL — в случае ошибки.

Перед входом в критичную секцию нужно вызвать функцию WaitForSingleObject() и передать ей в первом параметре дескриптор мьютекса, а во втором — время ожидания. Если время ожидания не ограничено, то в параметре dwMilliseconds задается значение макроса INFINITE. Прототип функции:

```
#include <windows.h>
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
#define INFINITE 0xffffffff
```

Получив блокировку, можно выполнять инструкции внутри критичной секции. После выхода из критичной секции нужно обязательно снять блокировку, вызвав функцию ReleaseMutex(), иначе секция будет заблокирована навсегда. Прототип функции:

```
#include <windows.h>
WINBOOL ReleaseMutex(HANDLE hMutex);
```

После завершения работы с мьютексом нужно освободить ресурсы, вызвав функцию CloseHandle(). Прототип функции:

```
#include <windows.h>
WINBOOL CloseHandle(HANDLE hObject);
```

Давайте рассмотрим синхронизацию потоков на примере (листинг 15.2).

Листинг 15.2. Синхронизация потоков (способ 1)

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

#define NUM_THREADS 3
HANDLE mutex_print;

void print(DWORD id) {
   // Ожидаем получения блокировки
   DWORD status = WaitForSingleObject(mutex_print, INFINITE);
   if (status == WAIT_OBJECT_0) {
      printf("1. %lu\n", id);
      printf("2. %lu\n", id);
      printf("3. %lu\n", id);
      fflush(stdout);
      ReleaseMutex(mutex_print); // Снимаем блокировку
   }
   else {
      // Здесь обрабатываем ошибки
   }
}

DWORD WINAPI thread_func(LPVOID param) {
   DWORD thread_id = GetCurrentThreadId();
   for (int i = 0; i < 10; ++i) {
      Sleep(1000);      // Имитация выполнения задачи
      print(thread_id); // Выводим сообщения в консоль
   }
   return 0;
}

int main(void) {
   HANDLE arr_threads[NUM_THREADS];   // Массив с дескрипторами

   mutex_print = CreateMutexA(NULL, FALSE, NULL);
   if (!mutex_print) {
      puts("Не удалось создать объект мьютекса");
      return 1;
   }

   setlocale(LC_ALL, "Russian_Russia.1251");
   DWORD main_id = GetCurrentThreadId();
   printf("Вход поток main: %lu\n", main_id);
```

```
    // Запускаем потоки
    for (int i = 0; i < NUM_THREADS; ++i) {
       arr_threads[i] = CreateThread(NULL, 0, thread_func, NULL, 0, NULL);
       if (!arr_threads[i]) return 1;
    }

    // Ожидаем завершения всех потоков
    WaitForMultipleObjects(NUM_THREADS, arr_threads, TRUE, INFINITE);

    // Закрываем все дескрипторы
    for (int i = 0; i < NUM_THREADS; ++i) {
       CloseHandle(arr_threads[i]);
    }
    CloseHandle(mutex_print);

    printf("Выход поток main: %lu\n", main_id);
    return 0;
}
```

Для синхронизации потоков можно также воспользоваться следующими функциями:

- `InitializeCriticalSection()` — **выполняет инициализацию. Прототип функции:**

```
#include <windows.h>
VOID InitializeCriticalSection(
                LPCRITICAL_SECTION lpCriticalSection);
```

- `EnterCriticalSection()` — **запрашивает блокировку и ожидает разрешение на вход в критичную секцию. Прототип функции:**

```
#include <windows.h>
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

- `TryEnterCriticalSection()` — **запрашивает блокировку без ожидания разрешения. Возвращает значение, отличное от нуля, если блокировка получена, и** `0` — **в противном случае. Прототип функции:**

```
#include <windows.h>
WINBOOL TryEnterCriticalSection(
                LPCRITICAL_SECTION lpCriticalSection);
```

- `LeaveCriticalSection()` — **снимает блокировку. Прототип функции:**

```
#include <windows.h>
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

- `DeleteCriticalSection()` — **удаляет объект. Прототип функции:**

```
#include <windows.h>
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Пример синхронизации потоков приведен в листинге 15.3.

Листинг 15.3. Синхронизация потоков (способ 2)

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

#define NUM_THREADS 3
CRITICAL_SECTION lock_print;

void print(DWORD id) {
   // Ожидаем получения блокировки
   EnterCriticalSection(&lock_print);
   printf("1. %lu\n", id);
   printf("2. %lu\n", id);
   printf("3. %lu\n", id);
   fflush(stdout);
   LeaveCriticalSection(&lock_print); // Снимаем блокировку
}

DWORD WINAPI thread_func(LPVOID param) {
   DWORD thread_id = GetCurrentThreadId();
   for (int i = 0; i < 10; ++i) {
      Sleep(1000);      // Имитация выполнения задачи
      print(thread_id); // Выводим сообщения в консоль
   }
   return 0;
}

int main(void) {
   HANDLE arr_threads[NUM_THREADS];   // Массив с дескрипторами

   InitializeCriticalSection(&lock_print);

   setlocale(LC_ALL, "Russian_Russia.1251");
   DWORD main_id = GetCurrentThreadId();
   printf("Вход поток main: %lu\n", main_id);

   // Запускаем потоки
   for (int i = 0; i < NUM_THREADS; ++i) {
      arr_threads[i] = CreateThread(NULL, 0, thread_func, NULL, 0, NULL);
      if (!arr_threads[i]) return 1;
   }

   // Ожидаем завершения всех потоков
   WaitForMultipleObjects(NUM_THREADS, arr_threads, TRUE, INFINITE);
```

```
    // Закрываем все дескрипторы
    for (int i = 0; i < NUM_THREADS; ++i) {
        CloseHandle(arr_threads[i]);
    }
    DeleteCriticalSection(&lock_print);

    printf("Выход поток main: %lu\n", main_id);
    return 0;
}
```

На заметку

Для синхронизации потоков помимо рассмотренных способов можно использовать и другие средства, например семафоры. За подробной информацией обращайтесь к документации.

15.2. Функции для работы с потоками, объявленные в файле process.h

Вместо функции `CreateThread()` из WinAPI для создания потока управления лучше использовать функцию `_beginthreadex()`, объявленную в заголовочном файле process.h, т. к. в этом случае внутри потока можно вызывать функции из стандартной библиотеки языка C. Прототип функции:

```
#include <process.h>
uintptr_t _beginthreadex(void *lpThreadAttributes,
          unsigned dwStackSize,
          unsigned (__stdcall *lpStartAddress) (void *),
          void *lpParameter, unsigned dwCreationFlags,
          unsigned *lpThreadId);
```

Все параметры аналогичны одноименным параметрам в функции `CreateThread()` *(см. разд. 15.1.1)*. Поток завершает работу при выходе из функции потока или при вызове функции `_endthreadex()`. В качестве параметра функция принимает код возврата. Прототип функции:

```
#include <process.h>
void _endthreadex(unsigned retval);
```

Переделаем код из листинга 15.1 и используем функцию `_beginthreadex()` вместо функции `CreateThread()` (листинг 15.4).

Листинг 15.4. Создание потоков с помощью функции `_beginthreadex()`

```
#include <stdio.h>
#include <locale.h>
#include <process.h>
#include <windows.h>
```

```
#define NUM_THREADS 3

unsigned thread_func(void *param) {
   DWORD thread_id = GetCurrentThreadId();
   for (int i = 0; i < 10; ++i) {
      Sleep(1000);      // Имитация выполнения задачи
      // Выводим сообщение в консоль
      printf("%d %lu\n", i, thread_id);
      fflush(stdout);
   }
   return 0;
}

int main(void) {
   HANDLE arr_threads[NUM_THREADS];   // Массив с дескрипторами

   setlocale(LC_ALL, "Russian_Russia.1251");
   DWORD main_id = GetCurrentThreadId();
   printf("Вход поток main: %lu\n", main_id);

   // Запускаем потоки
   for (int i = 0; i < NUM_THREADS; ++i) {
      arr_threads[i] = (HANDLE)_beginthreadex(NULL, 0, thread_func,
                                              NULL, 0, NULL);
      if (!arr_threads[i]) return 1;
   }

   // Ожидаем завершения всех потоков
   WaitForMultipleObjects(NUM_THREADS, arr_threads, TRUE, INFINITE);

   // Закрываем все дескрипторы
   for (int i = 0; i < NUM_THREADS; ++i) {
      CloseHandle(arr_threads[i]);
   }

   printf("Выход поток main: %lu\n", main_id);
   return 0;
}
```

Для создания потока можно также воспользоваться функцией `_beginthread()`. Прототип функции:

```
#include <process.h>
uintptr_t _beginthread(void (__cdecl *lpStartAddress) (void *),
                       unsigned dwStackSize, void *lpParameter);
```

В параметре `lpStartAddress` указывается адрес пользовательской функции, инструкции внутри которой будут выполняться в отдельном потоке. Обратите внима-

ние: функция потока ничего не возвращает. Для завершения работы потока можно вызвать функцию _endthread(), которая закроет дескриптор потока, поэтому использовать функцию CloseHandle() не нужно. Функция _endthread() вызывается автоматически при выходе из пользовательской функции потока. Прототип функции:

```
#include <process.h>
void _endthread(void);
```

В параметре dwStackSize задается размер стека или значение 0 для размера по умолчанию. Параметр lpParameter задает данные, указатель на которые будет доступен через параметр в функции потока. Если данные не передаются, то указываем значение NULL. Функция _beginthread() возвращает дескриптор потока или значение -1 в случае ошибки.

Пример использования функции _beginthread() приведен в листинге 15.5.

Листинг 15.5. Создание потоков с помощью функции _beginthread()

```
#include <stdio.h>
#include <locale.h>
#include <process.h>
#include <windows.h>

#define NUM_THREADS 3

void thread_func(void *param) {
   DWORD thread_id = GetCurrentThreadId();
   for (int i = 0; i < 10; ++i) {
      Sleep(1000);      // Имитация выполнения задачи
      // Выводим сообщение в консоль
      printf("%d %lu\n", i, thread_id);
      fflush(stdout);
   }
   printf("Выход поток: %lu\n", thread_id);
}

int main(void) {
   uintptr_t handle;

   setlocale(LC_ALL, "Russian_Russia.1251");
   DWORD main_id = GetCurrentThreadId();
   printf("Вход поток main: %lu\n", main_id);

   // Запускаем потоки
   for (int i = 0; i < NUM_THREADS; ++i) {
      handle = _beginthread(thread_func, 0, NULL);
      if (handle <= 0) return 1;
   }
```

```
    Sleep(15000); // Ожидаем завершения потоков

    printf("Выход поток main: %lu\n", main_id);
    return 0;
}
```

15.3. Потоки POSIX

Для создания потоков и управления ими можно воспользоваться функциями, объявленными в заголовочном файле pthread.h. В этом случае при компиляции внутри команды нужно указать флаги `-lpthread` и `-D_REENTRANT` или только флаг `-pthread`:

```
C:\book>gcc -Wall -Wconversion -O3 -finput-charset=cp1251
            -fexec-charset=cp1251 -lpthread -D_REENTRANT
            -o test.exe test.c
```

Компиляторы, которые мы установили в *главе 1*, собраны с различной поддержкой потоков: `win32` или `posix`. Для того чтобы увидеть поддерживаемую компилятором модель потоков, выполняем следующую команду:

```
C:\book>gcc -v
```

При использовании компилятора из каталога C:\msys64\mingw64\bin (этот компилятор мы используем по умолчанию) в результате выполнения команды можно найти следующие строки:

```
--enable-threads=posix
Thread model: posix
```

При использовании компилятора из каталога C:\MinGW64\mingw64\bin результат будет таким:

```
--enable-threads=win32
Thread model: win32
```

Если модель `win32`, то нужно при компиляции указать флаг `-pthread` или при возникновении ошибки установить компилятор с поддержкой модели `posix`:

```
C:\book>gcc -Wall -Wconversion -O3 -finput-charset=cp1251
            -fexec-charset=cp1251 -pthread -o test.exe test.c
```

15.3.1. Создание и завершение потока

Создать поток в модели `posix` позволяет функция `pthread_create()`. Прототип функции:

```
#include <pthread.h>
int pthread_create(pthread_t *th, const pthread_attr_t *attr,
                   void *(* func)(void *), void *arg);
typedef uintptr_t pthread_t;
typedef struct pthread_attr_t pthread_attr_t;
```

```
struct pthread_attr_t
{
    unsigned p_state;
    void *stack;
    size_t s_size;
    struct sched_param param;
};
```

Рассмотрим параметры функции `pthread_create()`:

- `th` — указывается адрес переменной, в которую будет записан дескриптор созданного потока;
- `attr` — задает атрибуты потока. Для использования атрибутов по умолчанию следует указать значение `NULL`;
- `func` — указатель на пользовательскую функцию, инструкции внутри которой будут выполнены в отдельном потоке. Пример функции:

```
void *thread_func(void *param) {
   // Инструкции, выполняемые в отдельном потоке
   return NULL;
}
```

- `arg` — задает данные, указатель на которые будет доступен через параметр `param` функции `thread_func()`. Если данные не передаются, то указываем значение `NULL`.

Если поток успешно создан, то функция `pthread_create()` возвращает значение `0`, в противном случае — код ошибки:

```
#include <errno.h>
#define EPERM 1
#define EAGAIN 11
#define EINVAL 22
```

Максимально возможное количество потоков содержится в макросе `PTHREAD_THREADS_MAX`:

```
#include <pthread.h>
#define PTHREAD_THREADS_MAX 2019
```

Поток завершает работу в следующих случаях.

- Функция `thread_func()` завершила свою работу (например, поток управления достиг оператора `return`). С помощью оператора `return` указывается статус завершения. Получить статус завершения можно посредством функции `pthread_join()`.
- Внутри функции `thread_func()` вызывается функция `pthread_exit()`. В параметре указывается статус завершения. Прототип функции:

```
#include <pthread.h>
void pthread_exit(void *res);
```

- В другом потоке вызвана функция `pthread_cancel()`. В параметре функции указывается дескриптор потока. Прототип функции:

```
#include <pthread.h>
int pthread_cancel(pthread_t th);
```

- Завершается процесс, например, поток управления достиг конца функции `main()`.

После запуска потоков мы должны дождаться завершения работы потоков в основном потоке, иначе они прекратят работу на любой стадии выполнения при завершении процесса. Дождаться завершения работы потока позволяет функция `pthread_join()`. В первом параметре указывается дескриптор потока, а через второй параметр доступно значение, возвращаемое функцией потока. Прототип функции:

```
#include <pthread.h>
int pthread_join(pthread_t t, void **res);
```

Если ошибок не было, то функция `pthread_join()` возвращает значение `0`, в противном случае — код ошибки:

```
#include <errno.h>
#define ESRCH 3
#define EINVAL 22
#define EDEADLK 36
```

Для того чтобы дождаться завершения работы потоков, можно в основном потоке (например, внутри функции `main()`) вызвать функцию `pthread_exit()`. В этом случае основной поток будет блокирован до момента завершения всех потоков.

Создадим три потока, внутри которых будем выводить данные в окно консоли (листинг 15.6).

Листинг 15.6. Создание потоков с помощью функции `pthread_create()`

```
#include <stdio.h>
#include <locale.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 3

void *thread_func(void *param) {
   int thread_id = *(int *)param;
   for (int i = 0; i < 10; ++i) {
      sleep(1);      // Имитация выполнения задачи
      // Выводим сообщение в консоль
      printf("i = %d поток: %d\n", i, thread_id);
      fflush(stdout);
   }
```

```
    printf("Выход поток: %d\n", thread_id);
    return NULL;
}

int main(void) {
    pthread_t arr_threads[NUM_THREADS];
    int arr_threads_id[NUM_THREADS];
    int status = 0;

    setlocale(LC_ALL, "Russian_Russia.1251");
    printf("Вход поток main\n");

    // Запускаем потоки
    for (int i = 0; i < NUM_THREADS; ++i) {
        arr_threads_id[i] = i + 1;
        status = pthread_create(&arr_threads[i], NULL, thread_func,
                                (void *)&arr_threads_id[i]);
        if (status != 0) {
            printf("Ошибка pthread_create(): %d\n", status);
            return 1;
        }
    }

    // Ожидаем завершения потоков
    for (int j = 0; j < NUM_THREADS; ++j) {
        status = pthread_join(arr_threads[j], NULL);
        if (status != 0) {
            printf("Ошибка pthread_join(): %d\n", status);
        }
    }

    printf("Выход поток main\n");
    return 0;
}
```

15.3.2. Синхронизация потоков

Для того чтобы избежать проблем при доступе к одному ресурсу, необходимо выполнить синхронизацию критичных секций (секций, к которым имеют доступ несколько потоков), например, с помощью мьютексов. Для этого предназначены следующие функции:

```
#include <pthread.h>
```

- `pthread_mutex_init()` — выполняет инициализацию мьютекса. В первом параметре указывается адрес переменной, а во втором параметре можно передать атрибуты или значение `NULL`. При успешном выполнении функция возвращает значение `0`, а при возникновении ошибки — код ошибки. Прототип функции:

```
int pthread_mutex_init(pthread_mutex_t *m,
                       const pthread_mutexattr_t *a);
```

- `pthread_mutex_destroy()` — удаляет объект мьютекса. При успешном выполнении функция возвращает значение `0`, а при возникновении ошибки — код ошибки. Прототип функции:

```
int pthread_mutex_destroy(pthread_mutex_t *m);
```

- `pthread_mutex_lock()` — запрашивает блокировку и ожидает разрешение на вход в критичную секцию. При получении блокировки функция возвращает значение `0`, а при возникновении ошибки — код ошибки. Прототип функции:

```
int pthread_mutex_lock(pthread_mutex_t *m);
```

- `pthread_mutex_trylock()` — запрашивает блокировку без ожидания разрешения. При получении блокировки функция возвращает значение `0`, а при возникновении ошибки — код ошибки. Прототип функции:

```
int pthread_mutex_trylock(pthread_mutex_t *m);
```

- `pthread_mutex_unlock()` — снимает блокировку. После выхода из критичной секции нужно обязательно снять блокировку, иначе секция будет заблокирована навсегда. При успешном выполнении функция возвращает значение `0`, а при возникновении ошибки — код ошибки. Прототип функции:

```
int pthread_mutex_unlock(pthread_mutex_t *m);
```

Пример синхронизации потоков приведен в листинге 15.7.

Листинг 15.7. Синхронизация потоков

```
#include <stdio.h>
#include <locale.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 3
pthread_mutex_t mutex_print;

void print(int id) {
   // Ожидаем получения блокировки
   if (pthread_mutex_lock(&mutex_print) == 0) {
      printf("1. %d\n", id);
      printf("2. %d\n", id);
      printf("3. %d\n", id);
      fflush(stdout);
      pthread_mutex_unlock(&mutex_print); // Снимаем блокировку
   }
   else {
      // Здесь обрабатываем ошибки
   }
}
```

```
void *thread_func(void *param) {
   int thread_id = *(int *)param;
   for (int i = 0; i < 10; ++i) {
      sleep(1);          // Имитация выполнения задачи
      print(thread_id); // Выводим сообщения в консоль
   }
   return NULL;
}

int main(void) {
   pthread_t arr_threads[NUM_THREADS];
   int arr_threads_id[NUM_THREADS];
   int status = 0;

   setlocale(LC_ALL, "Russian_Russia.1251");
   printf("Вход поток main\n");

   pthread_mutex_init(&mutex_print, NULL); // Инициализация мьютекса

   // Запускаем потоки
   for (int i = 0; i < NUM_THREADS; ++i) {
      arr_threads_id[i] = i + 1;
      status = pthread_create(&arr_threads[i], NULL, thread_func,
                              (void *)&arr_threads_id[i]);
      if (status != 0) {
         printf("Ошибка pthread_create(): %d\n", status);
         return 1;
      }
   }

   // Ожидаем завершения потоков
   for (int j = 0; j < NUM_THREADS; ++j) {
      status = pthread_join(arr_threads[j], NULL);
      if (status != 0) {
         printf("Ошибка pthread_join(): %d\n", status);
      }
   }
   pthread_mutex_destroy(&mutex_print); // Удаляем мьютекс

   printf("Выход поток main\n");
   return 0;
}
```

НА ЗАМЕТКУ

Для синхронизации потоков помимо мьютексов можно использовать и другие средства. За подробной информацией обращайтесь к документации.

15.4. Запуск процессов

Для запуска дочернего процесса предназначены следующие функции:

```
#include <process.h>
intptr_t _execl(const char *filename, const char *argList, ...);
intptr_t _wexecl(const wchar_t *filename, const wchar_t *argList, ...);
intptr_t _execle(const char *filename, const char *argList, ...);
intptr_t _wexecle(const wchar_t *filename, const wchar_t *argList, ...);
intptr_t _execlp(const char *filename, const char *argList, ...);
intptr_t _wexeclp(const wchar_t *filename, const wchar_t *argList, ...);
intptr_t _execlpe(const char *filename, const char *argList, ...);
intptr_t _wexeclpe(const wchar_t *filename, const wchar_t *argList, ...);
intptr_t _execv(const char *filename, const char *const *argList);
intptr_t _wexecv(const wchar_t *filename,
                 const wchar_t *const *argList);
intptr_t _execve(const char *filename, const char *const *argList,
                 const char *const *env);
intptr_t _wexecve(const wchar_t *filename,
                  const wchar_t *const *argList,
                  const wchar_t *const *env);
intptr_t _execvp(const char *filename, const char *const *argList);
intptr_t _wexecvp(const wchar_t *filename,
                  const wchar_t *const *argList);
intptr_t _execvpe(const char *filename, const char *const *argList,
                  const char *const *env);
intptr_t _wexecvpe(const wchar_t *filename,
                   const wchar_t *const *argList,
                   const wchar_t *const *env);
typedef __int64 intptr_t;
```

Буквы после префиксов _exec и _wexec имеют следующий смысл:

- p — файл filename будет искаться в путях, перечисленных в системной переменной PATH;
- l — аргументы командной строки передаются по отдельности. Первый аргумент должен содержать путь к исполняемому файлу. В последнем аргументе нужно передать значение NULL:

```
// Код программы test.exe приведен в листинге 17.3
_execl("C:\\book\\test.exe", "C:\\book\\test.exe",
       "-param1", "-param2", NULL);
// Сюда мы попадем только при ошибке
printf("%s\n", "Не удалось запустить процесс");
```

- e — передается массив указателей env на параметры среды (последний элемент массива должен иметь значение NULL):

```
const char *my_env[] = {
   "MYDIR=C:\\book\\",
```

```
    NULL
};
_execle("C:\\book\\test.exe", "C:\\book\\test.exe",
        "-param1", "-param2", NULL, my_env);
// Сюда мы попадем только при ошибке
printf("%s\n", "Не удалось запустить процесс");
```

- v — передается массив указателей `argList` на аргументы командной строки (последний элемент массива должен иметь значение `NULL`):

```
const char *my_args[4] = {
    "C:\\book\\test.exe",
    "-param1",
    "-param2",
    NULL
};
const char *my_env[] = {
    "MYDIR=C:\\book\\",
    NULL
};
_execve("C:\\book\\test.exe", my_args, my_env);
// Сюда мы попадем только при ошибке
printf("%s\n", "Не удалось запустить процесс");
```

Если процесс успешно запущен, то функции ничего не возвращают. Текущий процесс при этом будет замещен запущенным процессом. Инструкции после вызова этих функций выполняются только в случае ошибки при запуске процесса.

Следует учитывать, что строки, описывающие аргументы, не должны содержать пробелов. Если пробелы присутствуют, то нужно дополнительно заключить значение в кавычки:

```
setlocale(LC_ALL, "Russian_Russia.1251");
_wexecl(L"C:\\book\\test.exe", L"C:\\book\\test.exe",
        L"\"-param 1\"", L"\"-param 2\"", NULL);
// Сюда мы попадем только при ошибке
wprintf(L"%s\n", L"Не удалось запустить процесс");
```

Для запуска процесса можно также воспользоваться следующими функциями:

```
#include <process.h>
intptr_t _spawnl(int mode, const char *filename,
                 const char *argList, ...);
intptr_t _wspawnl(int mode, const wchar_t *filename,
                  const wchar_t *argList, ...);
intptr_t _spawnle(int mode, const char *filename,
                  const char *argList, ...);
intptr_t _wspawnle(int mode, const wchar_t *filename,
                   const wchar_t *argList, ...);
intptr_t _spawnlp(int mode, const char *filename,
                  const char *argList, ...);
```

```
intptr_t _wspawnlp(int mode, const wchar_t *filename,
                   const wchar_t *argList, ...);
intptr_t _spawnlpe(int mode, const char *filename,
                   const char *argList, ...);
intptr_t _wspawnlpe(int mode, const wchar_t *filename,
                    const wchar_t *argList, ...);
intptr_t _spawnv(int mode, const char *filename,
                 const char *const *argList);
intptr_t _wspawnv(int mode, const wchar_t *filename,
                  const wchar_t *const *argList);
intptr_t _spawnve(int mode, const char *filename,
                  const char *const *argList, const char *const *env);
intptr_t _wspawnve(int mode, const wchar_t *filename,
                   const wchar_t *const *argList,
                   const wchar_t *const *env);
intptr_t _spawnvp(int mode, const char *filename,
                  const char *const *argList);
intptr_t _wspawnvp(int mode, const wchar_t *filename,
                   const wchar_t *const *argList);
intptr_t _spawnvpe(int mode, const char *filename,
                   const char *const *argList,
                   const char *const *env);
intptr_t _wspawnvpe(int mode, const wchar_t *filename,
                    const wchar_t *const *argList,
                    const wchar_t *const *env);
typedef __int64 intptr_t;
```

В параметре `mode` **указываются следующие макросы:**

```
#include <process.h>
```

- `_P_OVERLAY` — **дочерний процесс перекрывает текущий процесс. Аналог вызова функций с префиксами** `_exec` **и** `_wexec`. **Определение макроса:**

  ```
  #define _P_OVERLAY 2
  ```

  Пример:

  ```
  // Код программы test.exe приведен в листинге 17.3
  _spawnl(_P_OVERLAY, "C:\\book\\test.exe", "C:\\book\\test.exe",
          "-param1", "-param2", NULL);
  // Сюда мы попадем только при ошибке
  printf("%s\n", "Не удалось запустить процесс");
  ```

- `_P_WAIT` — **текущий процесс будет ждать завершения нового процесса. Функции в этом случае возвращают код возврата нового процесса или значение** `-1` **в случае ошибки. Определение макроса:**

  ```
  #define _P_WAIT 0
  ```

Пример:

```
const char *my_args[4] = {
   "C:\\book\\test.exe",
   "-param1",
   "-param2",
   NULL
};
intptr_t st;
st = _spawnv(_P_WAIT, "C:\\book\\test.exe", my_args);
// Сюда мы попадем после завершения программы test.exe
printf("Код завершения: %I64d\n", st);
```

- **_P_NOWAIT и _P_NOWAITO — новый процесс будет выполняться параллельно с текущим процессом. Функции в этом случае возвращают дескриптор процесса. Определения макросов:**

```
#define _P_NOWAIT 1
#define _P_NOWAITO 3
```

Пример:

```
intptr_t d;
d = _spawnl(_P_NOWAIT, "C:\\book\\test.exe",
            "C:\\book\\test.exe", "-param1", "-param2", NULL);
// Сюда мы попадем не дожидаясь завершения программы test.exe
printf("Дескриптор процесса: %I64d\n", d);
```

- **_P_DETACH — новый процесс будет выполняться параллельно с текущим процессом в фоновом режиме без доступа к консоли и клавиатуре. Определение макроса:**

```
#define _P_DETACH 4
```

15.5. Получение идентификатора процесса

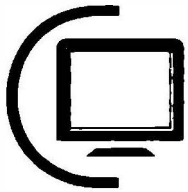
Получить идентификатор текущего процесса позволяет функция _getpid(). Прототип функции:

```
#include <process.h>
int _getpid(void);
```

Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
printf("Идентификатор текущего процесса: %d\n", _getpid());
```

ГЛАВА 16



Создание библиотек

В *разд. 10.2* мы научились размещать функции в отдельных файлах, а затем подключать их к основной программе. В больших программах функции принято объединять в библиотеки — *статические* (файлы с расширением lib в Visual C или с расширением a в MinGW) или динамические (файлы с расширением dll в Windows). Статические библиотеки становятся частью программы при компиляции, а динамические библиотеки подгружаются при запуске программы или при ее выполнении. Давайте научимся создавать библиотеки, а также пользоваться ими.

16.1. Статические библиотеки

При запуске программы из редактора Eclipse вы наверняка заметили, что процесс компиляции программы состоит из нескольких этапов. На первом этапе файл с исходным кодом преобразуется в объектный файл (файл с расширением o). Причем каждый файл с исходным кодом преобразуется отдельно. Этим достигается ускорение компиляции, т. к. нужно преобразовать лишь файлы, которые были изменены, а не все файлы с исходным кодом. На втором этапе на основе объектных файлов создается EXE-файл. Так вот на втором этапе вместо EXE-файла мы можем создать статическую библиотеку.

16.1.1. Создание статической библиотеки из командной строки

Вначале попробуем создать статическую библиотеку из командной строки. Для этого в каталоге C:\book\mylib создаем файлы module1.h (листинг 16.1), module1.c (листинг 16.2), module2.h (листинг 16.3) и module2.c (листинг 16.4).

Листинг 16.1. Содержимое файла C:\book\mylib\module1.h

```
#ifndef MODULE1_H_
#define MODULE1_H_
```

```
#ifdef __cplusplus
extern "C" {
#endif

int sum_int(int x, int y);

#ifdef __cplusplus
}
#endif

#endif /* MODULE1_H_ */
```

Листинг 16.2. Содержимое файла C:\book\mylib\module1.c

```
#include "module1.h"

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

Листинг 16.3. Содержимое файла C:\book\mylib\module2.h

```
#ifndef MODULE2_H_
#define MODULE2_H_

#ifdef __cplusplus
extern "C" {
#endif

double sum_double(double x, double y);

#ifdef __cplusplus
}
#endif

#endif /* MODULE2_H_ */
```

Листинг 16.4. Содержимое файла C:\book\mylib\module2.c

```
#include "module2.h"

double sum_double(double x, double y) {
   return x + y;
}
```

Таким образом, мы создали два модуля, содержащих по одной функции. Можно все функции объединить в один модуль, но в качестве примера компиляции нескольких файлов будем использовать два модуля. Обратите внимание на содержимое заголо-

вочных файлов: чтобы библиотеку можно было использовать в программах на языке C++, мы поместили объявления функций внутри следующего условия:

```
#ifdef __cplusplus
extern "C" {
#endif

// Объявления функций и т. д.

#ifdef __cplusplus
}
#endif
```

Теперь создадим объектные файлы:

```
C:\Users\Unicross>cd C:\book\mylib

C:\book\mylib>set Path=C:\msys64\mingw64\bin;%Path%

C:\book\mylib>gcc -Wall -Wconversion -c -finput-charset=cp1251
                  -fexec-charset=cp1251 -o module1.o module1.c

C:\book\mylib>gcc -Wall -Wconversion -c -finput-charset=cp1251
                  -fexec-charset=cp1251 -o module2.o module2.c
```

В этих командах мы указали флаг `-c`, чтобы создавались объектные файлы, а также флаг `-o`, для указания названий объектных файлов. Последние две команды можно объединить в одну команду:

```
C:\book\mylib>gcc -Wall -Wconversion -c -finput-charset=cp1251
                  -fexec-charset=cp1251 module1.c module2.c
```

В результате компиляции были созданы два файла: module1.o и module2.o. Эти два объектных файла можно объединить в статическую библиотеку с помощью следующей команды:

```
ar rcs lib<Название библиотеки>.a <Объектные файлы через пробел>
```

В этой команде мы воспользовались программой ar.exe, расположенной в каталоге C:\msys64\mingw64\bin. По существу программа создает архив, содержащий объектные файлы статической библиотеки. После названия программы указываются следующие флаги:

- `r` — замена существующих или вставка новых файлов в архив;
- `c` — создать новый архив, если нужно;
- `s` — создание индекса архива (действовать как программа ranlib.exe).

Далее указывается слово `lib`, после которого задается название статической библиотеки и расширение a. В конце команды через пробел перечисляются объектные файлы, добавляемые в архив. Давайте создадим статическую библиотеку с названием `modules_1_2`:

```
ar rcs libmodules_1_2.a module1.o module2.o
```

В результате в каталоге C:\book\mylib будет создан файл libmodules_1_2.a.

Создавать статическую библиотеку научились, теперь попробуем ее использовать внутри программы. Для этого в каталоге C:\book создаем файл test.c с кодом из листинга 16.5.

Листинг 16.5. Содержимое файла C:\book\test.c

```
#include <stdio.h>
#include <locale.h>
#include <module1.h>
#include <module2.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   printf("Функция sum_int(): %d\n", sum_int(5, 10));
   printf("Функция sum_double(): %.2f\n", sum_double(3.125, 5.6));
   return 0;
}
```

Скомпилируем и запустим программу:

```
C:\book\mylib>cd C:\book

C:\book>gcc -Wall -Wconversion -IC:\book\mylib -c
            -finput-charset=cp1251
            -fexec-charset=cp1251 test.c

C:\book>gcc test.o -LC:\book\mylib -lmodules_1_2 -o test.exe

C:\book>test.exe
Функция sum_int(): 15
Функция sum_double(): 8,72
```

В первой команде мы переходим в каталог C:\book. Вторая команда преобразует файл с исходным кодом в объектный файл. На этом этапе компилятору нужно знать сигнатуры методов, которые находятся в заголовочных файлах. Указать местоположение заголовочных файлов можно с помощью флага `-I`. В третьей команде выполняется сборка приложения. На этом этапе с помощью флага `-L` нужно передать местоположение статических библиотек, а после флага `-l` задать название подключаемой библиотеки. Обратите внимание: название библиотеки указывается без префикса `lib` и расширения файла. При использовании статических библиотек заголовочные файлы обычно размещают в каталоге include (например, C:\msys64\mingw64\include), а сами библиотеки — в каталоге lib (например, C:\msys64\mingw64\lib).

Последние две команды можно объединить в одну:

```
C:\book>gcc test.c -Wall -Wconversion -finput-charset=cp1251
            -fexec-charset=cp1251 -IC:\book\mylib
            -LC:\book\mylib -lmodules_1_2 -o test.exe
```

Если все библиотеки нужно компоновать статически, то в составе команды можно указать флаг `-static`:

```
C:\book>gcc test.c -Wall -Wconversion -finput-charset=cp1251
            -fexec-charset=cp1251 -static -IC:\book\mylib
            -LC:\book\mylib -lmodules_1_2 -o test.exe
```

Если библиотеки должны компоноваться по-разному, то можно воспользоваться флагами `-Wl,-Bdynamic` и `-Wl,-Bstatic`:

```
-Wl,-Bdynamic <динамические библиотеки через пробел>
-Wl,-Bstatic <статические библиотеки через пробел>
```

Используя флаг `-Wl,-Bstatic`, предыдущую команду можно записать так:

```
C:\book>gcc test.c -Wall -Wconversion -finput-charset=cp1251
            -fexec-charset=cp1251 -IC:\book\mylib -LC:\book\mylib
            -Wl,-Bstatic -lmodules_1_2 -o test.exe
```

16.1.2. Создание статической библиотеки в редакторе Eclipse

Теперь попробуем создать статическую библиотеку в редакторе Eclipse. В меню **File** выбираем пункт **New | Project**. В открывшемся окне (см. рис. 1.30) в списке выбираем пункт **C/C++ | C Project** и нажимаем кнопку **Next**. На следующем шаге (рис. 16.1) в поле **Project name** вводим `module_3`, в списке слева выбираем пункт **Static Library | Empty Project**, а в списке справа — пункт **MinGW GCC**. Для создания проекта нажимаем кнопку **Finish**. Проект статической библиотеки отобразится в окне **Project Explorer**.

Далее в корневой каталог проекта добавляем файл с исходным кодом **module3.c** (листинг 16.6) и заголовочный файл **module3.h** (листинг 16.7).

Листинг 16.6. Содержимое файла module3.c

```
#include "module3.h"

long sum_long(long x, long y) {
   return x + y;
}
```

Листинг 16.7. Содержимое файла module3.h

```
#ifndef MODULE3_H_
#define MODULE3_H_

#ifdef __cplusplus
extern "C" {
#endif
```

```
long sum_long(long x, long y);

#ifdef __cplusplus
}
#endif

#endif /* MODULE3_H_ */
```

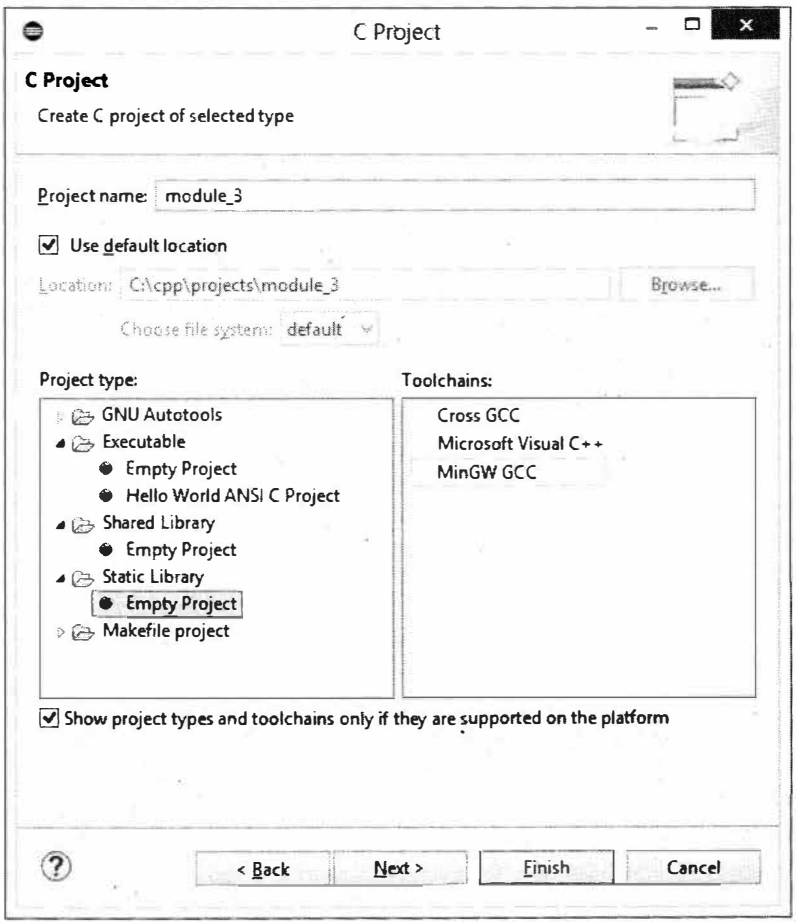


Рис. 16.1. Создание проекта

Открываем свойства проекта и на вкладке **C/C++ Build | Environment** проверяем настройки компилятора. Значение переменной окружения `MINGW_HOME` должно быть таким: `C:\msys64\mingw64`. Переходим на вкладку **C/C++ Build | Settings** и из списка **Configuration** выбираем пункт **All configurations**. На вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Miscellaneous**. В поле **Other flags** через пробел к имеющемуся значению добавляем следующие инструкции:

```
-finput-charset=cp1251 -fexec-charset=cp1251
```

Содержимое поля **Other flags** после изменения должно быть таким:

```
-c -fmessage-length=0 -finput-charset=cp1251 -fexec-charset=cp1251
```

Переходим на вкладку **GCC Archiver | General** и в поле **Archiver flags** вводим значение: `rcs`. На вкладке **Build Artifact** (рис. 16.2) можно настроить название библиотеки, префикс и расширение файла. Оставляем значения по умолчанию. Сохраняем настройки проекта.

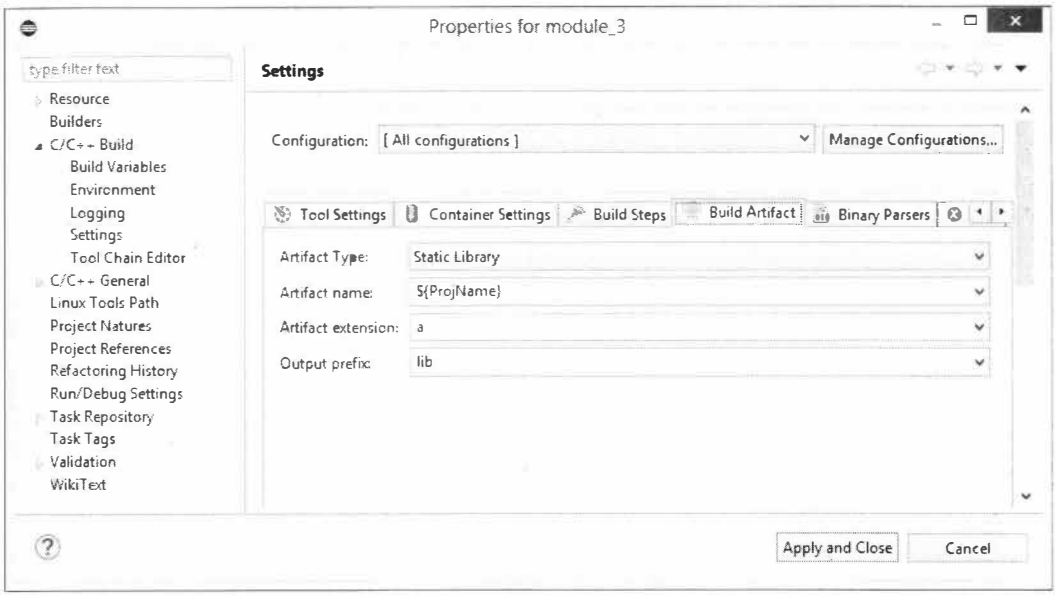


Рис. 16.2. Вкладка **Build Artifact**

Для создания статической библиотеки в меню **Project** выбираем пункт **Build Configurations | Set Active | Release**, а затем пункт **Build Project**. После компиляции в каталоге C:\cpp\projects\module_3\Release будет создан файл libmodule_3.a. В итоге мы имеем две статические библиотеки: libmodules_1_2.a (расположена в каталоге C:\book\mylib) и libmodule_3.a (расположена в каталоге C:\cpp\projects\module_3\Release). Давайте подключим их к проекту `Test64c`.

Открываем свойства проекта `Test64c` и из списка **Configuration** выбираем пункт **All configurations**. Из списка слева выбираем пункт **C/C++ Build | Settings**, а затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Includes** (рис. 16.3). В список **Include paths** добавляем пути к каталогам с заголовочными файлами: C:\book\mylib и C:\cpp\projects\module_3. Далее из списка выбираем пункт **MinGW C Linker | Libraries** (рис. 16.4). В список **Library search path** добавляем пути к каталогам со статическими библиотеками: C:\book\mylib и C:\cpp\projects\module_3\Release, а в список **Libraries** — библиотеки `modules_1_2` и `module_3`. Обратите внимание: название библиотеки указывается без префикса `lib` и расширения файла. Сохраняем настройки проекта.

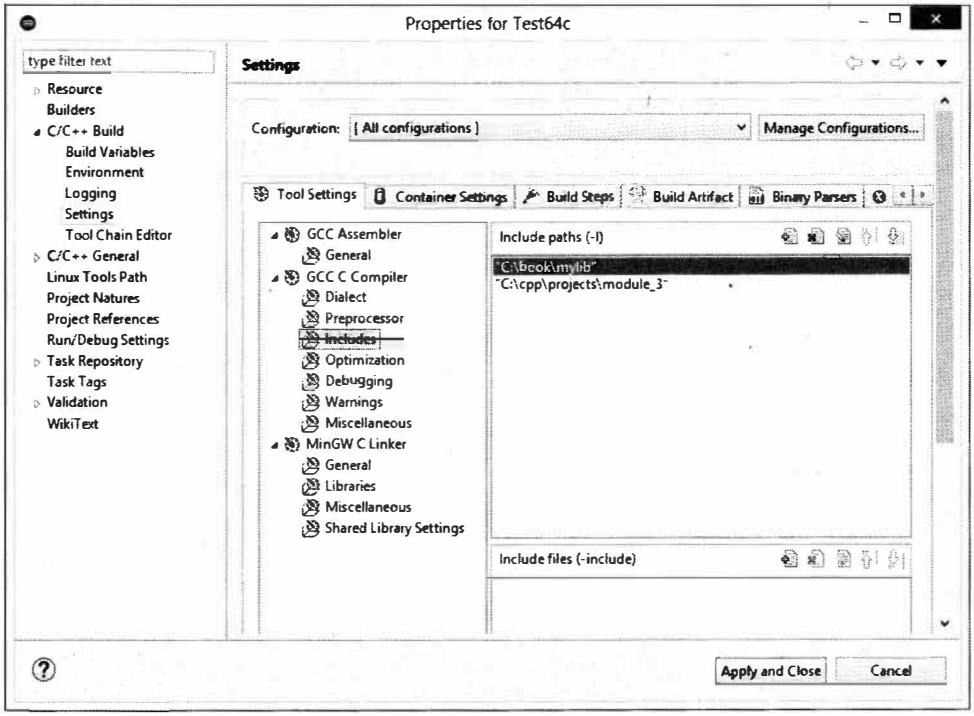


Рис. 16.3. Указание пути к заголовочным файлам

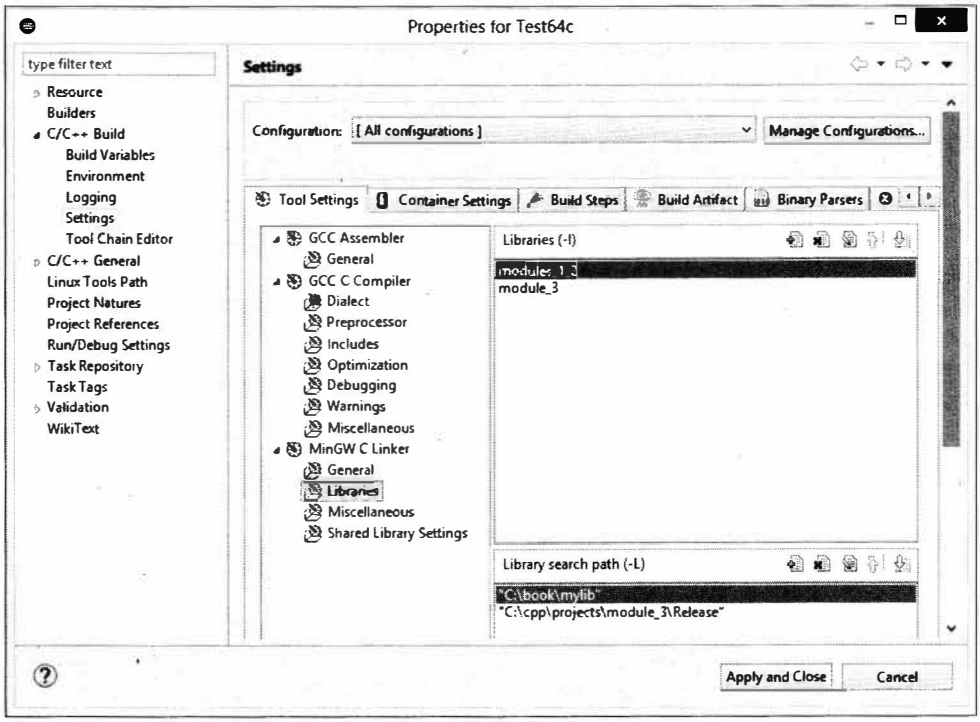


Рис. 16.4. Указание пути к библиотекам и названий библиотек

В файл Test64c.c добавляем код из листинга 16.8 и компилируем проект в режиме Release. В каталоге C:\cpp\projects\Test64c\Release будет создан файл Test64c.exe, который мы можем запустить из командной строки:

```
C:\book>C:\cpp\projects\Test64c\Release\Test64c.exe
Функция sum_int(): 15
Функция sum_double(): 8,72
Функция sum_long(): 11
```

Листинг 16.8. Содержимое файла Test64c.c

```
#include <stdio.h>
#include <locale.h>
#include <module1.h>
#include <module2.h>
#include <module3.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   printf("Функция sum_int(): %d\n", sum_int(5, 10));
   printf("Функция sum_double(): %.2f\n", sum_double(3.125, 5.6));
   printf("Функция sum_long(): %ld\n", sum_long(6L, 5L));
   return 0;
}
```

16.2. Динамические библиотеки

Статические библиотеки становятся частью программы при компиляции, а динамические библиотеки подгружаются при запуске программы. В итоге размер программы при использовании статических библиотек будет больше размера программы, использующей динамические библиотеки.

16.2.1. Создание динамической библиотеки из командной строки

Давайте рассмотрим процесс создания динамических библиотек из командной строки. В примере используем файлы module1.h (см. листинг 16.1), module1.c (см. листинг 16.2), module2.h (см. листинг 16.3) и module2.c (см. листинг 16.4), на основе которых мы создавали статическую библиотеку *(см. разд. 16.1.1)*. Запускаем командную строку и выполняем следующие команды:

```
C:\Users\Unicross>cd C:\book\mylib

C:\book\mylib>set Path=C:\msys64\mingw64\bin;%Path%

C:\book\mylib>gcc -Wall -Wconversion -fPIC -c -finput-charset=cp1251
                  -fexec-charset=cp1251 module1.c module2.c

C:\book\mylib>gcc -shared -o libmodules_1_2_0.dll module1.o module2.o
```

В первой команде мы делаем текущим каталог C:\book\mylib, а во второй — добавляем путь к компилятору в переменную окружения PATH. Третья команда создает объектные файлы на основе файлов с исходным кодом. Обратите внимание: мы дополнительно указали флаг -fPIC, который задает использование относительной адресации. Четвертая команда создает динамическую библиотеку. Во-первых, в составе команды указывается флаг -shared, а во-вторых, расширение файла dll, а не a. В результате в каталоге C:\book\mylib будет создан файл libmodules_1_2_0.dll.

Теперь скомпилируем файл C:\book\test.c с кодом из листинга 16.5.

```
C:\book\mylib>cd C:\book

C:\book>gcc -Wall -Wconversion -IC:\book\mylib -c
            -finput-charset=cp1251
            -fexec-charset=cp1251 test.c

C:\book>gcc test.o -o test.exe -LC:\book\mylib -lmodules_1_2_0
```

Последняя команда почти ничем не отличается от команды подключения статической библиотеки. Если в каталоге C:\book\mylib расположены одноименные статические и динамические библиотеки, то выбор библиотеки будет зависеть от используемого компилятора. Для того чтобы избежать проблем, лучше размещать статические и динамические библиотеки в каталогах с разными названиями (обычно lib и bin) или, как мы поступили в этом примере, давать библиотекам разные имена.

Если мы сейчас попробуем запустить программу test.exe из командной строки, то будет выведено сообщение о том, что библиотека libmodules_1_2_0.dll не найдена (рис. 16.5). Избежать этой ошибки в Windows можно, воспользовавшись следующими способами:

- поместить динамическую библиотеку рядом с EXE-файлом;
- скопировать динамическую библиотеку в системный каталог, например в каталог C:\Windows\System32 (однако лучше так не делать);
- поместить динамическую библиотеку в каталог, прописанный в системной переменной окружения PATH.

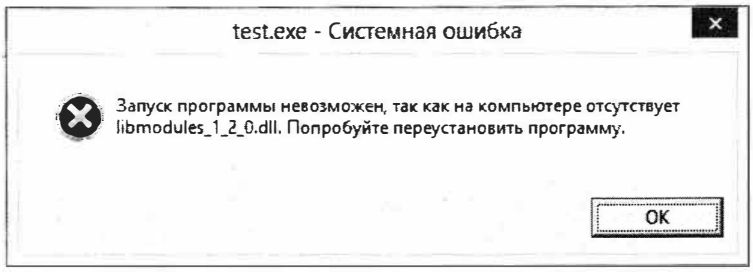


Рис. 16.5. Сообщение при отсутствии динамической библиотеки

В *разд. 1.1* мы создали каталог C:\cpp\lib, а в *разд. 1.2* добавили его в пути поиска, поэтому копируем библиотеку libmodules_1_2_0.dll в этот каталог и запускаем программу из командной строки:

```
C:\book>test.exe
Функция sum_int(): 15
Функция sum_double(): 8,72
```

16.2.2. Создание динамической библиотеки в редакторе Eclipse

Теперь попробуем создать динамическую библиотеку в редакторе Eclipse. В меню **File** выбираем пункт **New | Project**. В открывшемся окне (см. рис. 1.30) в списке выбираем пункт **C/C++ | C Project** и нажимаем кнопку **Next**. На следующем шаге (см. рис. 16.1) в поле **Project name** вводим `module_3_0`, в списке слева выбираем пункт **Shared Library | Empty Project**, а в списке справа — пункт **MinGW GCC**. Для создания проекта нажимаем кнопку **Finish**. Проект динамической библиотеки отобразится в окне **Project Explorer**.

Далее в корневой каталог проекта добавляем файл с исходным кодом module3.c (см. листинг 16.6) и заголовочный файл module3.h (см. листинг 16.7).

Открываем свойства проекта и на вкладке **C/C++ Build | Environment** проверяем настройки компилятора. Значение переменной окружения `MINGW_HOME` должно быть таким: `C:\msys64\mingw64`. Переходим на вкладку **C/C++ Build | Settings** и из списка **Configuration** выбираем пункт **All configurations**. На вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Miscellaneous**. В поле **Other flags** через пробел к имеющемуся значению добавляем следующие инструкции:

```
-finput-charset=cp1251 -fexec-charset=cp1251
```

Содержимое поля **Other flags** после изменения должно быть таким:

```
-c -fmessage-length=0 -finput-charset=cp1251 -fexec-charset=cp1251
```

Далее из списка выбираем пункт **MinGW C Linker | Shared Library Settings** и проверяем установку флажка **Shared (–shared)**. На вкладке **Build Artifact** (рис. 16.6) можно настроить название библиотеки, префикс и расширение файла. Оставляем значения по умолчанию. Сохраняем настройки проекта.

Для создания динамической библиотеки в меню **Project** выбираем пункт **Build Configurations | Set Active | Release**, а затем пункт **Build Project**. После компиляции в каталоге C:\cpp\projects\module_3_0\Release будет создан файл libmodule_3_0.dll. Для того чтобы не было проблем с запуском программы, копируем файл libmodule_3_0.dll в каталог C:\cpp\lib.

В итоге мы имеем две динамические библиотеки: libmodules_1_2_0.dll (в каталоге C:\book\mylib) и libmodule_3_0.dll (в каталоге C:\cpp\projects\module_3_0\Release). Копии этих библиотек должны быть расположены в каталоге C:\cpp\lib. Давайте подключим их к проекту `Test64c`.

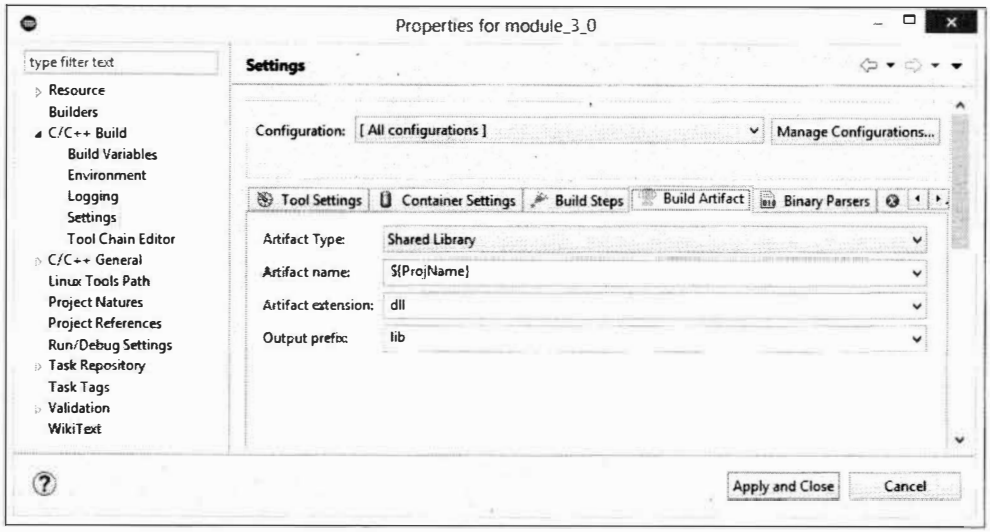


Рис. 16.6. Вкладка Build Artifact

Открываем свойства проекта `Test64c` и из списка **Configuration** выбираем пункт **All configurations**. Из списка слева выбираем пункт **C/C++ Build | Settings**, а затем на вкладке **Tool Settings** из списка выбираем пункт **GCC C Compiler | Includes** (рис. 16.7). В список **Include paths** добавляем пути к каталогам с заголовочными

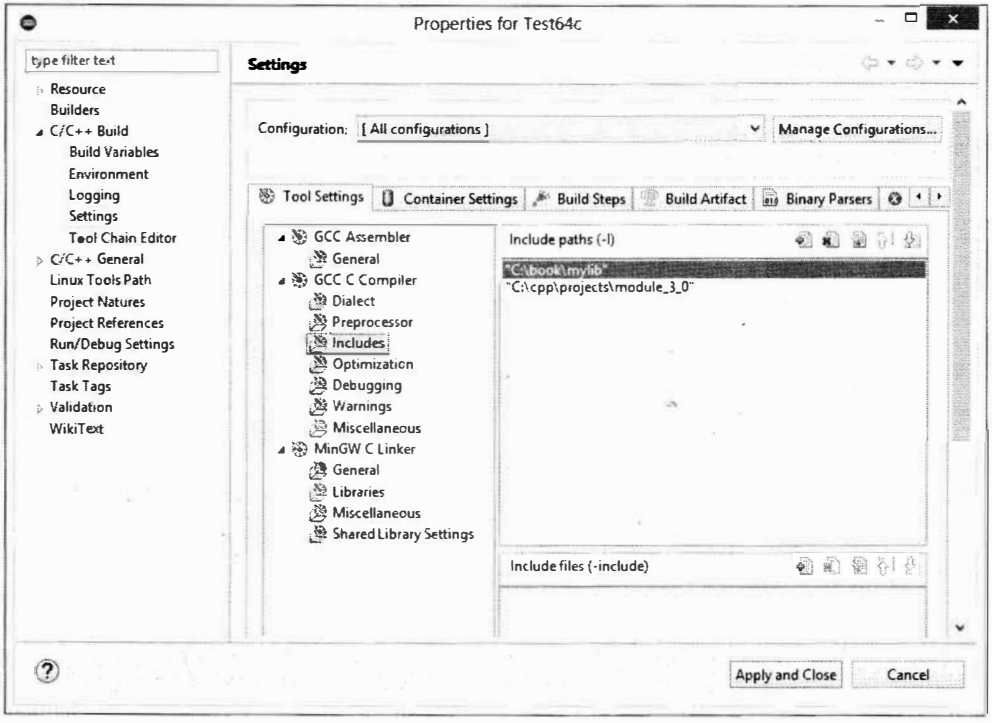


Рис. 16.7. Указание пути к заголовочным файлам

файлами: C:\book\mylib и C:\cpp\projects\module_3_0. Далее из списка выбираем пункт **MinGW C Linker | Libraries** (рис. 16.8). В список **Library search path** добавляем пути к каталогам с динамическими библиотеками: C:\book\mylib и C:\cpp\projects\module_3_0\Release, а в список **Libraries** — библиотеки `modules_1_2_0` и `module_3_C`. Обратите внимание: название библиотеки указывается без префикса `lib` и расширения файла. Сохраняем настройки проекта.

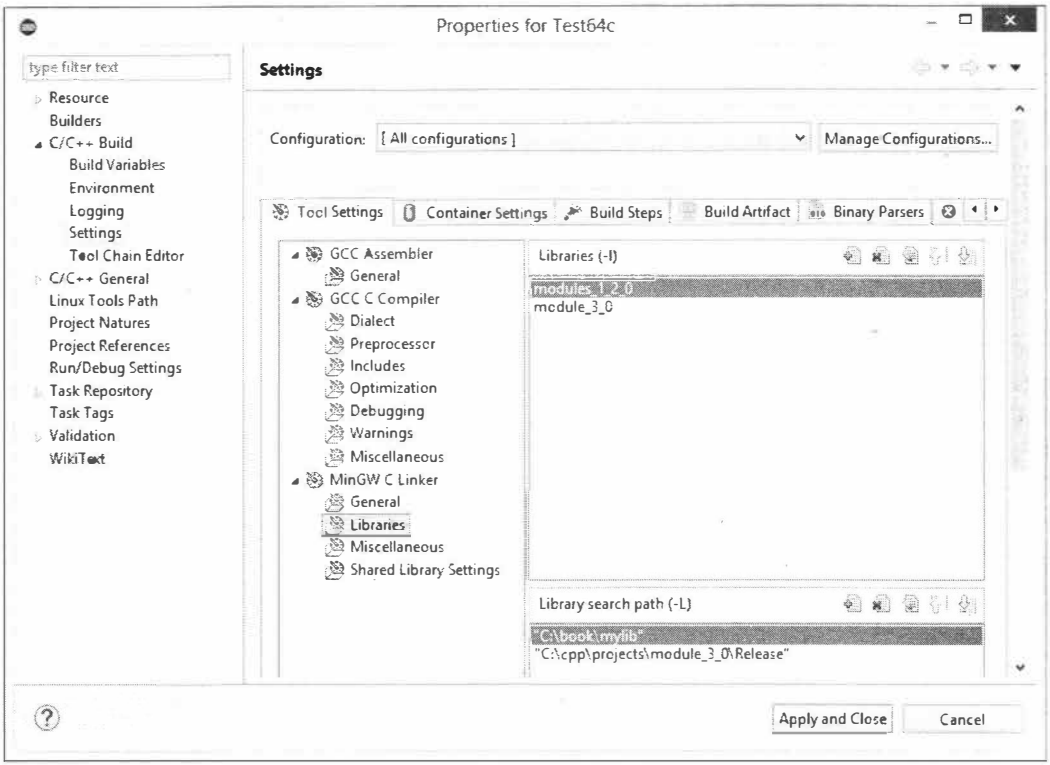


Рис. 16.8. Указание пути к библиотекам и названий библиотек

В файл Test64c.c добавляем код из листинга 16.8 и компилируем проект в режиме `Release`. В каталоге C:\cpp\projects\Test64c\Release будет создан файл Test64c.exe, который мы можем запустить из командной строки:

```
C:\book>C:\cpp\projects\Test64c\Release\Test64c.exe
Функция sum_int(): 15
Функция sum_double(): 8,72
Функция sum_long(): 11
```

Для того чтобы увидеть все динамические библиотеки, используемые программой, и полные пути к ним, следует запустить режим отладки и отобразить окно **Modules** (рис. 16.9), выбрав в меню **Window** пункт **Show View | Modules**.

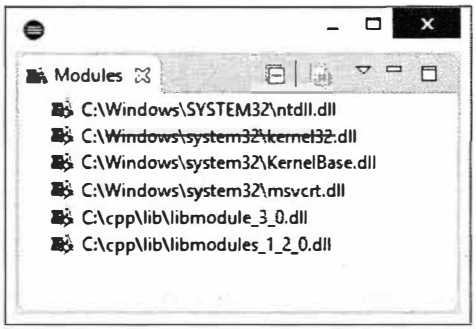


Рис. 16.9. Окно **Modules**

16.2.3. Загрузка динамической библиотеки во время выполнения программы

В предыдущих примерах динамические библиотеки загружаются при запуске программы. Если библиотека не найдена, то выводится сообщение об этом (см. рис. 16.5) и выполнение программы прекращается. В Windows существует возможность загрузки динамической библиотеки во время выполнения программы. Для этого предназначены функции `LoadLibraryA()` и `LoadLibraryW()` из WinAPI. Функции возвращают указатель на библиотеку или значение `NULL` в случае ошибки. Прототипы функций:

```
#include <windows.h>
HMODULE LoadLibraryA(LPCSTR lpLibFileName);
HMODULE LoadLibraryW(LPCWSTR lpLibFileName);
```

В параметре `lpLibFileName` можно указать просто название библиотеки (например, `libmodules_1_2_0`) или название с расширением (например, `libmodules_1_2_0.dll`). Перед названием можно задать абсолютный или относительный путь:

```
HINSTANCE hdll = LoadLibraryA("C:\\book\\mylib\\libmodules_1_2_0.dll");
```

Указать местоположение библиотек позволяют функции `SetDllDirectoryA()` и `SetDllDirectoryW()`. Прототипы функций:

```
#include <windows.h>
WINBOOL SetDllDirectoryA(LPCSTR lpPathName);
WINBOOL SetDllDirectoryW(LPCWSTR lpPathName);
```

Пример:

```
SetDllDirectoryA("C:\\book\\mylib");
HINSTANCE hdll = LoadLibraryA("libmodules_1_2_0.dll");
```

После завершения работы с библиотекой следует вызвать функцию `FreeLibrary()`. Прототип функции:

```
#include <windows.h>
WINBOOL FreeLibrary(HMODULE hLibModule);
```

Получить указатель на функцию, расположенную в загруженной библиотеке, позволяет функция `GetProcAddress()`. Если функции с названием `lpProcName` нет в библиотеке `hModule`, то возвращается значение `NULL`. Прототип функции:

```
#include <windows.h>
FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName);
```

Получить полный путь к загруженной библиотеке позволяют функции `GetModuleFileNameA()` и `GetModuleFileNameW()`. Прототипы функций:

```
#include <windows.h>
DWORD GetModuleFileNameA(HMODULE hModule, LPSTR lpFilename, DWORD nSize);
DWORD GetModuleFileNameW(HMODULE hModule, LPWSTR lpFilename, DWORD nSize);
```

Пример загрузки динамической библиотеки **libmodules_1_2_0.dll** при выполнении программы приведен в листинге 16.9. Предварительно в свойствах проекта удалите пути к заголовочным файлам, к библиотекам и названия подключаемых библиотек, т. к. эти данные не нужны при компиляции и компоновке программы.

Листинг 16.9. Содержимое файла C:\book\test.c

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

typedef int (*psum_int)(int, int);
typedef double (*psum_double)(double, double);

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   // Путь поиска библиотек
   SetDllDirectoryA(".\\mylib");
   // Загружаем библиотеку
   HINSTANCE hdll = LoadLibraryA("libmodules_1_2_0.dll");
   if (hdll == NULL) {
      puts("Библиотека не найдена");
      return 1;
   }
   // Получаем полный путь к библиотеке
   char buf[500] = {0};
   GetModuleFileNameA(hdll, buf, 499);
   printf("Путь: %s\n", buf);
   // Получаем указатели на функции из библиотеки
   psum_int sum_int = (psum_int)GetProcAddress(hdll, "sum_int");
   if (sum_int == NULL) {
      puts("Функция sum_int() не найдена");
      return 1;
   }
```

```
    psum_double sum_double =
                    (psum_double)GetProcAddress(hdll, "sum_double");
    if (sum_double == NULL) {
       puts("Функция sum_double() не найдена");
       return 1;
    }
    // Используем функции
    printf("Функция sum_int(): %d\n", sum_int(5, 10));
    printf("Функция sum_double(): %.2f\n", sum_double(3.125, 5.6));
    // Освобождаем ресурсы библиотеки
    FreeLibrary(hdll);
    return 0;
}
```

Скомпилируем программу в командной строке и запустим ее на выполнение:

```
C:\Users\Unicross>cd C:\book

C:\book>set Path=C:\msys64\mingw64\bin;%Path%

C:\book>gcc -Wall -Wconversion -O3 -finput-charset=cp1251
            -fexec-charset=cp1251 -o test.exe test.c

C:\book>test.exe
Путь: C:\book\mylib\libmodules_1_2_0.dll
Функция sum_int(): 15
Функция sum_double(): 8,72
```

В программе из листинга 16.9 мы указали относительный путь к библиотекам. Библиотека **libmodules_1_2_0.dll** должна быть расположена во вложенном каталоге **mylib**. Если библиотека не найдена, то в окне консоли получим следующее сообщение:

```
C:\book>test.exe
Библиотека не найдена
```

При использовании функций `LoadLibraryA()` и `LoadLibraryW()` поиск динамической библиотеки выполняется в следующем порядке:

- в каталоге с запускаемой программой;
- в каталоге, указанном с помощью функций `SetDllDirectoryA()` и `SetDllDirectoryW()`;
- в системном каталоге **Windows\System32**. Путь к этому каталогу можно получить с помощью функций `GetSystemDirectoryA()` и `GetSystemDirectoryW()`. Прототипы функций:

```
#include <windows.h>
UINT GetSystemDirectoryA(LPSTR lpBuffer, UINT uSize);
UINT GetSystemDirectoryW(LPWSTR lpBuffer, UINT uSize);
```

- в системном каталоге Windows\system;
- в системном каталоге Windows. Путь к этому каталогу можно получить с помощью функций `GetWindowsDirectoryA()` и `GetWindowsDirectoryW()`. Прототипы функций:

```
#include <windows.h>
UINT GetWindowsDirectoryA(LPSTR lpBuffer, UINT uSize);
UINT GetWindowsDirectoryW(LPWSTR lpBuffer, UINT uSize);
```

- в каталогах, прописанных в системной переменной окружения `PATH`.

16.2.4. Экспортируемые и внутренние функции

Экспортируемые функции могут использоваться внешними программами, а внутренние функции могут вызываться только из функций внутри динамической библиотеки. По умолчанию MinGW делает все функции экспортируемыми, однако если в программе перед определением функции явно указана инструкция `__declspec(dllexport)`, то экспортироваться будут только функции с этой инструкцией, а другие станут недоступными для внешнего использования. Например, если мы изменим код из листинга 16.2 следующим образом:

```
#include "module1.h"

__declspec(dllexport) int sum_int(int x, int y) {
   return x + y;
}
```

то функция `sum_int()` будет доступна, а вот функция `sum_double()` из листинга 16.4 станет недоступной.

16.2.5. Функция *DllMain()*

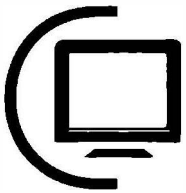
Динамические библиотеки могут иметь глобальные переменные, которые для каждого процесса создаются отдельно. Для инициализации этих переменных, а также для других целей, внутри библиотеки можно создать функцию с предопределенным названием `DllMain()`, которая является точкой входа в библиотеку. Функция вызывается при подключении библиотеки процессом или потоком, а также при отключении от нее. Определить причину вызова можно через второй параметр функции `DllMain()`. Помните, что внутри функции `DllMain()` можно выполнять далеко не все операции. Существует очень большой список ограничений, который можно найти в документации.

Пример реализации функции:

```
// #include <windows.h>
BOOL WINAPI DllMain(HINSTANCE hInst, DWORD dwReason, LPVOID lpRes) {
   switch (dwReason) {
      case DLL_PROCESS_ATTACH:
         // Загрузка DLL процессом
         break;
```

```
      case DLL_THREAD_ATTACH:
         // Загрузка DLL потоком
         break;
      case DLL_THREAD_DETACH:
         // Отсоединение от DLL потоком
         break;
      case DLL_PROCESS_DETACH:
         // Отсоединение от DLL процессом
         break;
      }
   return TRUE;
}
```

ГЛАВА 17



Прочее

В этой главе мы рассмотрим возможность запуска функции при завершении работы приложения, научимся выполнять системные команды и получать значения переменных окружения, изучим директивы препроцессора, а также создадим значок для EXE-файла.

17.1. Регистрация функции, выполняемой при завершении программы

Функции `atexit()` и `_onexit()` позволяют зарегистрировать функцию, которая будет вызвана при завершении программы. Внутри этой функции можно выполнить различные завершающие действия. Прототипы функций:

```
#include <stdlib.h>
int atexit(void (*func)(void));
_onexit_t _onexit(_onexit_t func);
typedef int (*_onexit_t)(void);
```

Функция `atexit()` возвращает значение `0`, если функция успешно зарегистрирована и ненулевое значение при ошибке. Функция `_onexit()` возвращает указатель на зарегистрированную функцию или значение `NULL` при ошибке.

Можно зарегистрировать сразу несколько функций. В этом случае они будут вызываться в обратном порядке: вначале будет вызвана последняя зарегистрированная функция, а последней — самая первая зарегистрированная функция (листинг 17.1).

Листинг 17.1. Регистрация функций, выполняемых при завершении программы

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

void func1(void) {
   puts("Выход из программы: func1");
}
```

```
void func2(void) {
   puts("Выход из программы: func2");
}
int func3(void) {
   puts("Выход из программы: func3");
   return 0;
}
int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   if (atexit(func1) != 0) {
      puts("Не удалось зарегистрировать функцию func1");
   }
   if (atexit(func2) != 0) {
      puts("Не удалось зарегистрировать функцию func2");
   }
   if ( !_onexit(func3) ) {
      puts("Не удалось зарегистрировать функцию func3");
   }
   printf("%s\n", "Текст, выводимый на консоль");
   return 0;
}
```

Результат в окне консоли:

```
Текст, выводимый на консоль
Выход из программы: func3
Выход из программы: func2
Выход из программы: func1
```

Зарегистрированные функции будут вызваны при нормальном завершении программы, а также при вызове функции `exit()`. **Если для досрочного завершения программы использовались функции** `_Exit()` **и** `_exit()`, **то зарегистрированные функции вызваны не будут. Прототипы функций:**

```
#include <stdlib.h>
void exit(int code);
void _Exit(int code);
void _exit(int code);
```

17.2. Выполнение системных команд

Функции `system()` **и** `_wsystem()` **позволяют выполнить системную команду** `command`. **Прототипы функций:**

```
#include <stdlib.h>
int system(const char *command);
int _wsystem(const wchar_t *command);
```

Если в качестве параметра указать значение NULL, то функция проверит наличие интерпретатора команд и вернет ненулевое значение, если интерпретатор найден, или значение 0 — в противном случае. Если задана команда, то при удачном выполнении команды функция вернет значение 0, в противном случае — значение -1. Код ошибки присваивается переменной errno.

Очистим окно консоли от команд с помощью функции system():

```
system("cls");
```

Зададим для консоли кодировку windows-1251 с помощью функции _wsystem():

```
_wsystem(L"chcp 1251");
```

Если нужно не просто выполнить команду, но и получить результат ее выполнения, то можно воспользоваться функциями _popen() и _wpopen(). Прототипы функций:

```
#include <stdio.h>
FILE *_popen(const char *command, const char *mode);
FILE *_wpopen(const wchar_t *command, const wchar_t *mode);
```

В первом параметре функции принимают команду в виде строки. Во втором параметре можно указать следующие режимы (или их комбинацию):

- r — чтение;
- w — запись;
- b — бинарный режим;
- t — текстовый режим.

Функции возвращают указатель на структуру FILE или нулевой указатель в случае ошибки. Используя этот указатель, можно читать и записывать данные с помощью функций для работы с файлами. После окончания работы нужно вызвать функцию _pclose(). Функция возвращает код завершения или значение -1 в случае ошибки. Прототип функции:

```
#include <stdio.h>
int _pclose(FILE *File);
```

Пример получения результата выполнения команды приведен в листинге 17.2.

Листинг 17.2. Получение результата выполнения системной команды

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <string.h>
#include <windows.h>

int main(void) {
   setlocale(LC_ALL, "Russian_Russia.1251");
   FILE *fp = _popen("dir", "rt");
```

```
   if (fp) {
      char buf[200] = {0};
      while (fgets(buf, 200, fp)) {
         if (ferror(fp)) {          // Проверяем наличие ошибок
            printf("Error: %s\n", strerror(ferror(fp)));
            exit(1);
         }
         // Преобразование windows-866 в windows-1251
         OemToCharA(buf, buf);
         printf("%s", buf);
      }
      printf("\n Код возврата: %d\n", _pclose(fp));
   }
   else puts("Ошибка");
   return 0;
}
```

17.3. Получение и изменение значений системных переменных

Все системные переменные доступны через третий параметр функции `main()`. Давайте сохраним код из листинга 17.3 в файле C:\book\test.c, скомпилируем его и запустим.

Листинг 17.3. Получение значений системных переменных

```
#include <stdio.h>

int main(int argc, char *argv[], char **penv) {
   printf("argc = %d\n", argc);
   for (int i = 0; i < argc; ++i) {
      printf("%s\n", argv[i]);
   }
   while (*penv != NULL) {
      printf("%s\n", *(penv++));
   }
   return 0;
}
```

Запускаем командную строку и компилируем программу:

```
C:\Users\Unicross>cd C:\book

C:\book>set Path=C:\msys64\mingw64\bin;%Path%

C:\book>gcc -Wall -Wconversion -O3 -o test.exe test.c
```

Для запуска программы вводим команду:

```
C:\book>test.exe -param1 -param2
```

В результате получим все аргументы, переданные в командной строке, а затем список всех системных переменных в формате переменная=значение:

```
ComSpec=C:\Windows\system32\cmd.exe
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
```

Для того чтобы получить значение какой-либо одной переменной по ее имени, можно воспользоваться следующими функциями.

- `getenv()` и `_wgetenv()` — возвращают указатель на значение или нулевой указатель в случае ошибки. Прототипы функций:

```
#include <stdlib.h>
char *getenv(const char *varName);
wchar_t *_wgetenv(const wchar_t *varName);
```

Пример получения значения системной переменной `PATH`:

```
setlocale(LC_ALL, "Russian_Russia.1251");
char *p = getenv("PATH");
if (p) printf("%s\n", p);
else puts("Не удалось получить значение");
```

Вместо функций `getenv()` и `_wgetenv()` можно использовать функции `getenv_s()` и `_wgetenv_s()`. Функция `getenv_s()` в MinGW недоступна. Прототипы функций:

```
#include <stdlib.h>
errno_t getenv_s(size_t *returnSize, char *destBuf,
            rsize_t destSize, const char *varName);
errno_t _wgetenv_s(size_t *returnSize, wchar_t *destBuf,
            size_t destSizeInWords, const wchar_t *varName);
```

Если в параметре `destBuf` указать значение `NULL`, то в переменную `returnSize` будет записан требуемый размер буфера. Если переменная `varName` не найдена, то переменная `returnSize` будет иметь значение `0`. В третьем параметре указывается размер буфера `destBuf`. Функции вернут значение `0`, если операция успешно произведена, или код ошибки, например, если размер буфера недостаточен или переменная не существует. Пример:

```
setlocale(LC_ALL, "Russian_Russia.1251");
wchar_t buf[256] = {0};
size_t returnSize = 0;
errno_t st = _wgetenv_s(&returnSize, buf, 256, L"PATHEXT");
if (st == 0) {
   wprintf(L"%I64u\n", returnSize);
   wprintf(L"%s\n", buf);
}
else _putws(L"Не удалось получить значение");
```

- `_dupenv_s()` и `_wdupenv_s()` — получают значение переменной `varName` и сохраняют его в динамической памяти. После завершения работы с переменной нужно освободить память с помощью функции `free()`. В MinGW функции не работают. Прототипы функций:

```
#include <stdlib.h>
errno_t _dupenv_s(char **pBuf, size_t *pBufSizeInBytes,
                  const char *varName);
errno_t _wdupenv_s(wchar_t **pBuf, size_t *pBufSizeInWords,
                   const wchar_t *varName);
```

Пример (в проектах `Test32c` и `Test64c` пример не работает):

```
setlocale(LC_ALL, "Russian_Russia.1251");
char *p = NULL;
size_t bufSize = 0;
errno_t st = _dupenv_s(&p, &bufSize, "PATH");
if (st == 0) {
   printf("%I64u\n", bufSize);
   printf("%s\n", p);
   free(p);
}
else puts("Не удалось получить значение");
```

Добавить новую переменную для текущего процесса, изменить значение существующей переменной или удалить переменную позволяют функции `_putenv()` и `_wputenv()`. Прототипы функций:

```
#include <stdlib.h>
int _putenv(const char *envString);
int _wputenv(const wchar_t *envString);
```

В качестве параметра указывается строка, имеющая формат `переменная=значение`. Если указать значение `переменная=`, то переменная будет удалена. Функции возвращают значение `0`, если операция успешно выполнена, или значение `-1` — в случае ошибки. Пример:

```
_putenv("MY_VAR=10");
char *p = getenv("MY_VAR");
if (p) printf("%s\n", p); // 10
```

Можно также воспользоваться функциями `_putenv_s()` и `_wputenv_s()`. Прототипы функций:

```
#include <stdlib.h>
errno_t _putenv_s(const char *name, const char *value);
errno_t _wputenv_s(const wchar_t *name, const wchar_t *value);
```

В первом параметре задается название переменной, а во втором — ее значение. Для того чтобы удалить переменную, нужно во втором параметре передать пустую

строку. Функции возвращают значение 0, если операция успешно выполнена, или код ошибки. Пример:

```
_putenv_s("MY_VAR", "10");
char *p = getenv("MY_VAR");
if (p) printf("%s\n", p); // 10
```

17.4. Директивы препроцессора

Перед компиляцией запускается специальная программа — *препроцессор*, которая подготавливает код к компиляции. В исходном коде мы можем использовать следующие *директивы препроцессора*:

- `#include` — позволяет вставить все содержимое указанного файла на место директивы (директиву мы уже рассматривали в *разд. 2.4*):

```
#include <stdio.h>
#include "HelloWorld.h"
#include "C:\\cpp\\projects\\HelloWorld\\src\\HelloWorld.h"
```

- `#define` — создает константу *(см. разд. 3.8)* или встраиваемую функцию *(см. разд. 10.12)*. Название, указанное в директиве `#define`, принято называть *макроопределением* или *макросом*. Пример определения макросов:

```
#define _MAX_PATH 260
#define __max(a,b) (((a) > (b)) ? (a) : (b))
```

- `#undef` — удаляет макрос:

```
#undef MY_CONST
```

- `#if` — проверяет условие. Формат директивы:

```
#if <Условие 1>
<Если условие 1 истинно>
#elif <Условие 2>
<Если условие 2 истинно>
#elif <Условие N>
<Если условие N истинно>
#else
<Если все условия ложны>
#endif
```

Пример:

```
#define MY_CONST 3
#if MY_CONST == 1
puts("MY_CONST == 1");
#elif MY_CONST >= 2
puts("MY_CONST >= 2");
#else
puts("Блок else");
#endif
```

После обработки препроцессор оставит только одну инструкцию (остальные инструкции будут удалены):

```
puts("MY_CONST >= 2");
```

- `#ifdef` — проверяет наличие макроса с указанным именем. Формат директивы:

```
#ifdef <Макрос>
<Если макрос определен>
#else
<Если макрос не определен>
#endif
```

Пример:

```
#ifdef _POSIX_
#define _P_tmpdir "/"
#else
#define _P_tmpdir "\\"
#endif
```

- `#ifndef` — проверяет отсутствие макроса с указанным именем. Формат директивы:

```
#ifndef <Макрос>
<Если макрос не определен>
#else
<Если макрос определен>
#endif
```

- `#pragma` — задает действие, зависящее от реализации компилятора. Например, указать, что заголовочный файл должен быть включен только один раз, можно так:

```
#pragma once
```

Включить вывод предупреждающих сообщений при использовании Visual C можно так:

```
#pragma warning( push, 4)
```

Добавить библиотеку User32.lib при использовании Visual C позволяет такая директива:

```
#pragma comment(lib, "User32.lib")
```

- `#warning` — выводит предупреждающее сообщение;
- `#error` — выводит сообщение об ошибке.

17.5. Создание значка приложения

По умолчанию для EXE-файла используется стандартный значок. В Windows мы можем добавить собственный значок, внедрив его в EXE-файл как ресурс. Попробуем создать значок для приложения. Вначале в каталог C:\book добавляем файл со

значком, например my_icon.ico. Затем создаем текстовый файл с названием resources.rc и вставляем в него следующую инструкцию:

```
001 ICON "my_icon.ico"
```

Запускаем командную строку и переходим в каталог C:\book:

```
C:\Users\Unicross>cd C:\book
```

Настраиваем переменную окружения PATH:

```
C:\book>set Path=C:\msys64\mingw64\bin;%Path%
```

Компилируем файл ресурсов с помощью программы windres.exe:

```
C:\book>windres --use-temp-file -iresources.rc -oresources.o
```

В результате в каталоге C:\book будет создан файл resources.o, который мы должны передать компилятору при сборке программы. Для этого в редакторе Eclipse открываем свойства проекта (в меню **Project** выбираем пункт **Properties**) и из списка слева выбираем пункт **C/C++ Build | Settings**. На отобразившейся вкладке из списка **Configuration** выбираем пункт **Release**, а затем на вкладке **Tool Settings** выбираем пункт **MinGW C Linker | Miscellaneous**. В список **Other objects** (рис. 17.1) добавляем следующее значение: `C:\\book\\resources.o`. Сохраняем настройки проекта, нажимая кнопку **Apply and Close**.

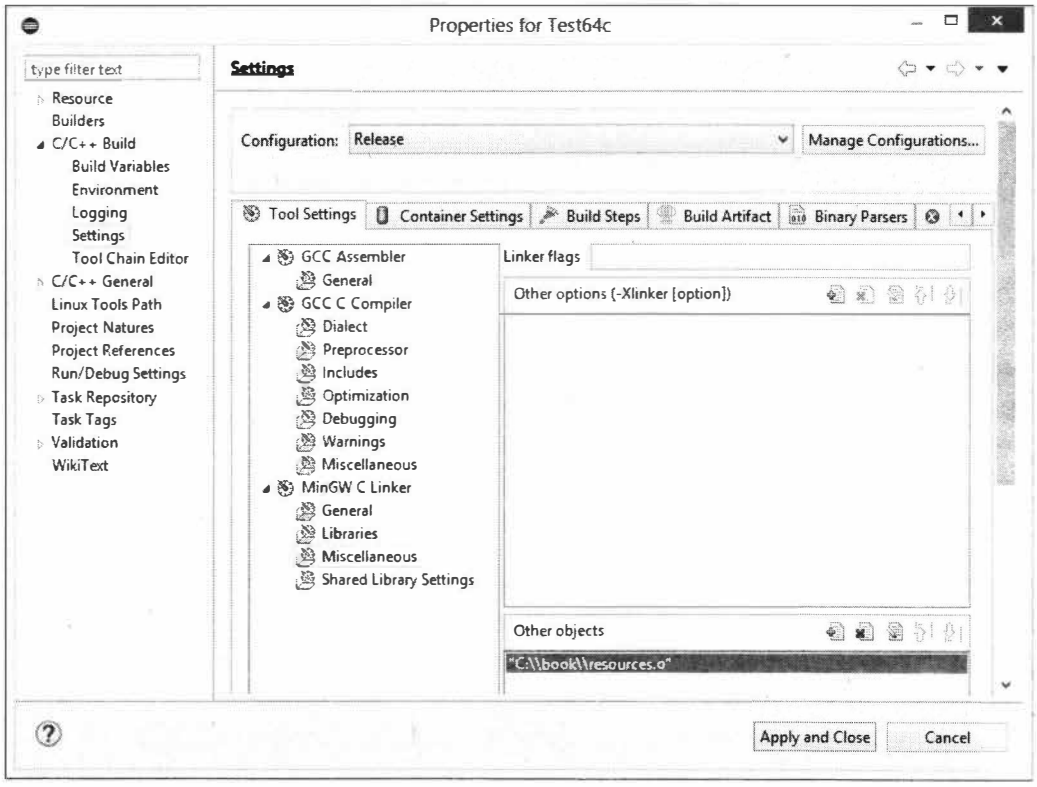


Рис. 17.1. Указание местоположения файла ресурсов

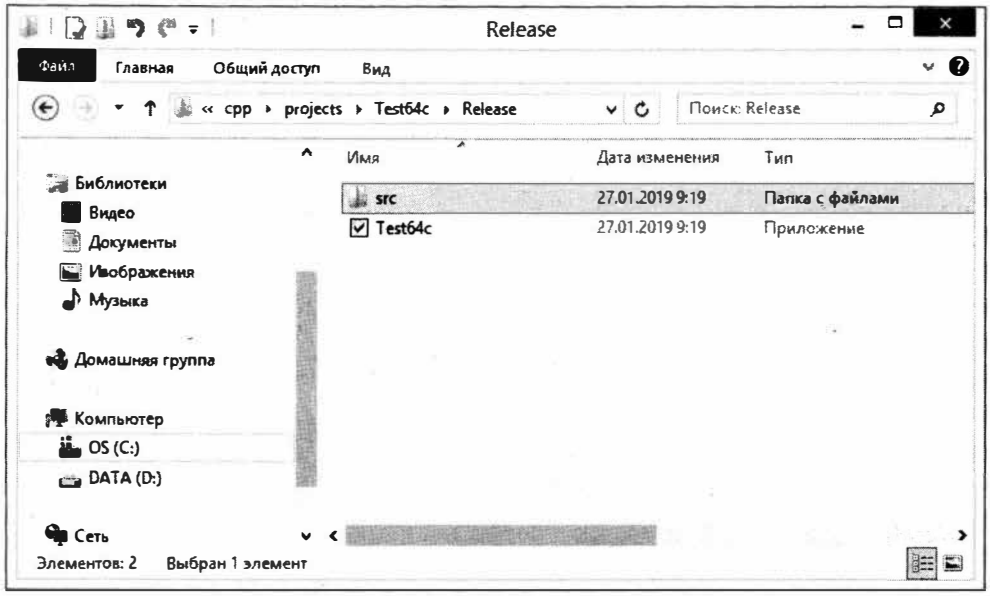


Рис. 17.2. Пользовательский значок слева от названия приложения

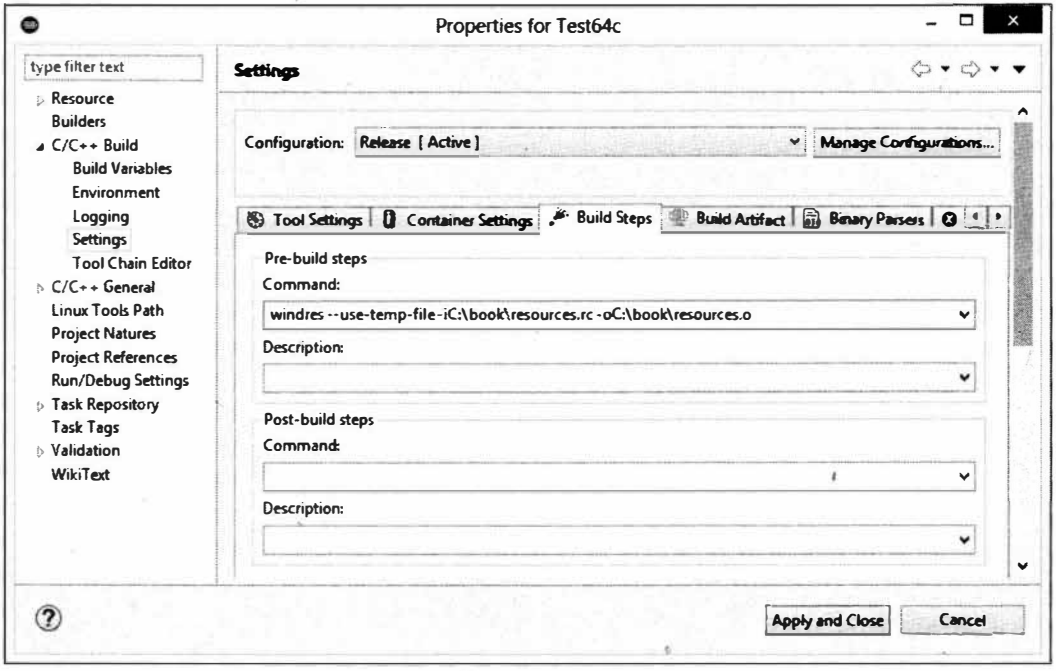


Рис. 17.3. Добавление команды для компиляции файла ресурсов

Осталось скомпилировать программу в режиме `Release`. В окне **Console** редактора Eclipse должна быть инструкция:

```
gcc -o Test64c.exe "src\\Test64c.o" "C:\\book\\resources.o"
```

Заходим в каталог **C:\cpp\projects\Test64c\Release**. Слева от названия исполняемого файла должен быть виден значок (на рис. 17.2 значок отображается в виде установленного флажка).

Если вы хотите, чтобы файл ресурсов компилировался каждый раз перед компиляцией программы, то на вкладке **Build Steps** в разделе **Pre-build steps** в поле **Command** (рис. 17.3) нужно ввести следующую инструкцию:

```
windres --use-temp-file -iC:\book\resources.rc -oC:\book\resources.o
```

В окне **Console** редактора Eclipse перед командами компиляции должна быть команда:

```
windres --use-temp-file "-iC:\\book\\resources.rc"
"-oC:\\book\\resources.o"
```

Заключение

Вот и закончилось наше путешествие в мир языка С. Материал книги описывает лишь основы языка. А здесь мы уточним, где найти дополнительную информацию.

Самым важным источником информации является официальный сайт библиотеки MinGW: **http://www.mingw.org/**. На этом сайте вы найдете новости, а также ссылки на все другие ресурсы в Интернете, посвященные MinGW. Не следует также забывать о существовании страницы **https://sourceforge.net/projects/mingw-w64/**, на которой доступна для скачивания самая последняя версия библиотеки MinGW-W64.

На странице **https://gcc.gnu.org/** расположена документация по GCC, которая обновляется в режиме реального времени. Библиотека постоянно совершенствуется, появляются новые функции, изменяются параметры функций и т. д. Регулярно посещайте эту страницу, и вы будете в курсе самых свежих новшеств.

На странице **https://ru.wikipedia.org/wiki/Си_(язык_программирования)** вы найдете историю развития языка С, а также действующие ссылки на все другие ресурсы в Интернете, посвященные этому языку.

На языке С можно создавать оконные приложения с помощью библиотеки GTK+. Подробную информацию смотрите на странице: **https://www.gtk.org/**.

Если в процессе изучения языка С у вас возникнут какие-либо недопонимания, то не следует забывать, что Интернет предоставляет множество ответов на самые разнообразные вопросы. Достаточно в строке запроса поискового портала (например, **http://www.google.com/**) набрать свой вопрос. Наверняка уже кто-то сталкивался с подобной проблемой и описал ее решение на каком-либо сайте.

Свои замечания и пожелания вы можете оставить на странице книги на сайте издательства "БХВ-Петербург": **http://www.bhv.ru/**. Все замеченные опечатки и неточности прошу присылать на e-mail: **mail@bhv.ru** — не забудьте только указать название книги и имя автора.

ПРИЛОЖЕНИЕ

Описание электронного архива

По ссылке **ftp://ftp.bhv.ru/9785977541169.zip** можно скачать электронный архив с исходными кодами рассмотренных примеров к книге. Ссылка доступна также со страницы книги на сайте **www.bhv.ru**.

Структура архива представлена в табл. П1.

Таблица П1. *Структура электронного архива*

Файл	Описание
Listings.doc	Содержит все пронумерованные листинги из книги, а также некоторые полезные фрагменты кода
Readme.txt	Описание электронного архива

Предметный указатель

#

#define 51, 63, 101, 102, 187, 317, 334, 344, 454
#elif 454
#else 455
#endif 51, 63, 317, 454, 455
#error 455
#if 454
#ifdef 455
#ifndef 51, 63, 317, 455
#include 45, 51, 62, 316, 318, 454
#pragma 51, 63, 455
#undef 103, 454
#warning 455

_

__DATE__ 103
__declspec(dllexport) 446
__FILE__ 103
__inline 333
__int16 97
__int32 97
__int64 97
__int8 97
__LINE__ 103
__max() 163
__min() 163
__mingw_printf() 69, 70, 100
__STDC_VERSION__ 9
__TIME__ 103
__time32_t 300
__time64_t 300
__VERSION__ 9
__wcserror() 340
__wcserror_s() 340
_A_ARCH 404
_A_HIDDEN 404
_A_NORMAL 404
_A_RDONLY 404
_A_SUBDIR 404
_A_SYSTEM 404
_abs64() 160
_access() 391
_access_s() 392
_atof_l() 171
_atoi_l() 165
_atoi64() 167
_atol_l() 166
_beginthread() 418
_beginthreadex() 417
_Bool 93, 156
_chdir() 402
_chdrive() 399
_chmod() 393
_close() 377
_commit() 378
_countof() 183
_create_locale() 214
_CRT_SECURE_NO_WARNINGS 232, 277
_ctime64() 306
_ctime64_s() 307
_difftime64() 304
_dup() 384
_dup2() 371, 384
_dupenv_s() 453
_endthread() 419
_endthreadex() 417
_eof() 380
_execl() 426
_execle() 426
_execlp() 426
_execlpe() 426

_execv() 426
_execve() 426
_execvp() 426
_execvpe() 426
_exit() 449
_Exit() 449
_fcloseall() 354
_fdopen() 383
_filelength() 397
_filelengthi64() 397
_fileno() 371, 383
_findclose() 403, 404
_finddata64i32_t 404
_findfirst() 403
_findfirst32() 406
_findfirst32i64() 406
_findfirst64() 406
_findfirst64i32() 403, 404
_findnext() 403
_findnext32() 406
_findnext32i64() 406
_findnext64() 406
_findnext64i32() 403, 404
_finite() 179
_finitef() 179
_free_locale() 214
_fseeki64() 366
_fsopen() 353
_fstat() 397
_fstat32() 397
_fstat32i64() 397
_fstat64() 397
_fstat64i32() 397
_fstati64() 397
_ftelli64() 366
_fullpath() 386
_futime() 398
_futime32() 398
_futime64() 398
_get_current_locale() 214
_getch() 80, 81, 84
_getche() 80
_getcwd() 354, 401
_getdcwd() 401
_getdrive() 399
_getdrives() 399
_getpid() 429
_getwch() 248
_getwche() 248
_getws() 261
_gmtime64() 302
_gmtime64_s() 302
_i64toa() 175
_i64toa_s() 175
_i64tow() 295
_i64tow_s() 295
_isalnum_l() 219
_isalpha_l() 219
_iscntrl_l() 221
_isdigit_l() 218
_isgraph_l() 221
_islower_l() 220
_isnan() 180
_isnanf() 180
_isprint_l() 221
_ispunct_l() 220
_isspace_l() 219
_isupper_l() 220
_isxdigit_l() 218
_itoa() 174
_itoa_s() 174
_itow() 294
_itow_s() 294
_locale_t 214
_localtime64() 303
_localtime64_s() 303
_lseek() 381
_lseeki64() 381
_ltoa() 175
_ltoa_s() 175
_ltow() 295
_ltow_s() 295
_makepath() 389
_makepath_s() 389
_MAX_PATH 354, 386, 401
_memicmp() 203, 241
_memicmp_l() 203, 241
_mkdir() 402
_mktime64() 304
_O_APPEND 375
_O_BINARY 375
_O_CREAT 375, 376
_O_EXCL 375
_O_NOINHERIT 375
_O_RANDOM 375
_O_RDONLY 374
_O_RDWR 374
_O_SEQUENTIAL 375
_O_SHORT_LIVED 375
_O_TEMPORARY 375, 381
_O_TEXT 375
_O_TRUNC 375
_O_U16TEXT 375
_O_U8TEXT 375

_O_WRONLY 374
_O_WTEXT 375
_onexit() 448
_open() 374
_P_DETACH 429
_P_NOWAIT 429
_P_NOWAITO 429
_P_OVERLAY 428
_P_WAIT 428
_pclose() 450
_popen() 450
_printf_l() 65, 66
_putenv() 453
_putenv_s() 453
_putws() 260
_read() 377
_rmdir() 402
_S_IEXEC 396
_S_IFCHR 396
_S_IFDIR 396
_S_IFIFO 396
_S_IFMT 396
_S_IFREG 396
_S_IREAD 375, 393, 396
_S_IWRITE 376, 393, 396
_SH_DENYNO 353, 376
_SH_DENYRD 353, 376
_SH_DENYRW 353, 376
_SH_DENYWR 353, 376
_snscanf() 173
_snwscanf() 293
_sopen() 376
_sopen_s() 376
_spawnl() 427
_spawnle() 427
_spawnlp() 427
_spawnlpe() 428
_spawnv() 428
_spawnve() 428
_spawnvp() 428
_spawnvpe() 428
_splitpath() 388
_splitpath_s() 387
_sprintf_l() 176, 242
_sprintf_s_l() 176, 242
_stat() 395, 396
_stat32() 395
_stat32i64() 395
_stat64 396
_stat64() 395, 396
_stat64i32 395
_stat64i32() 395
_stati64() 395
_strcoll_l() 238
_strdup() 230
_strerror() 340
_strerror_s() 340
_stricmp() 240
_stricmp_l() 240
_stricoll() 240
_stricoll_l() 240
_strlwr() 217, 236
_strlwr_s() 218, 237
_strncoll() 238
_strncoll_l() 238
_strnicmp() 240
_strnicmp_l() 240
_strnicoll() 240
_strnicoll_l() 240
_strnset() 235
_strnset_s() 235
_strrev() 236
_strset() 234
_strset_s() 235
_strtod_l() 171
_strtoi64() 169
_strtol_l() 168
_strtoui64() 169
_strtoul_l() 168
_strupr() 217, 236
_strupr_s() 217, 236
_strxfrm_l() 239
_tell() 380
_telli64() 380
_tempnam() 369
_time64() 301
_tolower_l() 217
_toupper_l() 216
_ui64toa() 176
_ui64toa_s() 176
_ui64tow() 296
_ui64tow_s() 296
_ultoa() 175
_ultoa_s() 175
_ultow() 295
_ultow_s() 295
_unlink() 390
_USE_MATH_DEFINES 160
_utimbuf 398
_utime() 398
_utime32() 398
_utime64() 398
_waccess() 391
_waccess_s() 392

_wasctime() 305
_wasctime_s() 306
_wchdir() 402
_wchmod() 393
_wcreate_locale() 214
_wcsdup() 275
_wcserror() 339
_wcserror_s() 339
_wcsicmp() 284
_wcsicoll() 285
_wcslwr() 250, 281
_wcslwr_s() 251, 282
_wcsncoll() 283
_wcsnicmp() 284
_wcsnicoll() 285
_wcsnset() 280
_wcsnset_s() 280
_wcsrev() 281
_wcsset() 279
_wcsset_s() 280
_wcstoi64() 290
_wcstoui64() 290
_wcsupr() 250, 281
_wcsupr_s() 250, 281
_wctime() 306
_wctime_s() 307
_wctime64() 306
_wctime64_s() 307
_wdupenv_s() 453
_wexecl() 426
_wexecle() 426
_wexeclp() 426
_wexeclpe() 426
_wexecv() 426
_wexecve() 426
_wexecvp() 426
_wexecvpe() 426
_wfdopen() 383
_wfinddata64i32_t 406
_wfindfirst32() 406
_wfindfirst32i64() 406
_wfindfirst64() 406
_wfindfirst64i32() 406
_wfindnext32() 406
_wfindnext32i64() 406
_wfindnext64() 406
_wfindnext64i32() 406
_wfopen() 351, 356
_wfopen_s() 352
_wfreopen() 370
_wfreopen_s() 371
_wfsopen() 353
_wfullpath() 386
_wgetcwd() 354, 401
_wgetdcwd() 401
_wgetenv() 452
_wgetenv_s() 452
_wmakepath() 389
_wmakepath_s() 389
_wmkdir() 402
_wopen() 374
_wperror() 341
_wpopen() 450
_wputenv() 453
_wputenv_s() 453
_wremove() 390
_wrename() 390
_write() 377
_wrmdir() 402
_wsetlocale() 213, 247, 260
_wsopen() 376
_wsopen_s() 376
_wspawnl() 427
_wspawnle() 427
_wspawnlp() 428
_wspawnlpe() 428
_wspawnv() 428
_wspawnve() 428
_wspawnvp() 428
_wspawnvpe() 428
_wsplitpath() 388
_wsplitpath_s() 387
_wstat() 395
_wstat32() 395
_wstat32i64() 395
_wstat64() 395, 396
_wstat64i32() 395
_wstati64() 395
_wsystem() 449, 450
_wtempnam() 369
_wtmpnam() 368
_wtmpnam_s() 368
_wtof() 291
_wtoi() 286
_wtoi64() 288
_wtol() 286
_wtoll() 287
_wunlink() 390
_wutime() 398
_wutime32() 398
_wutime64() 398

A

abort() 89
abs() 160
acos() 165
acosf() 165
acosl() 165
ar.exe 432
ASCII 207
asctime() 305
asctime_s() 306
asin() 165
asinf() 165
asinl() 165
atan() 165
atanf() 165
atanl() 165
atexit() 448
atof() 171
atoi() 165
atol() 166
atoll() 167
auto 104

B

BOM 356
bool 93
break 148, 149, 151, 153
bsearch() 200
btowc() 255
Byte Order Mark 356

C

-c 432
c16rtomb() 298
c32rtomb() 298
calloc() 119, 189
case 147, 148, 149
ceil() 164
ceilf() 164
ceill() 164
char 93, 110, 156, 206, 207, 222, 255
CHAR_BIT 93
CHAR_MAX 93
CHAR_MIN 93
char16_t 297
char32_t 297
CharToOemA() 266
CharToOemBuffA() 266
clearerr() 363
clearerr_s() 363
clock() 300, 311
clock_t 300
CLOCKS_PER_SEC 311
CloseHandle() 410, 413
Cmake 28
const 101, 116, 182, 183, 187, 225, 258, 325
continue 153
cos() 165
cosf() 165
cosl() 165
cp1251 209
cp866 209
CreateMutexA() 413
CreateMutexW() 413
CreateThread() 407
ctime() 306
ctime_s() 307
C-строка 110, 206

D

-D_REENTRANT 420
Debug 54
default 147
DeleteCriticalSection() 415
difftime() 304
div() 162
DLL 438
DllMain() 446
do...while 152, 153
double 94, 158
DWORD 71

E

Eclipse 29, 34, 47, 64
else 142–144
EnterCriticalSection() 415
enum 129
EOF 354
ERANGE 339
errno 339, 341, 358, 374
exit() 89, 449
EXIT_FAILURE 61, 89
EXIT_SUCCESS 61, 89
ExitThread() 409
exp() 161
expf() 161
expl() 161
extern 104

F

F_OK 391
fabs() 160
fabsf() 160
fabsl() 160
false 93
fclose() 353
feof() 363
ferror() 363
-fexec-charset 38, 39, 47, 244
fflush() 71, 75, 85, 358, 372
fgetc() 360
fgetpos() 365
fgets() 77, 358, 361
fgetwc() 360
fgetws() 261, 361
FILE 351, 352
FILE_ATTRIBUTE_HIDDEN 394
FILENAME_MAX 354
-finput-charset 38, 39, 47, 244
float 94, 158
floor() 164
floorf() 164
floorl() 164
fmax() 163
fmaxf() 163
fmaxl() 163
fmin() 163
fminf() 163
fminl() 163
fmod() 162
fmodf() 162
fmodl() 162
fopen() 351, 356
FOPEN_MAX 352
fopen_s() 352
for 150, 151, 185, 226, 259
-fPIC 439
fpos_t 365
fprintf() 358, 359
fprintf_s() 360
fputc() 358
fputs() 358, 359
fputwc() 359
fputws() 359
fread() 364
free() 119, 189
FreeLibrary() 443
freopen() 370
freopen_s() 371
fscanf() 361
fseek() 366
fsetpos() 366
ftell() 365
-fwide-exec-charset 244
fwprintf() 360
fwprintf_s() 360
fwrite() 363
fwscanf() 362

G

g++.exe 18
gcc.exe 9, 15, 18, 20, 47
gdb.exe 345
getc() 360
getchar() 71, 77, 85
GetCurrentThread() 410
GetCurrentThreadId() 410
GetDiskFreeSpaceEx() 400
GetDiskFreeSpaceExA() 400
GetDiskFreeSpaceExW() 400
GetDriveType() 400
GetDriveTypeA() 400
GetDriveTypeW() 400
getenv() 452
getenv_s() 452
GetExitCodeThread() 409
GetFileAttributes() 394
GetLastError() 409
GetModuleFileNameA() 444
GetModuleFileNameW() 444
GetProcAddress() 444
gets() 77
GetSystemDirectoryA() 445
GetSystemDirectoryW() 445
getwc() 360
getwchar() 248, 261
GetWindowsDirectoryA() 446
GetWindowsDirectoryW() 446
Glade 28
gmtime() 301
gmtime_s() 302
goto 154
GTK+ 28

H

HUGE_VAL 172
Нулевой указатель 112

I

-I 51, 433
if 142, 144, 146
imaxabs() 161
imaxdiv() 163
INFINITE 409
INFINITY 134, 179
InitializeCriticalSection() 415
inline 333
int 93, 97, 156
INT_MAX 94
INT_MIN 94
INT16_MAX 98
INT16_MIN 98
int16_t 98
INT32_MAX 98
INT32_MIN 98
int32_t 98
INT64_MAX 98
INT64_MIN 98
int64_t 98, 99
INT8_MAX 98
INT8_MIN 98
int8_t 98
INTMAX_MAX 170
INTMAX_MIN 170
intmax_t 170
isalnum() 219
isalpha() 219
isblank() 221
iscntrl() 221
isdigit() 218
isgraph() 221
isinf() 180
islower() 220
isnan() 180
iso8859-5 209
isprint() 221
ispunct() 220
isspace() 219
isupper() 220
iswalnum() 252
iswalpha() 252
iswblank() 254
iswcntrl() 254
iswdigit() 251
iswgraph() 253
iswlower() 252
iswprint() 253
iswpunct() 253
iswspace() 252
iswupper() 253
iswxdigit() 251
isxdigit() 218

J

Java 29
Java Development Kit 29
Java Runtime Environment 29
JDK 29
JRE 29

K

koi8-r 209

L

-l 433
-L 433
labs() 160
LC_ALL 88, 212
LC_COLLATE 88, 212
LC_CTYPE 88, 212
LC_MONETARY 88, 212
LC_NUMERIC 88, 212
LC_TIME 88, 212
lconv 215
ldiv() 162
LeaveCriticalSection() 415
llabs() 160
lldiv() 162
LLONG_MAX 97
LLONG_MIN 97
LoadLibraryA() 443
LoadLibraryW() 443
localeconv() 215
localtime() 302
localtime_s() 303
log() 161
log10() 161
log10f() 161
log10l() 161
logf() 161
logl() 161
long 69, 96, 156
long double 70, 97, 158
long int 96
long long 69, 96, 156
long long int 96
LONG_LONG_MAX 97
LONG_LONG_MIN 97

LONG_MAX 96
LONG_MIN 96
-lpthread 420
Lucida Console 86
L-строки 244

M

M_1_PI 159
M_2_PI 159
M_2_SQRTPI 160
M_E 159
M_LN10 160
M_LN2 159
M_LOG10E 159
M_LOG2E 159
M_PI 159
M_PI_2 159
M_PI_4 159
M_SQRT1_2 160
M_SQRT2 160
mac-cyrillic 209
main() 45, 60, 61, 82, 451
Make 28
make.exe 23
malloc() 118, 119, 189
mbrtoc16() 298
mbrtoc32() 299
mbstowcs() 263, 265
mbstowcs_s() 264
memchr() 233
memcmp() 203, 241
memcpy() 201
memcpy_s() 202
memmove() 202
memmove_s() 202
memset() 235
Microsoft Visual C++ 42
MinGW 14
MINGW_HOME 37, 41
mingw32.exe 24
mingw64.exe 24
MinGW-W64 20, 24
mktime() 303
modf() 162
modff() 162
modfl() 162
MSYS 14
MSYS_HOME 37, 41
MSYS2 24
msys2.exe 24
MultiByteToWideChar() 267

N

NAN 134, 179, 180
nanosleep() 310
Notepad++ 19
NULL 79, 112, 113

O

-o 47, 432
-O3 47
OemToCharA() 266
OemToCharBuffA() 266

P

PATH 12, 17, 37, 426, 446, 452
pause 85
perror() 341, 358
pow() 161
powf() 161
powl() 161
PRId16 99
PRId32 99
PRId64 99
PRId8 99
printf() 45, 65, 66, 324, 344
PRIu16 99
PRIu32 99
PRIu64 99
PRIu8 99
-pthread 420
pthread_cancel() 422
pthread_create() 420
pthread_exit() 421, 422
pthread_join() 421, 422
pthread_mutex_destroy() 424
pthread_mutex_init() 423
pthread_mutex_lock() 424
pthread_mutex_trylock() 424
pthread_mutex_unlock() 424
PTHREAD_THREADS_MAX 421
putc() 358
putchar() 65
puts() 65
putwc() 359
putwchar() 247

Q

qsort() 196

R

R_OK 391
rand() 177
RAND_MAX 177
ranlib.exe 432
realloc() 122, 189
register 104
Release 54
ReleaseMutex() 413
remove() 390
rename() 390
return 45, 60, 312, 313, 328
rewind() 363, 366
round() 164
roundf() 164
roundl() 164

S

scanf() 72, 343
SCHAR_MAX 95
SCHAR_MIN 95
SEEK_CUR 366, 381
SEEK_END 366, 381
SEEK_SET 366, 381
SetDllDirectoryA() 443
SetDllDirectoryW() 443
SetFileAttributes() 394
setlocale() 73, 74, 78, 88, 212, 247, 260
-shared 439
short 69, 95, 156
short int 95
SHRT_MAX 96
SHRT_MIN 96
signed 94, 156
signed char 94
signed int 95
signed long 96
signed long int 96
signed long long 96
signed long long int 96
signed short 95
signed short int 95
sin() 165
sinf() 165
sinl() 165
size_t 99, 225, 258
sizeof 99, 118, 124, 183, 246
sleep() 309
Sleep() 71, 310, 311, 410
sprintf() 176, 242
sprintf_s() 176, 242
sqrt() 161
sqrtf() 161
sqrtl() 161
srand() 177
sscanf() 173
static 104, 105, 327, 333
-static 434
-std 9
stderr 341, 358, 369, 382, 383
STDERR_FILENO 382
stdin 75, 78, 369, 382
STDIN_FILENO 382
stdout 71, 369, 382, 384
STDOUT_FILENO 382
strcat() 229
strcat_s() 230
strchr() 232
strcmp() 237
strcoll() 238
strcpy() 223, 228
strcpy_s() 223, 228
strcspn() 234
strerror() 339, 358
strerror_s() 339
strftime() 307, 308
strlen() 225–227
strncat() 230
strncat_s() 230
strncmp() 237
strncpy() 228
strncpy_s() 229
strnlen() 228
strpbrk() 233
strrchr() 233
strspn() 234
strstr() 234
strtod() 171
strtof() 172
strtoimax() 170
strtok() 231
strtok_s() 231
strtol() 168
strtold() 172
strtoll() 168
strtoul() 168
strtoull() 168
strtoumax() 170
struct 123, 126
strxfrm() 239
switch 147, 148
swprintf_s() 296

swscanf() 293
system() 73, 78, 85, 87, 449, 450

T

tan() 165
tanf() 165
tanl() 165
TerminateThread() 409
time() 301
time_t 300
tm 300
tmpfile() 367
tmpfile_s() 368
tmpnam() 368
tmpnam_s() 368
tolower() 217
toupper() 216
towlower() 250
towupper() 249
true 93
TryEnterCriticalSection() 415
typedef 100

U

UCHAR_MAX 95
UINT_MAX 95
UINT16_MAX 98
uint16_t 98
UINT32_MAX 98
uint32_t 98
UINT64_MAX 98
uint64_t 98
UINT8_MAX 98
uint8_t 98
UINTMAX_MAX 170
uintmax_t 170
ULLONG_MAX 97
ULONG_LONG_MAX 97
ULONG_MAX 96
UNICODE 356, 375
union 127
unsigned 95, 156, 207
unsigned char 95, 156, 222
unsigned int 95, 222
unsigned long 96, 156
unsigned long int 96
unsigned long long 156
unsigned long long int 97
unsigned short 96, 156
unsigned short int 96
USHRT_MAX 96
usleep() 310
UTF-16 297
UTF-16BE 267
UTF-16LE 267, 356, 375
UTF-32 297
UTF-32BE 267
UTF-32LE 267
UTF-8 244, 267, 356, 375

V

va_arg() 324
va_end() 324
va_list 324
va_start() 324
Visual C++ 42
Visual Studio 42
void 60, 94, 118, 312, 314, 328
void* 331
volatile 97

W

W_OK 391
WaitForMultipleObjects() 409
WaitForSingleObject() 409, 413
-Wall 38, 39, 47, 337
WCHAR_MAX 245
WCHAR_MIN 245
wchar_t 244, 245, 255, 256
-Wconversion 38, 39, 47, 58, 131, 337
wcscat() 274
wcscat_s() 274
wcschr() 277
wcscmp() 282
wcscoll() 283
wcscpy() 256, 272
wcscpy_s() 256, 272
wcscspn() 278
wcsftime() 307, 308
wcslen() 258, 271
wcsncat() 275
wcsncat_s() 275
wcsncmp() 282
wcsncpy() 273
wcsncpy_s() 273
wcsnlen() 271
wcspbrk() 278
wcsrchr() 278
wcsspn() 279
wcsstr() 279

wcstod() 292
wcstof() 292
wcstoimax() 291
wcstok() 276
wcstok_s() 276
wcstol() 288
wcstold() 293
wcstoll() 289
wcstombs() 264, 265
wcstombs_s() 264
wcstoul() 288
wcstoull() 289
wcstoumax() 291
wcsxfrm() 284
wctob() 255
WEOF 255, 360
while 152, 185, 227, 260
WideCharToMultiByte() 268
windows-1251 86, 209, 265, 267
windows-866 86, 209, 265, 267
windres.exe 456
WINT_MAX 246
WINT_MIN 246
wint_t 246
-Wl,-Bdynamic 434
-Wl,-Bstatic 434
wmemchr() 278
wmemcmp() 285
wmemcpy() 272
wmemcpy_s() 272
wmemmove() 273
wmemmove_s() 273
wmemset() 280
wprintf() 248, 261
wscanf() 249, 262
wtoll() 287

А

Абсолютное значение 161
Адрес переменной 112
Адресная арифметика 116
Арккосинус 165
Арксинус 165
Арктангенс 165

Б

Бесконечность 134, 179
Библиотека 430
◊ динамическая 438
◊ статическая 430
Бинарный поиск 199
Битовые поля 126
Блок 106, 144, 155

В

Ввод данных 71
Вещественные числа 158
◊ точность вычислений 158
Возведение в степень 161
Восьмеричные числа 157, 175, 295
Время 300
Встраиваемая функция 333
Вывод данных 65
Выполнение программы по шагам 345
Вычитание 133

Г

Глобальная переменная 105

Д

Дата 300
◊ форматирование 305
Двоичные числа 175, 295
Декремент 134
Деление 133
Десятичные числа 157
Десятичный логарифм 161
Динамическая библиотека 438
Динамическое выделение памяти 118
Директивы препроцессора 454
Диск 399

З

Завершение выполнения программы 89, 448
Заголовочный файл 50
Значок приложения 455

И

Именование переменных 92
Инициализация переменных 100
Инкремент 134

К

Каталог 401
◊ перебор объектов 403
◊ преобразование пути 386
◊ создание 402
◊ текущий рабочий 354
◊ удаление 402
Квадратный корень 161
Ключевые слова 92
Кодировка 49, 86, 263, 265
◊ файла с программой 244
Командная строка 14
Комментарии 63
◊ многострочные 64
◊ однострочные 63
Компилятор 14
Консоль 14
◊ предотвращение закрытия окна 84
Консольное приложение 44
Константа 101
Косинус 165

Л

Логарифм
◊ десятичный 161
◊ натуральный 161
Локаль 212
Локальная переменная 105

М

Макроопределение 101, 334
Макрос 101, 334
Массив 107, 181
◊ двумерный 109, 190
◊ динамический 189
◊ доступ к элементам 184
◊ зубчатый 121
◊ изменение значения элемента 184
◊ инициализация 181
◊ количество элементов 183
◊ копирование элементов 201
◊ максимальное значение 193
◊ минимальное значение 193
◊ многомерный 109, 190
◊ объявление 181
◊ перебор элементов 185
◊ переворачивание 204
◊ поиск значения 198
◊ получение значения элемента 184
◊ проверка наличия значения 198
◊ размер 183
◊ сортировка 195
◊ составной литерал 188, 320
◊ сравнение 203
◊ указателей 114, 189
◊ указатель на элемент 186
Метка порядка байтов 356
Многострочный комментарий 64
Мьютекс 413, 423

Н

Натуральный логарифм 161
Нулевой указатель 79

О

Области видимости переменных 105
Объединение 127
Объявление
◊ переменной 91
◊ функции 312
Однострочный комментарий 63
Операторы 133
◊ ?: 146
◊ break 153
◊ continue 153
◊ do...while 152
◊ for 149
◊ goto 154
◊ if 142
◊ switch 147
◊ while 152
◊ ветвления 142
◊ выбора 147
◊ декремента 134
◊ запятая 138
◊ инкремента 134
◊ математические 133
◊ побитовые 135
◊ приоритет выполнения 141
◊ присваивания 138
◊ сравнения 139
◊ циклы 149

Определение
◊ переменной 91
◊ функции 60, 312
Остаток от деления 134, 162
Отладка программы 345
Ошибки 336
◊ времени выполнения 337
◊ логические 337, 341
◊ отладка программы 345
◊ поиск 341
◊ синтаксические 336
◊ типы ошибок 336

П

Пароль
◊ ввод 80
◊ генерация 178
Переменная 91
◊ автоматическая 104
◊ адрес 112
◊ глобальная 58, 105
◊ именование 92
◊ инициализация 100
◊ локальная 58, 105
◊ область видимости 105
◊ объявление 91
◊ объявление в другом месте 104
◊ окружения 451
◊ регистровая 104
◊ статическая 104, 105, 326
Перенаправление потоков 369, 384
Перечисление 129
Потоки ввода/вывода 369
Потоки управления 407
◊ POSIX 420
◊ WinAPI 407
◊ синхронизация 412, 423
Препроцессор 454
Приведение типов 130, 131
Приоритет выполнения операторов 141
Присваивание 100
Прототип функции 59, 312
Процесс 426
Псевдоним 100
Пузырьковая сортировка 195
Путь
◊ абсолютный 354
◊ относительный 354
◊ преобразование 386

Р

Разыменование указателя 112
Рекурсия 332

С

Символы 206
◊ широкие 245
Синус 165
Синхронизация потоков 412, 423
Системные команды 449
Системные переменные 451
Сложение 133
Сортировка массива 195
Специальные символы 206
Спецификаторы хранения 104
Статическая библиотека 430
Статическая переменная 105, 326, 327
Статическая функция 327
Строки 110, 206
◊ длина 225, 258
◊ доступ к символам 224, 257
◊ замена 234, 279
◊ из широких символов 244
◊ кодировки 263, 265
◊ конкатенация 223, 257
◊ поиск 232, 277
◊ регистр символов 216, 249
◊ специальные символы 206
◊ сравнение 237, 282
◊ форматирование 242
Структура 123

Т

Тангенс 165
Текущий рабочий каталог 354
Типы данных 93
◊ приведение 131
Точки останова 346

У

Указатель 111
◊ void* 331
◊ арифметические и логические операции 116
◊ на структуру 126
◊ на функцию 330
◊ нулевой 112
◊ разыменование 112

Умножение 133
Унарный минус 133
Установка программ 11

Ф

Файл 351
◊ временный 367, 381
◊ двоичный 363
◊ закрытие 353, 377
◊ запись 358, 377
◊ низкоуровневые потоки 374
◊ открытие 351, 374
◊ переименование 390
◊ перемещение 390
◊ получение информации 395
◊ права доступа 393
◊ преобразование пути 386
◊ произвольного доступа 365, 380
◊ режим открытия 356, 374
◊ ресурсов 456
◊ скрытый 394
◊ удаление 390
◊ чтение 360, 377
Файл с программой 49
Факториал 332
Форматирование программы 143, 341
Функция 312
◊ возврат значения 328
◊ встраиваемая 333
◊ объявление 312
◊ определение 312
◊ передача данных произвольного типа 331
◊ передача массива 320
◊ передача параметров 318
◊ передача строки 320
◊ переменное число параметров 324
◊ прототип 312
◊ расположение определений 315
◊ рекурсия 332
◊ статическая 327

Ц

Целые числа 156, 157
Цикл
◊ do...while 152
◊ for 149
◊ while 152
◊ переход на следующую итерацию 153
◊ прерывание 153

Ч

Числа 156
◊ NAN 179
◊ бесконечность 179
◊ вещественные 158
▫ точность вычислений 158
◊ восьмеричные 157, 175, 295
◊ двоичные 175, 295
◊ десятичные 157
◊ округление 164
◊ основные функции 160
◊ преобразование 165, 286
◊ преобразование в строку 174, 294
◊ случайные 177
◊ тригонометрические функции 165
◊ целые 156, 157
◊ шестнадцатеричные 157, 175, 295

Ш

Шестнадцатеричные числа 157, 175, 295

Э

Экспонента 161

www.bhv.ru

Культин Н.

C/C++ в задачах и примерах
3-е изд.

Отдел оптовых поставок:
e-mail: opt@bhv.ru

Более 200 задач

- Ввод данных с клавиатуры
- Вывод на экран
- Циклы
- Сортировка и поиск в массиве
- Рекурсия
- Операции с файлами
- Работа с объектами

Сборник примеров и задач для самостоятельного решения по программированию на языке C/C++ охватывает практически все разделы начального курса программирования: от задач консольного ввода/вывода, использования циклов и операций с массивами до работы со строками, файлами и объектами. Примеры представлены в виде хорошо документированных исходных текстов программ. Книга содержит справочник — описание основных типов данных, операторов и наиболее часто используемых функций. Адресована студентам, школьникам старших классов и всем тем, кто изучает программирование в учебном заведении или самостоятельно. В третьем издании добавлены и обновлены примеры.

Никита Борисович Культин, кандидат технических наук, доцент Санкт-Петербургского государственного политехнического университета, где читает курс «Теория и технология программирования». Автор книг по программированию в Delphi, Microsoft Visual C++, Microsoft Visual C# и др., которые вышли общим тиражом более 350 тыс. экземпляров.

www.bhv.ru

Полубенцева М.

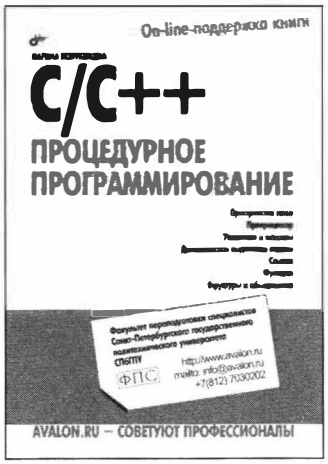
C/C++. Процедурное программирование

Отдел оптовых поставок

E-mail: opt@bhv.ru

«Программировать — значит понимать»

Кристиан Нюгард (Kristen Nygaard), норвежский математик, один из пионеров в разработке языков программирования



- Пространства имен
- Препроцессор
- Указатели и массивы
- Динамическое выделение памяти
- Ссылки
- Функции
- Структуры и объединения

«... В современных книгах незаслуженно мало внимания уделяется процедурным возможностям языков C/C++. Книга призвана заполнить этот пробел. В ней рассмотрены как основные понятия процедурного программирования на C/C++, так и взаимосвязи между ними. В результате по мере прочтения в голове у читателя формируется модель внутренней структуры изучаемого предмета, и после этого существенно облегчается восприятие последующего материала, поскольку начинает активно работать ассоциативная память.

Начинающий программист найдет в книге структурированный материал «для старта», а программист, уже имеющий опыт программирования на C/C++, откроет для себя возможности языка, которыми он до сих пор не пользовался (или пользовался неосознанно). Книга будет полезна разработчикам программного обеспечения для встраиваемых систем, поскольку в настоящее время большинство программ для микроконтроллеров пишут на языке C.

Для понимания правил, по которым действует компилятор, приводятся пояснения на уровне языка Ассемблера...»

М. И. Полубенцева, старший преподаватель компьютерных дисциплин СПбГПУ

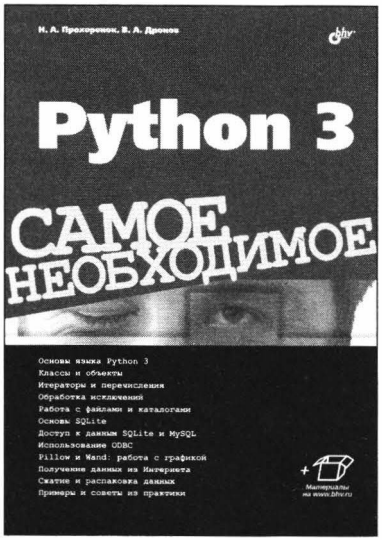
Полубенцева Марина Игоревна, старший преподаватель компьютерных дисциплин в Санкт-Петербургском государственном политехническом университете (СПбГПУ). Имеет многолетний преподавательский и практический опыт в области разработки приложений на C++. Читает курсы по процедурному программированию на C/C++, объектно-ориентированному программированию на C++, Windows-программированию, системному программированию в ОС Windows.

www.bhv.ru

Прохоренок Н., Дронов В.
Python 3. Самое необходимое

Отдел оптовых поставок:
e-mail: opt@bhv.ru

Быстро и легко осваиваем Python — самый стильный язык программирования



- Основы языка Python 3
- Классы и объекты
- Итераторы и перечисления
- Обработка исключений
- Работа с файлами и каталогами
- Основы SQLite
- Доступ к данным SQLite и MySQL
- Использование ODBC
- Pillow и Wand: работа с графикой
- Получение данных из Интернета
- Сжатие и распаковка данных
- Примеры и советы из практики

В книге описан базовый синтаксис языка Python 3: типы данных, операторы, условия, циклы, регулярные выражения, встроенные функции, классы и объекты, итераторы и перечисления, обработка исключений, часто используемые модули стандартной библиотеки. Даны основы SQLite, описан интерфейс доступа к базам данных SQLite и MySQL, в том числе посредством ODBC. Рассмотрена работа с изображениями с помощью библиотек Pillow и Wand, получение данных из Интернета и работа с архивами различных форматов. Книга содержит более двухсот практических примеров, помогающих начать программировать на языке Python самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания и продвижения динамических сайтов с использованием HTML, JavaScript, PHP, Perl и MySQL, автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Разработка Web-сайтов с помощью Perl и MySQL», «Python 3 и PyQt. Разработка приложений», «Python. Самое необходимое».

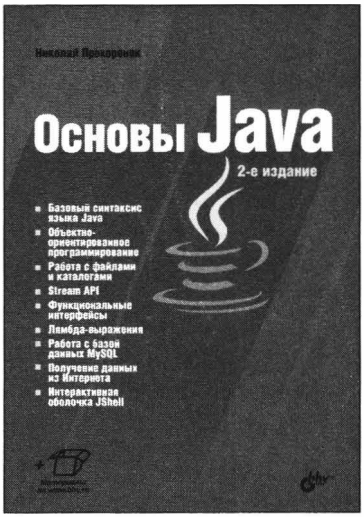
Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 20 популярных компьютерных книг, в том числе «HTML 5, CSS 3 и Web 2.0. Разработка современных Web-сайтов», «PHP 5/6, MySQL 5/6 и Dreamweaver CS4. Разработка интерактивных Web-сайтов», «JavaScript и AJAX в Web-дизайне», «Windows 8: разработка Metro-приложений для мобильных устройств» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

www.bhv.ru

Прохоренок Н.

Основы Java
2-е изд.

Отдел оптовых поставок
E-mail: opt@bhv.ru



Просто о сложном

- Базовый синтаксис языка Java
- Объектно-ориентированное программирование
- Работа с файлами и каталогами
- Stream API
- Функциональные интерфейсы
- Лямбда-выражения
- Работа с базой данных MySQL
- Получение данных из Интернета
- Интерактивная оболочка JShell

Если вы хотите научиться программировать на языке Java, то эта книга для вас. В книге описан базовый синтаксис языка Java: типы данных, операторы, условия, циклы, регулярные выражения, лямбда-выражения, ссылки на методы, объектно-ориентированное программирование. Рассмотрены основные классы стандартной библиотеки, получение данных из сети Интернет, работа с базой данных MySQL. Во втором издании приводится описание большинства нововведений: модули, интерактивная оболочка JShell, инструкция var и др.

Книга содержит большое количество практических примеров, помогающих начать программировать на языке Java самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Прохоренок Николай Анатольевич, профессиональный программист, автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений», «OpenCV и Java. Обработка изображений и компьютерное зрение» и др.

Язык С

Прикоснись к легенде

Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания и продвижения сайтов, анализа и обработки данных (работает с компьютерами с 1990 года). Автор книг «HTML, JavaScript, PHPи MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений», «Основы Java», «OpenCV и Java. Обработка изображений и компьютерное зрение», «Разработка Web-сайтов с помощью Perl и MySQL» и др., многие из которых выдержали несколько переизданий и стали бестселлерами.

Если вы хотите научиться программировать на языках С или С++, то эта книга для вас. В книге описан базовый синтаксис современного языка С: типы данных, операторы, условия, циклы, работа с числами, строками, массивами и указателями, создание пользовательских функций, модулей, статических и динамических библиотек. Рассмотрены основные функции стандартной библиотеки языка С, а также функции, применяемые только в операционной системе Windows. Для написания, компиляции и запуска программ используется редактор Eclipse, а для создания исполняемого файла — компилятор gcc.exe версии 8.2, входящий в состав популярной библиотеки MinGW-W64.

Книга содержит большое количество практических примеров, помогающих начать программировать на языке С самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник. Изучив основы языка С по этой книге, вы легко сможете научиться программировать на пяти языках, которые обеспечивают выполнение большинства прикладных задач:

- JavaScript и PHP — по книге «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера»;
- Perl — по книге «Разработка Web-сайтов с помощью Perl и MySQL»;
- Python — по книгам «Python 3. Самое необходимое» и «Python 3 и PyQt 5. Разработка приложений»;
- Java — по книге «Основы Java».

ISBN 978-5-9775-4116-9

Примеры из книги можно скачать по ссылке ftp://ftp.bhv.ru/9785977541169.zip, а также со страницы книги на сайте www.bhv.ru.

bhv

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru