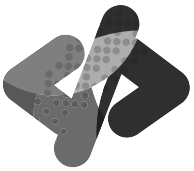
Майк МакГрат

C

ПРОГРАММИРОВАНИЕ ДЛЯ НАЧИНАЮЩИХ

ПЕРВЫЙ ШАГ НА ПУТИ К УСПЕШНОЙ КАРЬЕРЕ

Четвертое издание



ПРОГРАММИРОВАНИЕ
ДЛЯ НАЧИНАЮЩИХ

Mike McGrath

C PROGRAMMING EASY STEPS

4 edition

Майк МакГрат

ПРОГРАММИРОВАНИЕ НА C
ДЛЯ НАЧИНАЮЩИХ

Москва
2016

УДК 004.43
ББК 32.973-018.2
М15

Mike McGrath
C PROGRAMMING IN EASY STEPS, 4th edition

МакГрат, Майк.

М15 Программирование на С для начинающих / Майк МакГрат ; [пер. с англ. М. Райтмана]. – Москва : Эксмо, 2016. – 192 с. – (Программирование для начинающих).

В этой книге с помощью примеров программ и иллюстраций, показывающих результаты работы кода, разбираются все ключевые аспекты языка С. В этой книге описано даже то, как установить бесплатный компилятор для языка С и работать в нем, — у вас просто не будет шансов ошибиться!

Книга идеально подойдет программистам, переключающимся на работу с другим языком, студентам, изучающим язык С, а также тем, кто только начинает свою профессиональную деятельность и хочет научиться основам процедурного программирования.

УДК 004.43
ББК 32.973-018.2

Производственно-практическое издание

ПРОГРАММИРОВАНИЕ ДЛЯ НАЧИНАЮЩИХ

Майк МакГрат

ПРОГРАММИРОВАНИЕ НА С ДЛЯ НАЧИНАЮЩИХ
(орыс тілінде)

Директор редакции *Е. Капьёв*. Ответственный редактор *В. Обручев*
Художественный редактор *В. Брагина*

В оформлении обложки использованы иллюстрации: antishock, VikaSuh / Shutterstock.com
Используется по лицензии от Shutterstock.com

ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86, 8 (495) 956-39-21.
Home page: **www.eksmo.ru** E-mail: **info@eksmo.ru**

Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел. 8 (495) 411-68-86, 8 (495) 956-39-21
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Қазақстан Республикасында дистрибьютор және өнім бойынша
арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 2 51 59 89,90,91,92, факс: 8 (727) 251 58 12 вн. 107; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить по адресу: http://eksmo.ru/certification/
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 02.10.2015. Формат 84x108 1/16.
Печать офсетная. Усл. печ. л. 20,16.
Тираж экз. Заказ

ISBN 978-5-699-79117-0

Оглавление

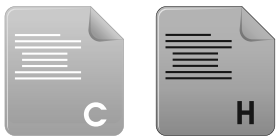
Введение

Создание этой книги позволило мне, Майку МакГрату (Mike McGrath), добавить в мои предыдущие книги по программированию на C новые приемы. Все примеры, предоставленные здесь, демонстрируют особенности C, которые поддерживаются современными компиляторами в операционных системах Windows и Linux, а также в среде разработки Microsoft Visual Studio. На снимках экранов показаны реальные результаты, полученные путем компилирования и выполнения предоставленного кода.

Соглашения, принятые в этой книге

Код каждого примера я сделал максимально наглядным.

В дополнение, для того, чтобы правильно идентифицировать каждый файл исходного кода, описанный в книге, используются соответствующие значки и имена файлов.



`main.c` `header.h`

Загрузка примеров к книге

Для удобства файлы с исходным кодом примеров, приведенных в этой книге, размещены в одном ZIP-архиве, с предоставлением отдельных версий для платформ Windows и Linux, а также для IDE Microsoft Visual Studio. Вы можете получить полный архив, перейдя по ссылке **www.eksmo.ru**. Далее извлеките содержимое архива в любое удобное место на вашем компьютере.

Я искренне надеюсь, что вы насладитесь исследованием возможностей программирования на языке C и получите столько же удовольствия, сколько получил я при написании этой книги.

Майк МакГрат

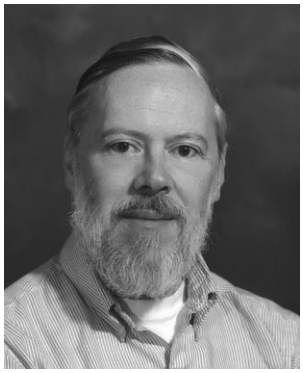
1 Приступаем к работе

Добро пожаловать в мир языка C. В этой главе показывается, как создать программу на языке C в текстовом виде, а затем скомпилировать ее в исполняемый файл.

- **Введение в язык C**
- **Установка компилятора языка C**
- **Написание программы на языке C**
- **Компилирование программы на языке C**
- **Понимание процесса компилирования**
- **Заключение**

Введение в язык C

C — это компактный компьютерный язык программирования общего назначения, созданный Деннисом Макалистером Ритчи (Dennis MacAlistair Ritchie) для операционной системы Unix на компьютере Digital Eauipment Corporartion PDP-11 в 1972-м году.



Деннис М. Ритчи, создатель языка программирования C.

Этот новый язык программирования был назван C (Си) в честь его предшественника — языка B, представленного в 1970-м году.

Операционная система Unix и фактически все ее приложения написаны на языке C. Однако он не ограничивается определенной платформой, поэтому программы с его использованием можно создавать на любой машине, поддерживающей этот язык, в том числе и на платформе Windows.

Гибкость и переносимость языка C сделали его очень популярным, этот язык был формализован Национальным Американским Институтом Стандартизации (American National Standards Institute, ANSI). Стандарт ANSI однозначно определил все аспекты языка, избавив нас от сомнений о его точном синтаксисе.

ANSI C стал узнаваемым стандартом языка C. В этой книге рассматривается и описывается именно он.

Зачем изучать программирование с помощью языка C?

Язык C существует уже долгое время, он застал создание новых языков программирования вроде Java, C++ и C#. Многие из них основаны на языке C, по крайней мере, отчасти, и при этом более громоздкие. C, как более компактный язык, лучше подходит для того, чтобы начать программировать, поскольку изучить его проще.

На заметку

Программы, написанные на языке C двадцать лет назад, в наши дни выполняются так же успешно, как и в те времена.

Как только освоите основные принципы программирования на языке C, вы сможете переключиться на изучение более новых языков программирования. Например, язык C++ является расширением языка C. Он может быть труден в изучении, если вы сперва не освоите программирование на C.

Несмотря на дополнительные возможности, доступные в новых языках, C остается популярным, поскольку он гибок и эффективен. Сегодня он используется на большом количестве платформ, начиная с микроконтроллеров и заканчивая продвинутыми научными системами. Программисты со всего мира используют язык C, поскольку он позволяет им получить максимальный контроль над выполнением программ, а также повышает их эффективность.

Стандартные библиотеки языка C

Для ANSI C определены несколько стандартных библиотек, содержащие опробованные и протестированные функции, которые могут быть использованы в ваших собственных программах, написанных на языке C.

Библиотеки содержатся в «заголовочных файлах», каждый из которых имеет расширение *.h*. Названия стандартных заголовочных файлов и их описания приведены в таблице ниже.

Совет

Функция — это фрагмент кода, который может быть повторно использован в программе на языке C. Описание каждой функции библиотеки C представлено в конце книги.

Библиотека	Описание
stdio.h	Содержит функции ввода и вывода, типы и описания макросов. Эта библиотека используется в большинстве программ, написанных на языке C, и представляет почти треть всех библиотек языка C
ctype.h	Содержит функции для работы с символами
string.h	Содержит функции для работы со строками
math.h	Содержит математические функции
stdlib.h	Содержит вспомогательные функции для преобразования чисел, выделения памяти и т. д.
assert.h	Содержит функции, которые могут быть использованы для диагностики программы
stdarg.h	Содержит функции, которые могут быть использованы для выполнения перебора аргументов функций
setjmp.h	Содержит функции, которые могут быть использованы для того, чтобы нарушить обычную последовательность входа в функции и выхода из них
signal.h	Содержит функции, предназначенные для обработки исключительных ситуаций, которые могут возникнуть в программе
time.h	Содержит функции для манипулирования компонентами, представляющими дату и время
limits.h	Содержит константные определения размеров типов данных языка C
float.h	Содержит определения констант, используемых в арифметике с плавающей точкой

Установка компилятора языка C

GNU является рекурсивным акронимом для фразы Gnu's Not Unix (Gnu — это не Unix) и произносится как «ГЭ-ЭН-У». Более подробную информацию вы можете найти, перейдя по адресу **www.gnu.org**.

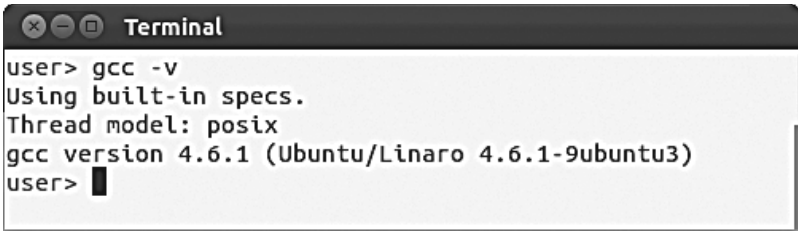
Программы на языке C изначально создаются как простые текстовые файлы, сохраняемые с расширением *.c*. Они могут быть написаны в любом текстовом редакторе, даже в программе Блокнот (Notepad) операционной системы Windows — никакого специального программного обеспечения не требуется.

Для того чтобы выполнить программу, написанную на языке C, необходимо ее «скомпилировать» в байт-код, который компьютер сможет понять. Компилятор языка C считывает оригинальную текстовую версию программы и переводит ее во второй файл, имеющий исполняемый байтовый формат, который сможет распознать компьютер.

Если текст программы содержит синтаксические ошибки, компилятор об этом сообщит, и исполняемый файл не будет построен.

Один из наиболее популярных компиляторов языка C — GNU C Compiler (GCC) — доступен бесплатно под лицензией General Public License (GPL). Он включен во все дистрибутивы операционной системы Linux. GNU C Compiler был использован для компилирования в исполняемый код всех примеров этой книги.

Чтобы определить, имеет ли ваша операционная система компилятор GNU C Compiler, наберите в командной строке `gcc -v`. Если компилятор доступен, он выведет на экран информацию о своей версии.



Если вы используете операционную систему Linux и компилятор GNU C Compiler недоступен, установите его с диска с дистрибутивом или из онлайн-репозитория, или же попросите выполнить установку системного администратора.

Если вы используете операционную систему Windows, с помощью следующих шагов вы сможете загрузить пакет Minimalist GNU for Windows (MinGW), который содержит компилятор GNU C Compiler.

На заметку

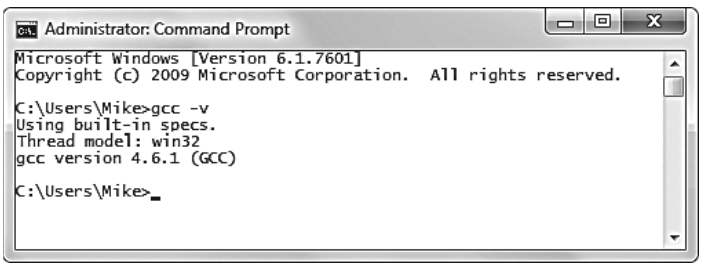
При установке компилятора устанавливаются также стандартные заголовочные файлы (указанные на предыдущей странице).

1. Запустите веб-браузер и перейдите на страницу проекта MinGW: **sourceforge.net/projects/mingw**.
2. На странице проекта MinGW нажмите кнопку **Download**, чтобы загрузить «Автоматический установщик MinGW» (Automated MinGW Installer) — он имеет имя, похожее на *mingw-get-inst-setup.exe*.
3. Дважды щелкните мышью по загруженному файлу.
4. Примите предложенное место установки *C:\MinGW*, а затем нажмите кнопку **Continue** (Продолжить), чтобы запустить процесс установки.

Как только установка завершится, исполняемый файл компилятора GNU C Compiler может быть найден в поддиректории *C:\MinGW\bin*. Довольно удобно добавить этот каталог к системным путям, чтобы компилятор можно было легко найти из любой директории вашей системы.

По окончании установки нажмите кнопку **Continue** (Продолжить), чтобы открыть окно менеджера установки. В левой части окна выберите пункт **All Packages** (Все пакеты), а в правой — отметьте флажками семь пунктов **mingw32-gcc** (в описании со значением The GNU C Compiler). Затем выберите команду меню **Installation** ⇒ **Apply Changes** (Установка ⇒ Применить изменения) и подтвердите действие нажатием кнопки **Apply** (Применить). После завершения установки закройте окно менеджера установки.

5. Откройте диалоговое окно **Environment Variables** (Переменные среды):
   - откройте окно **Control panel** (Панель управления) операционной системы Windows;
   - щелкните мышью по значку **System** (Система);
   - щелкните по ссылке **Advanced system settings** (Дополнительные параметры системы);
   - нажмите кнопку **Environment Variables** (Переменные среды).
6. В нижней части диалогового окна найдите переменную **Path**, а затем добавьте в конец представляющей ее строки пункт **;C:\MinGW\bin**.
7. Чтобы протестировать доступность компилятора GNU C Compiler, наберите в командной строке `gcc -v`, а затем нажмите кнопку **Enter**, чтобы компилятор вернул информацию о своей версии.



Внимание

Предоставленные шаги установки корректны на момент написания книги, но что-то может измениться. Перейдите на сайт **www.mingw.org**, чтобы получить свежие детали. Дальнейшие инструкции по загрузке и установке доступны после перехода по ссылке **Support** (Поддержка) на странице проекта MinGW на сайте **sourceforge.net**.

Совет

Поскольку язык C++ является расширением языка C, любой инструмент разработки для языка C++ также может быть использован и для компилирования программ, написанных на языке C.

Написание программы на языке C

В языке программирования C утверждения, которые должны быть выполнены, располагаются внутри функций, определяемых с использованием следующего синтаксиса:

```
тип-данных имя-функции () {утверждения-которые-нужно-выполнить}
```

После того, как функция вызывается с целью выполнения утверждений, которые в ней содержатся, она может вернуть значение вызывающей стороне. Это значение должно иметь тип, указанный перед именем функции.

Программа может содержать одну или несколько функций, при этом в ней должна присутствовать функция, которая называется `main`. Функция `main()` — это стартовая точка всех программ, написанных на языке C, и компилятор не будет компилировать код, если не найдет внутри программы функцию `main()`.

Другим функциям программы можно дать любые имена, содержащие буквы, цифры и символы подчеркивания, но следует иметь в виду, что имя функции не должно начинаться с цифры. Также следует избегать использования в качестве имен функций ключевых слов языка C, приведенных в конце книги.

Круглые скобки (),следующие за именем функции, могут содержать значения, которые используются функцией. Эти значения имеют вид разделенного запятыми списка и называются аргументами функции.

Фигурные скобки {} содержат утверждения, которые должны быть выполнены при вызове функции. Каждое утверждение обязано заканчиваться точкой с запятой.

Традиционно при изучении языка программирования в первую очередь пишут программу, выводящую на экран сообщение Hello World.

hello.c

1. Откройте простой текстовый редактор, например Блокнот (Notepad), а затем введите следующую строку кода в начале страницы именно так, как показано здесь:

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

   Программа начинается с инструкции для компилятора C, которая указывает подключить файл стандартной библиотеки функций ввода и вывода `stdio.h`. Что делает доступными в программе все функции, описанные внутри этого файла. Подходящее название данной инструкции — «инструкция препроцессора» или «дирек-

Внимание

Не используйте приложения — текстовые процессоры для написания кода программ, поскольку они хранят дополнительную информацию о форматировании, которая помешает скомпилировать код.

На заметку

Инструкции препроцессора начинаются с символа хэша — #, а имена стандартных библиотек заключаются в угловые скобки — <>.

тива препроцессора», она всегда должна появляться в начале страницы, до того, как будет обработан сам код программы.

2. Пропустите две строки после инструкции препроцессора и добавьте пустую функцию **main**:

```
int main()
{

}
```

Такое объявление функции определяет, что после выполнения функция должна будет вернуть значение типа **int**.

3. Внутри скобок вставьте строку кода, которая вызывает одну из функций, определенных в стандартной библиотеке ввода-вывода, ставшую доступной после написания инструкции препроцессора:

```
printf ( "Hello World!\n" ) ;
```

Внутри круглых скобок функции **printf()** определяется один строковой аргумент. В языке программирования C строки должны быть заключены в двойные кавычки. Эта строка содержит текст **Hello World** и управляющую последовательность **\n**, которая переводит каретку к левому краю следующей строки.

4. Внутри скобок вставьте последнюю строку кода, возвращающую число 0, что требуется в объявлении функции

```
return 0;
```

По традиции возвращение значения 0 после выполнения программы указывает операционной системе, что программа выполнилась корректно.

5. Проверьте, что код программы выглядит в точности так же, как и в листинге, приведенном внизу, а затем добавьте последний символ новой строки (нажмите клавишу **Enter** после закрывающей скобки) и сохраните программу под именем *hello.c*.

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Эта программа в текстовом формате уже готова к компилированию в понятный для машины байтовый формат.

Совет

Компилятор языка C игнорирует пробелы между кодом, но код программы всегда должен заканчиваться символом новой строки.

На заметку

Каждое утверждение должно завершаться точкой с запятой.

Компилирование программы на языке C

Совет

В командной строке можно ввести `gcc -help`, а затем нажать клавишу **Enter**, чтобы получить список всех опций компилятора.

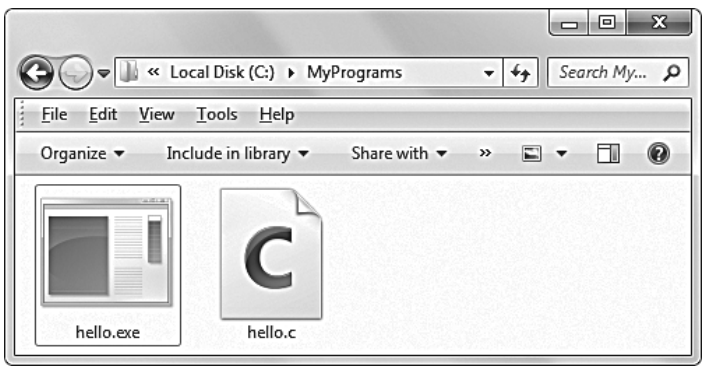
Исходный код примеров этой книги, написанный на языке C, хранится в директории, созданной специально для этой цели. Директория имеет название *MyPrograms*, ее абсолютный адрес в операционной системе Windows — *C:\MyPrograms*, а в операционной системе Linux — */home/MyPrograms*. Файл исходного кода *hello.c*, созданный с помощью указанных на предыдущей странице шагов, сохраняется в этой директории, ожидая запуска компилирования, что создаст исполняемый файл в формате байт-кода.

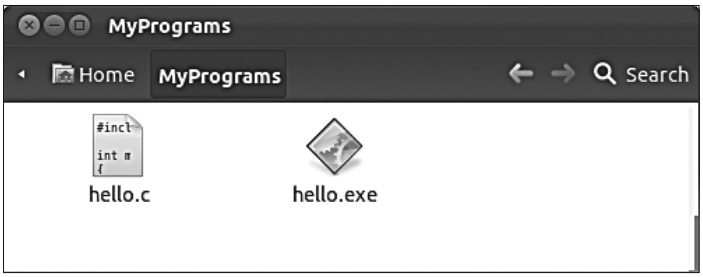
1. В командной строке введите команду `cd` с путем к директории `MyPrograms`, чтобы перейти в нее.
2. В командной строке, находясь в директории `MyPrograms`, введите команду `gcc hello.c`, а затем нажмите клавишу **Enter**, чтобы скомпилировать программу.

После того, как процесс успешно завершится, компилятор создаст исполняемый файл в каталоге с файлом исходного кода. По умолчанию этот файл будет иметь имя *a.out* в операционной системе Linux и *a.exe* — в операционной системе Windows. Компилирование другого файла исходного кода, написанного на языке C, в директории *MyPrograms* перезапишет первый исполняемый файл без предупреждения. Такое поведение очевидно является неудовлетворительным, поэтому при компилировании файла *hello.c* следует указывать имя исполняемого файла. Это делается с помощью опции `-o`, после которой указывается новое имя.

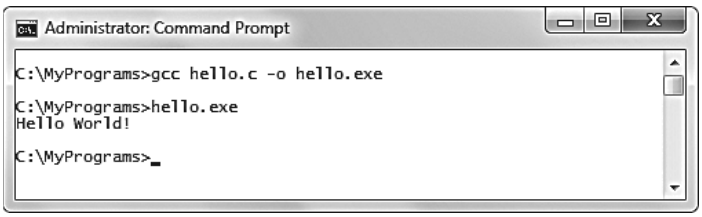
3. В командной строке, находясь в директории `MyPrograms`, введите `gcc hello.c -o hello.exe`, а затем нажмите клавишу **Enter**, чтобы скомпилировать программу еще раз.

В обеих операционных системах (как в Windows, так и в Linux) будет создан файл с именем *hello.exe* в соответствующей директории с файлом исходного кода:





4. В командной строке в операционной системе Windows введите имя исполняемого файла, а затем нажмите клавишу **Enter**, чтобы запустить программу — в результате будет выведена текстовая строка, а каретка переместится к следующей строке.



Поскольку операционная система Linux по умолчанию не станет проверять текущую директорию на наличие исполняемых файлов, если это не указано специально, для выполнения программы необходимо добавлять к имени файла символы ./.

5. В командной строке в операционной системе Linux введите `./hello.exe`, а затем нажмите клавишу **Enter**, чтобы запустить программу — в результате будет выведена текстовая строка, а каретка переместится к следующей строке.

Вы создали, скомпилировали и запустили простую программу Hello World, которая является стартовой точкой программирования. Все другие примеры этой книги будут созданы, скомпилированы и запущены точно таким же способом.

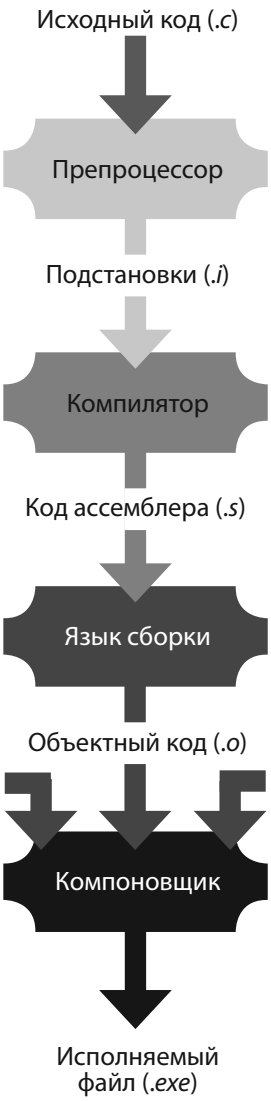
На заметку

Если компилятор выведет предупреждение о том на то, что в конце программы нет символа перехода к новой строке, разместите этот символ после исходного кода, а затем сохраните программу и повторите попытку компилирования.

Совет

Пользователи операционной системы Windows могут не указывать расширение файла при запуске программ. В этом примере достаточно написать лишь слово `hello`.

Понимание процесса компилирования



При создании исполняемого файла из оригинального файла исходного кода на языке C процесс компилирования проходит через четыре отдельных этапа, каждый из которых генерирует новый файл.

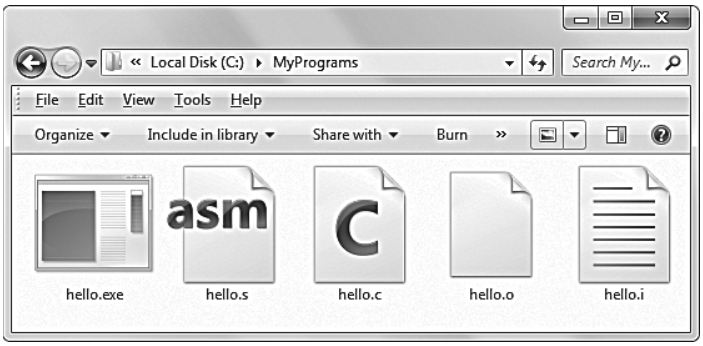
- **Предварительная обработка** — препроцессор заменяет все директивы препроцессора в оригинальном коде на языке C на код библиотек, которые реализуют эти директивы. Например, подобная замена выполняется для директив `#include`. Сгенерированный файл, содержащий замены, имеет текстовый формат и, как правило, расширение *.i*.
- **Преобразование (трансляция)** — компилятор транслирует высокоуровневые инструкции файла с расширением *.i* в низкоуровневые инструкции языка ассемблера. Сгенерированный файл, содержащий транслированные инструкции, имеет текстовый формат и, как правило, расширение *.s*.
- **Сборка** — сборщик преобразует текстовые инструкции языка ассемблера файла с расширением *.s* в машинный код. Сгенерированный файл объекта имеет двоичный формат и, как правило, расширение *.o*.
- **Связывание** — компоновщик связывает полученные двоичные объектные файлы с расширением *.o* в единый исполняемый файл. Сгенерированный файл имеет двоичный формат и, как правило, расширение *.exe*.

Вкратце, «компилирование» представляет собой первые три вышеописанных этапа работы с одним файлом исходного хода, из которого создается один двоичный объектный файл. Если исходный код программы содержит синтаксические ошибки, например, отсутствует точка с запятой или круглые скобки, компилятор сообщит о них и компиляция не завершится.

Компоновщик, с другой стороны, способен работать с несколькими объектами и по окончании своей работы сгенерирует один исполняемый файл. Это позволяет создавать большие программы из отдельных файлов, и каждый из них может содержать повторно используемые функции. Если компоновщик находит функцию, имя которой уже встречалось в другом файле, он сообщит об ошибке и исполняемый файл не будет создан.

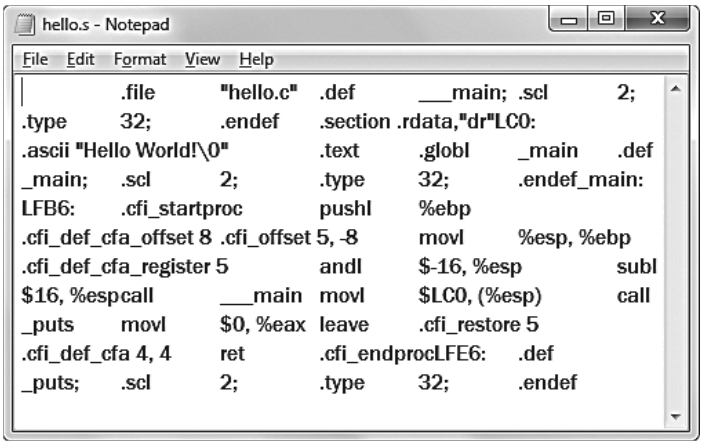
Обычно временные файлы, созданные во время промежуточных шагов процесса компилирования, удаляются автоматически, но их можно восстановить с помощью опции `-save-temps`, переданной в команде компилятору.

1. В командной строке директории `MyPrograms` введите `gcc hello.c -save-temps -o hello.exe`, а затем нажмите клавишу **Enter**, чтобы повторно скомпилировать программу и сохранить временные файлы.



2. Откройте файл *hello.i* в простом текстовом редакторе, например Блокнот (Notepad). Ваш исходный код окажется в самом конце файла, а перед ним будет располагаться код библиотеки `stdio.h`.

3. Теперь откройте файл *hello.s* в простом текстовом редакторе, чтобы увидеть предыдущий файл, преобразованный к низкоуровневым инструкциям ассемблера. Обратите внимание, насколько менее дружелюбным он кажется по сравнению с кодом на языке C.



Совет

Программы, написанные на языке ассемблера, могут работать быстрее, чем те, которые написаны на языке C, но их гораздо сложнее писать и обслуживать. Для традиционного программирования язык C приоритетнее.

Заключение

- Национальный Американский Институт Стандартов создал узнаваемый стандарт для языка программирования C.
- Другие языки программирования, такие как C++ и C#, как минимум частично созданы на базе языка C.
- Язык программирования C имеет несколько стандартных библиотек, содержащих проверенные годами функции, которые могут быть использованы в любой программе.
- Библиотеки языка C содержатся в заголовочных файлах, чьи имена имеют расширение *.h*.
- Программы на языке C создаются как простые текстовые файлы, чьи имена имеют расширение *.c*.
- Популярный компилятор GNU C Complier (GCC) можно установить с помощью пакета Minimalist GNU for Windows (MinGW).
- Добавление к системному пути каталога, где располагается компилятор, позволит последнему запускаться из любой директории.
- Программы могут иметь одну или более функций, содержащих утверждения, которые должны быть выполнены при вызове функции.
- Каждая программа, написанная на языке C, обязана иметь функцию `main()`.
- Объявление функций начинается с указания типа данных значения, что должно быть возвращено после выполнения функции.
- Утверждения, которые следует выполнить, содержатся в фигурных скобках {}, каждое утверждение обязано заканчиваться точкой с запятой.
- Инструкции препроцессора располагаются в начальной части программы и, как правило, будут заменены кодом библиотек.
- Компилятор GNU C Compiler запускается с помощью команды `gcc`, могущую включать в себя опцию `-o`, вместе с которой передается имя исполняемого файла.
- Временные файлы, созданные во время процесса компилирования, могут быть получены с помощью опции команды компилятора `-save-temps`.

2 Сохранение значений переменных

Здесь показывается, как сохранять, получать и манипулировать различными типами данных с использованием контейнеров переменных в программах на языке C.

- **Создание переменных в программе**
- **Отображение значений переменных**
- **Ввод значений переменных**
- **Спецификаторы типов данных**
- **Использование глобальных переменных**
- **Размещение переменных в регистрах**
- **Преобразование типов данных**
- **Создание массивов переменных**
- **Описание нескольких измерений**
- **Заключение**

Создание переменных в программе

Внимание

В языке C имена переменных чувствительны к регистру — переменные `VAR`, `Var` и `var` будут рассматриваться как три разные переменные.

Переменная — это контейнер в программе, написанной на языке C, с помощью которого в памяти компьютера могут быть сохранены данные. К сохраненному значению можно обратиться с помощью имени переменной. Программисту позволено выбрать любое имя для переменной, соответствующее соглашениям, перечисленным в таблице ниже.

Соглашение	Пример
Не может содержать ключевые слова языка C	`Volatile`
Не может содержать арифметические операции	`a+b*c`
Не может содержать знаки пунктуации	`%$#@!`
Не может содержать пробелы	`no spaces`
Не может начинаться с цифры	`2bad`
Может содержать цифры в любом другом месте	`good1`
Может содержать буквы в разных регистрах	`UPdown`
Может содержать подчеркивания	`is_ok`

На заметку

Ключевые слова языка C перечислены в конце книги.

Переменным принято давать осмысленные имена, чтобы код программы был более доступен для понимания. Для того чтобы создать переменную, необходимо просто *объявить* ее. Объявление переменной выглядит так:

```
тип-данных имя-переменной;
```

При объявлении переменной вначале указывается тип данных, которые будут храниться в переменной. Можно выбрать один из типов, описанных на следующей странице. После типа данных следует пробел, а затем — выбранное имя переменной, соответствующее соглашениям, представленным выше. Как и любое другое утверждение языка C, объявление переменной должно заканчиваться точкой с запятой. Несколько переменных, имеющих один тип данных, могут быть объявлены одновременно следующим образом:

```
тип-данных имя-переменной1, имя-переменной2, имя-переменной3;
```

В языке C существуют четыре простых типа данных. Они определяются с помощью ключевых слов языка C и перечислены вместе с их описаниями в следующей таблице.

Тип данных	Описание	Пример
`char`	Один байт, в котором сохраняется один символ	`'A'`
`int`	Целое число	`100`
`float`	Число с плавающей точкой с точностью до шести знаков	`0.123456`
`double`	Число с плавающей точкой с точностью до десяти знаков	`0.123456789`

Четыре типа данных выделяют разные объемы машинной памяти для хранения данных. Самый малый объем — у типа `char`, для этого типа выделяется всего один байт памяти. Самый большой объем выделяется для типа `double` — как правило, восемь байт. Этот объем памяти в два раза превышает тот, что выделяется для типа `float`, поэтому тип `double` должен использоваться только в том случае, если это действительно необходимо.

Внимание

Значения типа `char` должны быть заключены в одинарные кавычки. Использование двойных кавычек некорректно.

Объявление переменных должно происходить до того, как в программе появится код, который следует выполнить. Когда значение назначается переменной говорят, что переменная была *инициализирована*. Иногда переменная бывает инициализирована в момент объявления.

В примере кода, приведенном ниже, различные переменные объявляются, а затем и инициализируются подходящими значениями, что описано в комментариях к коду — описательных текстах, заключенных между символами `/*` и `*/`, которые компилятор игнорирует:

```
int num1, num2;          /*объявляем две целочисленные переменные*/
char letter;             /*объявляем символьную переменную*/
float decimal = 7.5;     /*объявляем и инициализируем переменную с плавающей точкой*/
num1 = 100;              /*инициализируем целочисленные переменные*/
num2 = 200;
letter = 'A';            /*инициализируем символьную переменную*/
```

Совет

При присвоении значения переменной левая часть операции `=` называется объектом L-значения, а правая — данными R-значения.

Отображение значений переменных

Значение переменной может быть отображено с помощью функции `printf()`, которая уже была использована нами в главе 1 для отображения сообщения Hello World. Формат отображения значения переменной должен быть определен как аргумент функции `printf()`, располагающийся в скобках — сначала следует «спецификатор формата», а затем — имя переменной.

Спецификатор	Описание	Пример
`%d`	Целое число от -32768 до +32767	`100`
`%ld`	Длинное целое число от -2^{31} до $+2^{31}$	`123456789`
`%f`	Число с плавающей точкой	`0.123456`
`%c`	Один символ	`'A'`
`%s`	Строка символов	`"Hello World"`
`%p`	Адрес в памяти компьютера	`0x0022FF34`

На заметку

Единичные символы должны обрамляться в одинарные кавычки, а строки — в двойные.

Спецификатор формата позволяет убедиться, что выходные данные займут определенный объем места, если указать число сразу же после символа `%`. Например, чтобы убедиться, что целое число всегда будет иметь как минимум семь разрядов, следует использовать спецификатор `%7d`. Если необходимо заполнить все свободное пространство нулями, между спецификатором и числом следует добавить `0`. Например, чтобы убедиться, что целое число всегда будет как минимум семь разрядов, и при этом свободные разряды окажутся заполнены нулями, следует использовать спецификатор `%07d`.

Спецификатор точности, который представляет собой точку и число, может быть использован со спецификатором `%f` для указания отображаемого количества знаков после запятой. Например, чтобы отобразить только два разряда, следует использовать спецификатор `%.2f`. Спецификатор точности может быть использован вместе со спецификатором минимального пустого пространства, чтобы управлять как минимальным объемом пустого места, так и количеством разрядов числа. Например, чтобы отобразить семь разрядов числа, включая два после запятой, и заполнить пустые разряды нулями, используйте идентификатор `%07.2f`.

По умолчанию пустые разряды следуют перед самим числом, соответственно, число имеет выравнивание по правому краю. Однако пустые разряды могут быть добавлены и слева. Это достигается путем добавления символа к спецификатору.

1. Начните новую программу с инструкции препроцессора, чтобы включить в код стандартные библиотечные функции ввода/вывода.

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

vars.c

2. Добавьте функцию **main**, в которой объявляются и инициализируются две переменные:

```
int main()
{
int num = 100;
double pi = 3.1415926536;
}
```

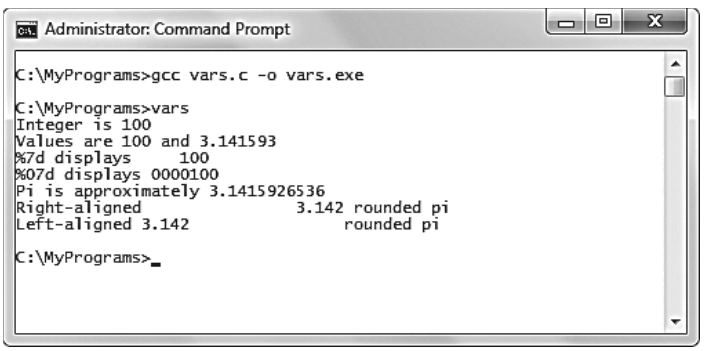
3. В функции **main** после объявления переменных добавьте утверждения, позволяющие вывести значения переменных в разных форматах.

```
printf("Integer is %d \n", num);
printf("Values are %d and %f \n", num, pi);
printf("%%7d displays %7d \n", num);
printf("%%07d displays %07d \n", num);
printf("Pi is approximately %1.10f \n", pi);
printf("Right-aligned %20.3f rounded pi \n", pi);
printf("Left-aligned %-20.3f rounded pi \n", pi);
```

4. В конце функции **main** добавьте финальное утверждение, которое возвращает 0, чего требует объявление функции.

```
return 0;
```

5. Теперь сохраните файл и затем в командной строке скомпилируйте и запустите программу, чтобы увидеть, как значения переменных будут выведены в разных форматах.



Совет

Чтобы отобразить символ % с помощью функции printf(), добавьте перед ним еще один символ %, как это показано в примере.

Внимание

Обратите внимание, что в случае, если указано меньше десятичных разрядов, чем того требует число, значение с плавающей точкой будет округлено, а не обрезано.

Ввод значений переменных

Стандартная библиотека функций ввода-вывода `stdio.h` предоставляет функцию `scanf()`, которая может использоваться для получения данных от пользователя. Функция `scanf()` требует, чтобы в нее передавали два аргумента, определяющие тип данных и место, где они должны быть сохранены.

Совет

Строки — это особый случай. Работа с ними продемонстрирована на странице 36 в разделе, посвященном работе с массивами.

Первый аргумент функции `scanf()` должен быть одним из спецификаторов формата из таблицы, приведенной на странице 22, он помещается в двойные кавычки. Например, `"%d"`, если вводится целочисленное значение. Вторым аргументом функции `scanf()` должно быть имя переменной, перед которым стоит символ `&`, если только вводится не строка. Символ `&` имеет несколько применений в программировании на языке C, но в этом контексте он используется как операция адресации, что значит, что вводимые данные должны храниться в том участке памяти, который был зарезервирован для этой переменной.

При объявлении переменной в памяти компьютера резервируется место для хранения данных, записанных в эту переменную. Количество байт зависит от типа переменной. К выделенной памяти можно обратиться, используя уникальное имя переменной.

Представьте, что память компьютера — это очень большой ряд ячеек. Каждая ячейка имеет уникальный адрес, который записывается в шестнадцатеричном формате. Это похоже на очень большой ряд домов — в каждом доме содержатся люди, он имеет уникальный адрес. В программах, написанных на языке C, дома — это ячейки, а люди — это данные, расположенные по этим адресам.

Внимание

Обратите внимание на то, что функция `scanf()` прекращает считывание введенных значений, если встретит пробел.

Функция `scanf()` может назначать значения нескольким переменным одновременно. Для этого в первый аргумент помещается несколько спецификаторов формата, разделенных пробелами, а весь список должен располагаться в двойных кавычках. Второй аргумент должен содержать разделенный запятыми список имен переменных, перед каждым из которых следует поставить операция адресации `&`. Имена переменных следует располагать соответственно указанным форматам.

Операция адресации `&` также может использоваться для того, чтобы возвращать шестнадцатеричный адрес, по которому хранится переменная.

Функции `printf()` и `scanf()` позволяют программам взаимодействовать с пользователем.

1. Начните новую программу с инструкции препроцессора, включающей стандартную библиотеку функций ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляются три переменные.

```
int main()

{

    char letter;

    int num1, num2;

}
```

3. Далее в блоке функции **main** после объявления переменных добавьте утверждения, позволяющие пользователю ввести данные.

```
printf("Enter any one keyboard character");

scanf("%c", letter);

printf("Enter two integers separated by a space:");

scanf("%d, %d", &num1, &num2);
```

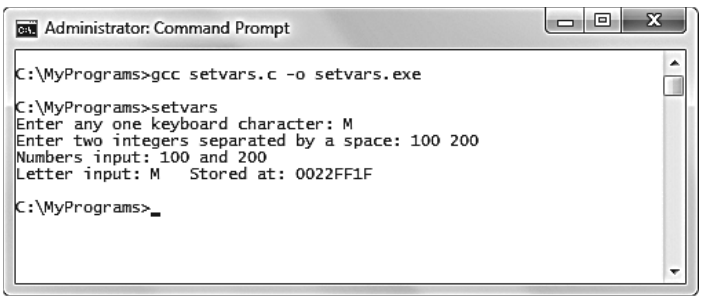
4. Теперь добавьте утверждения, выводящие на экран сохраненные данные.

```
printf("Numbers input: %d and %d \n", num1, num2);

printf("Letter input: %c", letter);

printf("Stored at: %p \n", &letter);
```

5. В конце блока функции **main** верните значение 0, чего требует объявление функции.

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и выполните программу, введя требуемые данные, а затем проверьте сохраненные данные.



setvars.c

На заметку

Спецификатор формата для адреса в памяти компьютера: `%p`.

Спецификаторы типов данных

Числовая переменная типа **int** способна содержать положительные и отрицательные значения. Они называются *знаковыми*. Диапазон допустимых значений может определяться системой как *длинный* или *короткий*.

Если переменная типа **int** по умолчанию создается как «длинная» (что более вероятно), она, как правило, может иметь значения от –2.147.483.648 до +2.147.483.647.

С другой стороны, если переменная типа **int** создается как «короткая» (что менее вероятно), она, как правило, способна иметь значения от –32.768 до +32.767.

Размер диапазона может быть указан явно при помощи ключевых слов-спецификаторов **short** и **long** в объявлении переменной, например, так:

```
short int num1; /*позволяет сэкономить память*/

long int num2; /*позволяет работать с крупными числами*/
```

Библиотечный заголовочный файл **limits.h** содержит определяемые реализацией размеры каждого типа данных. Они доступны через константы. Для переменных типа **int**, размер которых не указан, это **INT_MAX** и **INT_MIN**. Аналогично для переменных типа **short int** это переменные **SHRT_MAX** и **SHRT_MIN**, для переменных типа **long int** — **LONG_MAX** и **LONG_MIN**.

Неотрицательные *беззнаковые* переменные типа **int** могут быть объявлены с помощью ключевого слова-спецификатора **unsigned** в случае, если переменная никогда не получит отрицательное значение. Переменная типа **unsigned short int**, как правило, будет иметь диапазон от 0 до 65.535, при этом она займет столько же места в памяти, сколько и обычная переменная типа **short int**. Переменная типа **unsigned long int**, как правило, будет иметь диапазон от 0 до 4.294.967.295, при этом она займет столько же места в памяти, сколько и обычная переменная типа **long int**.

Операция **sizeof** языка C может быть использована для того, чтобы узнать, какой объем памяти резервируется под переменные разного типа. Рекомендуется использовать минимально возможный объем памяти. Например, если переменная будет содержать только положительные значения меньше 65.535, предпочтительнее использовать тип данных **unsigned short int**, который резервирует только 2 байта памяти, вместо **long int**, резервирующего 4 байта.

Совет

Имена констант всегда пишутся в верхнем регистре, что позволяет отличать их от переменных.

На заметку

Объем памяти, выделяемый для переменной типа **int** неопределенного размера, зависит от настроек системы, но часто по умолчанию выделяется столько же памяти, сколько и для переменной типа **long int**.

1. Начните новую программу с инструкций препроцессора, включающих стандартную библиотеку функций ввода/вывода и константы.

sizeof.c

```
#include <stdio.h>

#include <limits.h>
```

2. Добавьте функцию **main**, в которой содержатся утверждения, позволяющие вывести на экран размер и диапазон типа данных **short int**.

```
int main()

{

    printf("short int… \tsize: %d bytes %d байт \t", sizeof(short int));

    printf("от %d до %d", SHRT_MAX, SHRT_MIN);

}
```

3. Далее добавьте утверждения, позволяющие вывести на экран размер и диапазон типа данных **long int**.

```
printf("long int… \tsize: %d bytes %d байт \t", sizeof(long int));

printf("от %d до %d", LONG_MAX, LONG_MIN);
```

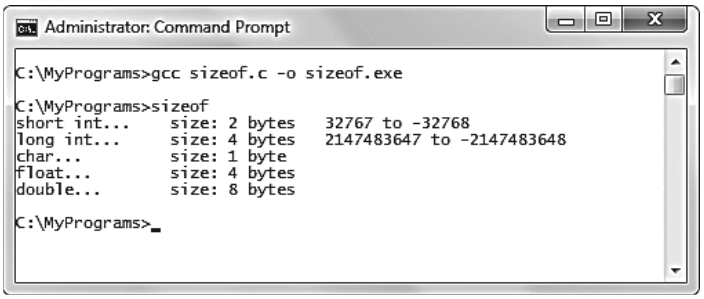
4. Теперь добавьте утверждения, выводящие на экран размер других типов данных.

```
printf("char… \tsize: %d bytes %d байт \n", sizeof(char));

printf("float… \tsize: %d bytes %d байт \n", sizeof(float));

printf("double… \tsize: %d bytes %d байт \n", sizeof(double));
```

5. Добавьте финальное утверждение, чтобы вернуть значение 0, чего требует объявление функции.

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и выполните программу, чтобы увидеть размеры и диапазоны типов данных.



Совет

Обратите внимание на то, как в данном примере используется управляющая последовательность символов табуляции \t.

Использование глобальных переменных

Фрагменты программы, в которых доступны переменные, называются *областью видимости*. Переменные, объявленные внутри функции, называются *локальными* переменными, а переменные, объявленные вне функций — *глобальными*.

Внимание

Широкое использование глобальных переменных может вызвать конфликты имен.

Локальные переменные могут быть использованы только внутри той функции, в которой они объявлены. Это поведение применяется по умолчанию, однако его можно использовать намеренно с помощью редко используемого ключевого слова `auto` в объявлении переменной. Такие переменные появляются во время вызова функции и, как правило, исчезают, как только функция отработает.

Глобальные переменные, с другой стороны, могут использоваться внутри любой функции программы. Они появляются во время запуска программы и существуют в течение всего времени выполнения программы.

Внешние глобальные переменные должны быть объявлены всего один раз в начале программы. Они также должны быть объявлены внутри каждой функции, которая будет их использовать. Такое объявление должно начинаться с ключевого слова `extern`, что указывает, что переменная является внешней, а не локальной. Подобные объявления не следует использовать для объявления глобальных переменных.

На заметку

Постарайтесь использовать только локальные переменные. Глобальные переменные удобно использовать для получения их значений из любой функции программы, но рекомендуется избегать глобальных переменных и передавать значения между функциями как аргументы.

Крупные программы на языке C часто состоят из нескольких файлов исходного кода, которые компилируются вместе для создания одного исполняемого файла. Глобальные переменные обычно доступны из любой функции любого файла программы. Все функции, как правило, также доступны глобально. Но область действия функций и, что более типично, глобальных переменных может быть ограничена только тем файлом, в котором они располагаются, с помощью размещения в объявлении ключевого слова `static`.

Обычно программа не способна использовать несколько переменных с одинаковым именем, но это становится возможным, если каждая из таких переменных объявлена с помощью ключевого слова `static` и является уникальной в текущем файле исходного кода. Создание программы, использующей несколько файлов, может привести к тому, что имя одной глобальной переменной будет использоваться в двух разных файлах. Использование ключевого слова `static` позволяет избежать переименования одной из этих переменных в исходном коде.

Внутренние глобальные переменные также могут быть объявлены с использованием ключевого слова `static`. Они станут доступны только внутри функции, в которой они были объявлены, но не пропадут

по завершении работы функции. Это позволяет создать перманентное частное хранилище внутри функции, которое будет существовать до конца работы программы.

1. Начните новую программу с инструкций препроцессора, включающих стандартную библиотеку функций ввода/вывода.

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

global_1.c

2. Определите и инициализируйте глобальную статическую переменную, получить доступ к которой можно только из текущего файла исходного кода.

```
static int sum = 100;
```

3. Добавьте функцию **main**, в которой объявляется необходимость использовать глобальную статическую переменную, а также выводится ее значение.

```
int main()
{
    extern int sum;
    printf("Sum is %d \n", sum);
}
```

Совет

Ключевое слово **static**, добавленное к объявлению переменной **num**, ограничит ее область действия рамками только этого файла, что приведет к ошибке компилирования.

4. Далее добавьте объявление второй глобальной переменной и выведите ее значение.

```
extern int num;
printf("Num is %d \n", num);
```

5. Добавьте финальное утверждение, чтобы вернуть значение 0, чего требует объявление функции, а затем сохраните файл.

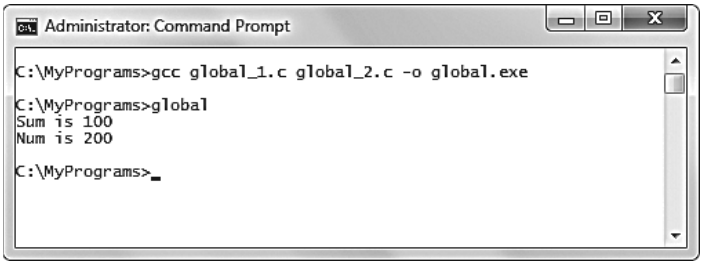
```
return 0;
```

6. В отдельном файле объявите вторую глобальную переменную, а затем сохраните этот файл.

```
int num = 200;
```

global_2.c

7. Скомпилируйте оба файла исходного кода в один исполняемый файл, передав имя каждого из них команде компилирования, а затем запустите программу, чтобы увидеть на экране значения глобальных переменных.



Размещение переменных в регистрах

Объявление переменной, которое включает в себя ключевое слово **register**, указывает компилятору, что эта переменная будет часто использоваться в программе. Это делается для того, чтобы компилятор разместил переменные, зарегистрированные таким способом, в регистрах компьютера, чтобы ускорить доступ к ним. Полезность этого действия довольно сомнительная, поскольку компиляторы могут свободно проигнорировать это указание.

С использованием ключевого слова **register** могут быть объявлены только внутренние локальные переменные. В любом случае, таким способом могут быть зарегистрированы только несколько переменных, и они могут иметь только определенные типы. Точные ограничения могут варьироваться от компьютера к компьютеру.

Несмотря на возможные недостатки, использование ключевого слова **register** безвредно, поскольку это ключевое слово будет проигнорировано, если компилятор не сможет поместить переменные в регистры. Вместо этого переменные создаются обычным образом, как если бы ключевое слово **register** не было указано.

Полной противоположностью ключевого слова **register** является ключевое слово **volatile**. Это значит, что переменная не должна помещаться в регистры, поскольку ее значение способно измениться в любой момент даже без участия кода, окружающего их. Это бывает важно для глобальных переменных в крупных программах, которые могут изменяться сразу несколькими потоками.

Использование ключевого слова **register** может оказаться полезным для локальных переменных, применяющихся для хранения управляющего значения в цикле. На каждой итерации цикла происходит обращение к переменной, которая хранит управляющее значение. Хранение этого значения в регистре способно ускорить выполнение цикла.

С другой стороны, если для хранения такого значения используется глобальная переменная, которая может быть изменена за пределами цикла, рекомендуется использовать ключевое слово **volatile**.

Структура циклов описана в главе 5, но пример, приведенный далее, включен для того, чтобы продемонстрировать повторяющиеся обращения к переменной, объявленной с помощью ключевого слова **register**.

На заметку

Ключевые слова **register** и **volatile** описаны здесь лишь для полноты картины — в действительности они редко используются, поскольку в большинстве программ для хранения данных используются обычные переменные.

1. Начните новую программу с инструкций препроцессора, включающих стандартную библиотеку функций ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляется и инициализируется переменная с использованием ключевого слова **register**. Эта переменная содержит управляющее значение цикла, равное 0.

```
int main()

{

register int num = 0;

}
```

3. Теперь добавьте условие, позволяющее проверить, не превысило ли управляющее значение число 5, а затем пару фигурных скобок.

```
while (num < 5)

{                    }
```

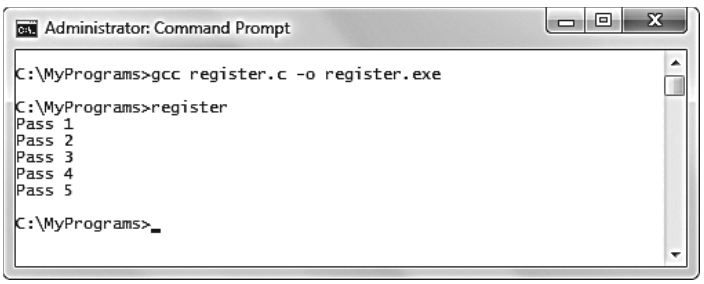
4. Между скобками цикла добавьте утверждение, позволяющее в каждой итерации цикла увеличить управляющее значение и вывести его на экран.

```
++num;

printf("Pass %d \n", num);
```

5. В конце блока функции **main** верните значение 0, чего требует объявление функции, а затем сохраните файл.

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и запустите ее, чтобы увидеть на экране значение переменной, размещенной в регистре, на каждой итерации цикла.



register.c

Внимание

Если для переменной, объявленной с помощью ключевого слова **register**, использовать операцию **&**, компилятор разместит переменную в памяти компьютера, а не в регистре.

Преобразование типов данных

Компилятор будет неявно заменять один тип данных другим, если это кажется логичным. Например, если переменной типа `float` назначено целочисленное значение, компилятор преобразует это значение к типу `float`.

Любые данные, хранящиеся в переменной, могут быть явно записаны в переменную, имеющую другой тип данных, с помощью процесса, называющегося *приведением*. При приведении переменной необходимо указать в скобках перед ее именем тип данных, к которому переменная должна быть приведена. Также в скобках можно разместить любые модификаторы, такие как `unsigned`, где это необходимо. Синтаксис приведения выглядит так:

```
имя-переменной2 = (модификаторы тип-данных) имя-переменной1;
```

Обратите внимание — приведение не изменяет исходный тип данных переменной, выполняется копирование ее значения в переменную, имеющую другой тип данных.

Если переменная типа `float` приводится к переменной типа `int`, ее значение просто обрезается после запятой — никакого округления не выполняется. Например, если выполнить приведение переменной `float num = 5.75`, результатом будет значение `5`, а не `6`.

Если переменная типа `char` приводится к переменной типа `int`, результатом окажется ее ASCII-код, который соответствует символу. Символы в верхнем регистре располагаются в диапазоне 65-90, а символы в нижнем регистре — 97-122. Например, если выполнить приведение переменной `char letter = 'A'`, результатом будет значение `65`. Приведение целочисленных переменных к типу `char` создаст соответствующий символ.

Довольно часто необходимо выполнить приведение целочисленных значений к типу `float`, чтобы выполнить точные подсчеты — в противном случае деление может создать «обрезанный» целочисленный результат. Например, если объявить переменные `int x=7, y=5`, а затем выполнить операцию деления `float z = x / y`, результатом будет `1.000000`. Для получения более точного результата необходимо использовать приведение `(float) x / (float) y`, что приведет к получению результата `1.400000`.

Приведение длинных точных чисел с плавающей точкой от типа `double` к более «короткому» типу `float` приводит не к обрезанию длинного зна-

Совет

ASCII (произносится как «аски») — это акроним «American Standard Code for Information Interexchange» («American Standard Code for Information Interexchange»), он является стандартом для простого текста. Вы можете просмотреть весь диапазон стандартных ASCII-кодов в конце книги.

Совет

Прямой слеш, /, представляет собой операцию деления в языке C.

чения, а к его округлению в шестом знаке после запятой. Например, приведение переменной `double decimal = 0.1234569` даст результат `0.123457`.

cast.c

1. Начните новую программу с инструкций препроцессора, включающих стандартную библиотеку функций ввода/вывода.

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

2. Добавьте функцию `main`, в которой объявляются и инициализируются переменные, имеющие разные типы данных.

```
int main()

{

    float num = 5.75;

    char letter = 'A';

    int zee = 90;

    int x = 7, y = 5;

    double decimal = 0.1234569;

}
```

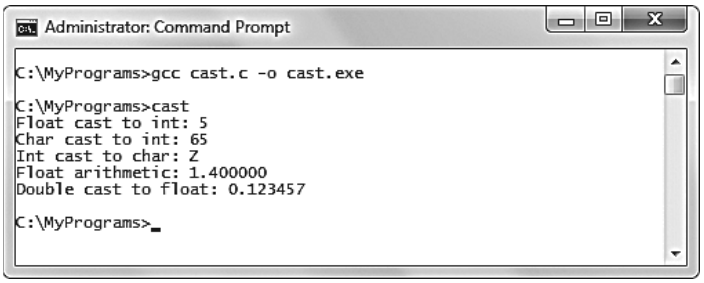
3. Теперь добавьте утверждение, позволяющее вывести результат преобразования каждой переменной.

```
printf( "Float cast to int: %d \n" , (int)num ) ;

printf( "Char cast to int: %d \n" , (int)letter ) ;

printf( "Int cast to char: %c \n" , (char)zee ) ;

printf( "Float arithmetic: %f \n" , (float)x / (float)y ) ;

printf( "Double cast to float: %f \n" , (float)decimal ) ;
```

4. В конце блока функции `main` верните значение 0, чего требует объявление функции, а затем сохраните файл.

```
return 0;
```

5. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть на экране результат каждого приведения.



На заметку

Значения типа `float`, приведенные к типу `int`, обрезаются, а значения типа `double`, приведенные к типу `float` — округляются.

Создание массивов переменных

Массив переменных может хранить несколько элементов данных в отличие от обычных переменных, которые могут хранить только один элемент.

Внимание

Элементы в массиве нумеруются с нуля, а не с единицы.

Элементы данных хранятся последовательно в «ячейках» массива, которые нумеруются начиная с нуля. Первый элемент массива хранится в ячейке с номером 0, второй — в ячейке с номером 1, и т. д.

Массив с программой на языке C объявляется как и обычная переменная, однако в объявлении следует также указать размер массива (количество его ячеек). Размер указывается внутри квадратных скобок после имени массива, синтаксис выглядит так:

```
тип-данных имя-массива [количество-элементов]
```

Опционально массив может быть инициализирован при объявлении, значения каждого элемента оформляются как разделенный запятыми список, обрамленный фигурными скобками. Например, массив из трех целочисленных элементов может быть создан и проинициализирован следующим образом: `int arr[3] = {1, 2, 3};`

К элементу массива допустимо обратиться с помощью имени массива, после которого ставятся квадратные скобки, содержащие номер элемента. Например, запись `arr[0]` позволит обратиться к первому элементу.

При объявлении массива можно опустить число элементов массива, указываемое в квадратных скобках — в таком случае размер массива будет автоматически подогнан в соответствии с количеством значений, помещаемых в массив.

Одна из наиболее используемых особенностей массива при программировании на языке C заключается в том, что они могут хранить текстовые строки. Каждый элемент массива типа `char` способен хранить один символ. Добавление специального символа-терминатора `\0` в конец массива «повышает» его статус до строки. Например, строка может быть создана и проинициализирована так: `char str[4] = {'C', 'a', 't', '\0'} ;`.

На заметку

При создании массива для строки не забудьте оставить место для символа `\0`.

Использование массива символов, являющегося строкой, позволяет обратиться ко всей строке с помощью всего одного имени массива. Строка также может быть отображена с помощью функции `printf()` и спецификатора формата `%s`. Например, отобразить строку `str` можно так: `printf("%s", str);`.

1. Начните новую программу с инструкций препроцессора, включающих стандартную библиотеку функций ввода/вывода.

array.c

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

2. Добавьте функцию **main**, в которой объявляется массив из трех целочисленных элементов.

```
int main()
{
    int arr[3];
}
```

3. Теперь добавьте утверждения, инициализирующие массив целочисленными элементами, индивидуально назначая значение каждого элемента.

```
arr[0] = 100;
arr[1] = 200;
arr[2] = 300;
```

4. Теперь добавьте утверждение, создающее и инициализирующее массив символов, позволяющий хранить текстовую строку.

```
char str[10] = {'C', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
```

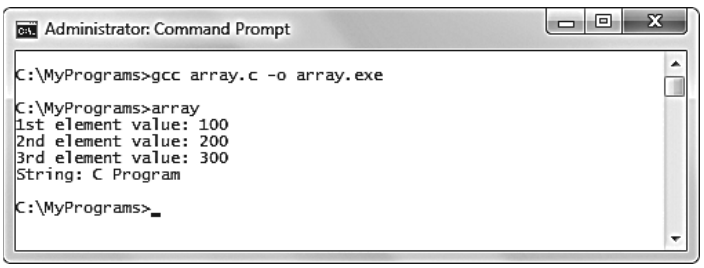
5. Далее добавьте утверждения, позволяющие вывести все элементы целочисленного массива и строки из массива символов.

```
printf( "1st element value: %d \n" , arr[0] ) ;
printf( "2nd element value: %d \n" , arr[1] ) ;
printf( "3rd element value: %d \n" , arr[2] ) ;
printf( "String: %s \n" , str ) ;
```

6. В конце блока функции **main** верните значение 0, чего требует объявление функции, а затем сохраните файл.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть на экране значения, сохраненные в массивах переменных.



Совет

При создании массива для каждого элемента память выделяется соответственно его типу — один байт для каждого элемента типа **char**. Размер массива не может быть изменен динамически.

Описание нескольких измерений

Нумерация элементов массива часто упоминается как «*индекс массива*», и, поскольку нумерация элементов начинается с нуля, иногда используется выражение «*основанный на нуле индекс*». Массив с одним индексом является одномерным, его элементы размещаются в одном ряду.

Содержимое ячейки…	A	B	C	D	E
Номера индексов…	[0]	[1]	[2]	[3]	[4]

Массивы также могут иметь несколько индексов, такие массивы являются многомерными. Массив, созданный с двумя индексами, называется двухмерным, его элементы располагаются в нескольких рядах:

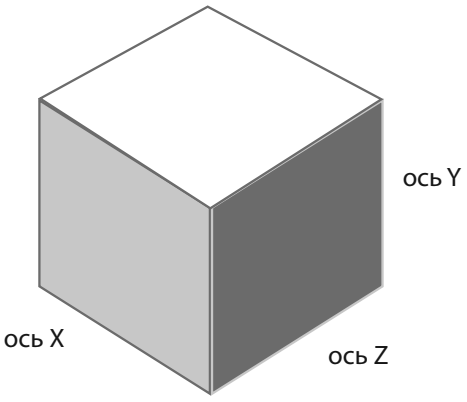
Первый индекс	[0]	A	B	C	D	E
	[1]	F	G	H	I	J
Второй индекс		[0]	[1]	[2]	[3]	[4]

В многомерных массивах к значениям, содержащимся в каждой ячейке, можно обратиться, указав номер каждого индекса. Например, в случае двухмерного массива, приведенного выше, элемент по адресу [1][2] содержит букву «H».

Внимание

Использование массивов, имеющих более двух индексов, может привести к трудностям при чтении кода и к ошибкам.

Двухмерные массивы годятся для хранения информации в форме сети, такой как координаты по осям X и Y. Создание трехмерного массива, имеющего три индекса, позволит хранить трехмерные координаты:



Использование большего числа индексов позволит создавать массивы с большим количеством измерений, но в действительности массивы

более чем с тремя измерениями, применяются редко, поскольку их трудно визуализировать.

1. Начните новую программу с инструкций препроцессора, включающих стандартную библиотеку функций ввода/вывода.

matrix.c

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

2. Добавьте функцию **main**, в которой объявляется двухмерный массив целочисленных элементов.

```
int main()

{

    int matrix[2][3] = {{'A', 'B', 'C'}, {1, 2, 3}};

}
```

3. Теперь добавьте утверждения, позволяющие отобразить содержимое всех элементов первого индекса.

```
printf( "Element [0][0] contains %c \n" , matrix[0][0] ) ;

printf( "Element [0][1] contains %c \n" , matrix[0][1] ) ;

printf( "Element [0][2] contains %c \n" , matrix[0][2] ) ;
```

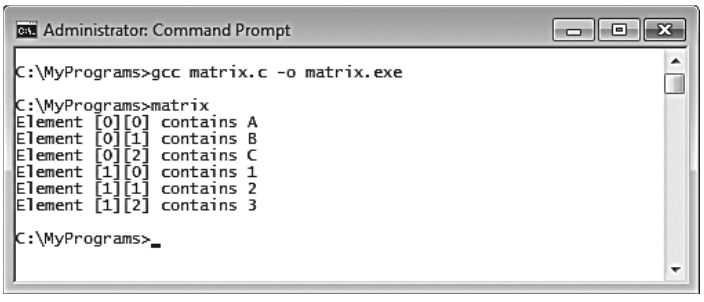
4. Теперь добавьте утверждения, позволяющие отобразить содержимое всех элементов второго индекса.

```
printf( "Element [1][0] contains %d \n" , matrix[1][0] ) ;

printf( "Element [1][1] contains %d \n" , matrix[1][1] ) ;

printf( "Element [1][2] contains %d \n" , matrix[1][2] ) ;
```

5. В конце блока функции **main** верните значение **0**, чего требует объявление функции, а затем сохраните файл.

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть на экране все значения, сохраненные в двухмерном массиве элементов.



Совет

Обратите внимание на то, как с помощью неявного приведения ASCII-коды букв сохраняются в массиве целочисленных элементов, но спецификатор формата **%c** позволяет отобразить их как буквы.

Заключение

- Переменная является контейнером в программе, написанной на языке C, с помощью которого данные могут быть сохранены в памяти компьютера.
- Имена переменных должны соответствовать соглашениям языка C об именовании.
- Четыре базовых типа данных в языке C это `char`, `int`, `float` и `double`.
- Спецификаторы формата `%d`, `%f`, `%c`, `%s` и `%p` могут использоваться в функции `printf()`, чтобы отобразить на экране значения переменных разных типов.
- Данные, введенные пользователем, разрешается помещать в переменные с помощью функции `scanf()`.
- Допустимый диапазон значений целочисленных переменных может быть явно указан с помощью ключевых слов `short` и `long`.
- Переменные, в которые никогда не будет записано отрицательное значение, разрешается пометить с помощью ключевого слова `unsigned`, чтобы расширить диапазон возможных положительных значений.
- Количество байт памяти, резервируемое для каждой переменной, можно узнать с помощью операции `sizeof`.
- Область действия описывает доступность переменных — они могут быть локальными и глобальными.
- Ключевое слово `extern` указывает, что переменная объявлена вне блока кода, где она используется, а ключевое слово `static` ограничивает доступность переменной тем файлом, где она описана.
- Производительность часто используемых переменных может быть повышена с помощью ключевого слова `register`, указывающего компилятору поместить переменную в регистры процессора.
- Переменные, которые не следует помещать в регистры, могут быть отмечены ключевым словом `volatile`.
- Компилятор способен явно изменять типы данных там, где это логично, или же тип данных может быть изменен явно путем выполнения приведения.
- Массивы переменных могут хранить несколько элементов данных внутри своих ячеек, которые пронумерованы последовательно начиная с нуля.
- Доступ к значениям массивов можно получить, если после имени массива в квадратных скобках указать номер требуемой ячейки.
- Массивы могут иметь несколько индексов, что позволяет создавать многомерные массивы.

Установка значений переменных

Здесь показано, как создать и использовать константные значения и типы внутри программы, написанной на языке C.

- **Объявление констант в программе**
- **Перечисление значений констант**
- **Создание константного типа**
- **Определение констант**
- **Отладка с помощью определений**
- **Заключение**

Объявление констант в программе

В случае, когда в программе требуется использовать константное значение, которое никогда не изменится, оно должно быть объявлено точно так же, как и обычная переменная, но с использованием в объявлении ключевого слова `const`. Например, константа, представляющая неизменяющееся число «миллион», может быть объявлена и инициализирована следующим образом:

```
const int MILLION = 1000000;
```

В объявлениях констант объект всегда должен инициализироваться. Программа не способна изменить исходное значение константы. Это охраняет его от случайностей — компилятор сообщит об ошибке, если программа попытается изменить исходное значение константы.

Имена констант должны соответствовать соглашениям именования переменных, приведенных на странице 20, но традиционно в именах констант используются символы в верхнем регистре, что позволяет с легкостью отличить их от имен переменных при чтении исходного кода.

Ключевое слово `const` также может быть использовано при объявлении массива, если все его элементы не будут изменяться во время работы программы.

`constant.c`

1. Начните новую программу с инструкции препроцессора, включающей стандартную библиотеку функций ввода/вывода.

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

2. Добавьте функцию `main`, в которой объявляется константное значение, приблизительно равное числу π.

```
int main()
{
    const float PI = 3.141593;
}
```

3. Далее в блоке функции `main` добавьте утверждения, объявляющие четыре переменные.

```
float diameter;
float radius;
float circ;
float area;
```

4. Теперь добавьте утверждения, требующие от пользователя ввести значение переменной.

```
printf("Enter the diameter of a circle in millimeters:");

scanf("%f", &diameter);
```

5. Добавьте утверждения, рассчитывающие значения трех остальных переменных с использованием константного значения и значения, введенного пользователем.

```
circ = PI * diameter;

radius = diameter / 2;

area = PI * (radius * radius);
```

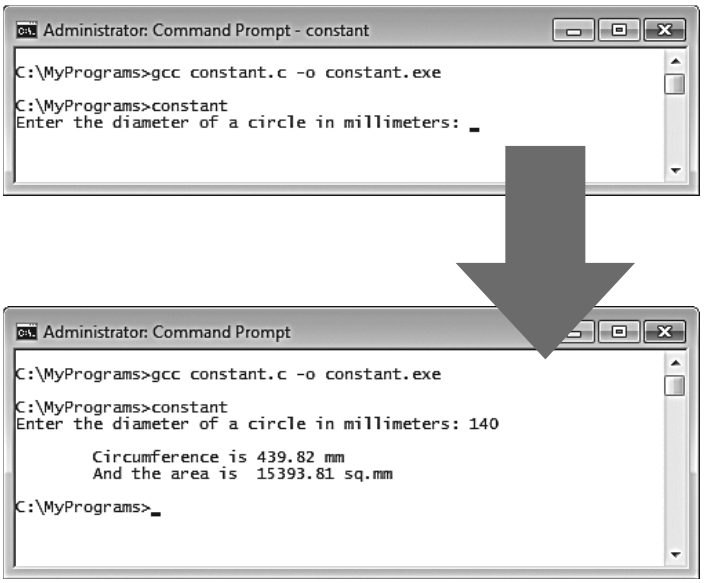
6. Теперь вставьте утверждения, позволяющие вывести рассчитанные значения с округлением до двух десятичных знаков.

```
printf( "\n\tCircumference is %.2f mm" , circ ) ;

printf( "\n\tAnd the area is %.2f sq.mm\n" , area ) ;
```

7. В конце блока функции **main** верните значение 0, чего требует объявление функции.

```
return 0;
```

8. Сохраните файл программы, а затем скомпилируйте и выполните ее, введя требуемые данные, чтобы увидеть рассчитанный результат



Совет

Здесь знак * является арифметической операцией умножения, а прямой слеш / — операцией деления. Арифметические операции рассматриваются более подробно в следующей главе.

На заметку

Помните, что операция адресации & должна следовать перед именем переменной при вызове функции scanf(), предназначенной для указания значений переменных.

Перечисление значений констант

Ключевое слово **enum** позволяет удобным способом создать последовательность числовых констант. Объявление последовательности может включать в себя имя, располагающееся после слова **enum**. Имена констант размещаются внутри скобок, разделенные запятыми.

Каждая константа по умолчанию будет иметь значение, превосходящее предыдущее на 1. Если значение первой константы не указано, она будет иметь значение 0, следующая — 1, и т. д. Например, именованные константы дней недели могут получить числовые значения, начиная с нуля — **enum {MON,TUE,WED,THU,FRI}**. В этом случае константа **WED** будет иметь значение 2.

Константе при объявлении может быть назначено любое числовое значение, но следующая константа в списке будет иметь значение, превосходящее предыдущее на 1, если только ей также не будет назначено отдельное значение. Например, для того, чтобы начать последовательность с 0, а не с 1, назначим первой константе значение 1 — **enum {MON=1,TUE,WED,THU,FRI}**. В этом случае константа **WED** будет иметь значение 3.

Список перечисленных констант также известен как «перечисление» и порой содержит повторяющиеся значения. Например, значение 0 может быть назначено константам с именами **NIL** и **NONE**.

В следующем примере перечисление представляет собой очки, начисляемые за шары при игре в снукер. Оно содержит необязательные имена переменных, записанные прописными буквами, поскольку они являются константами.

enum.c

1. Начните новую программу с инструкции препроцессора, включающей стандартную библиотеку функций ввода/вывода.

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

2. Добавьте функцию **main**, которая определяет и объявляет перечисление констант, чьи значения начинаются с единицы.

```
int main()

{

    enum SNOOKER

    { RED=1,YELLOW, GREEN, BROWN, BLUE, PINK, BLACK};

}
```

3. Далее в блоке функции **main** объявите числовую переменную, в которой будет храниться сумма нескольких значений констант.

```
int total;
```

4. Теперь добавьте утверждения, предназначенные для отображения значений некоторых перечисленных констант.

```
printf( "\nI potted a red worth %d\n" , RED ) ;

printf( "Then a black worth %d\n" , BLACK ) ;

printf( "Followed by another red worth %d\n" , RED ) ;

printf( "And finally a blue worth %d\n" , BLUE ) ;
```

5. Добавьте следующее утверждение, позволяющее рассчитать общую сумму константных значений, отображенных на предыдущем шаге.

```
total = RED + BLACK + RED + BLUE;
```

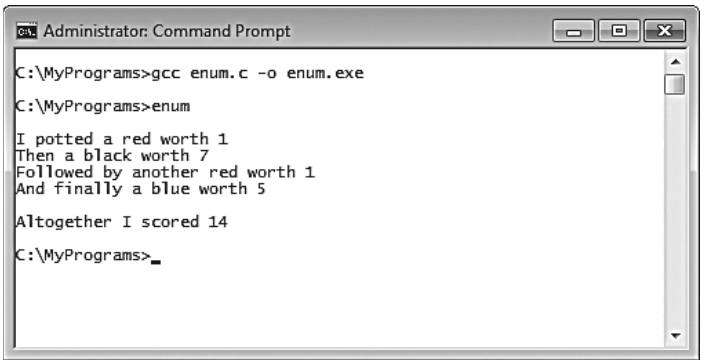
6. Теперь добавьте утверждение, предназначенное для вывода рассчитанной суммы значений.

```
printf( "\nAltogether I scored %d\n" , total ) ;
```

7. В конце блока функции **main** верните значение 0, чего требует объявление функции.

```
return 0;
```

8. Сохраните файл программы, а затем скомпилируйте и выполните программу, чтобы увидеть значения перечисленных констант и рассчитанную сумму.



Внимание

Присвоение значения 1 константе **BLUE** в этом примере перезапустит процесс инкрементирования — константа **PINK** будет иметь значение 2, а константа **BLACK** — 3.

На заметку

Имя перечисления (в этом случае **SNOOKER**) является опциональным, оно может быть опущено.

Создание константного типа

После того, как перечисление было объявлено, оно может быть рассмотрено как новый тип данных, его свойствами являются указанные имена констант и связанные с ними значения.

Переменные нашего типа **enum** могут быть объявлены точно так же, как и переменные любого другого типа данных — с помощью следующего синтаксиса:

```
тип-данных имя-переменной ;
```

В примере, приведенном на предыдущей странице, создается перечисление с именем **SNOOKER**, которое допустимо рассмотреть как тип данных **enum SNOOKER**. Поэтому переменная с именем **pair**» может быть создана с помощью объявления **enum SNOOKER pair**; она способна хранить значения перечисления, определяемого этим типом.

Для того чтобы явно назначить численное значение переменной подобного типа, стандарт языка С рекомендует приводить тип данных **int** к типу данных **enum**, например, так:

```
pair = (enum SNOOKER) 7;
```

На практике это делать необязательно, поскольку перечисленные значения всегда являются числами, и потому эквивалентны типу данных **int**.

С помощью объявления **enum** также можно создать переменную, указав имя переменной после последней скобки. Например, объявление **enum BOOLEAN {FALSE, TRUE} flag**; определяет тип данных **enum** и создает переменную **flag** этого типа.

Совет

Хотя это необязательно, явное указание значений перечисляемого типа служит для напоминания его типа.

Пользовательские типы данных могут быть определены с помощью ключевого слова **typedef** и следующего синтаксиса:

```
typedef определение имя-типа ;
```

Объявления пользовательских типов данных помогут сделать код программы более понятным. Например, если в программе часто объявляются переменные с типом **unsigned short int**, можно создать пользовательский тип данных, имеющий такие же модификаторы, например так:

```
typedef unsigned short int USINT;
```

Каждая переменная с типом **unsigned short int USINT** может быть объявлена с использованием типа данных **USINT** вместо **unsigned short int**.

1. Начните новую программу с инструкции препроцессора, включающей стандартную библиотеку функций ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляется и инициализируется перечисление констант, чьи значения начинаются с единицы.

```
int main()

{

    enum SNOOKER

    { RED=1,YELLOW, GREEN, BROWN, BLUE, PINK, BLACK};

}
```

3. Далее в блоке функции **main** объявите и проинициализируйте переменную определенного типа **enum**, а затем отобразите ее значение.

```
enum SNOOKER pair = RED + BLACK;

printf( "Pair value: %d \n" , pair ) ;
```

4. Теперь добавьте утверждение, предназначенное для создания пользовательского типа данных.

```
typedef unsigned short int USINT;
```

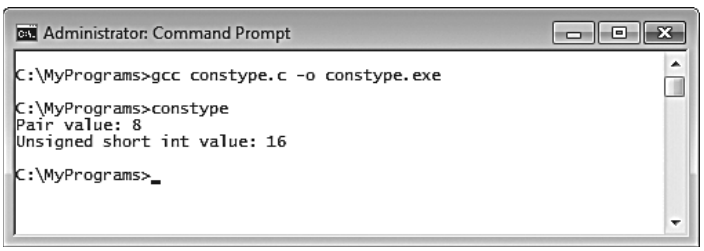
5. Далее объявите и проинициализируйте переменную пользовательского типа данных и отобразите ее значение.

```
USINT num = 16;

printf( "Unsigned short int value: %d \n" , num ) ;
```

6. В конце блока функции **main** верните значение 0, чего требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и выполните программу, чтобы увидеть значение, назначенное переменной перечисляемого типа, и значение, назначенное переменной пользовательского типа данных.



consttype.c

На заметку

Пользовательские типы данных должны быть определены в программе до того, как переменные этого типа будут созданы.

Определение констант

Директива препроцессора **#define** может быть использована для указания текстовых значений констант, которые впоследствии могут быть использованы в программе, с помощью следующего синтаксиса:

```
#define ИМЯ-КОНСТАНТЫ "текстовая-строка"
```

Как и **#include**, эта директива должна размещаться в самом начале файла с кодом программы. Все включения константы с указанным именем в коде программы перед компиляцией будут заменены препроцессором соответствующей текстовой строкой.

Условная директива препроцессора **#ifdef** также может быть использована для проверки того, существует ли определение. В зависимости от результата проверки далее может идти директива **#define**, позволяющая указать значение константы. Эта инструкция препроцессора также называется «макросом». Каждый макрос должен оканчиваться директивой **#endif**.

К счастью, макросы препроцессора могут проверять определенные компилятором константы, чтобы определить текущую операционную систему. Значения этих констант будут варьироваться на разных компьютерах, но на платформе Windows значением константы обычно является **_WIN32**, а на платформе Linux — **linux**. Директива препроцессора **#ifdef** может применить соответствующую текстовую строку для определения текущей платформы.

define.c

1. Начните новую программу с инструкции препроцессора, включающей стандартную библиотеку функций ввода/вывода.

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

2. Добавьте еще три директивы препроцессора, которые определяют заменяемые текстовые строки в исходном коде.

```
#define LINE "_____________________"

#define TITLE "C Programming in easy steps"

#define AUTHOR "Mike McGrath"
```

3. Далее добавьте условный макрос, который определяет текстовую строку, позволяющую идентифицировать платформу Windows.

```
#ifdef _WIN32

#define SYSTEM "Windows"

#endif
```

4. Далее добавьте условный макрос, который определяет текстовую строку, позволяющую идентифицировать платформу Linux.

```
#ifdef linux

#define SYSTEM "Linux"

#endif
```

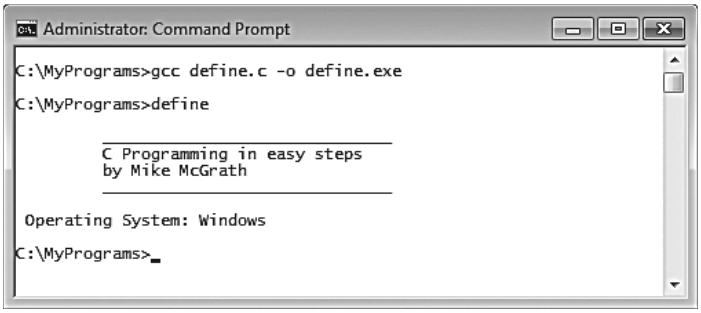
5. Далее добавьте функцию **main**, в которой выводится текстовая строка, подмененная препроцессором.

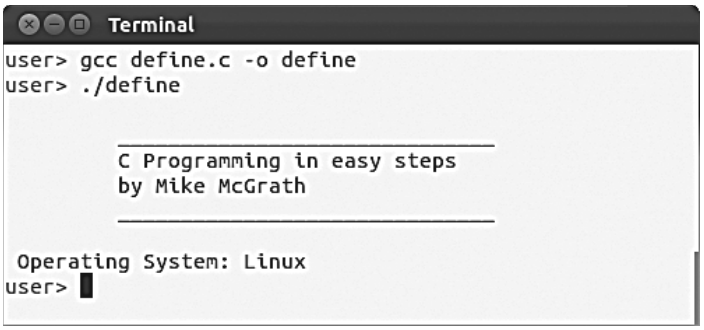
```
int main()

{

   printf("\n \t %s \n \t %s \n", LINE, TITLE);

   printf("\t by %S \n \t %s \n", AUTHOR, LINE);

   printf("\n Operating System: %s \n", SYSTEM);

}
```

6. В конце блока функции **main** верните значение 0, чего требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и выполните программу, чтобы увидеть отображенную строку.





На заметку

Строка **LINE** в этом примере представляет собой простой набор подчеркиваний.

Совет

Попробуйте вызвать команду **cpp –dM define**, чтобы увидеть все определенные константы.

Отладка с помощью определений

В качестве альтернативы можно использовать директивы препроцессора **#if**, **#else** и **#elif** (**else if**). Они позволяют использовать условное ветвление в соответствии с результатом оценки.

Константа, определенная директивой препроцессора **#define**, может быть разопределена с помощью директивы **#undef**. Оценка также рассматривается в случае, если константа не определена, это делается с помощью директивы препроцессора **#ifndef**.

Все оценки могут быть выполнены внутри блока функции, смешавшись с обычными утверждениями языка C, они могут быть вложены друг в друга. Подобные макросы довольно полезны для отладки исходного кода, поскольку целые разделы могут быть спрятаны или показаны путем простого изменения состояния макроса **DEBUG**.

debug.c

1. Начните новую программу с инструкции препроцессора, включающей стандартную библиотеку функций ввода/вывода.

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

2. Добавьте еще одну директиву препроцессора, предназначенную для создания макроса.

```
#define DEBUG 1
```

3. Далее добавьте функцию **main**, содержащую директивы препроцессора, предназначенные для оценки и отчета о состоянии макросов.

```
#if DEBUG == 1
    printf( "Debug status is 1 \n" );
#elif DEBUG == 2
    printf( "Debug status is 2 \n" ) ;
#else
    #ifdef DEBUG
        printf( "Debug is defined! \n" ) ;
    #endif
    #ifndef DEBUG
        printf( "Debug is not defined! \n" ) ;
    #endif
#endif
```

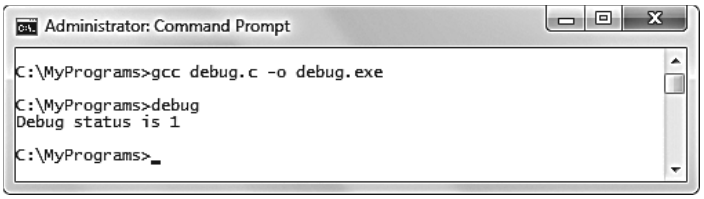
Совет

Операция равенства, **==**, использованная в этом примере, означает «равен».

4. В конце блока функции `main` верните значение 0, чего требует объявление функции.

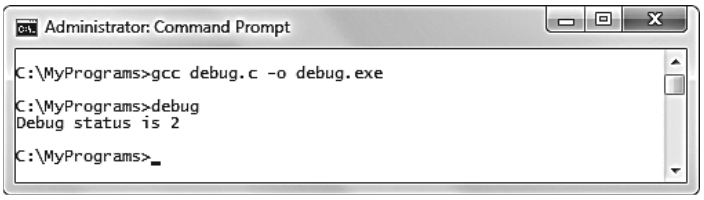
```
return 0;
```

5. Сохраните файл программы, а затем скомпилируйте и выполните программу, чтобы увидеть сообщение о состоянии макроса.



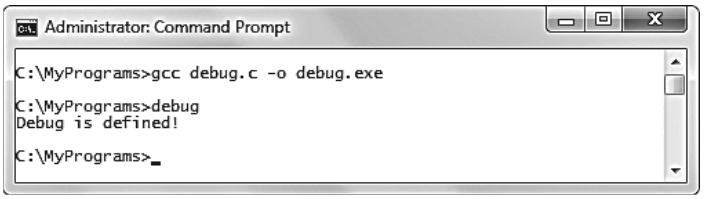
6. Измените значение макроса, а затем сохраните, перекомпилируйте и выполните программу снова, чтобы увидеть, что состояние макроса изменилось.

```
#define DEBUG 2
```



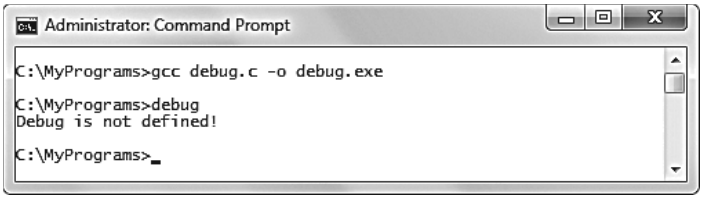
7. Измените значение макроса еще раз, а затем сохраните, перекомпилируйте и выполните программу снова, чтобы увидеть изменение.

```
#define DEBUG 3
```



8. Добавьте еще одну директиву препроцессора в начале блока функции, а затем сохраните, перекомпилируйте и выполните программу снова, чтобы увидеть изменение.

```
#undef DEBUG
```



На заметку

Каждая проверка условий с помощью препроцессора должна оканчиваться директивой `#endif`

Заключение

- Фиксированное значение, которое не будет изменяться, должно быть сохранено как константа, объявленная с помощью ключевого слова `const`.
- При объявлении константы всегда необходимо инициализировать ее фиксированным значением.
- Ключевое слово `enum` создает последовательность констант, чьи значения по умолчанию начинаются с нуля.
- Любой константе в последовательности может быть назначено числовое значение, которое будет увеличиваться в последующих константах.
- Последовательность констант допустимо рассматривать как новый тип данных. Переменные такого типа могут быть созданы для хранения перечислений, определенных этим типом.
- Пользовательский тип данных может быть определен с помощью ключевого слова `typedef`. Переменные такого типа могут быть созданы с помощью синтаксиса, используемого для создания обычных переменных.
- Директива препроцессора `#define` может быть использована для указания значения константы, которое перед компиляцией будет заменено.
- Условная директива препроцессора `#ifdef` проверяет, существует ли заданное определение.
- Макрос — это процедура препроцессора, которая должна оканчиваться директивой `#endif`.
- Определенные компилятором константы, например `_WIN32` и `linux`, могут помочь в определении операционной системы.
- Выполнить проверку макросов на разные значения можно с помощью директив препроцессора `#if`, `#else` и `#elif`.
- Определения макросов-констант могут быть «разопределены» с помощью директивы `#undef`. С помощью директивы `#ifndef` можно проверить, определен ли заданный макрос.
- Макросы могут быть полезны при отладке исходного кода, позволяя с легкостью спрятать или показать разделы кода.

4 Выполнение операций

Здесь показывается, как использовать операции языка C, чтобы манипулировать данными внутри программы.

- **Выполнение арифметических операций**
- **Присваивание значений**
- **Сравнение значений**
- **Логические значения**
- **Проверка условий**
- **Измерение размера**
- **Сравнение битовых значений**
- **Флаги**
- **Знакомство с приоритетами**
- **Заключение**

Выполнение арифметических операций

Арифметические операторы, повсеместно используемые в программах, написанных на языке C, приведены в таблице, расположенной ниже, где также перечислены операции, которые они могут выполнять.

Символ	Операция
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Деление с остатком
++	Инкремент
--	Декремент

Операции сложения, вычитания, умножения и деления ведут себя в соответствии с вашими ожиданиями. Однако для того, чтобы код выглядел прозрачно, следует использовать скобки в тех случаях, когда нужно выполнить более одного действия:

```
a = b * c – d % e /f;          /*Этот вариант непрозрачен*/

a = (b * c) – ((d % e) / f);   /*Этот вариант прозрачнее*/
```

Совет

Числа, используемые при работе с операциями для формирования выражений, называются *операндами* — в выражении `2 + 3` числа `2` и `3` являются операндами.

Операция деления с остатком, `%`, делит первое заданное число на второе и возвращает остаток от такого деления. Это полезно для определения четности числа.

Операции инкремента, `++`, и декремента, `--`, изменяют заданное число на 1 и возвращают результирующее значение. Чаще всего эти операции используются для подсчета итераций цикла. Операция инкремента увеличивает значение на 1, а операция декремента уменьшает на 1.

Операции инкремента и декремента могут быть размещены перед операндом или после него, эффект от них будет разным. Если операция размещена перед операндом (префикс), значение операнда изменится мгновенно, в противном случае (постфикс) его значение сначала записывается, а затем изменяется.

1. Начните новую программу с инструкции препроцессора, которая позволит включить стандартную библиотеку функций ввода/вывода.

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

arithmetic.c

2. Добавьте функцию **main**, в которой объявляется и инициализируется несколько целочисленных переменных.

```
int main()
{
    int a = 4, b = 8, c = 1, d = 1;
}
```

3. Далее в блоке функции **main** выведите результат арифметических операций, произведенных над значениями переменных.

```
printf( "Addition: %d \n" , a + b ) ;
printf( "Subtraction: %d \n" , b - a ) ;
printf( "Multiplication: %d \n" , a * b ) ;
printf( "Division: %d \n" , b / a ) ;
printf( "Modulus: %d \n" , a % b ) ;
```

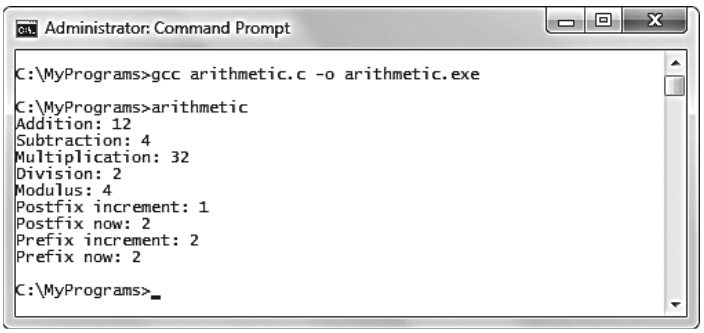
4. Теперь в блоке функции **main** выведите результат постфиксных и префиксных операций инкремента.

```
printf( "Postfix increment: %d \n" , c++ ) ;
printf( "Postfix now: %d \n" , c ) ;
printf( "Prefix increment: %d \n" , ++d ) ;
printf( "Prefix now: %d \n" , d ) ;
```

5. В конце блока функции **main** верните значение 0, чего требует объявление функции.

```
return 0;
```

6. Сохраните файл, а затем запустите программу, чтобы увидеть результат выполнения арифметических операций.



Внимание

Обратите внимание на то, что значение переменной мгновенно увеличивается только при префиксных операциях. При использовании постфиксных операций операнд оказывается увеличенным только при следующем обращении к нему.

Присваивание значений

Операции, которые используются с программах, написанных на языке C, для присваивания значений, перечислены в таблице ниже. Все они, помимо простой операции присваивания, `=`, являются краткой формой более длинного выражения, поэтому для ясности каждому из них приведен эквивалент.

Операция	Пример	Эквивалент
=	`a = b`	`a = b`
+=	`a += b`	`a = (a + b)`
-=	`a -= b`	`a = (a - b)`
*=	`a *= b`	`a = (a * b)`
/=	`a /= b`	`a = (a / b)`
%=	`a %= b`	`a = (a % b)`

Очень важно рассматривать операцию `=` в отношении присваивания, а не равенства, чтобы избежать путаницы с операцией равенства, `==`.

В примере, приведенном выше, переменной `a` присваивается значение, содержащееся в переменной `b`. Значением переменной `a` становится значение переменной `b`.

Внимание

Операция равенства, `==`, сравнивает значения операндов. Она описана далее в этой главе.

Операция `+=` может пригодиться для того, чтобы добавить значение к уже существующему значению, которое хранится в переменной `a`. В табличном примере операция `+=` сначала добавляет значение, хранимое в переменной `a`, к значению, хранимому в переменной `b`. Затем он присваивает результат этого действия переменной `a`.

Все остальные операции, приведенные в таблице, работают точно так же, сначала выполняя арифметические действия над двумя значениями, а затем присваивая результат первой переменной.

В случае операции `%=` первый операнд `a` делится на второй операнд `b`, а затем остаток от операции присваивается переменной `a`.

`assign.c`

1. Начните новую программу с инструкции препроцессора, которая позволит включить стандартную библиотеку функций ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляются и инициализируются две целочисленные переменные.

```
int main()
{
   int a, b;
}
```

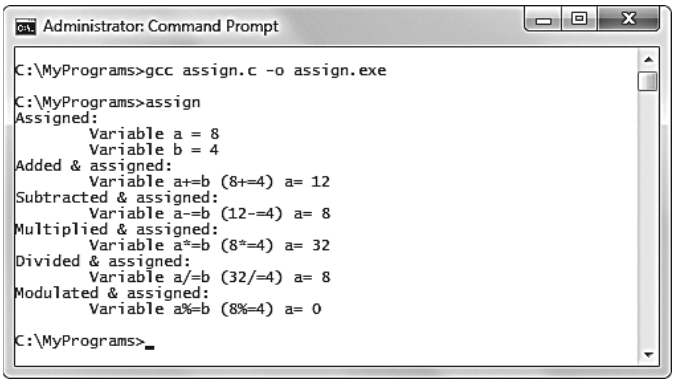
3. Далее в блоке функции **main** выведите результаты работы операций присваивания, произведенных над значениями переменных.

```
printf( "Assigned: \n" ) ;
printf( "\tVariable a = %d \n" , a = 8 ) ;
printf( "\tVariable b = %d \n" , b = 4 ) ;
printf( "Added & assigned: \n" ) ;
printf( "\tVariable a+=b (8+=4) a= %d \n", a += b ) ;
printf( "Subtracted & assigned: \n" ) ;
printf( "\tVariable a-=b (12-=4) a= %d \n", a -= b ) ;
printf( "Multiplied & assigned: \n" ) ;
printf( "\tVariable a*=b (8*=4) a= %d \n", a *= b ) ;
printf( "Divided & assigned: \n" ) ;
printf( "\tVariable a/=b (32/=4) a= %d \n", a /= b ) ;
printf( "Modulated & assigned: \n" ) ;
printf( "\tVariable a%=b (8%=4) a= %d \n", a %= b ) ;
```

4. В конце блока функции **main** верните значение 0, чего требует объявление функции.

```
return 0;
```

5. Сохраните файл, а затем запустите программу, чтобы увидеть результат выполнения операций присваивания.



Совет

Обратите внимание на то, что для того, чтобы отобразить символ `%`, внутри функции `printf()` следует использовать комбинацию символов `%%`.

Сравнение значений

Операции, часто используемые при программировании на языке C для сравнения двух числовых значений, перечислены в следующей таблице.

Операция	Сравнение
==	Равенство
!=	Неравенство
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Операция равенства, **==**, сравнивает два операнда и возвращает 1 (**true**), если их значения равны, в противном случае он вернет значение 0 (**false**). Операнды равны, если содержат одинаковое число, или же, если они являются символами, они равны в случае, когда совпадает числовое значение их ASCII-кодов.

Операция неравенства,**!=**, поступает наоборот — возвращает 1 (**true**), если операнды не равны, используя те же правила, что и операция равенства, **==**, в противном случае она вернет значение 0 (**false**).

Операции равенства и неравенства полезны при проверке состояния двух переменных с целью выполнения условного ветвления в программе.

Операция **>** (больше) сравнивает два операнда и вернет значение 1 (**true**), если значение первого операнда больше, чем значение второго. Если значение второго операнда больше или равно значению первого, операция вернет значение 0 (**false**). Эта операция часто используется для проверки счетчиков цикла.

Операция **<** (меньше) выполняет такое же сравнение, но возвращает значение 1 (**true**), если значение второго операнда больше значения первого, в противном случае он возвращает значение 0 (**false**).

Добавление операции **=** после операций **>** или **<** заставляет их также возвращать значение 1, если значения двух операндов равны.

Совет

Более подробную информацию о значениях ASCII-кодов можно найти в конце книги.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

comparison.c

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

2. Добавьте функцию **main**, в которой объявляются и инициализируются три целочисленные переменные и две символьные переменные.

```
int main()

{

    int zero = 0, nil = 0, one = 1;

    char upr = 'A', lwr = 'a';

}
```

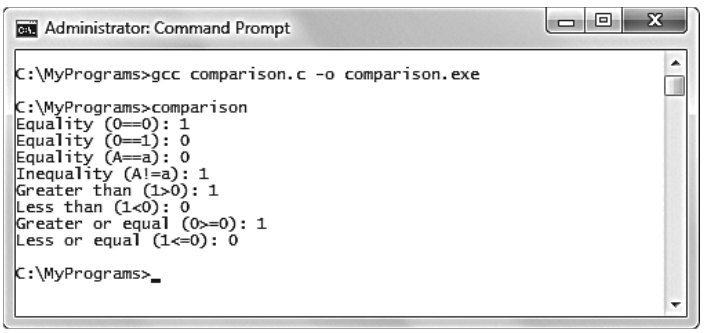
3. Далее в блоке функции **main** выведите результат выполнения над переменными операций сравнения.

```
printf( "Equality (0==0): %d \n" , zero == nil ) ;

printf( "Equality (0==1): %d \n" , zero == one ) ;

printf( "Equality (A==a): %d \n" , upr == lwr ) ;

printf( "Inequality (A!=a): %d \n" , upr != lwr ) ;

printf( "Greater than (1>0): %d \n" , one > nil ) ;

printf( "Less than (1<0): %d \n" , one < nil ) ;

printf( "Greater or equal (0>=0): %d \n" , zero >= nil ) ;

printf( "Less or equal (1<=0): %d \n" , one <= nil ) ;
```

4. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

5. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть результат выполнения операций сравнения.



На заметку

Число 1 представляет собой значение **true**, число 0 — значение **false**.

Совет

ASCII-код для символа верхнего регистра **A** равен 65, а для символа нижнего регистра **a** — 97, поэтому при их сравнении будет возвращено значение 0.

Логические значения

Логические операции, наиболее часто используемые в программировании на языке C, перечислены в следующей таблице.

Символ	Операция
&&	Логическое И
||	Логическое ИЛИ
!	Логическое НЕ

Логические операции используются для операндов, которые имеют булевы значения **true** или **false**, а также в выражениях, которые могут быть преобразованы к значениям **true** или **false**.

Операция логического И, **&&**, оценит два операнда и вернет значение **true** только в том случае, если оба операнда имеют значение **true**. В противном случае, операция **&&** вернет значение **false**.

Эта операция используется в условном ветвлении, когда направление выполнения программы, написанной на языке C, определяется двумя условиями. Если оба условия были удовлетворены, выполнение программы продолжится в заданном направлении, в противном случае направление придется сменить.

В отличие от операции **&&**, требующей, чтобы оба операнда имели значение **true**, операция логического ИЛИ, ||, оценивает два операнда и возвращает значение **true**, если хотя бы один из них имеет значение **true**. Если ни один из операндов не имеет значения **true**, операция || вернет значение **false**. В языке C эта операция может быть полезной, если требуется проверить, было ли соблюдено хотя бы одно условие.

Совет

Термин «булев» относится к системе логического мышления, разработанной английским математиком Джорджем Булем (1815–1864).

Третья операция — логическое НЕТ, !, — является унарной операцией, она применяется только для одного операнда. Она возвращает инвертированное значение заданного операнда, поэтому, если переменная **var** имеет значение **true**, выражение **!var** вернет значение **false**. Операция NOT полезна при программировании на языке C, если нужно инвертировать значение переменной в успешных итерациях цикла с помощью утверждения вроде **var = !var**. Это гарантирует, что каждый раз значение будет инвертировано, и напоминает щелчки выключателем.

В программировании на языке C 0 представляет собой значение **false**, а любое ненулевое значение, например, 1 — значение **false**.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

logic.c

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

2. Добавьте функцию **main**, в которой объявляются и инициализируются две целочисленные переменные.

```
int main()

{

    int yes = 0, no = 1;

}
```

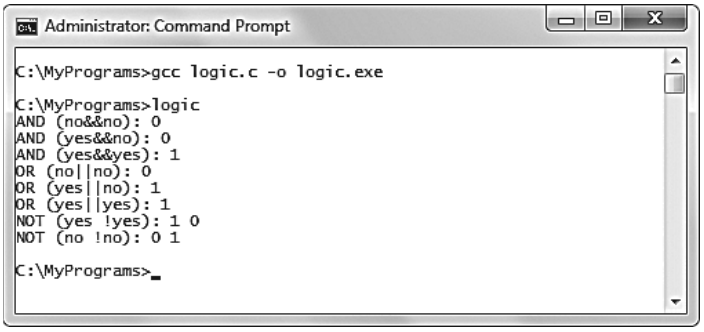
3. Далее в блоке функции **main** выведите результат выполнения над переменными логических операций.

```
printf( "AND (no&&no): %d \n" , no && no ) ;

printf( "AND (yes&&no): %d \n" , yes && no ) ;

printf( "AND (yes&&yes): %d \n" , yes && yes ) ;

printf( "OR (no||no): %d \n" , no || no ) ;

printf( "OR (yes||no): %d \n" , yes || no ) ;

printf( "OR (yes||yes): %d \n" , yes || yes ) ;

printf( "NOT (yes !yes): %d %d\n" , yes , !yes ) ;

printf( "NOT (no !no): %d %d\n" , no , !no ) ;
```

4. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

5. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть результат выполнения логических операций.



На заметку

Обратите внимание на то, что выражение `0 && 0` возвращает значение 0 (**false**), что иллюстрирует поговорку «Минус на минус — не всегда плюс».

Проверка условий

Возможно, самой любимой операцией программистов, пишущих на C, является условная операция ?:, также известная как *тернарная* операция. Она сначала оценивает результат выполнения выражения (**true** или **false**), а затем выполняет одно из двух заданных утверждений в зависимости от результата оценки.

Условная операция имеет следующий синтаксис:

```
(выражение-для-проверки) ? если-true-выполнить-это : если-false-выполнить-это);
```

Эта операция может быть использована, например, для оценки того, является ли заданное число четным или нечетным путем проверки наличия остатка от деления на 2, а затем вывода соответствующей строки:

```
(7 % 2 != 0) ? printf ("Четное число") : printf("Нечетное число");
```

В этом примере деление числа 7 на число 2 оставляет ненулевой остаток, поэтому выражение имеет значение **true**, а значит, будет выполнено первое утверждение, правильно описывая число как нечетное.

Условная операция также может быть полезна для контроля за грамматикой в тексте, выводимом на экран, когда речь идет о единственном и множественном числе, избегая неловких фраз наподобие «Имеем пять ножницы». Такую проверку легко выполнить внутри утверждения **printf()**:

```
printf("Имеем %d %s", (num == 1 ? "ножницы" : "ножниц", num );
```

В этом примере в случае, если значение выражения равно **true**, значение переменной **num** равно 1 и будет использован вариант «ножницы», в противном случае будет использован вариант «ножниц».

Внимание

Несмотря на то, что условная операция проста в применении, ее использование может снизить читабельность кода.

Условная операция также может быть использована для присвоения соответствующих значений переменной в зависимости от результата проверки. Синтаксис выражения будет выглядеть так:

```
переменная = (проверочное-выражение) ? если-true-назначить-это-значение : если-false-
    назначить-это-значение
```

Эта операция может использоваться, например, для того, чтобы проверить, превосходит ли одно число другое, а затем присвоить большее из них переменной **a**, например так:

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

num = (a > b) ? a : b;
```

В данном примере значение переменной **a** больше значения переменной **b**, поэтому переменной **num** присваивается большее значение — 5.

1. Начните новую программу с инструкции препроцессора, которая позволит включить стандартную библиотеку функций ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляется и инициализируется одна целочисленная переменная.

```
int main()

{

    int num = 7;

}
```

3. Далее в блоке функции **main** добавьте условное утверждение, позволяющее вывести четность значения переменной.

```
(num % 2 != 0) ?

printf("%d is odd\n", num) : printf("%d is even\n", num) ;
```

4. Далее добавьте условное утверждение, позволяющее вывести грамматически правильную фразу.

```
printf("There %s", (num == 1) ? "is" : "are";

printf("%d %s\n", num (num == 1) ? "apple" : "apples";
```

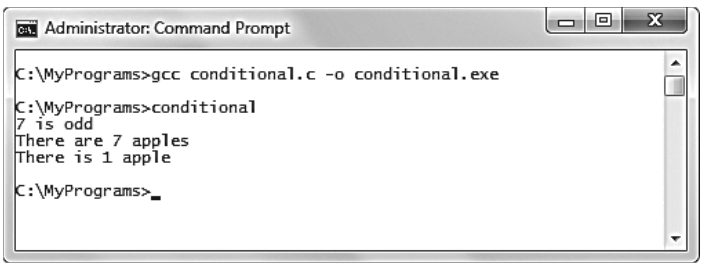
5. Уменьшите значение переменной, а затем снова добавьте условное утверждение, позволяющее вывести грамматически правильную фразу.

```
num=1;

printf("There %s", (num == 1) ? "is" : "are";

printf("%d %s\n", num (num == 1) ? "apple" : "apples";
```

6. В конце блока функции **main** верните значение 0, чего требует объявление функции.

```
return 0;
```

7. Сохраните файл, а затем запустите программу, чтобы увидеть результат выполнения условных операций.



conditional.c

Совет

Условную операцию лучше всего использовать для утверждений, имеющих всего две простые альтернативы.

Измерение размера

В программировании на языке C операция **sizeof** возвращает целочисленное значение, представляющее количество байт, необходимое для хранения содержимого данного операнда в памяти.

Когда операнд, переданный операции **sizeof**, является именем типа данных, например, **int**, он должен быть помещен в скобки — как в примерах, приведенных ранее на странице 27.

В качестве альтернативы в случае, если переданный операнд является именем объекта данных, например именем переменной, скобки опционально можно опустить. На практике многие программисты всегда будут помещать операнд операции **sizeof** в скобки, чтобы избежать необходимости запоминать подобное разграничение.

Простейшим хранилищем в языке C является тип данных **char**, который хранит один символ в одном байте памяти. Это значит, что утверждение **sizeof(char)** вернет 1 (один байт).

Как правило, для типов данных **int** и **float** выделяется 4 байта машинной памяти, а для типа данных **double** — 8 байт, что позволит разместить весь диапазон его значений. Эти размеры не являются стандартизированными, они зависят от реализации и потому могут изменяться. Хорошим тоном при программировании является использование операции **sizeof** для измерения объема выделенной памяти.

Совет

Всегда обрамляйте в скобки операнд, переданный операции **sizeof**.

Объем памяти, выделенный для массивов переменных, складывается из количества байт, выделяемых для заданного типа данных, умноженного на число элементов массива. Например, выражение **sizeof(int[3])**, как правило, вернет число 12 (3 × 4 байта).

Особенно важно использовать операцию **sizeof** для того, чтобы точно определить объем памяти, выделенный для определенных пользователем структур, которые могут иметь члены различных типов данных, поскольку часто между ними автоматически добавляется так называемая *прослойка*. Это значит, что общий объем выделенной памяти может превосходить сумму объемов памяти, выделенной для каждого члена. Логично было бы ожидать, что для структуры, содержащей переменные **int score** и **char grade**, будет выделено 5 байт (4 + 1), но, как правило, в таких ситуациях выделяется 8 байт. Это происходит потому, что (32-битные) компьютерные системы обычно считывают данные *словами*, имеющими размер 4 байта. Поэтому при выделении памяти добавляется прослойка, позволяющая дополнить объем выделяемой памяти до числа, кратного 4.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляется и инициализируется одна целочисленная переменная.

```
int main()

{

    int num = 1234567890;

}
```

3. Далее в блоке функции **main** добавьте утверждения, позволяющие узнать объем памяти, выделенной для типа данных **int** (в данной реализации), с помощью имени типа данных и имени объекта.

```
printf( "Size of int data type is %d bytes\n" , sizeof (int) ) ;

printf( "Size of int variable is %d bytes\n" , sizeof (num) ) ;
```

4. Теперь добавьте утверждение, позволяющее вывести на экран объем памяти, который выделяется для каждого элемента массива.

```
printf( "Size of an int array is %d bytes\n", sizeof (int[3]) ) ;
```

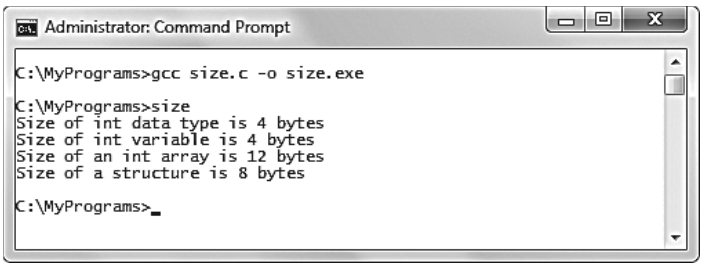
5. Определите структуру, содержащую одну переменную типа **char** и одну переменную типа **int**, а затем выведите на экран объем памяти, выделенный структуре, включая прослойку.

```
struct {int score; char grade; } result;

printf("Size of a structure is %d bytes\n" , sizeof (result) ) ;
```

6. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть результат выполнения логических операций.



size.c

На заметку

В этом примере приведен код для создания объекта типа **struct**, который представляет собой набор различных переменных. Тема структуры рассмотрена в главе 9, в данном примере структуры включены для того, чтобы продемонстрировать, как можно измерить объем выделенной для них памяти с помощью операции **sizeof**.

Сравнение битовых значений

На заметку

Многие программисты, использующие язык C, никогда не используют битовые операции, однако понимать их и знать, где они могут использоваться, может быть полезно.

Каждый байт состоит из восьми бит, каждый из которых может содержать 1 или 0. Все вместе они дают двоичное значение, представляющее десятичное число от 0 д 255. Бит вносит свою лепту в формирование десятичного числа только в том случае, если он содержит 1. Компоненты были созданы в формате «справа-налево» от наименее значащего бита (Less Significant Bit, LSB) до наиболее значащего бита (Most Significant Bit, MSB). Двоичное число, приведенное ниже, представляет собой десятичное число 50.

Номер бита	8 MSB	7	6	5	4	3	2	1 LSB
Десятичные разряды	128	64	32	16	8	4	2	1
Двоичные разряды	0	0	1	1	0	0	1	0

Совет

Каждая половина байта называется *тетрадой* (4 бита). Двоичные числа, использованные в примерах таблицы, описывают значения, хранящиеся в тетраде.

Хотя тип данных `char` в языке C представляет собой простое однобайтовое хранилище, как описано на предыдущей странице, существует возможность манипулировать его отдельными частями с помощью *битовых* операций.

Символ	Имя	Операция
|	ИЛИ	Возвращает 1 в бите, если хотя бы один из сравниваемых битов имеет значение 1 Пример: 1010 | 0101 = 1111
&	И	Возвращает 1 в бите, если оба сравниваемых бита имеют значение 1 Пример: 1010 & 1100 = 1000
~	НЕ	Возвращает 1 в бите, если ни один из сравниваемых битов не имеет значения 1 Пример: 1010 ~ 0011 = 0100
^	Исключающее ИЛИ	Возвращает 1 в бите, если только один из сравниваемых битов имеет значение 1 Пример: 1010 ^ 0100 = 1110
<<	Битовый сдвиг влево	Перемещает каждый бит, имеющий значение 1, на определенное количество разрядов влево Пример: 0010 << 2 = 1000
>>	Битовый сдвиг вправо	Перемещает каждый бит, имеющий значение 1, на определенное количество разрядов вправо Пример: 1000 >> 2 = 0010

Если вы не программируете для устройств с ограниченными ресурсами, битовые операции вам скорее всего не понадобятся, однако они могут быть довольно полезны. Например, операция **XOR** (исключающее ИЛИ) позволяет вам менять значения двух переменных без необходимости в наличии третьей.

xor.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляются и инициализируются две целочисленные переменные.

```
int main()

{

    int x = 10, y = 5;

    printf("\nx = %d y = %d\n", x, y);

}
```

3. Далее в блоке функции **main** трижды добавьте утверждение **XOR**, чтобы поменять местами значения переменных путем битовых манипуляций.

```
x = x ^ y;  /* 1010 ^ 0101 = 1111 (десятичное число 15)*/

y = x ^ y;  /* 1111 ^ 0101 = 1010 (десятичное число 10)*/

x = x ^ y;  /* 1111 ^ 1010 = 0101 (десятичное число 5)*/
```

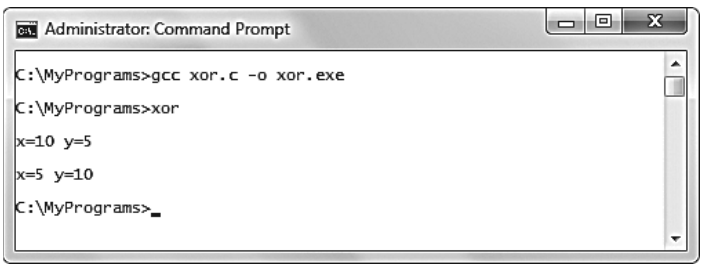
4. Теперь добавьте утверждение, позволяющее вывести на экран новые значения переменных.

```
printf("\nx = %d y = %d\n", x, y);
```

5. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть битовую магию операции **XOR**.



Внимание

Не путайте битовые операции с логическими. Битовые операции сравнивают двоичные числа, а логические проверяют булевы значения.

Флаги

По этого момента наиболее распространенным вариантом использования битовых операций было манипулирование компактным битовым полем, содержащим набор булевых флагов. Такое использование памяти гораздо более эффективно, нежели хранение значений булевых флагов в отдельных переменных. Например, переменная типа **char** занимает всего один байт, в котором разрешается хранить целых восемь флагов, а создание восьми отдельных переменных типа **char** потребует использования целых восьми байт.

Исходное значение битовых флагов может быть установлено путем присвоения переменной десятичного значения, двоичный эквивалент которого имеет единицы в тех битах, которые должны «включить» битовый флаг. Например, десятичное число 8 имеет двоичный эквивалент 1000 (1 × 8 0 × 4 0 × 2 0 × 1), поэтому «включенным» будет четвертый флаг, если считать справа налево от наименее значащего бита.

Значения битовых флагов могут быть обращены с помощью битовой операции НЕ (**~**), однако, следует использовать маску для нулей, идущих перед битовыми флагами, иначе каждый из них получит значение 1. Например, чтобы обратить битовое поле из четырех флагов, располагающееся с правой стороны байта, к четырем левым битам должна быть применена маска. Десятичное число 15 имеет двоичное представление 00001111 (0 × 128 0 × 64 0 × 32 0 × 16 1 × 8 1 × 4 1 × 2 1 × 1), его разрешается использовать как маску с помощью операции И (**&**).

Шаблон битовых флагов также может быть изменен путем смещения флагов, имеющих значение 1, на определенное количество бит с помощью операций побитового сдвига влево (**<<**) и вправо (**>>**).

bitflag.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляется и инициализируется одна символьная переменная.

```
int main()
{
    int flags = 8;  /*Двоичное представление 1000 (1 x 8 0 x 4 0 x 2 0 x 1)*/
}
```

3. Далее в блоке функции **main** назначьте переменной новое значение, чтобы значение 1 получил также второй флаг.

```
flags = flags | 2;  /* 1000 | 0010 = 1010 (десятичное число 10)*/
```

4. Теперь добавьте утверждения, позволяющие вывести на экран значения всех битовых флагов.

```
printf( "Flag 1: %s\n" , ( (flags & 1) > 0) ? "ON" : "OFF") ;
printf( "Flag 2: %s\n" , ( (flags & 2) > 0) ? "ON" : "OFF") ;
printf( "Flag 3: %s\n" , ( (flags & 4) > 0) ? "ON" : "OFF") ;
printf( "Flag 4: %s\n\n" , ((flags & 8) > 0) ? "ON" : "OFF") ;
```

5. Далее добавьте утверждения, накладывающие маску на первые четыре бита байта, а затем инвертируйте все значения битовых флагов.

```
char mask = 15;     /*Двоичное представление 00001111*/
flags = ~flags & mask;  /*~(1010 & 111 = 1010) = 0101*/
```

6. Теперь добавьте утверждения, позволяющие вывести на экран измененные значения всех битовых флагов, а также десятичное значение, представляющее этот шаблон.

```
printf( "Flag 1: %s\n" , ( (flags & 1) > 0) ? "ON" : "OFF") ;
printf( "Flag 2: %s\n" , ( (flags & 2) > 0) ? "ON" : "OFF") ;
printf( "Flag 3: %s\n" , ( (flags & 4) > 0) ? "ON" : "OFF") ;
printf( "Flag 4: %s\n\n" , ((flags & 8) > 0) ? "ON" : "OFF") ;
printf( "Flags decimal value is %d\n" , flags ) ;
```

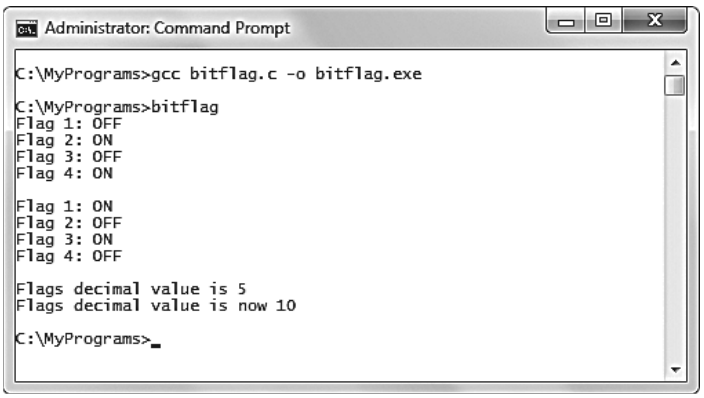
7. Добавьте утверждение, позволяющее выполнить сдвиг «включенных» флагов на один бит влево, а затем выведите десятичное значение нового шаблона.

```
flags = flags << 1; /*0101 << 1 = 1010*/
printf( "Flags decimal value is now %d\n" , flags ) ;
```

8. В конце блока функции `main` верните значение 0, как того требует объявление функции.

```
return 0;
```

9. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть битовую магию операции `XOR`.



Совет

Часто довольно удобно использовать заранее определенные константы, представляющие значение каждого битового флага. Например,

```
#define FLAG1 1
#define FLAG2 2
#define FLAG3 4
#define FLAG4 8
```

На заметку

Более крупные битовые поля могут быть созданы с использованием переменной, резервирующей больший объем памяти. Например, переменная типа `int`, для которой резервируется 4 байта, может вместить 32 флага.

Знакомство с приоритетами

Совет

Операция умножения, *, находится выше, чем операция сложения, +, поэтому в выражении `a = 6 + b *3` сначала выполняется операция умножения.

Приоритет операций определяет порядок, в котором в языке C выполняются выражение. Например, в выражении `a = 6 + b * 3` приоритет операций определяет, в каком порядке будут выполнены операции сложения и умножения.

В следующей таблице перечислен приоритет операций в убывающем порядке. Операции, расположенные выше, имеют больший приоритет, чем операции, стоящие ниже, поэтому они будут выполнены раньше. Операции из одной ячейки имеют одинаковый приоритет, поэтому порядок их выполнения зависит только от направления ассоциативности операций. Например, при ассоциативности слева направо, операции, расположенные слева, будут выполнены раньше.

На заметку

Операции структур -> и . описываются далее в этой книге; они включены в эту таблицу для полноты картины.

Операция	Ассоциативность
() Вызов функции [] Индекс массива -> Указатель на структуру. Член структуры	Слева направо
! НЕ ~ Битовое НЕ ++ Инкремент -- Декремент + Знак «плюс» `sizeof` - Знак «минус» * Указатель & `addressof`	Справа налево
* Умножение / Деление % Деление с остатком	Слева направо
+ Сложение - Вычитание	Слева направо
<< Сдвиг влево >> Сдвиг вправо	Слева направо
< Меньше <= Меньше или равно > Больше >= Больше или равно	Слева направо
== Равенство != Неравенство	Слева направо
& Битовое И	Слева направо
^ Битовое «исключающее И»	Слева направо
| Битовое ИЛИ	Слева направо
&& И	Слева направо
|| ИЛИ	Слева направо
?: Условная операция	Справа налево
= += -= *= /= %= &= ^= |= <<= >>= Операции присваивания	Справа налево
, Запятая	Справа налево

precedence.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой выводится результат вычисления выражения, следующего низкоуровневому порядку приоритета операций, а также результат вычисления выражения с явно заданным приоритетом.

```
int main()
{
    printf("\nDefault precedence ((2*3)+4)-5 : %d\n", 2*3+4-5 ) ;
    printf("Explicit precedence 2* ((3+4)-5 ): %d\n", 2*((3+4)-5) ) ;
}
```

3. Далее в блоке функции **main** выведите число-результат вычисления выражения, следующего правилам приоритета для ассоциативности слева направо, а затем результат вычисления выражения с явно определенным приоритетом.

```
printf( "\nDefault precedence (7*3) %% 2: %d\n" , 7*3%2 );
printf( "Explicit precedence 7* (3%%2) : %d\n" , 7*(3%2) ) ;
```

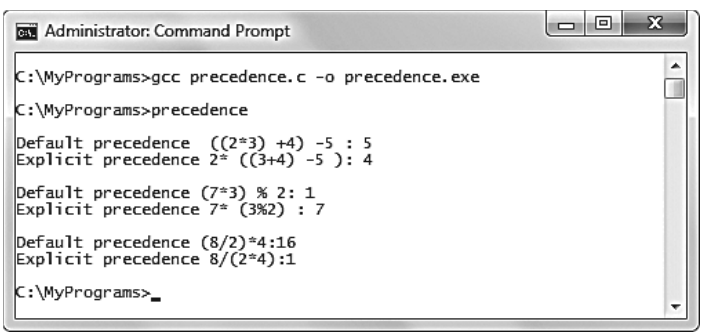
4. Теперь выведите число-результат вычисления выражения, следующего правилам приоритета для ассоциативности справа налево, а затем результат вычисления выражения с явно определенным приоритетом.

```
int num = 9;
printf("\nDefault precedence(8/2)*4:%d\n", --num/2*sizeof(int));
num = 9;
printf("Explicit precedence 8/(2*4):%d\n", --num/(2*sizeof(int)));
```

5. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть результат, следующий правилам приоритета.



Внимание

Для того чтобы вывести на экран символ `%` в функции `printf()` следует использовать комбинацию `%%`.

Заключение

- Арифметические операции, которые могут образовывать выражения с двумя операндами — сложение, `+`, вычитание, `-`, умножение, `*`, деление, `/`, и деление с остатком, `%`.
- Операции инкремента, `++`, и декремента, `--`, изменяют значение их единственного операнда на 1.
- Операция присваивания = может быть объединен с арифметической операцией, чтобы выполнить расчет выражения, а затем сразу присвоить результат.
- Операции сравнения могут образовывать выражения, сравнивающие два операнда с точки зрения равенства `==` или неравенства `!=`, а также проверяющие, является ли одно число больше `>`, меньше `<`, больше или равно `>=` или меньше или равно `<=` другому.
- Операции «Логическое И» `&&` и «Логическое ИЛИ» `||` образуют выражения, оценивающие два операнда и возвращающие булево значение `true` или `false`, а логическая операция `NOT !` возвращает инвертированное булево значение одного операнда.
- Условная операция `?:` оценивает заданное булево выражение, а затем возвращает один из двух операндов в зависимости от результата.
- Операция `sizeof` возвращает размер выделяемой памяти в байтах для заданного типа данных или объекта данных.
- Один байт памяти состоит из восьми бит, которые могут содержать значение 0 или 1.
- Битовые операции ИЛИ `|`, И `&`, НЕ `~` или исключающее ИЛИ `^` возвращают значение после выполнения сравнения значений двух битов, а операции побитового сдвига влево `<<` и побитового сдвига вправо `>>` смещают значение переданных битов на заданное число позиций.
- Наиболее частое использование битовых операций заключается в том, чтобы изменить компактное поле бит, содержащее набор булевых флагов.
- В выражениях, содержащих несколько операций, операции будут выполняться в соответствии со стандартным приоритетом операций, если только порядок выполнения не определен явно путем добавления скобок.

Создание утверждений

Здесь показывается, как с помощью утверждений можно оценивать выражения, чтобы определить направление, в котором следует создавать программу.

- **Проверка значений выражений**
- **Ветвление с помощью операции** `switch`
- **Зацикливание с помощью счетчика**
- **Зацикливание с помощью условия**
- **Досрочный выход из циклов**
- **Переход к меткам**
- **Заключение**

Проверка значений выражений

Ключевое слово **if** используется для выполнения простой условной проверки, которая оценивает булево значение заданного выражения. Утверждения внутри скобок, проверка которых выполняется, могут быть выполнены только в том случае, если значение выражения равно **true**. Синтаксис проверочного утверждения **if** выглядит так:

```
if (проверочное-выражение) {утверждения-которые-следует-выполнить-если-true}
```

Если значение выражения равно **true**, можно выполнить и несколько утверждений, каждое из них должно отделяться точкой с запятой.

Иногда возникает необходимость проверить несколько выражений, чтобы определить, должны ли быть выполнены следующие далее утверждения. Этого можно достигнуть двумя способами. Операция логическое И **&&** используется для того, чтобы убедиться в том, что утверждения будут выполнены только в том случае, если оба выражения имеют значение **true**. Синтаксис выглядит так:

```
if ((проверочное-выражение) && (проверочное-выражение)) {утверждения}
```

Кроме того, можно использовать несколько утверждений **if**, *вложенных* друг в друга. Это также поможет убедиться в том, что последующие утверждения окажутся выполнены только в том случае, когда оба проверочных выражения будут иметь значение **true**, например так:

```
if (проверочное-выражение)
{
    if(проверочное-выражение) {утверждения}
}
```

Когда одно или более выражений, оцениваемые с помощью утверждения **if**, имеют значение **false**, утверждения, расположенные внутри скобок не выполняются, и программа продолжает выполнение последующего кода.

Часто является предпочтительным расширять утверждение **if**, добавляя к нему утверждение **else**, указав внутри скобок утверждения, которые должны быть выполнены в случае, если проверочное выражение будет иметь значение **false**. Синтаксис выглядит так:

```
if (проверочное-выражение)
    {утверждения-которые-следует-выполнить-если-true}
else
    {утверждения-которые-следует-выполнить-если-false}
```

Этот прием программирования является фундаментальным, он предлагает программе пойти в одном из двух направлений в зависимости от результатов проверки. Этот прием также известен как «условное ветвление».

Совет

Если код, который должен быть выполнен, содержит только одно утверждение, фигурные скобки можно опустить.

Совет

Несколько выражений сразу можно оценить, объединив утверждения **if** и **else**, например, так: **if** () {…} **else if** () {…}.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой располагается утверждение, выводящее текст в случае, когда проверочное выражение имеет значение **true**.

```
int main()
{
    if( 5 > 1 ) { printf( "Yes, 5 is greater than 1\n" ) ; }
}
```

3. Далее в блоке функции **main** добавьте утверждение, которое выводит на экран текст в случае, когда оба проверочных выражения имеют значение **true**.

```
if ( 5 > 1)
{
    if(7 > 2)

    { printf("5 is greater than 1 and 7 is greater than 2\n") ; }

}
```

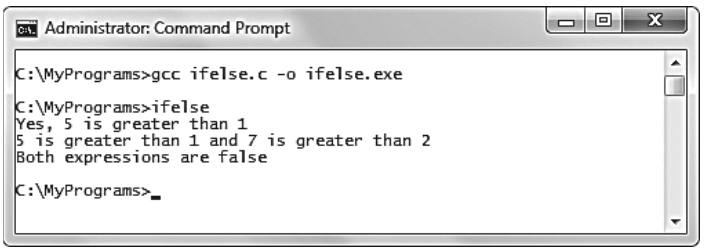
4. Теперь после двух проверочных выражений добавьте утверждение по умолчанию, которое выполняется в случае, когда оба выражения имеют значение **false**.

```
if( 1 > 2 )
{ printf( "1st Expression is true\n" ) ; }
else if( 1 > 3 )
{ printf( "2nd expression is true\n" ) ; }
else
{ printf( "Both expressions are false\n" ) ; }
```

5. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть выведенный на экран результат работы условных проверок.



ifelse.c

На заметку

В языке программирования C значение **true** имеет числовое представление 1, а значение **false** – числовое представление 0. Поэтому выражение (5 > 1) является сокращенной версией выражения (5 > 1 == 1).

Ветвление с помощью операции switch

Условное ветвление, которое выполняется с помощью нескольких утверждений **if else**, можно выполнить более эффективно с помощью утверждения **switch**, где проверочное выражение проверяет единственное условие.

Утверждение **switch** работает довольно необычно. Оно получает переданное значение как параметр, а затем ищет совпадение среди некоторого количества утверждений **case**.

Код, который должен быть выполнен при совпадении, находится в каждом утверждении **case**.

Совет

Утверждение с меткой **default** стоит включать всегда, даже если оно будет использоваться только для того, чтобы вывести сообщение об ошибке.

Важно помнить заканчивать утверждения **case** ключевым словом **break**, что позволит утверждению **switch** завершиться при нахождении совпадения, а не продолжать дальнейший поиск, если только это не является необходимостью.

Опционально список утверждений **case** может завершаться финальным утверждением **default**, позволяющим указать код, который должен быть выполнен в случае, если в утверждениях **case** не было найдено совпадений.

Синтаксис утверждений **switch** обычно выглядит так:

```
switch(проверочное-значение)
{
    case значение: утверждения-которые-нужно-выполнить-при-совпадении; break;
    case значение: утверждения-которые-нужно-выполнить-при-совпадении; break;
    case значение: утверждения-которые-нужно-выполнить-при-совпадении; break;
    default: утверждения-которые-нужно-выполнить-при-отсутствии-совпадений;
}
```

Согласно стандарту ANSI C, утверждение **switch** может иметь до 257 утверждений **case**, но никакие два утверждения **case** не могут иметь одинаковых значений.

На заметку

Утверждение с меткой **default** не должно обязательно появляться в конце блока **switch**, но поместить его туда логично, поскольку в этом случае для него не потребуется утверждения **break**.

Когда для некоторого количества элементов нужно выполнить одинаковые утверждения, только итоговое утверждение **case** должно их содержать. Например, для того, чтобы вывести одинаковое сообщение при значениях 0, 1 и 2, следует использовать следующий код:

```
switch (num)
{
    case 0:
    case 1:
    case 2 : printf( "Less than 3\n" ) ; break ;
    case 3 : printf( "Exactly 3\n" ) ; break ;
    default : printf( "Greater than 3 or less than zero\n" ) ;
}
```

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

switch.c

2. Добавьте функцию **main**, в которой объявляются и инициализируются одна целочисленная и одна символьная переменные.

```
int main()
{
    int num = 2; char letter = 'b';
}
```

3. Далее в блоке функции **main** добавьте утверждение **switch**, которое пытается найти соответствие целочисленному значению.

```
switch(num)
{
    case 1 : printf( "Number is one\n" ) ; break ;
    case 2 : printf( "Number is two\n" ) ; break ;
    case 3 : printf( "Number is three\n" ) ; break ;
    default : printf( "Number is unrecognized\n" ) ;
}
```

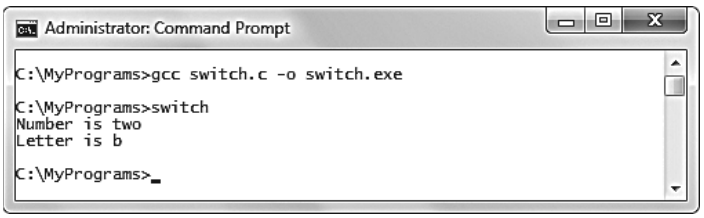
4. Теперь добавьте утверждение **switch**, которое пытается найти соответствие символьному значению.

```
switch(letter)
{
    case 'a': case 'b': case 'c':
    printf( "Letter is %c\n" , letter ) ; break ;
    default : printf( "Letter is unrecognized\n" ) ;
}
```

5. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть выведенный на экран результат работы операции **switch**.



Совет

В утверждениях **switch** ключевое слово **case**, значение, указанное рядом, и символ двоеточия рассматриваются как уникальная *метка*.

Зацикливание с помощью счетчика

Цикл — это фрагмент кода программы, который повторяется автоматически. Однократное выполнение всех утверждений внутри цикла называется *итерацией* или *проходом*. Длина цикла контролируется условной проверкой, выполняемой внутри цикла. Пока проверяемое выражение имеет значение **true**, цикл станет продолжаться (до тех пор, пока значение не будет равно **false**, в этот момент цикл закончится).

В программировании на языке C существуют три варианта структуры цикла — циклы **for**, циклы **while** и циклы **do while**. Возможно, наиболее распространенным являются циклы **for**, которые имеют следующий синтаксис:

```
for (начальное-выражение; проверочное-выражение; инкремент) {утверждения}
```

Начальное выражение используется для того, чтобы задать начальное значение счетчика количества итераций цикла. Для этих целей используется целочисленная переменная, которой по традиции присваивают имя **i**.

В каждой итерации цикла выполняется оценка проверочного выражения, и следующая итерация будет выполнена только в том случае, если значение этого выражения окажется равно **true**. Как только выражение получает значение **false**, цикл моментально прекращается и утверждения выполнены не будут. С каждой итерацией счетчик увеличивается, а затем выполняются утверждения.

Циклы могут быть вложенными, что позволяет выполнить все итерации внутреннего цикла на каждой итерации внешнего цикла.

forloop.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляются и инициализируются две целочисленные переменные, которые будут использованы впоследствии как счетчики цикла.

```
int main()
{
    int i, j;
}
```

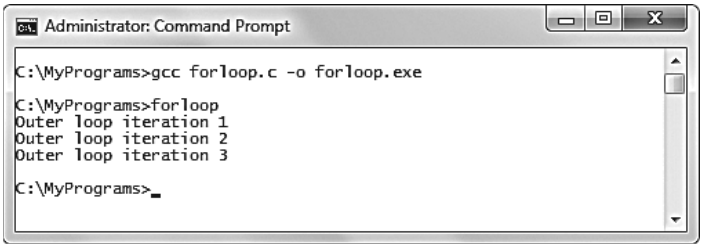
3. Далее в блоке функции **main** добавьте цикл **for**, позволяющий вывести на экран значение счетчика цикла на каждой из трех итераций.

```
for (i = 1; i < 4; i++)
{
    printf( "Outer loop iteration %d\n" , i ) ;
}
```

4. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

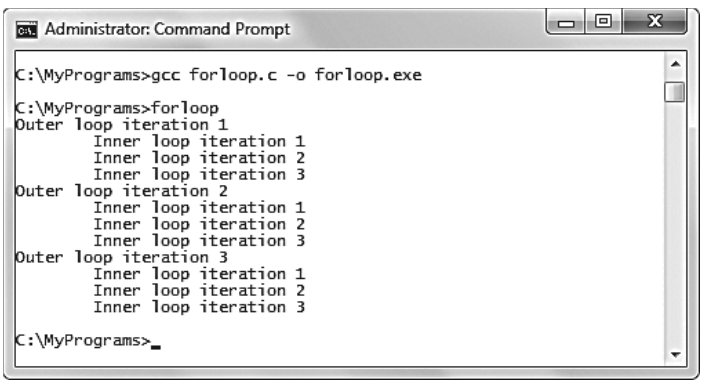
5. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть выведенные на экран значения счетчика цикла.



6. Теперь добавьте в блок цикла еще один цикл сразу после существующего утверждения, выводящего на экран количество итераций. В этом цикле выведите значение счетчика на каждой из трех итераций внутреннего цикла.

```
for (j = 1; j < 4; j++)
{
    printf( "\tInner loop iteration %d\n" , j ) ;
}
```

7. Сохраните файл программы, а затем скомпилируйте и сохраните ее, чтобы увидеть выведенные на экран значения счетчиков обоих циклов.



Совет

Начальное значение цикла **for** может также уменьшаться путем использования операции декремента — **i--** вместо **i++**.

Зацикливание с помощью условия

Цикл **while** является альтернативой циклу **for**, описанному в предыдущем примере. Цикл **while** также требует наличия начального выражения, проверочного выражения и инкремента, но они указываются не так явно, как в цикле **for**. Вместо того, чтобы расположиться в круглых скобках, начальное выражение должно находиться перед началом блока цикла, проверочное выражение должно быть помещено в круглые скобки после ключевого слова **while**, после которых располагаются фигурные скобки, содержащие в себе инкремент и утверждения, которые должны быть выполнены на каждой итерации:

```
начальное-выражение
while(проверочное-выражение)
{утверждения; инкремент}
```

Внимание

Обратите внимание на то, что после утверждения **while** в цикле **do while** необходимо поставить точку с запятой.

Цикл **do while** несколько отличается синтаксисом — ключевое слово **do** помещается перед блоком цикла, а утверждение **while** находится после блока цикла:

```
начальное-выражение
do {утверждения; инкремент}
while (проверочное-выражение);
```

Циклы **while** и **do while** будут выполнять свои итерации до тех пор, пока значение проверочного выражения не станет равно **false** — в этот момент цикл прервется. Поэтому необходимо, чтобы тело цикла содержало код, который в определенный момент изменит значение проверочного выражения, в противном случае создастся бесконечный цикл, который заблокирует систему.

Совет

Если так случилось, что запустился бесконечный цикл, в операционных системах Windows и Linux нажмите сочетание клавиш **Ctrl**+C, чтобы прекратить выполнение цикла.

Значительное различие между циклами **while** и **do while** заключается в том, что первый не сделает и единой итерации, если при стартовой оценке значение проверочного выражения равно **false**. Цикл **do while** напротив всегда будет делать хотя бы одну итерацию, поскольку утверждения выполняются до оценки выражения. Если такое поведение является необходимым, цикл **do while** очевидно является лучшим выбором, в противном случае выбор между циклами **for** и **while** является скорее делом вкуса. Как правило, циклами **for** пользуются, когда нужно выполнить определенное количество итераций, а циклами **while** — когда нужно выполнять итерации до тех пор, пока не будет соблюдено некоторое условие.

Циклы идеально подходят для работы с массивами, поскольку каждая итерация поможет легко считать или записать последовательные элементы массива.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода

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

2. Добавьте функцию **main**, в которой объявляется переменная счетчика, а также объявляется и инициализируется массив переменных из трех элементов

```
int main()
{
    int i, arr[3] = {10, 20, 30};
}
```

3. Далее в блоке функции **main** добавьте цикл **while**, позволяющий вывести на экран порядковый номер каждого элемента и его значение

```
i = 0;
while(i < 3)
{
    printf("While: arr[%d] = %d\n", i, arr[i]); i++;
}
```

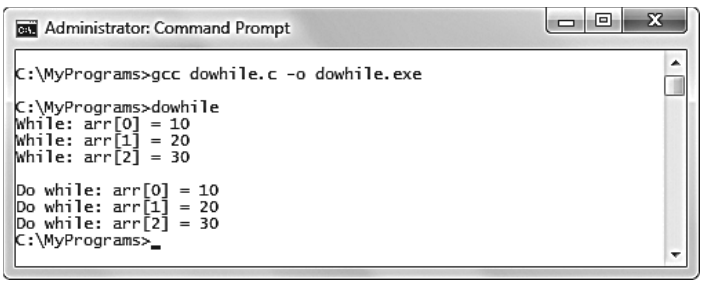
4. Далее добавьте цикл **do while**, который также выводит порядковый номер каждого элемента и его значение

```
i = 0;
do
{
    printf("\nDo while: arr[%d] = %d", i, arr[i]); i++;
}
while(i < 3);
```

5. В конце блока функции **main** верните значение 0, как того требует объявление функции

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть кажущиеся идентичными значения элементов массива из обоих циклов, выведенные на экран



dowhile.c

На заметку

Измените значение в проверочном выражении на 0 (`while(i < 0)`) в обоих циклах, а затем перекомпилируйте программу, чтобы увидеть, что будет выполнена только первая итерация цикла `do while`.

Досрочный выход из циклов

Ключевое слово **break** может быть использовано для того, чтобы преждевременно прекратить выполнение цикла при соблюдении определенного условия. Утверждение **break** находится внутри блока утверждений цикла, перед ним должно располагаться проверочное выражение. Когда проверочное выражение возвращает значение **true**, цикл немедленно заканчивается и программа переходит к выполнению следующей задачи. Например, при завершении внутреннего цикла таким способом начнется новая итерация внешнего цикла.

breakcontinue.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода

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

2. Добавьте функцию **main**, в которой объявляются две целочисленные переменные, которые в дальнейшем будут использоваться как счетчики цикла

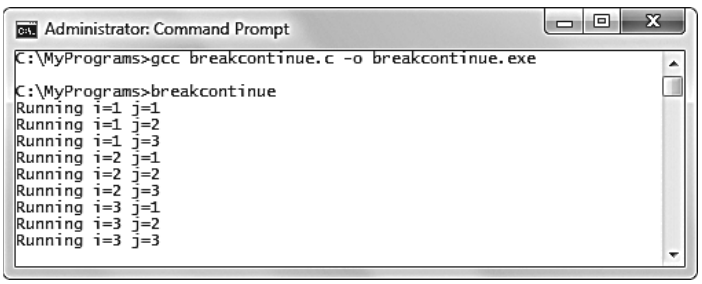
```
int main()
{
    int i, j;
}
```

3. Далее в блоке функции **main** добавьте два цикла, вложенных друг в друга, которые выводят значения их счетчиков на каждой из трех итераций

```
for(i = 1; i < 4; i++)
{
    for(j = 1; j < 4; j++)
    {
        printf( "Running i=%d j=%d\n", i, j ) ;
    }
}
```

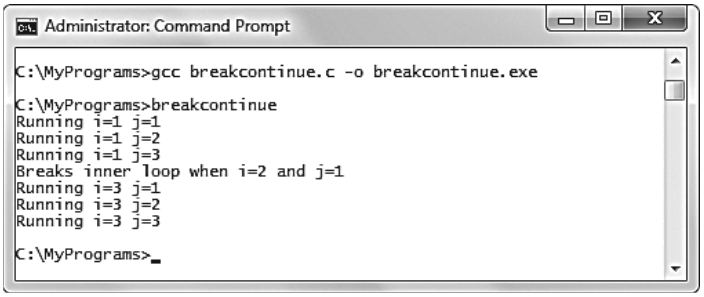
4. В конце блока функции **main** верните значение 0, как того требует объявление функции

```
return 0;
```



5. Сохраните файл программы, а затем скомпилируйте и сохраните ее, чтобы увидеть выведенные на экран значения обоих счетчиков циклов
6. Теперь добавьте утверждение **break** в самом начале блока внутреннего цикла, затем сохраните, скомпилируйте и запустите программу еще раз.

```
if(i == 2 && j == 1)
{
    printf( "Breaks inner loop when i=%d and j=%d\n" , i, j ) ;
    break;
}
```



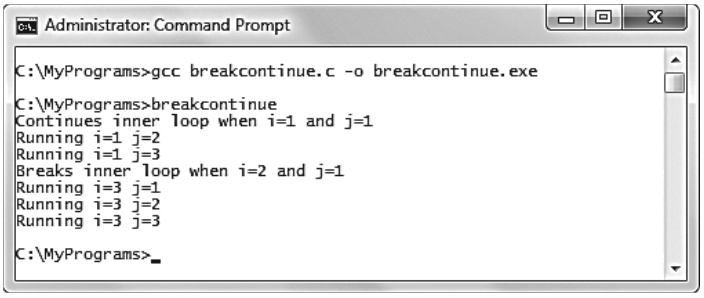
На заметку

В этом примере утверждение **break** прерывает все три итерации внутреннего цикла, когда внешний цикл пытается запустить его во второй раз.

Ключевое слово **continue** может использоваться для того, чтобы пропустить одну итерацию цикла, если соблюдено некоторое условие. Утверждение **statement** располагается внутри блока утверждений, перед ним находится проверочное выражение. Когда проверочное выражение возвращает значение **true**, заканчивается одна итерация цикла.

7. Добавьте утверждение **continue** в начало блока внутреннего цикла, чтобы пропустить первую итерацию внутреннего цикла — затем сохраните, скомпилируйте и запустите программу снова

```
if(i == 1 && j == 1)
{
    printf( "Continues inner loop when i=%d and j=%d\n", i, j ) ;
    continue;
}
```



На заметку

В этом примере утверждение **continue** просто пропускает первую итерацию внутреннего цикла, когда внешний пытается запустить его в первый раз.

Переход к меткам

Ключевое слово `goto` в соответствии со своим названием позволяет потоку программы переходить к меткам, расположенным в других частях программы, примерно как гиперссылка на веб-странице. Однако в реальности это может привести к ошибкам и считается плохим приемом программирования.

Переход с помощью ключевого слова `goto` — это мощная функциональность, которая существовала в компьютерных программах десятилетиями, но ее мощью злоупотребляли многие первые программисты, создававшие программы, в которых переходы выполнялись непостижимым образом. Это приводило к написанию нечитаемого программного кода, поэтому использование ключевого слова `goto` стало весьма непопулярным.

Одним из возможных корректных использований ключевого слова `goto` может быть выход их внутреннего цикла путем перехода к метке, расположенной после блока внешнего цикла. Это мгновенно завершит оба цикла, ни одна из итераций любого цикла не будет выполнена.

`jump.c`

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода

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

2. Добавьте функцию `main`, в которой объявляются две целочисленные переменные, которые в дальнейшем будут использоваться как счетчики цикла

```
int main()
{
    int i, j;
}
```

3. Далее в блоке функции `main` добавьте два цикла, вложенных друг в друга, которые выводят значения их счетчиков на каждой из трех итераций

```
for(i = 1; i < 4; i++)
{
    for(j = 1; j < 4; j++)
    {
        printf( "Running i=%d j=%d\n", i, j ) ;
    }
}
```

На заметку

Обратите внимание на то, что рядом с утверждением `goto` указывается только имя метки. Сама метка должна заканчиваться двоеточием.

4. В первую строку внутреннего цикла добавьте утверждение, позволяющее перейти к метке с именем `end` при определенных значениях счетчиков

```
if(i == 2 && j == 1) {goto end;}
```

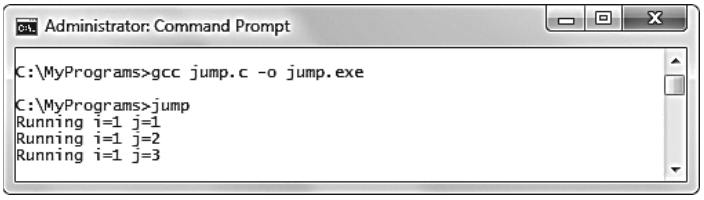
5. Теперь добавьте метку после закрывающей скобки внешнего цикла

```
} end:
```

6. В конце блока функции `main` верните значение 0, как того требует объявление функции

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и сохраните ее, чтобы увидеть, что после перехода к метке циклы завершатся.



Если вы не хотите использовать ключевое слово `goto`, существует альтернативный вариант — проверять значение переменной на равенство булеву значению `true` (1) на каждой итерации каждого цикла.

8. Добавьте еще одну переменную к уже существующим объявлениям и инициализируйте ее значением 1

```
int i, j, flag = 1;
```

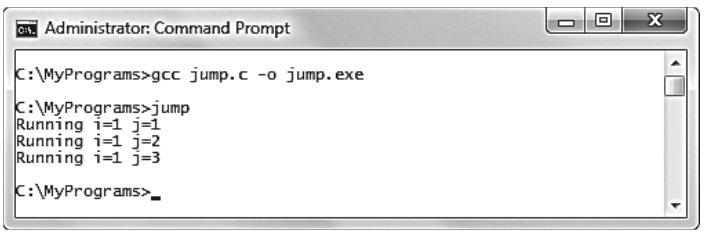
9. Добавьте условную проверку значения новой переменной в начале блока внешнего цикла и в начале блока внутреннего цикла прямо перед утверждением, выводящим текст

```
if (flag)
```

10. Измените первую строку внутреннего цикла, заменив утверждение `goto` другим, присваивающим переменной значение 0

```
if(i == 2 && j == 1) {flag = 0;}
```

11. Теперь можно удалить ставшую бесполезной метку `end:`, располагавшуюся после закрывающей фигурной скобки, а затем сохраните, скомпилируйте и выполните программу снова, чтобы увидеть на экране точно такой же результат



Совет

В языке программирования C булевы значения представлены числами 1 (`true`) и 0 (`false`).

Внимание

Избегайте создания исполняемого файла программы с именем goto.exe, поскольку это может привести к конфликтам с внутренней командой Windows `goto`.

Заключение

- Ключевое слово **if** выполняет простую условную проверку, чтобы оценить значение заданного выражения — оно может быть **true** или **false**.
- Ключевое слово **else** разрешается использовать для предоставления альтернативного набора утверждений, которые должны быть выполнены в случае, если утверждение **if** оценит значение как **false**.
- Предоставление программе альтернативных направлений выполнения после оценки называется *условным ветвлением*.
- Условное ветвление, выполняемое с помощью нескольких утверждений **if else**, может быть выполнено более эффективно с помощью утверждения **switch**.
- Обычно утверждения **case** внутри блока **switch** должны заканчиваться утверждением **break**.
- Опционально блок **switch** может содержать утверждение **default**, с помощью которого указываются утверждения, которые должны быть выполнены в случае отсутствия совпадений.
- После ключевого слова **for** следуют скобки, в которых указывается начальное выражение, проверочное выражение и инкремент, позволяющие управлять циклом.
- После ключевого слова **while** следуют скобки, в которых указывается проверочное выражение, позволяющее определить, должен ли цикл продолжаться.
- Перед блоком цикла **while** должно располагаться начальное выражение, а внутри него — инкремент.
- После ключевого слова **do** следует блок утверждений, после которого обязательно нужно добавить утверждение **while**, которое должно заканчиваться двоеточием.
- Перед циклом **do while** должно находиться начальное выражение, а внутри него — инкремент.
- В отличие от циклов **for** и **while** утверждения цикла **do while** всегда будут выполнены хотя бы один раз.
- Ключевое слово **break** может использоваться для того, чтобы завершить цикл, а ключевое слово **continue** — для того, чтобы пропустить одну итерацию цикла.
- Циклы могут быть вложенными друг в друга, и ключевое слово **goto** может быть использовано для того, чтобы выйти из них всех и перейти к указанной метке, хотя использовать именно это ключевое слово не рекомендуется.

6 Использование функций

Здесь показывается, как следует разместить утверждения внутри функции, чтобы их можно было выполнить в любой момент по запросу программы.

- **Объявление функций**
- **Передача аргументов**
- **Рекурсивные вызовы**
- **Размещение функций в заголовках**
- **Ограничение доступности**
- **Заключение**

Объявление функций

В предыдущих примерах этой книги использовали обязательную функцию `main()` и стандартные функции, содержащиеся в библиотеке заголовочных файлов, такие как `printf()` из файла `stdio.h`. Однако в большинстве программ, написанных на языке C, содержится некоторое количество пользовательских функций, которые могут быть вызваны по требованию во время выполнения программы.

Совет

Прототип функции иногда называют ее *заголовком*.

Блок функции содержит только набор утверждений, который выполняется при вызове функции. Как только утверждения функции выполняются полностью, поток программы продолжает свою работу с точки, следующей непосредственно после вызова функции. Такая модульность очень полезна при программировании на языке C, поскольку она позволяет изолировать набор процедур, чтобы впоследствии их можно было вызвать не один раз.

Для того чтобы ознакомить программу с пользовательской функцией, последнюю необходимо объявить, точно так же, как и переменные, которые необходимо сначала объявить, а лишь затем использовать. Объявления функций должны быть добавлены перед блоком функции `main()`.

Как и функция `main`, пользовательские функции могут возвращать значения. Тип данных возвращаемого значения должен быть включен в объявление функции. Если функция не возвращает никакого значения, она должна быть объявлена с помощью ключевого слова `void`. Имя функции следует выбирать исходя из соглашений именования переменных.

Объявление функции часто более корректно называют *прототипом функции*, с его помощью мы можем информировать компилятор о том, что функция существует. Определение же функции, включающее в себя утверждения, которые необходимо выполнить, располагается после функции `main`. Пользовательские функции могут быть вызваны из функции `main` для того, чтобы выполнить их утверждения.

Значение, возвращаемое пользовательской функцией, может быть присвоено переменной подходящего типа данных или просто отображено с использованием подходящего спецификатора формата.

`first.c`

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Объявите три пользовательских прототипа функции.

```
void first();
int square5();
int cube5();
```

3. Добавьте функцию **main**, в которой объявляется целочисленная переменная.

```
int main()
{
    int num;
}
```

4. После функции **main** определите пользовательские функции.

```
void first()
{
    printf( "Hello from the first function\n" ) ;
}
int square5()
{
    int square = 5 * 5;
    return square;
}
int cube5()
{
    int cube = (5 * 5) * 5;
    return cube;
}
```

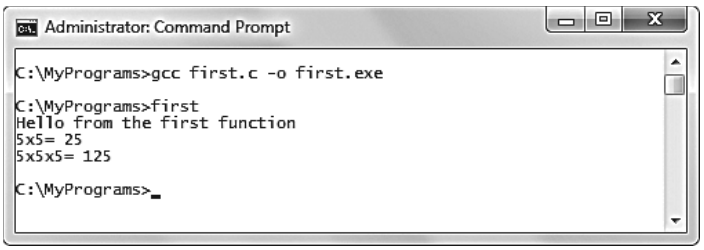
5. Теперь добавьте вызовы пользовательских функций в блок функции **main**.

```
first();
num = square5();
printf("5x5= %d\n", num);
printf("5x5x5= %d\n", cube5());
```

6. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть выведенный на экран результат работы пользовательских функций.



Внимание

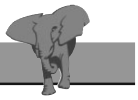
Обратите внимание на то, что каждый прототип функции должен заканчиваться точкой с запятой.

Совет

Определения пользовательских функций технически должны появляться перед функцией **main**(), но по соглашению там следует писать только прототипы, а функцию **main** размещать в начале кода.

Передача аргументов

В пользовательские функции данные могут быть переданы как *аргументы*. Там они могут использоваться для выполнения их утверждений. Прототип функции должен включать имя и тип данных каждого аргумента.



На заметку

Передача данных по значению присваивает значение переменной в вызываемой функции. Далее функция может работать с этой копией, но оригинальные данные она не затронет.

Важно понимать, что данные в языке C передаются *по значению* в переменную, указанную как аргумент функции. Это отличает язык программирования C от других языков программирования, таких как Pascal, где аргументы передаются *по ссылке* — функция работает с оригинальным значением, а не локальной копией.

Аргументы в прототипе функции называются формальными параметрами функции. Они могут иметь разные типы данных, несколько аргументов могут быть указаны для одной функции, их следует разделять запятой. Например, прототип функции с аргументами, имеющими каждый из четырех типов данных, может выглядеть так:

```
void action(char c, int I, float f, double d);
```

Компилятор проверяет, совпадают ли указанные в прототипе функции формальные параметры с параметрами в определении функции. Он сообщит об ошибке в случае несовпадения.

args.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Объявите три пользовательских прототипа функции, в каждую из которых при вызове передается один аргумент.

   ```
   void display(char str[]);

   int square(int x);

   int cube(int y);
   ```

3. Добавьте функцию `main`, в которой объявляется целочисленная переменная и массив символьных переменных, который инициализируется текстовой строкой.

   ```
   int main()

   {

       int num;

       char msg[50] = "String to be passed to a function" ;

   }
   ```

4. После функции **main** определите пользовательские функции.

```
void display(char str[])
{
    printf("%s\n", str);
}
int square(int x)
{
    return x * x;
}
int cube(int y)
{
    return (y * y) * y;
}
```

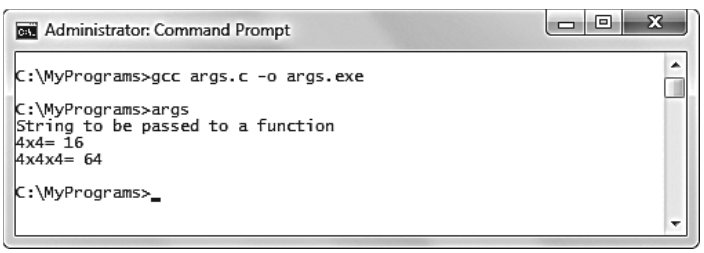
5. Теперь добавьте вызовы пользовательских функций в блок функции **main**.

```
display(msg);
num = square(4);
printf("4x4= %d\n", num);
printf("4x4x4= %d\n", cube(4));
```

6. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и сохраните программу, чтобы увидеть выведенный на экран результат работы пользовательских функций, использующих переданные значения аргументов.



Совет

Имена аргументов, использованных в определении функции, могут отличаться от имен аргументов, указанных в прототипе функции, но типы данных аргументов, их количество и порядок должны оставаться одинаковыми. Впрочем, использование одинаковых имен позволит сделать код более прозрачным.

Совет

Функция может не возвращать значения, но по-прежнему использовать слово **return**, рядом с которым не указано никаких значений. Это делается только для того, чтобы явно обозначить возврат управления вызывающей стороне.

Рекурсивные вызовы

Утверждения, находящиеся внутри пользовательских функций, могут свободно вызывать другие пользовательские функции точно так же, как они могут вызывать стандартные библиотечные функции наподобие `printf()`.

Также функции могут вызывать сами себя, это называется *рекурсией*. Как и в случае с циклами, очень важно, чтобы рекурсивные функции изменяли проверочное выражение, чтобы функция в конечном итоге завершилась.

`recur.c`

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Объявите пользовательский прототип функции, в которую при вызове передается один аргумент.

```
void count_down_from(int num);
```

3. Добавьте функцию `main`, в которой объявляется целочисленная переменная.

```
int main()
{
    int start;
}
```

4. Далее в блоке функции `main` добавьте утверждение, запрашивающее у пользователя целочисленное значение для переменной.

```
printf( "Enter a positive integer to count down from: " ) ;
scanf("%d", &start);
```

5. Теперь в блоке функции `main` добавьте вызов пользовательской функции, передав туда значение, введенное пользователем.

```
count_down_from(start);
```

6. В блоке функции `main` добавьте утверждение, выводящее сообщение в момент, когда управление возвращается из пользовательской функции.

```
printf( "Lift Off!\n" ) ;
```

На заметку

При использовании функции `scanf()` имя переменной должен предварять оператор адресации `&`, как говорится в главе 2.

7. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

8. После блока функции **main** начните определять пользовательскую функцию с утверждения, позволяющего вывести на экран значение аргумента, переданное при вызове функции.

```
void count_down_from(int num)
{
    printf("%d\n", num);
}
```

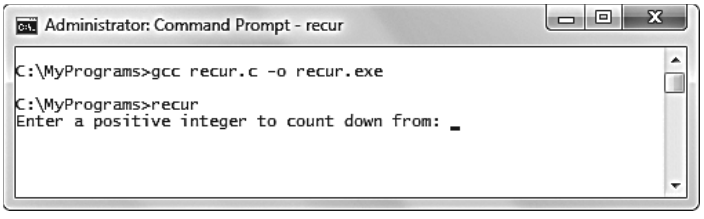
9. Далее в блоке пользовательской функции уменьшите значение, переданное как аргумент.

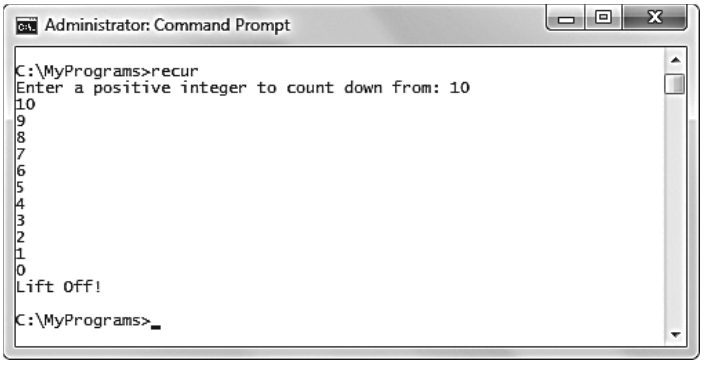
```
--num;
```

10. Теперь в блоке пользовательской функции добавьте условную проверку, позволяющую вернуть управление функции **main**, если уменьшенное значение меньше нуля, или снова передать это значение как аргумент путем рекурсивного вызова функции.

```
if(num < 0) return;
else
count_down_from(num);
```

11. Сохраните файл программы, а затем скомпилируйте и запустите программу, введя по требованию целое число, чтобы увидеть выведенный на экран результат работы функции, вызванной рекурсивно.





Внимание

Использование рекурсивных функций может быть менее эффективно, чем использование циклов.

Размещение функций в заголовках

Программы в примерах, приведенных в этой книге, специально сделаны небольшими из-за того, что размер книги ограничен, но в действительности большинство программ, написанных на языке C, будут содержать гораздо больше кода.

При разработке крупных программ следует продумать структуру программы. Поддержка программного кода, размещенного в одном файле, может стать неудобной по мере развития программы.

Для того чтобы упростить структуру программы, разрешается создать пользовательский заголовочный файл, содержащий в себе функции, которые могут быть использованы много раз. Такой файл должен иметь расширение `.h`, как и обычные заголовочные файлы библиотеки C.

Функции, расположенные в пользовательском заголовочном файле, могут быть доступны программе, если добавить в начало файла, содержащего функцию `main()`, директиву препроцессора `#include`. Имя пользовательского заголовочного файла должно размещаться в двойных кавычках (не следует использовать символы `<` и `>`, которые используются для стандартных заголовочных файлов).

utils.h

1. Создайте пользовательский заголовочный файл с именем `utils.h`, содержащий определение одной функции.

```
int square(int num)
{
    return (num * num);
}
```

square.c

2. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

3. Объявите пользовательский прототип функции, не имеющей аргументов.

```
void getnum();
```

4. Добавьте функцию `main`, в которой вызывается другая функция, а затем возвращается число 0, чего требует ее объявление.

```
int main()
{
    getnum();
    return 0;
}
```

5. После блока функции `main` добавьте второе определение функции, в которой объявляются две переменные.

```
void getnum()
{
    int num;
    char again;
}
```

6. Далее в этой функции добавьте утверждения, запрашивающие у пользователя данные, которые впоследствии будут присвоены переменной.

```
printf( "Enter an integer to be squared: " ) ;
scanf("%d", &num);
```

7. Теперь выведите результат вызова функции из пользовательского заголовочного файла, передав в нее как аргумент введенное число.

```
printf( "%d squared is %d\n" , num, square(num) ) ;
```

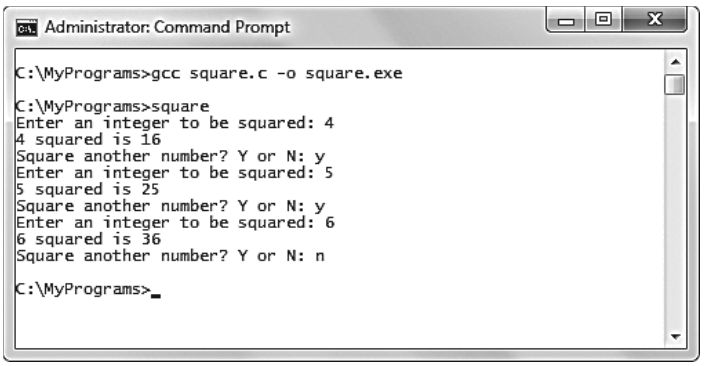
8. Добавьте утверждения, запрашивающие у пользователя данные, которые впоследствии будут присвоены переменной.

```
printf( "Square another number? Y or N: " ) ;
scanf("%1s", &again);
```

9. Наконец, добавьте условную проверку, позволяющую выполнить эту функцию еще раз или вернуть управление функции `main`.

```
if((again == 'Y') || (again == 'y')) getnum();
else return;
```

10. Сохраните файл программы, а затем скомпилируйте и запустите программу, введя по требованию целое число и символ, чтобы увидеть выведенный на экран результат работы функций.



На заметку

Файл программы и пользовательский заголовочный файл должны находиться в одной директории, но компилируются они с помощью обычный команды — компилятор считывает заголовочный файл автоматически благодаря директиве `#include`.

Совет

Обратите внимание на то, что спецификатор формата `%1s` в этом примере используется для того, чтобы считать следующий символ, введенный пользователем.

Ограничение доступности

Ключевое слово **static** может быть использовано для того, чтобы ограничить доступность функций рамками файла, в котором они были созданы, точно так же, как и для переменных.

Этот прием рекомендуется использовать в больших программах, которые располагаются в нескольких файлах **.c**, чтобы оградить их от случайного использования их функций. Например, функции **square()** и **multiply()** не могут быть вызваны непосредственно из функции **main** в этом примере.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

menu.c

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

2. Объявите пользовательский прототип функции, в которую при вызове не передается никаких аргументов.

```
void menu();
```

3. Добавьте функцию **main**, в которой вызывается пользовательская функция, а затем возвращается число 0, чего требует объявление функции.

```
int main()
{
    menu();
    return 0;
}
```

4. После блока функции **main** определите пользовательскую функцию, позволяющую передать номер пункта меню как аргумент в другую функцию.

```
void menu()
{
    int option();
    printf( "\n\tWhat would you like to do?" ) ;
    printf( "\n\t1. Square a number" ) ;
    printf( "\n\t2. Multiply two numbers" ) ;
    printf( "\n\t3. Exit\n" ) ;
    scanf("%d", &option);
    action(option);
}
```

5. Теперь начните второй файл программы с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

action.c

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

6. Далее определите две простые статические функции

```
static square(int a) {return (a * a); }
static multiply(int a, int b) {return a * b; }
```

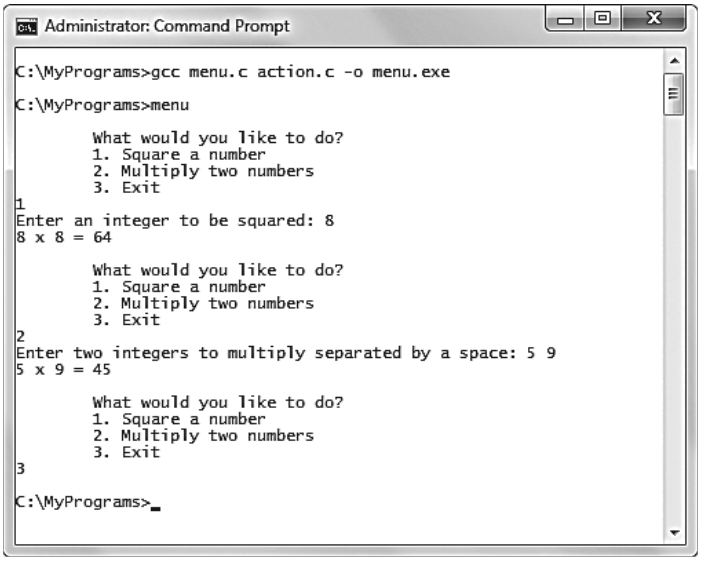

Совет

Пример, в котором показывается работа статических переменных, рассматривался в главе 2.

7. Теперь определите функцию, в которую станет передаваться вариант меню из функции **main**, на основе чего будет выполняться соответствующее действие — вызов одной из статических функций, созданных в этой файле.

```
void action(int option)
{
    int n1, n2;
    if(option == 1)
    {
        printf( "Enter an integer to be squared: " ) ;
        scanf("%d", &n1);
        printf("%d x %d = %d \n", n1, n1, square(n1));
        menu();
    }
    else if (option == 2)
    {
        printf( "Enter two integers to multiply " ) ;
        printf( "separated by a space: " ) ;
        scanf("%d", &n1); scanf("%d", &n2);
        printf("%d x %d = %d\n", n1, n2, multiply(n1, n2));
        menu();
    }
    else return;
}
```

8. Сохраните оба файла, а затем скомпилируйте и сохраните программу, чтобы увидеть выведенный на экран результат работы статических функций.



Совет

Обратите внимание на то, что для функций, расположенных вне файла, содержащего функцию `main()`, прототипы не нужны.

На заметку

Обратите внимание на то, что в команде компилятора упоминаются два файла исходного кода, но создается только один исполняемый файл.

Заключение

- Пользовательские функции объявляются путем указания типа данных, которые будут возвращены функцией, затем ее имени, а затем скобок. Закончить конструкцию следует точкой с запятой.
- Объявления функций также называются прототипами функций, они должны располагаться перед функцией `main` — поэтому компилятор будет знать об их существовании при чтении функции `main()`.
- Определения функций, непосредственно содержащие утверждения, которые необходимо выполнить, когда вызывается функция, должны располагаться после блока функции `main()`.
- Объявления функций опционально могут иметь внутри скобок разделенный запятыми список аргументов, передаваемых вызываемой стороной, для каждого из которых необходимо указать тип данных и имя.
- Аргументы, указанные в определении функции, должны соответствовать аргументам в описании, поскольку последние являются их формальными параметрами.
- В программировании на языке C аргументы передаются по значению — функция работает только с копией оригинального значения.
- Функция может рекурсивно вызывать саму себя, в этом случае она должна содержать утверждение, изменяющее проверочное выражение, чтобы в определенный момент завершиться.
- Пользовательские заголовочные файлы должны иметь расширение `.h`.
- Если с помощью директивы препроцессора `#include` добавляются пользовательские заголовочные файлы, имя файла указывается в двойных кавычках.
- Ключевое слово `static` может быть использовано в объявлениях и описаниях функций, чтобы ограничить к ним доступ рамками файла, в котором они находятся.
- Крупные программы должны объявлять функции с помощью ключевого слова `static`, если только нет какой-то особенной причины, по которой функция должна быть видима за пределами файла.
- Указывать прототипы необязательно для функций, располагающихся за пределами файла, содержащего функцию `main()`.

7 Указатели

Здесь показывается, как можно обратиться к данным путем использования их адреса в памяти.

- **Получение доступа к данным с помощью указателей**
- **Арифметика указателей**
- **Передача указателей в функции**
- **Создание массивов указателей**
- **Указатели на функции**
- **Заключение**

Получение доступа к данным с помощью указателей

Указатели — это очень полезный элемент языка C для эффективного программирования. Они являются переменными, хранящими адрес в памяти других переменных.

Когда объявляется обычная переменная, для нее выделяется некоторый объем памяти в соответствии с ее типом данных в свободном участке памяти. После этого, когда программа встречает имя переменной, она обращается к данным по ее адресу. Аналогично, когда программа встречает имя переменной-указателя, она обращается к данным, хранящимся по адресу, хранящемуся в ней. Но переменную-указатель разрешается *разыменовать*, чтобы можно было обратиться к данным, хранящимся по адресу, который лежит в указателе.

Совет

Операция разыменования, `*`, также известна как *косвенная* операция.

Переменные-указатели объявляются точно так же, как и другие переменные, но к имени указателя добавляется символ `*`. В этом случае символ `*` представляет собой операцию разыменования и указывает, что объявленная переменная является указателем.

После того, как переменная-указатель была объявлена, ей можно назначить адрес другой переменной с помощью операции адресации `&`. Перед переменной-указателем при присвоении ей значения операцию разыменования, `*`, помещать не нужно, если только инициализация не происходит непосредственно после объявления переменной-указателя.

- Имя переменной-указателя само по себе представляет адрес в памяти, выраженный в шестнадцатеричном формате.

Внимание

Убедитесь, что удалили указатель после удаления объекта, к которому он обращается, чтобы не оставлять в программе *висящие указатели*.

Когда операция разыменования, `*`, используется при объявлении переменной, он указывает, что переменная является указателем, но использование этой операции в другом месте программы вызовет обращение к данным, хранящимся по адресу, лежащему в переменной-указателе.

- С помощью имени переменной-указателя можно обратиться к данным, хранящимся по адресу, присвоенному этой переменной, если перед ним стоит операция разыменования, `*`.

Это означает, что программа способна получить адрес, присвоенный переменной-указателю, с помощью ее имени. Также она может получить доступ к данным, лежащим по адресу, который хранится в переменной-указателе, если поместить операцию разыменования, `*`, перед именем этой переменной.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода

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

2. Добавьте функцию **main**, в которой объявляется и инициализируется обычная целочисленная переменная и целочисленный указатель

```
int main()
{
    int num = 8;
    int *ptr = &num;
}
```

3. Далее в блоке функции **main** выведите содержимое обеих переменных, а также значение, к которому обращается указатель

```
printf( "Regular variable contains: %d\n" , num ) ;
printf( "Pointer variable contains: 0x%p\n" , ptr ) ;
printf( "Pointer points to value: %d\n\n" , *ptr ) ;
```

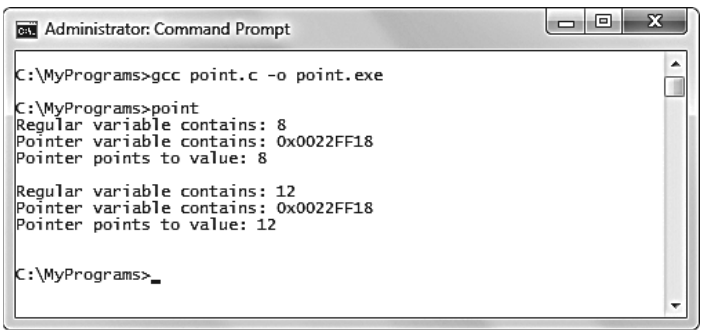
4. Теперь в блоке функции **main** присвойте новое значение обычной переменной с помощью указателя, а затем еще раз выведите ее содержимое и значение, к которому обращается указатель

```
*ptr = 12;
printf( "Regular variable contains: %d\n" , num ) ;
printf( "Pointer variable contains: 0x%p\n" , ptr ) ;
printf( "Pointer points to value: %d\n\n" , *ptr ) ;
```

5. В конце блока функции **main** верните значение 0, как того требует объявление функции

```
return 0;
```

6. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значение переменной и значение, к которому обращается указатель



point.c

На заметку

Объявляйте переменные до того, как в ход пойдут другие утверждения — например, в начале функции `main()`.

Арифметика указателей

После того, как указателю присваивается адрес в памяти, этот адрес можно изменить, присвоив ему другой адрес или воспользовавшись арифметическими операциями.

Операции инкремента, **++**, и декремента, **--**, сместят указатель вперед или назад к другому адресу в памяти для заданного типа данных — чем больше тип данных, тем больше прыжок.

Более длинные прыжки можно осуществить с помощью операций **+=** и **-=**, которые позволяют указать, на сколько позиций следует сместиться.

Арифметика указателей особенно полезна при работе с массивами, поскольку элемента массива занимают место в памяти последовательно.

Присвоение имени массива указателю автоматически присваивает ему адрес в памяти первого элемента этого массива. Увеличение значения указателя на 1 переместит его к следующему элементу массива.

movptr.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляется и инициализируется целочисленная переменная, а затем объявляется и инициализируется массив целочисленных переменных.

```
int main()

{

    int i;

    int nums[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

}
```

Внимание

Операции ***=** и **/=** не могут быть использованы для того, чтобы перемещать указатель.

3. Далее в блоке функции **main** объявите переменную-указатель и проинициализируйте ее адрес первого элемента массива, затем выведите этот адрес и значение, которое располагается по этому адресу в памяти.

```
int *ptr = nums;

printf( "\nAt Address: %p is Value: %d\n", ptr , *ptr ) ;
```

4. Теперь в блоке функции **main** увеличивайте значение указателя на 1, чтобы перемещать его к следующему элементу массива один за другим.

```
ptr++;

printf( "At Address: %p is Value: %d\n", ptr , *ptr ) ;

ptr++;

printf( "At Address: %p is Value: %d\n", ptr , *ptr ) ;
```

5. Теперь вернитесь на два элемента назад, чтобы указатель адресовал первый элемент массива

```
ptr -= 2;

printf( "At Address: %p is Value: %d\n\n", ptr , *ptr ) ;
```

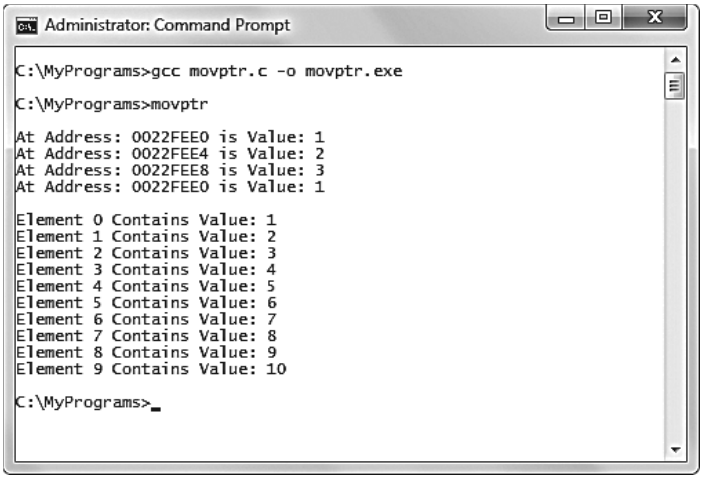
6. Далее добавьте цикл и выведите номер каждого элемента массива, а также их значения

```
for (i = 0; i < 10; i++)

{

   printf( "Element %d Contains Value: %d\n" , i , *ptr ) ;

   ptr++;

}
```

7. В конце блока функции **main** верните значение 0, как того требует объявление функции

```
return 0;
```

8. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значение переменной и значение, к которому обращается указатель



На заметку

Имя массива выступает в качестве указателя на его первый элемент.

Передача указателей в функции

В программах, написанных на языке C, аргументы функций передаются *по значению* в локальную переменную внутри вызываемой функции. Это значит, что функция работает не с оригинальным значением, а с его копией.

Передача указателя на оригинальное значение позволяет функции работать с оригинальным значением *по ссылке* и является основным преимуществом использования указателей.

passptr.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Объявите два прототипа пользовательских функций, в каждую из которых передается один целочисленный указатель.

```
void twice(int *ptr);

void thrice(int *ptr);
```

3. Добавьте функцию `main`, в которой объявляется и инициализируется обычная целочисленная переменная и целочисленный указатель, в котором содержится адрес предыдущей переменной.

```
int main()

{

    int num = 5;

    int *ptr = &num;

}
```

Внимание

Обратите внимание на то, что аргумент-указатель должен быть включен в объявление прототипа функции.

4. Далее в блоке функции `main` выведите адрес, хранящийся в указателе, а также значение, на которое ссылается указатель.

```
printf( "ptr stores address: %p\n" , ptr ) ;

printf( "*ptr dereferences value: %d\n\n" , *ptr ) ;
```

5. Теперь выведите оригинальное значение обычной целочисленной переменной.

```
printf( "The num value is %d\n" , num ) ;
```

6. После блока функции `main` определите две пользовательские функции, объявленные с помощью прототипов, каждая из которых получает в качестве аргумента целочисленный указатель.

```
void twice(int *number)
{
    *number = (*number * 2);
}
void thrice(int *number)
{
    *number = (*number * 3);
}
```

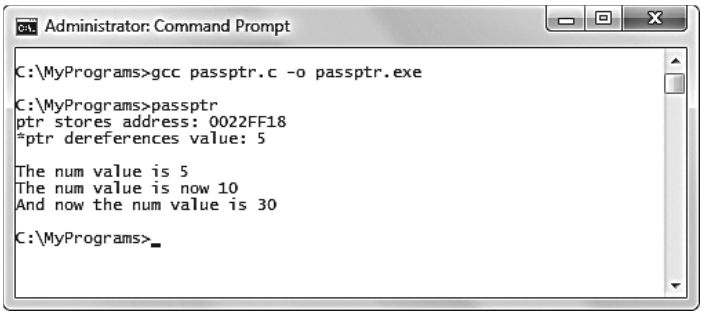
7. В блоке функции `main` добавьте вызовы пользовательских функций, передавая адрес обычной переменной как ссылку, а затем выведите измененное значение обычной переменной.

```
twice(ptr);
printf( "The num value is now %d\n", num ) ;
thrice( ptr ) ;
printf( "And now the num value is %d\n", num ) ;
```

8. В конце блока функции `main` верните значение 0, как того требует объявление функции.

```
return 0;
```

9. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть на экране значения переменной после передачи ссылки на него.



На заметку

Символ *, находясь в скобках определения функции, показывает, что аргумент является указателем, а находясь внутри утверждения, с его помощью можно получить аргумент по ссылке. В арифметическом утверждении этот символ является операцией умножения.

Создание массивов указателей

Программа, написанная на языке C, способна содержать массивы указателей, в каждом элементе которого хранятся адреса других переменных.

Эта возможность особенно полезна для работы с символьными строками. Массив символов, который заканчивается символом `\0` носит статус строки, поэтому его можно присвоить переменной-указателю. Имя символьного массива служит как указатель на его первый элемент, поэтому не требуется операция адресации `&` для того, чтобы присвоить строку переменной-указателю.

arrptr.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию `main`, в которой объявляется обычная целочисленная переменная, а также объявляется и инициализируется массив целочисленных переменных.

```
int main()

{

    int i;

    int nums[5] = {1, 2, 3, 4, 5};

}
```

3. Далее в блоке функции `main` объявите и проинициализируйте целочисленные переменные-указатели, содержащие адрес каждого элемента.

```
int *ptr0 = &nums[0];

int *ptr1 = &nums[1];

int *ptr2 = &nums[2];

int *ptr3 = &nums[3];

int *ptr4 = &nums[4];
```

Совет

Обратите внимание на то, как значение счетчика i заменяется номером элемента массива на каждой итерации цикла.

4. Теперь объявите и проинициализируйте массив целочисленных указателей, каждый элемент которого будет содержать один из целочисленных указателей.

```
int *ptrs[5] = {ptr0, ptr1, ptr2, ptr3, ptr4};
```

5. Далее объявите и проинициализируйте массив символов, указатель на этот массив и массив символьных указателей, содержащий строку внутри каждого элемента

```
char str[9] = { 'C', ' ', 'i', 's', ' ', 'F', 'u', 'n', '\0' } ;

char *string = str ;

char *strings[3] = { "Alpha", "Bravo", "Charlie" } ;
```

6. Добавьте цикл, с помощью которого выводится адрес каждого элемента массива целочисленных указателей и значения, на которые они ссылаются.

```
for(i = 0; i < 5; i++)

{

    printf( "The value at %p is: %d\n" , ptrs[i], *ptrs[i] ) ;

}
```

7. Добавьте утверждение, выводящее на экран значение, хранящееся в массиве символов.

```
printf( "\nString is: %s\n\n" , string ) ;
```

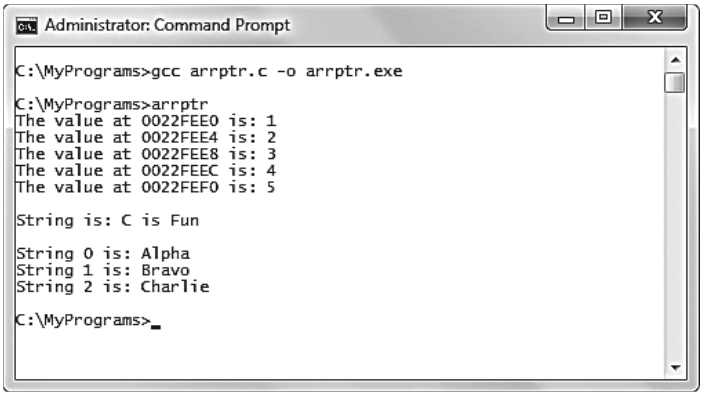
8. Добавьте цикл, с помощью которого выведите строки, содержащиеся в каждом элементе массива символьных указателей.

```
for(i = 0; i < 3; i++)

{

    printf( "String %d is: %s\n" , i , strings[i] ) ;

}
```

9. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

10. Сохраните файл программы, а затем скомпилируйте и запустите ее, чтобы увидеть на экране значения массивов указателей.



На заметку

С помощью имени указателя можно обратиться ко всей строке, хранящейся в символьном массиве, не прибегая к использованию операции *.

Внимание

Для того чтобы добавить в строку символ пробела, необходимо использовать комбинацию символов ' '. Две одинарные кавычки, расположенные рядом — '', рассматриваются как пустой элемент и вызывают ошибку компилятора.

Указатели на функции

Можно также создать указатели, указывающие на функции, хотя эта возможность используется реже, чем создание обычных указателей.

Указатель на функцию похож на указатель на данные, но он всегда должен помещаться в скобки при использовании операции разыменования, *, чтобы избежать возникновения ошибок компилятора. После этой конструкции также должны следовать скобки, содержащие аргументы, которые должны быть переданы в функцию, на которую ссылается указатель.

Указатель на функцию содержит адрес в памяти, по которому лежит начало функции. Когда указатель на функцию разыменовывается, вызывается функция, на которую он ссылается, и аргументы передаются в вызываемую функцию.

Указатель на функцию может быть передан как аргумент в другую функцию. Функция, получившая такой аргумент, способна вызвать функцию, на которую ссылается указатель.

fcnptr.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Объявите прототип одной пользовательской функции, которая имеет целочисленный аргумент, и прототип другой функции, имеющей в качестве аргументов указатель на функцию и целое число.

```
int bounce(int a);

int caller((int(*function) (int), int b);
```

3. Добавьте функцию main, в которой объявляется обычная целочисленная переменная, а также объявляется и инициализируется переменная-указатель на функцию.

```
int main()

{

    int num;

    int (*fptr)(int) = bounce;

}
```

4. После блока функции main определите первую пользовательскую функцию, которая выводит на экран полученное значение и возвращает целое число.

```
int bounce(int a)
{
    printf( "\nReceived Value: %d\n", a ) ;
    return((3 * a) + 3);
}
```

5. Далее определите вторую пользовательскую функцию, вызывающую обычную функцию из полученного указателя на функцию и передающую ей полученное целочисленное значение.

```
int caller(int (*function)(int), int b)
{
    (function*)(b);
}
```

6. В блоке функции `main` присвойте значение целочисленной переменной с помощью вызова обычной функции через указатель на нее и выведите значение, которое она вернет.

```
num = (*fptr)(10);
printf( "Returned Value: %d\n" , num ) ;
```

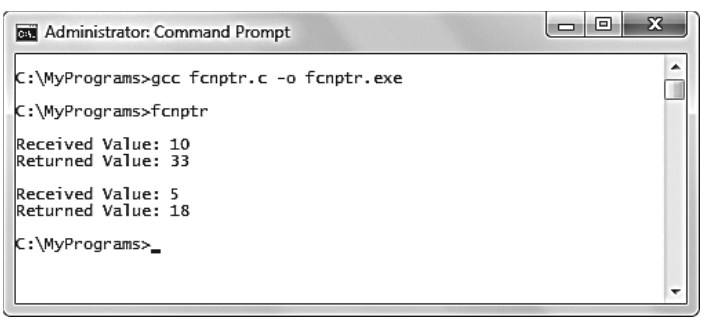
7. Теперь присвойте новое значение целочисленной переменной, передав указатель на функцию и целое число другой функции, которая, в свою очередь, вызовет обычную функцию, а затем выведет значение, которое та вернет.

```
num = caller(fptr, 5);
printf( "Returned Value: %d\n", num ) ;
```

8. В конце блока функции `main` верните значение 0, как того требует объявление функции.

```
return 0;
```

9. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значения, выведенные на экран указателями на функции.



Совет

Первым аргументом функции `caller()` в этом примере может быть указатель на любую функцию, которая получает один целочисленный аргумент и возвращает целочисленное значение, что указано в объявлении прототипа.

На заметку

Как и другие указатели в языке C, указатель на функцию просто хранит адрес в памяти. Когда указатель на функцию разыменовывается с помощью операции *, вызывается функция, расположенная по адресу, хранящемуся в указателе.

Заключение

- Указатель — это переменная, хранящая адрес в памяти другой переменной, выраженный в шестнадцатеричном формате.
- Операция разыменования, `*`, используется, чтобы получить значение, хранящееся в переменной, на которую ссылается указатель.
- При объявлении переменной указателя перед ее именем ставится символ `*`, чтобы указать, что переменная является указателем.
- Переменной-указателю можно присвоить адрес другой переменной с помощью операция адресации `&`.
- Арифметика указателей позволяет перемещать указатель между последовательными участками памяти, что особенно полезно при перемещении между элементами массива.
- Когда имя переменной — массива целых чисел присваивается указателю, этот указатель автоматически сохраняет адрес первого элемента массива.
- Указатели могут быть переданы как аргументы в другие функции.
- Передача указателя как аргумент функции означает передачу переменной по ссылке, что позволяет функции работать с оригинальным значением.
- Каждый элемент массива указателей может хранить адрес другой переменной.
- Массив указателей особенно полезно использовать для работы с символьными строками.
- Когда имя массива/символьной строки присваивается указателю, этот указатель автоматически сохраняет всю строку.
- Указатель на функцию всегда размещается в скобках при использовании операции разыменования `*`, чтобы избежать ошибок компилятора.
- Разыменование указателя на функцию приводит к вызову функции, на которую он ссылается, и происходит передача аргументов.

8 Работа со строками

Здесь показываются принципы работы с текстовыми строками.

- **Чтение строк**
- **Копирование строк**
- **Объединение строк**
- **Поиск подстрок**
- **Валидация строк**
- **Преобразование строк**
- **Заключение**

Чтение строк

В программировании на языке C строка является массивом символов, который содержит в последнем элементе специальный символ \0. Каждый символ, включая знаки пунктуации и непечатаемые символы, такие, как перевод каретки, имеют уникальное числовое значение кода ASCII. Это значит, что эти символы можно изменить арифметически. Например, если символ имеет значение `char letter = 'S'`, то операция `letter++` даст результат `'T'`.

Совет

Вы можете найти таблицу стандартных кодов ASCII в конце книги.

Значения кодов ASCII для символов нижнего регистра всегда имеют значение, превосходящее на 32 значения аналогичных символов верхнего регистра, из чего следует, что регистр любого символа может быть изменен путем добавления или вычитания числа 32. Например, если символ имеет значение `char letter = 'M'`, то операция `letter += 32` даст результат `'m'`.

Поскольку в языке C не существует специального типа данных для строки, переменные-строки должны создаваться как массивы символов, в последнем элементе которых лежит символ-терминатор \0, что повышает статус данного массива до строки. Это означает, что имя массива действует как **implied** указатель на целую строку символов. Операция `sizeof` вернет длину строки, если передать ему в качестве аргумента имя строки. Строка может быть присвоена любому массиву или указателю на тип `char`, например так:

```
char arr[6] = { 'A', 'l', 'p', 'h', 'a' } ;

char *ptr = "Beta" ;

printf("%s", arr); /*Выведет слово "Alpha"*/

printf("%s", ptr); /*Выведет слово "Beta"*/
```

На заметку

В языке программирования C символьные значения должны помещаться в одинарные кавычки, а строки — в двойные.

Пользователь может ввести строковое значение в программу, написанную на языке C, с помощью функции `scanf()`, что описано на странице 26. Эта функция хорошо работает как с отдельными символами, так и с их последовательностями, формируя из символов отдельное слово. Однако, функция `scanf()` имеет одно важное ограничение — считывание прекращается, когда функция встречает пробел. Это значит, что пользователь с помощью функции `scanf()` не может ввести предложение, поскольку строка будет обрезана после первого пробела.

Эту проблему можно решить с помощью двух альтернативных функций, также располагающихся в файле `stdio.h`. Первая из этих функций называется `gets()`. Она используется для чтения данных, введенных пользователем. Эта функция принимает все символы (включая пробелы), и присваивает строку массиву символов, указанному как аргумент. Она автоматически добавляет символ-терминатор \0 в конце строки, когда пользователь нажимает на клавишу `Enter`, чтобы гарантировать, что вве-

денные данные будут иметь статус строки. Компаньоном этой функции является функция `puts()`, которая выведет строку, переданную в качестве аргумента, и автоматически в конце добавит символ перевода каретки.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию `main`, в которой объявляется массив символов.

getsputs.c

```
int main()

{

    char str[51];

}
```

3. Далее в блоке функции `main` запросите у пользователя данные, которые впоследствии будут помещены в массив.

```
printf( "\nEnter up to 50 characters with spaces:\n" ) ;

gets(str);
```

4. Теперь в блоке функции `main` выведите строку, сохраненную в массиве.

```
printf( "fgets() read: " ) ;

puts(str);
```

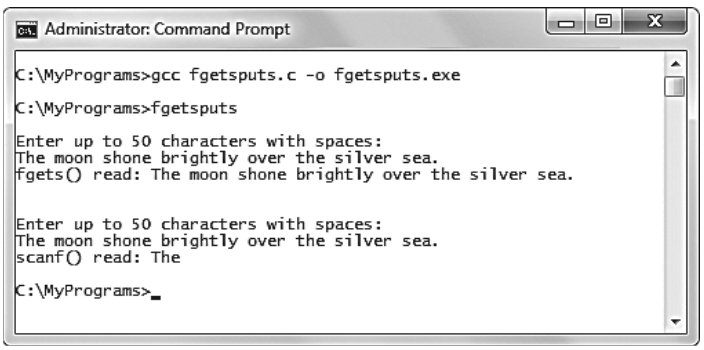
5. Повторите процесс, чтобы увидеть ограничения функции `scanf()`.

```
printf( "\nEnter up to 50 characters with spaces:\n" ) ;

scanf("%s", str);

printf( "scanf() read: %s\n" , str ) ;
```

6. В конце блока функции `main` верните значение 0, как того требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть итоговые строки.



Совет

Если вы не хотите, чтобы после вашей строки выводился символ перевода каретки, следует использовать функцию `printf()`, а не `puts()`.

Копирование строк

Стандартные библиотеки C включают в себя заголовочный файл с именем `string.h`, который содержит специальные функции работы со строками. Чтобы они были доступны, в программе необходимо добавить заголовочный файл `string.h` с помощью директивы `#include`, расположенной в начале программы.

Внимание

Убедитесь, что целевой массив имеет подходящий размер, позволяющий сохранить все копируемые символы, включая символ-терминатор `\0`.

Одна из функций файла `string.h` называется `strlen()`, она может быть использована для определения длины строки, переданной как аргумент. Функция `strlen()` возвращает число, которое является количеством символов строки, включая пробелы и исключая финальный символ-терминатор `\0`.

Заголовочный файл `string.h` предоставляет еще две полезные функции, позволяющие копировать строки из одного массива в другой. Первая из них называется `strcpy()`, в которую требуется передать два аргумента. Первый — это имя массива, в который требуется скопировать строку, а второй —имя массива, из которого будет выполняться копирование. Синтаксис этой функции выглядит так:

```
strcpy(целевой-массив, исходный-массив);
```

Все символы исходного массива будут скопированы в целевой, включая символ-терминатор `\0`. К элементам, расположенным после символа-терминатора, также будет добавлен символ `\0`. Это гарантирует, что остатки более длинной строки окажутся удалены при копировании в массив более короткой строки.

Вторая функция копирования строк имеет похожее название —`strncpy()`. Она используется точно так же, как и функция `strcpy()`, но принимает третий аргумент, позволяющий указать, сколько символов следует скопировать. Ее синтаксис выглядит так:

```
strncpy(целевой-массив, исходный-массив, длина);
```

Копируемая строка начнется с первого символа исходного массива, но закончится в позиции, указанной в третьем аргументе — финальный символ `\0` не будет скопирован автоматически. После того, как символы будут скопированы, в конец целевого массива запишется символ `\0`, что поднимет его статус до строки. Поэтому необходимо убедиться, что целевой массив всегда будет на один элемент больше, чем количество символов, которое будет в него скопировано, включая символы пробелов, чтобы разместить символ-терминатор `\0`.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода и функции для работы со строками.

strcpy.c

```
#include <stdio.h>

#include <string.h>
```

2. Добавьте функцию **main**, в которой объявляются и инициализируются строками два массива символов.

```
int main()

{

    char s1[] = "Larger text string", s2[] = "Smaller string" ;

}
```

3. Далее в блоке функции **main** выведите содержимое первого массива, его размер и длину строки.

```
printf( "\n%s: %d elements" , s1 , sizeof(s1) ) ;

printf( ", %d characters\n" , strlen(s1) ) ;
```

4. Теперь в блоке функции **main** скопируйте все содержимое второго массива в первый.

```
strcpy(s1, s2);
```

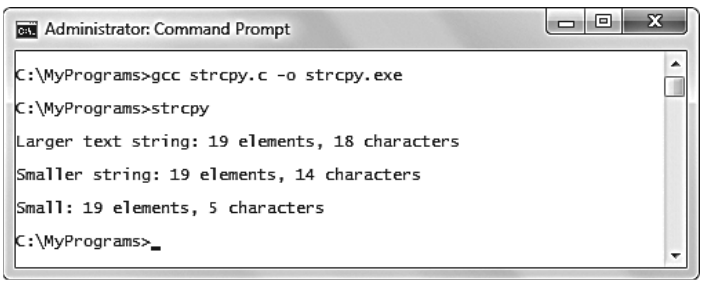
5. Повторите шаг 3, чтобы снова вывести характеристики первого массива, затем скопируйте в первый массив первые пять символов, содержащихся во втором массиве, далее поставьте символ-терминатор \0.

```
strncpy( s1, s2, 5 ) ; s1[5] = '\0' ;
```

6. Повторите шаг 3, чтобы снова вывести характеристики первого массива еще раз, затем верните конечное значение 0, как того требует объявление функции.

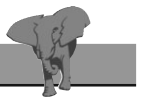
```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть скопированные строки.



Совет

Обратите внимание на то, что размеры массивов можно не указывать при их объявлении, поскольку массивы тут же инициализируются. Для других массивов, которые будут инициализированы позже, например, пользователем или при копировании из других массивов, размер необходимо указать.



На заметку

К пятому элементу итоговой строки `s1` можно обратиться с помощью конструкции `s1[5]`, поскольку нумерация индексов массива начинается с нуля, а не с единицы.

Объединение строк

Объединение двух строк в одну имеет более точное название — *конкатенация*.

Стандартный библиотечный заголовочный файл `string.h` содержит две функции, которые могут быть использованы для конкатенации строк. Чтобы эти функции были доступны, в программе необходимо добавить заголовочный файл `string.h` с помощью директивы `#include`, расположенной в начале программы.

Первая функция конкатенации строк называется `strcat()`, в качестве аргументов она требует имена двух строк, которые требуется объединить. Строка, идущая второй в списке аргументов, будет добавлена в конец строки, идущей первой, затем функция вернет сконкатенированную первую строку. Синтаксис этой функции выглядит так:

```
strcat(первая-строка, строка-которую-нужно-добавить-к-первой);
```

Необходимо помнить, что массив, хранящий первую строку, должен быть достаточно большим, чтобы разместить все символы объединенной строки, что позволит избежать ошибок.

Вторая функция конкатенации строк имеет похожее название — `strncat()`. Она используется точно так же, как и функция `strcat()`, но принимает третий аргумент, позволяющий указать, сколько символов второй строки будет добавлено к первой. Ее синтаксис выглядит так:

```
strncat(первая-строка, строка-которую-нужно-добавить-к-первой, длина);
```

На заметку

Эти функции изменяют длину исходной строки, поскольку они добавляют символы.

Присоединяемая часть строки по умолчанию начнется с первого символа второй строки и закончится в позиции, указанной в третьем аргументе. Но поскольку имя строки является указателем на первый символ, с помощью арифметики указателей можно указать другую позицию первого копируемого символа. Синтаксис будет выглядеть так:

```
strncat(первая-строка, строка-которую-нужно-добавить-к-первой + смещение, длина);
```

Опять же, как и в случае с функцией `strcat()`, важно, чтобы первый массив был достаточно большим, чтобы разместить все символы объединенной строки, что позволит избежать ошибок при использовании функции `strncat()`.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода и функции для работы со строками.

strcat.c

```
#include <stdio.h>

#include <string.h>
```

2. Добавьте функцию **main**, в которой объявляются и инициализируются строками четыре массива символов.

```
int main()

{

    char s1[100] = "A Place for Everything ";

    char s2[] = "and Everything in its Place";

    char s3[100] = "The Truth is Rarely Pure ";

    char s4[] = "and Never Simple. – Oscar Wilde";

}
```

3. Далее в блоке функции **main** присоедините вторую строку к первой и выведите результат.

```
strcat(s1, s2); printf("\n%s\n", s1);
```

4. Присоедините первые 17 символов четвертой строки к третьей и выведите результат.

```
strncat(s3, s4, 17); printf("\n%s\n", s3);
```

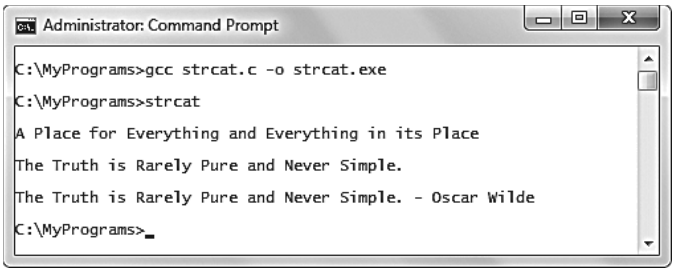
5. Присоедините последние 14 символов четвертой строки к третьей и выведите результат.

```
strncat(s3, (s4 + 17), 14); printf("\n%s\n", s3);
```

6. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть сцепленные строки.



Внимание

Удлиняемая строка должна иметь размер, достаточный для того, чтобы разместить все символы объединенной строки.

Поиск подстрок

Существует возможность выполнить поиск по строке, чтобы определить, содержит ли она определенную последовательность символов (*подстроку*). Это допустимо сделать с помощью функции `strstr()`. Она является частью стандартного заголовочного файла `string.h`, который необходимо добавить с помощью директивы `#include`, расположенной в начале программы.

Внимание

Функция `strstr()` прекращает поиск, когда встречает первое включение подстроки. Последующие включения этой подстроки не будут найдены.

Функция `strstr()` принимает два аргумента. Первый из них представляет собой строку, в которой выполняется поиск, а второй — подстроку, которую следует найти. Если подстрока не найдена, функция вернет значение `NULL`. Если подстрока найдена, функция вернет указатель на первый символ первого включения подстроки.

Номер элемента, содержащего первый символ подстроки, легко определить с помощью арифметики указателей. Вычтя адрес первого символа подстроки, возвращенного функцией `strstr()`, из адреса строки, в которой выполнялся поиск (на который указывает имя массива), можно получить целое число. Это число является индексом первого символа подстроки внутри строки, в которой выполнялся поиск.

В программировании на языке C операции сравнения `==` и `!=` могут быть использованы для того, чтобы сравнить результат со значением `NULL`, но их нельзя использовать для сравнения самих строк. В стандартном библиотечном заголовочном файле `string.h` существует функция `strcmp()`, которая позволяет выполнять сравнение строк. Она принимает два аргумента, которые являются строками, подлежащими сравнению. Сравнение выполняется на основе значений числовых кодов ASCII каждого символа и их позиции. Если строки абсолютно одинаковы, включая регистр, функция `strcmp()` вернет значение 0. В противном случае она вернет положительное или отрицательное значение в зависимости от значений строк.

`strstr.c`

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода и функции для работы со строками.

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

2. Добавьте функцию `main`, в которой объявляются и инициализируются строками два массива символов.

```
int main()
{
    char str[] = "No Time Like Present ";
    char sub[] = "Time";
}
```

3. Далее в блоке функции **main** выведите сообщение в случае, если вторая строка не будет найдена в первой.

```
if(strstr(str, sub) == NULL)
{
    printf( "Substring \"Time\" Not Found\n" ) ;
}
```

4. Теперь добавьте утверждения, позволяющие вывести адрес в памяти и индекс первого символа подстроки.

```
else
{
    printf( "Substring \"Time\" Found at %p\n" , strstr( str, sub) ) ;
    printf( "Element Index Number %d\n\n", strstr(str,sub) - str ) ;
}
```

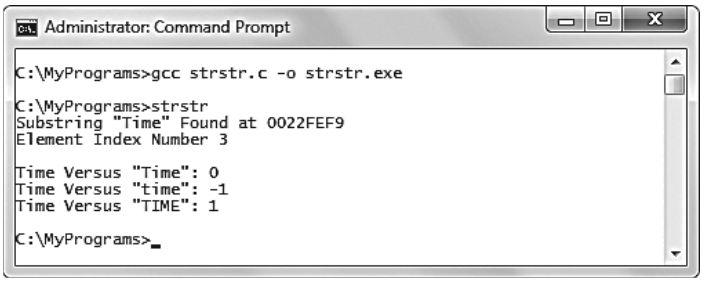
5. Выведите результат трех операций сравнения, проведенных со второй строкой.

```
printf("%s Versus \"Time\": %d\n",sub, strcmp(sub,"Time"));
printf("%s Versus \"time\": %d\n",sub, strcmp(sub,"time"));
printf("%s Versus \"TIME\": %d\n" ,sub, strcmp(sub,"TIME"));
```

6. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть результат поиска подстроки и сравнения трех строк.



На заметку

Обратите внимание на то, что символ \ в этом примере используется как экранирующий символ для двойных кавычек, что позволяет избежать преждевременного прерывания выводимых строк.

Совет

В заголовочном файле `string.h` имеются также две функции, позволяющие выполнять поиск символов. Функция `strchr()` ищет первое вхождение символа, функция `strrchr()` — последнее. Обе возвращают значение `NULL` в случае, если символ не найден.

Валидация строк

В стандартном библиотечном заголовочном файле `ctype.h` содержится несколько функций, позволяющих выполнять проверки символов. Чтобы эти функции были доступны, в программе необходимо добавить заголовочный файл `ctype.h` с помощью директивы `#include`, расположенной в начале программы.

В заголовочном файле `ctype.h` содержится функция `isalpha()`, которая проверяет, является ли символ буквой алфавита, а также функция `isdigit()`, которая проверяет, является ли символ цифрой. Аналогично, функция `ispunct()` проверяет, является ли символ каким-либо другим печатаемым символом, а функция `isspace()` —является ли символ пробелом.

В дополнение, в заголовочном файле `ctype.h` содержатся функции `isupper()` и `islower()`, которые проверяют регистр символа, наряду с функциями `toupper()` и `tolower()`, позволяющими изменять регистр символов.

Каждая проверочная функция возвращает ненулевое значение (необязательно 1), когда символ успешно проходит проверку, но всегда возвращает 0, если символ проверку не прошел.

Эти проверочные функции могут быть использованы для валидации строк, введенных пользователем, путем прохода по их символам в цикле и проверкой каждого из них. Если один из символов не подходит требуемым условиям, устанавливается значение переменной-флага, на основе которой пользователю рекомендуется исправить введенные данные.

`isval.c`

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода и функции для работы со строками

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

2. Добавьте функцию `main`, в которой объявляются три переменные, а также переменная-флаг инициализируется числовым аналогом значения `true` (1)

```
int main()
{
    char str[7];
    int i;
    int flag = 1;
}
```

3. Далее в блоке функции `main` запросите у пользователя ввести строку, затем присвойте ее переменной-массиву

```
puts( "Enter six digits without any spaces..." ) ;
gets(str);
```

4. Теперь в блоке функции **main** добавьте цикл, позволяющий проверить каждый символ массива

```
for(i = 0; i < 6; i++) {        }
```

5. Внутри блока цикла измените значение переменной-флага на **false** (0), если какой-либо символ не является цифрой

```
if(!isdigit(str[i]))
{
    flag = 0;
}
```

6. Далее в блоке **if** опишите любой символ, не являющийся цифрой

```
if(isalpha(str[i]))
{ printf( "Letter %c Found\n" , toupper(str[i]) ) ; }
else if(ispunct(str[i]))
{ printf( "Punctuation Found\n" ) ; }
else if(isspace(str[i]))
{ printf( "Space Found\n" ) ; }
```

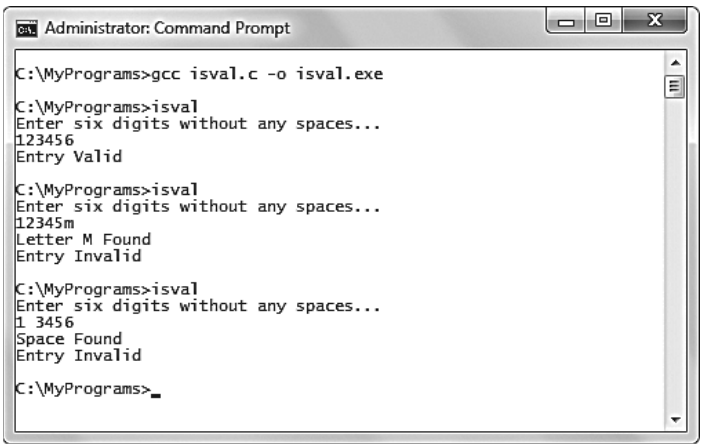
7. После блока цикла выведите сообщение, описывающее состояние переменной-флага

```
( flag ) ? puts("Entry Valid") : puts("Entry Invalid") ;
```

8. В конце блока функции **main** верните значение 0, как того требует объявление функции

```
return 0;
```

9. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть результат валидации введенных строк



На заметку

Проверка `isdigit()` может быть выражена как `isdigit() == 0`.

Совет

Полный набор функций, проверяющих символы, может быть найден в конце книги.

Преобразование строк

Стандартный библиотечный файл `stdlib.h` содержит полезную функцию, которая называется `atoi()`. Она может быть использована для преобразования строки в число. Чтобы эта функция была доступна, в программе необходимо добавить заголовочный файл `ctype.h` с помощью директивы `#include`, расположенной в начале программы.

Функция `atoi()` (расшифровывается как alpha-to-integer, буква-к-цифре) принимает в качестве своего единственного аргумента строку, которую нужно преобразовать. Если строка пустая или ее первый символ не является числом или знаком «минус», функция `atoi()` вернет значение 0. В противном случае строка (или хотя бы цифры, которые стоят в ее начале) будет преобразовываться к числу, пока функция `atoi()` не встретит в строке не являющийся числом символ. Если функция `atoi()` встретит символ, не являющийся числом, она вернет уже преобразованные в число цифры.

Также существует функция `itoa()` (расшифровывается как integer-to-alpha, цифра-к-букве), которая используется для преобразования значения, имеющего тип `int`, к строке. Эта функция широко используется, однако она не является частью стандартной спецификации ANSI C.

Функция `itoa()` принимает три аргумента. Первый из них — преобразовываемое число, второй — строка, которой будет присвоен сконвертированный результат, третий — основание, которое будет использовано при преобразовании. Например, если указать основание 2, строке присвоится бинарный эквивалент указанного числа.

В стандарте ANSI C существует аналог функции `itoa()`, который называется `sprintf()` и располагается в заголовочном файле `stdlib.h`. Эта функция менее могущественна, поскольку нельзя указать основание для преобразования. Функция `sprintf()` также принимает три аргумента — строка, которой будет присвоено число, спецификатор формата и число, которое необходимо преобразовать. Эта функция возвращает число, которое является количеством символов в преобразованной строке.

`conv.c`

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода и функции преобразования.

```
#include <stdio.h>

#include <stdlib.h>
```

2. Добавьте функцию `main`, в которой объявляются три целочисленные переменные, а также объявляются и инициализируются три символьных массива.

```
int main()
{
    int n1, n2, n3;
    char s1[10]= "12eight", s2[10]= "-65.8", s3[10]= "x13" ;
}
```

3. Далее в блоке функции **main** попробуйте преобразовать каждую строку к целому числу и выведите результат.

```
n1 = atoi( s1 ) ;
printf( "\nString %s converts to Integer: %d\n" , s1 , n1 ) ;

n2 = atoi( s2 ) ;
printf( "String %s converts to Integer: %d\n" , s2 , n2 ) ;

n3 = atoi( s3 ) ;
printf( "String %s converts to Integer: %d\n\n" , s3 , n3 ) ;
```

4. Далее в блоке функции **main** преобразуйте первую числовую переменную к строке, переведя ее в двоичную систему счисления с использованием нестандартной функции, а затем выведите результат.

```
itoa(n1, s1, 2);

printf( "Decimal %d is Binary: %s\n" , n1 , s1 ) ;
```

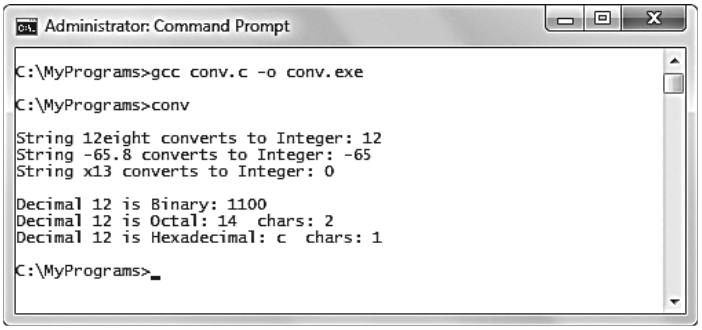
5. Далее преобразуйте первую числовую переменную к строке, переведя ее в восьмеричную и шестнадцатеричную систему счисления с использованием стандартной библиотечной функции, и сохраните длину каждой строки.

```
n2 = sprintf( s3, "%o", n1 ) ;
printf( "Decimal %d is Octal: %s chars: %d\n", n1, s3, n2) ;

n3 = sprintf( s3, "%x", n1 ) ;
printf("Decimal %d is Hexadecimal: %s chars: %d\n",n1,s3,n3);
```

6. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть результат преобразований.



Внимание

Функция `itoa()` очень полезна, но, поскольку она не является частью стандарта ANSI, ее поддерживают не все компиляторы. Однако компилятор GNU C, используемый на протяжении этой книги, эту функцию поддерживает.

На заметку

В отличие от функции `itoa()` функция `sprintf()` не может преобразовывать числа в двоичную систему счисления, поскольку не существует спецификатора формата для двоичных чисел.

Заключение

- В программировании на языке C строка является массивом символов, в конце которого размещается специальный символ-терминатор `\0`.
- Каждый символ имеет свой числовой код ASCII.
- Имя массива символов действует как указатель на всю строку.
- Функция `scanf()` прекращает считывать пользовательский ввод после того, как встретит первый пробел, но функция `gets()` может работать с пробелами и добавляет символ-терминатор `\0` автоматически.
- Функция `puts()` выводит на экран строку, переданную ее в качестве аргумента, и автоматически добавляет символ перевода каретки.
- В стандартном заголовочном библиотечном файле `string.h` содержатся специальные функции работы со строками, такие как `strlen()`, которая возвращает длину заданной строки, или `strcpy()` и `strncpy()`, которые копируют строки.
- В заголовочном файле `string.h` также содержатся функции `strcat()` и `strncat()`, которые могут использоваться для конкатенации строк.
- В дополнение, в файле `string.h` имеется функция `strstr()`, которая ищет в строке заданную подстроку, и функция `strcmp()`, которая выполняет сравнение двух заданных строк.
- Если функция `strstr()` не может найти заданную подстроку, она возвращает значение `NULL`.
- В стандартном библиотечном файле `ctype.h` содержатся функции, позволяющие проверять типы символов, например `isaplha()`, `isdigit()` и `ispunc()`.
- В заголовочном файле `ctype.h` также имеются функции `islower()`, `isupper()` и `tolower()`, `toupper()`, которые проверяют и изменяют регистр символов.
- В стандартном библиотечном заголовочном файле `stdlib.h` существует функция `atoi()`, которая преобразует строку к числу.
- В стандартном библиотечном заголовочном файле `stdlib.h` существует функция `sprintf()`, которая преобразует число к строке. Более могущественная нестандартная функция `itoa()` делает примерно то же самое.

9 Создание структур

Здесь вы познакомитесь со структурами и абстрактными типами данных `struct` *и* `union`*, а также увидите, как их можно использовать в программах, написанных на языке C, для того, чтобы сгруппировать несколько переменных разных типов.*

- **Группирование данных в структуру**
- **Определение типа данных с помощью структуры**
- **Использование указателей в структурах**
- **Указатели на структуры**
- **Передача структур в функции**
- **Группирование данных в объединение**
- **Выделение памяти**
- **Заключение**

Группирование данных в структуру

Структура в программе, написанной на языке C, способна содержать одну или несколько переменных, чей тип данных может как совпадать, так и отличаться. Они группируются в одну структуру, к ним разрешается обратиться с помощью ее имени. Переменные, расположенные внутри структур, называются ее *членами*.

Группирование связанных друг с другом переменных в структуру помогает организовать сложные данные, особенно в крупных программах. Например, структуру можно создать для описания записи платежной ведомости, ее переменные будут хранить имя сотрудника, его адрес, зарплату, налог и т. д.

В языке программирования C структура объявляется с помощью ключевого слова `struct`, после которого следует имя структуры и фигурные скобки, содержащие переменные-члены структуры. Наконец объявление структуры должно завершаться точкой с запятой после закрывающей скобки. Структура, содержащая члены, описывающие координаты *x* и *y* точки, расположенной на графе, будет выглядеть так:

```
struct coords:
{
    int x;
    int y;
};
```

Опционально разрешается добавить теги перед последней точкой с запятой в формате списка, разделенного запятыми. Объявление структуры определяет новый тип данных, а эти имена ведут себя как переменные этого типа данных. Например, переменная с именем `point` типа данных `coords` может быть объявлена следующим образом:

```
struct coords
{
    int x;
    int y;
} point;
```

К каждому члену структуры можно обратиться, добавив операцию . и имя члена и теги, например, `point.x`.

Также новая переменная, имеющая тип данных структуры, может быть объявлена с помощью имени существующей структуры. Эта перемен-

Внимание

Переменные-члены структуры не могут быть инициализированы внутри объявления структуры.

Совет

Обычно в программах, написанных на языке C, объявление структур располагается перед функцией `main()`.

ная унаследует оригинальные члены структуры. Например, чтобы создать новую переменную для структуры **coords** с именем **top**, следует использовать следующий код:

```
struct coords top;
```

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

struct.c

2. Объявите структуру, имеющую два члена и один тег.

   ```
   struct coords
   {
       int x;
       int y;
   } point;
   ```

3. Далее создайте еще один экземпляр структуры.

   ```
   struct coords top;
   ```

4. Добавьте функцию **main**, в которой инициализируются оба члена каждой структуры.

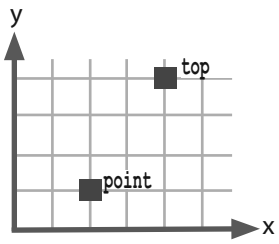
   ```
   int main()
   {
       point.x = 5; point.y = 8;
       top.x = 15; top.y = 24;
   }
   ```

5. Теперь выведите значения, хранящиеся в каждом члене структуры.

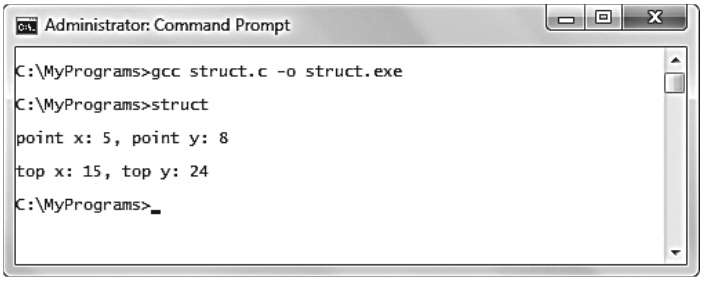
   ```
   printf( "\npoint x: %d, point y: %d\n", point.x , point.y ) ;
   printf( "\ntop x: %d, top y: %d\n" , top.x , top.y ) ;
   ```

6. В конце блока функции **main** верните значение 0, как того требует объявление функции

   ```
   return 0;
   ```



7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значения, хранящиеся в членах структуры.



Определение типа данных с помощью структуры

Тип данных, определяемый структурой, может быть объявлен так же, как и обычное определение типа данных с помощью ключевого слова **typedef**, размещенного в самом начале объявления структуры. Это определяет ее как прототип, с помощью которого можно объявить другие структуры без ключевого слова **struct** – понадобится только тег.

Использование ключевого слова **typedef** часто может помочь упростить код, поскольку не придется использовать ключевое слово **struct** для объявления переменных, имеющих тип данных структуры. Однако стоит начинать тег с большой буквы, чтобы можно было легко понять, что оно представляет собой тип данных, определяемый структурой.

В объявлениях переменных, имеющих тип данных, определяемый структурой, можно опционально инициализировать все их члены, присвоив им значения путем создания списка, разделенного запятыми, внутри фигурных скобок.

Структуры также могут быть вложены в другие структуры. В этом случае к отдельным членам можно обратиться, используя две операции .. Синтаксис будет выглядеть так: ***внешняя-структура.внутренняя-структура.член.***

typedef.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Объявите безымянную структуру, определяющую тип данных и имеющую два члена и один тег.

```
typedef struct
{
    int x;
    int y;
} Point;
```

3. Далее создайте две переменные определенного структурой типа данных, в одной из которых будут проинициализированы оба члена структуры.

```
Point top = {15, 24};
Point btm;
```

4. Далее добавьте функцию **main**, в которой инициализируются оба члена второй переменной.

```
int main()
{
```

```
    btm.x = 5;
    btm.y = 8;
}
```

5. Теперь выведите координаты обеих точек.

```
printf( "\nTop x: %d, y: %d\n" , top.x , top.y ) ;
printf( "Bottom x: %d, y: %d\n" , btm.x , btm.y ) ;
```

6. Перед блоком функции `main` добавьте вторую структуру, определяющую тип данных и имеющую в качестве членов два объекта структуры.

```
typedef struct
{
    Point a;
    Point b;
} Box;
```

7. Далее объявите переменную второго типа, определенного структурой, и инициализируйте все его переменные.

```
Box rect ={6, 12, 30, 20};
```

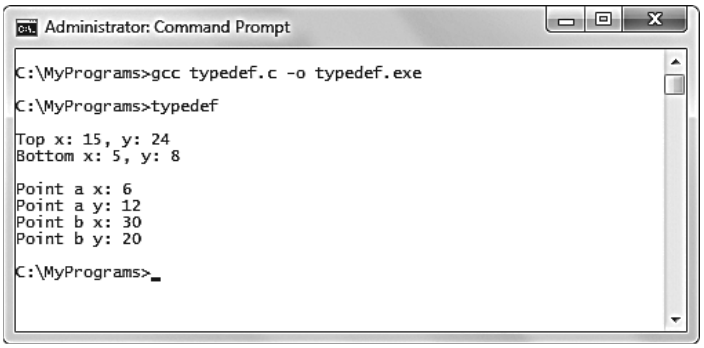
8. Вернитесь в блок функции `main` и выведите координаты всех точек, содержащихся во вложенных членах структуры.

```
printf( "\nPoint a x: %d" , rect.a.x ) ;
printf( "\nPoint a y: %d" , rect.a.y ) ;
printf( "\nPoint b x: %d" , rect.b.x ) ;
printf( "\nPoint b y: %d\n" , rect.b.y ) ;
```

9. В конце блока функции `main` верните значение 0, как того требует объявление функции.

```
return 0;
```

10. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значения, хранящиеся в членах структуры.



Совет

Обратите внимание на то, что структура может быть безымянной, но она должна иметь тег.

Внимание

Члены структур могут быть инициализированы с помощью списка, разделенного запятыми, только при объявлении переменных — далее их можно инициализировать только индивидуально.

Использование указателей в структурах

Использовать указатель на символ, находящийся в структуре, в качестве контейнера для строки порой более выгодно, чем использовать для этих же целей массив символов.

Совет

L-значения являются объектами, а R-значения — данными.

Целая строка может быть присвоена массиву символов только при его объявлении. Далее в программе массиву символов разрешается присвоить строки только последовательно, символ за символом, с помощью операции **=**.

При каждом присваивании значение, расположенное слева от операции **=**, называется L-значением, оно представляет фрагмент памяти. Значение, расположенное справа от операции **=**, называется R-значением, оно представляет собой данные, которые следует поместить в этот фрагмент.

В программировании на языке C существует важное правило, согласно которому R-значение не может располагаться слева от операции **=**. С другой стороны, L-значение способно располагаться с любой стороны операции **=**.

Каждый отдельный элемент массива символов является L-значение, ему может быть присвоен отдельный символ. Указатель на символ также является L-значение, которому после объявления может быть присвоена целая строка.

strmbr.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Далее объявите безымянную структуру, определяющую тип данных, которая имеет один член — массив символов — и один тег.

```
typedef struct
{
    char str[5];
} ArrType;
```

3. Теперь объявите безымянную строку, определяющую тип данных, которая имеет один член — указатель на символ — и один тег.

```
typedef struct
{
    char *str;
} PrtType;
```

4. Далее объявите одну переменную каждого из объявленных типов данных и инициализируйте их члены.

```
ArrType arr = {'B', 'a', 'd', ' ', '\0'};
PtrType ptr = {"Good"};
```

5. Далее добавьте функцию **main**, в которой выведите значение массива символов, являющегося членом структуры.

```
int main()
{
    printf( "\nArray string is a %s" , arr.str ) ;
}
```

6. Внутри блока функции **main** скрупулезно присвойте новое значение каждому элементу массива символов и выведите новую строку.

```
arr.str[0] = 'I';
arr.str[1] = 'd';
arr.str[2] = 'e';
arr.str[3] = 'a';
arr.str[4] = '\0';
printf("%s\n", arr.str);
```

7. Далее выведите строку, к которой обращается указатель-член структуры.

```
printf( "\nPointer string is a %s" , ptr.str ) ;
```

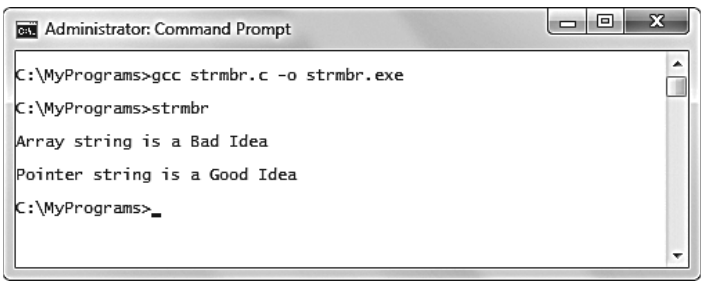
8. Теперь присвойте новое значение символьному указателю, что не вызовет у вас больших трудностей, и выведите на экран новую строку.

```
ptr.str = "Idea";
printf("%s\n", ptr.str);
```

9. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

10. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значения, хранящиеся в структурах.



Совет

Обратите внимание на то, что члены разных структур могут иметь одинаковые имена, и это не вызовет конфликтов.

На заметку

Элементы массива символов могут быть инициализированы во время объявления массива путем присвоения ему списка, разделенного запятыми.

Указатели на структуры

Указатели на типы данных, объявленные с помощью структур, могут быть созданы точно так же, как и указатели на обычные типы данных. В этом случае указатель хранит адрес начала фрагмента памяти, используемого для хранения данных членов структуры.

Использование указателей на структуры настолько типично в программах, написанных на языке C, что для этого была введена специальная операция. При работе с указателями на структуру операция . может быть заменен операцией **->**, представляющим собой дефис, за которым следует знак «больше». Эта комбинация называется «стрелкой». Например, указатель на член структуры **ptr->member** является эквивалентом конструкции **(*ptr).member**.

structptr.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Далее объявите безымянную структуру, определяющую тип данных, которая имеет два члена — оба являются символьными указателями — и один тег.

```
typedef struct

{

    char *name; char *popn;

} City;
```

Совет

Операция «стрелка» может быть полезна при разграничении указателей на структуру и прочих.

3. Теперь добавьте функцию **main**, в которой объявляются три обычные переменные и переменную-указатель. Все они имеют тип данных, определенный структурой.

```
int main()

{

    City ny, la, ch, *ptr;

}
```

4. Внутри блока функции **main** скрупулезно присвойте значения всем членам первой переменной, затем отобразите хранящиеся там значения с помощью операции ., чтобы сначала сохранить, а затем получить значения.

```
ny.name = "New York City" ;

ny.popn = "8,274,527" ;

printf( "\n%s , Population: %s\n", ny.name , ny.popn ) ;
```

5. Присвойте адрес второй переменной указателю.

```
ptr = &la;
```

6. Теперь присвойте значения членам второй переменной, а затем отобразите сохраненные значения с помощью операции «стрелка», позволяющей сохранить значения, и операции ., позволяющей получить эти значения.

```
ptr->name = "Los Angeles" ;

ptr->popn = "3,834,340" ;

printf( "\n%s, Population: %s\n" , la.name, la.popn ) ;
```

7. Далее присвойте адрес третьей переменной переменной-указателю.

```
ptr = &ch;
```

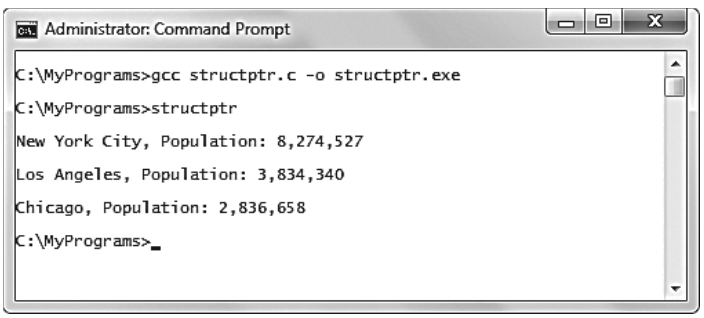
8. Теперь присвойте значения членам третьей переменной, а затем отобразите сохраненные значения с помощью операции «стрелка», позволяющей сохранить и получить эти значения.

```
ptr->name = "Chicago" ;

ptr->popn = "2,836,658" ;

printf( "\n%s, Population: %s\n", ptr->name, ptr->popn ) ;
```

9. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

10. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значения, хранящиеся в структурах.



На заметку

Значения, представляющие собой население городов, присваиваются как строки, что позволяет использовать в качестве разделителя разрядов запятые.

Передача структур в функции

Члены структур могут быть сохранены в массиве, как и значения любого другого типа данных. Массив объявляется как обычно, но метод присваивания значений его элементам несколько отличается. Каждый разделенный запятыми список значений-членов должен быть помещен в фигурные скобки.

Внимание

Простое присваивание всех значений с помощью разделенного запятыми списка не сработает — каждый набор значений должен быть помещен в свою пару скобок.

Аналогично, структура может быть передана в функцию в качестве аргумента, как и любая другая переменная. Тип данных структуры должен быть указан как в прототипе функции, так и в ее описании, наряду с именем переменной, с помощью которого можно адресовать значения структуры.

Но помните, что если вы передадите данные по значению с помощью обычной переменной, функция будет работать с копией структуры, а ее оригинальные значения останутся неизменными. С другой стороны, передача структуры по ссылке с помощью переменной-указателя означает, что функция будет работать с исходными членами структуры, что изменит их по завершении ее работы.

passstruct.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Объявите безымянную структуру, определяющую тип данных и имеющую два члена и один тег.

```
typedef struct
{
    char *name;
    int quantity;
} Item;
```

3. Теперь объявите массив переменных, имеющих тип, определенный структурой, и инициализируйте члены каждой из трех структур.

```
Item fruits[3] = { { "Apple" , 10 }, { "Orange" , 20 }, { "Pear" , 30 } } ;
```

4. Добавьте прототип функции, в которую в качестве аргументов будут передаваться переменная, имеющая тип структуры, и указатель на тип, определяемый структурой.

```
void display(Item val, Item *ref);
```

5. Теперь определите функцию, объявленную с помощью прототипа, в которой будут выводиться значения переданных аргументов.

```
void display(Item val, Item *ref)

{ printf("%s: %d\n", val.name, val.quantity); }
```

6. Далее в блоке функции измените значения членов переданной структуры-копии, а затем выведите новые значения.

```
val.name = "Banana"; val.quantity = 40;

{ printf("%s: %d\n", val.name, val.quantity); }
```

7. Убедитесь, что оригинальные значения не изменились.

```
printf("%s: %d\n", fruits[0].name, fruits[0].quantity);
```

8. Далее измените члены оригинальной структуры и выведите новые значения.

```
ref->name = "Peach";

ref->quantity = 50;

printf("%s: %d\n", fruits[0].name, fruits[0].quantity);
```

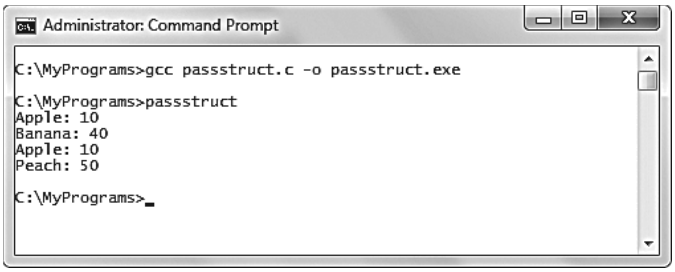
9. Прямо перед блоком функции добавьте функцию **main**, где вызывается другая функция, в которую в качестве аргументов передаются структура и указатель.

```
int main()

{

    display(fruits[0], &fruits[0]);

}
```

10. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

11. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значения, хранящиеся в членах структуры.



Совет

Передача крупных структур по значению неэффективна, поскольку в памяти создается копия целой структуры, а при копировании по ссылке требуется выделить объем памяти, равный размеру одного указателя.

Группирование данных в объединение

В программировании на языке C *объединение* позволяет хранить различные фрагменты любого типа данных в одном участке памяти в течение работы программы — присвоение значения объединению перезапишет данные, которые хранились там ранее. Это позволяет использовать память более эффективно.

Объединения похожи на структуры, но первые объявляют с помощью ключевого слова `union` и из-за их природы во время выполнения программы их члены могут получить значения только индивидуально.

Массив объединений можно создать точно так же, как и массив структур в предыдущем примере, но члены объединений могут быть инициализированы только при объявлении, если все они имеют одинаковый тип данных. Указатель на объединение создается точно так же, как и указатель на структуру. Также объединение можно передать в функцию, как и любую другую переменную.

`union.c`

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Объявите безымянную структуру, определяющую тип данных и имеющую три члена и один тег.

```
typedef struct

{ int num; char ltr, *str;} Distinct;
```

3. Теперь объявите безымянное объединение, определяющее тип данных и имеющее три члена и один тег.

```
typedef union { int num; char ltr, *str;} Unified;
```

4. Добавьте функцию `main`, в которой объявляется переменная типа данных, определенного структурой, и инициализируются все ее члены.

```
int main()

{

    Distinct sdata = {10, 'C', "Program"};

}
```

5. Теперь внутри блока функции `main` объявите переменную типа данных, определяемого объединением.

```
Unified udata;
```

На заметку

Именно программист несет ответственность за понимание того, какие типы данных хранятся в объединении в любой момент выполнения программы.

6. После объявления переменных выведите значение и адрес в памяти каждого члена структуры.

```
printf( "\nStructure:\nNumber: %d", sdata.num ) ;

printf( "\tStored at: %p\n" , &sdata.num ) ;

printf( "Letter: %c" , sdata.ltr ) ;

printf( "\tStored at: %p\n" , &sdata.ltr ) ;

printf( "String: %s" , sdata.str ) ;

printf( "\tStored at: %p\n" , &sdata.str ) ;
```

7. Теперь присвойте значение первому члену объединения, а затем выведите его значение и адрес в памяти.

```
udata.num = 16;

printf( "\nUnion:\nNumber: %d" , udata.num ) ;

printf( "\tStored at: %p\n" , &udata.num ) ;
```

8. Далее присвойте значение второму члену объединения, а затем выведите его значение и адрес в памяти.

```
udata.ltr = 'A';

printf( "Letter: %c" , udata.ltr ) ;

printf( "\tStored at: %p\n" , &udata.ltr ) ;
```

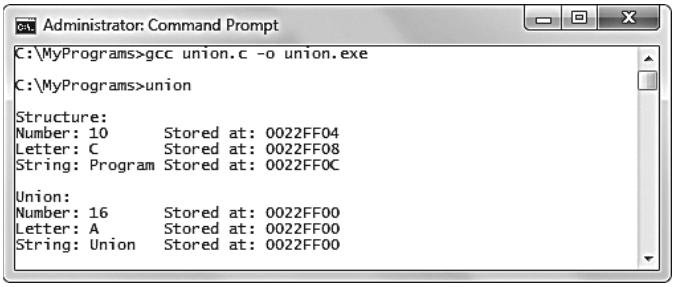
9. Наконец присвойте значение третьему члену объединения, а затем выведите его значение и адрес в памяти.

```
udata.str = "Union" ;

printf( "String: %s", udata.str ) ;

printf( "\tStored at: %p\n" , &udata.str ) ;
```

10. В конце блока функции `main` верните значение 0, как того требует объявление функции.

```
return 0;
```

11. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть значения, хранящиеся в членах структуры и объединения, а также их адреса в памяти.



Совет

Объединения наиболее полезны, если вам приходится работать в условиях ограниченной памяти.

Выделение памяти

Совет

Функция `calloc()` заполняет все выделенное пространство нулями, а функция `malloc()` сохраняет значения, лежащие в памяти до вызова этой функции.

Стандартная библиотека заголовочных файлов `stdlib.h` предоставляет функции управления памятью, с помощью которых программа может явно запросить память во время выполнения. Функция `malloc()` запрашивает один аргумент, позволяющий определить, сколько байт памяти требуется выделить. Функция `calloc()` требует наличия двух аргументов, которые будут перемножены между собой с целью определения объема памяти, который следует выделить. Обе эти функции в случае успеха возвращают указатель на начало блока памяти. В случае неудачи они возвращают значение `NULL`.

Объем памяти, выделенный с помощью функций `malloc()` и `calloc()`, может быть увеличен с помощью функции `realloc()`. Эта функция требует указатель на выделенный блок памяти в качестве первого аргумента и число, обозначающее размер нового блока, в качестве второго. Она возвращает указатель на начало увеличенного блока в случае успеха и значение `NULL` в случае неудачи.

`memory.c`

1. Начните новую программу с инструкций препроцессора, позволяющих включить функции стандартной библиотеки ввода/вывода и функции управления памятью.

```
#include <stdio.h>

#include <stdlib.h>
```

2. Добавьте функцию `main`, в которой объявляются целочисленная переменная и численный указатель.

```
int main()

{

    int size, *mem;

}
```

3. Запросите память в объеме, необходимом для размещения сотни целых чисел.

```
mem = malloc(100 * sizeof(int));
```

4. Далее выведите информацию о выделенном блоке памяти или сообщение, гласящее, что запрос не сработал.

```
if (mem != NULL)
{
    size = _msize(mem);
    printf( "\nSize of block for 100 ints: %d bytes\n", size ) ;
    printf( "Beginning at %p\n" , mem ) ;
}
else { printf( "!!! Insufficient memory\n") ; return 1 ; }
```

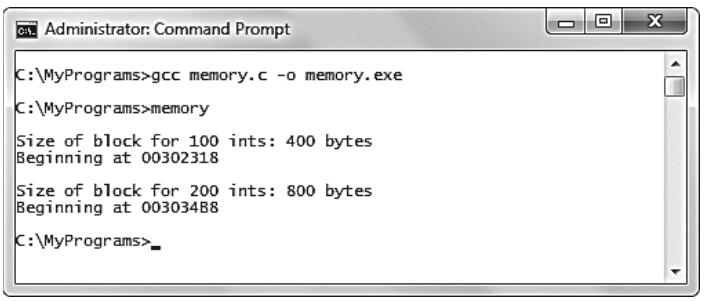
5. Теперь попытайтесь увеличить объем выделенного блока памяти и вывести информацию о нем или же сообщение, гласящее, что запрос не сработал.

```
mem = realloc(mem, size + (100 * sizeof(int));
if (mem != NULL)
{
    size = _msize(mem);
    printf( "\nSize of block for 200 ints: %d bytes\n" , size ) ;
    printf( "Beginning at %p\n" , mem ) ;
}
else { printf( "!!! Insufficient memory\n") ; return 1 ; }
```

6. В конце блока функции `main` не забудьте освободить выделенную память, а затем верните значение 0, как того требует объявление функции.

```
free( mem ) ;
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть выделенные блоки памяти в том случае, если запросы сработали успешно.



Совет

Несмотря на то, что тип `int`, как правило, занимает 4 байта, хорошим тоном является использование операции `sizeof()`, чтобы определить точный размер блока памяти на случай, если переменные типа `int` имеют другой размер.

На заметку

Утверждение, запрашивающее память, может выглядеть и так: `calloc(100, sizeof(int))`.

Заключение

- Структура способна содержать одну или несколько переменных любого типа данных, которые называются членами структуры.
- К каждому члену структуры можно обратиться, прибавив его имя к тегу с помощью операции ..
- Последующие экземпляры структуры унаследуют свойства-члены оригинальной структуры, от которой они образованы.
- Перед ключевым словом `struct` может располагаться ключевое слово `typedef`, указывающее, что структура является типом данных.
- Хорошим приемом программирования является написание тегов с большой буквы, что позволит проще разграничивать определенные структурами типы данных.
- При объявлении переменной, имеющей тип, определяемый структурой, можно инициализировать каждый член структуры, присвоив переменной разделенный запятыми список значений.
- Указатель на структуру хранит адрес в памяти начала фрагмента памяти, задействованного для хранения данных членов структуры.
- При работе с указателем на структуру операция . может быть заменена операцией `->`.
- Члены массива структур могут быть опционально инициализированы при объявлении, но каждый список значений должен размещаться в фигурных скобках.
- Структура может быть передана в функцию в качестве аргумента, как и любая другая переменная.
- Передача структуры в функцию через переменную означает, что функция будет работать с копией членов структуры. Передача структуры в функцию с помощью указателя означает, что последняя будет работать с оригинальными значениями.
- Объединение очень похоже на структуру — в нем так же хранятся разные данные, которые лежат в одном фрагменте памяти во время работы программы.
- Стандартный библиотечный заголовочный файл `stdlib.h` предоставляет заголовочные функции `malloc()`, `calloc()` и `realloc()`, которые позволяют выделить память, и функцию `free()`, предназначенную для освобождения выделенной памяти.

Получение результата

Здесь показывается, как в программах, написанных на языке C, можно использовать файлы, системное время, случайные числа и простые диалоговые окна Windows.

Создание файла

В заголовочном файле `stdio.h` определен специальный тип данных, предназначенный для работы с файлами. Он называется *указателем на файл* и имеет синтаксис `FILE *fp`. Указатели на файлы используются для открытия, чтения, записи и закрытия файлов. В программе, написанной на языке C, указатель на файл с именем `file_ptr` может быть создан с помощью объявления `FILE *file_ptr;` .

Совет

Все стандартные функции, предназначенные для работы с файлами, содержатся в заголовочном файле `stdio.h`.

Указатель на файл указывает на структуру, определенную в заголовочном файле `stdio.h`, которая содержит информацию о файле. Последняя включает в себя данные о текущем символе и о состоянии файла (выполняется ли его чтение или запись в данный момент).

Перед тем, как файл может быть записан или считан, он сначала должен быть открыт с помощью функции `fopen()`. Эта функция принимает два аргумента, определяющие имя и расположение файла, а также *режим*, в котором файл должен быть открыт. Функция `fopen()` возвращает указатель на файл в случае успеха и значение `NULL` в случае неудачи.

После успешного открытия файла его можно считать, добавить в него данные или же полностью переписать, действие зависит от режима, указанного при вызове функции `fopen()`. Открытый файл всегда должен быть закрыт путем вызова функции `fclose()`, которая принимает указатель на файл в качестве единственного аргумента.

В таблице, приведенной ниже, перечислены все возможные режимы открытия файлов, которые могут быть переданы в качестве второго аргумента функции `fopen()`:

На заметку

Для удобства все текстовые файлы, рассмотренные в примерах этой главы, расположены в директории *MyPrograms*, в которой также содержатся и файлы исходного кода. В операционной системе Windows это директория *C:\MyPrograms*, в операционной системе Linux — *\home\MyPrograms*.

Режим файла	Операция
`r`	Открытие существующего файла для чтения
`w`	Открытие существующего файла для записи. Создание нового файла, если файла с таким именем не существует, или же открытие уже существующего файла и затирание всех данных, которые там были ранее
`a`	Добавление текста. Открывает или создает текстовый файл для записи в конце файла
`r+`	Открывает текстовый файл для чтения или записи
`w+`	Открывает текстовый файл для записи или чтения
`a+`	Открывает или создает текстовый файл для чтения или записи в конец файла
Если режим включает в себя букву `b` после любого приведенного ранее режима, операция будет относиться к бинарному файлу, а не к текстовому. Например, `rb` или `w+b`.	

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию `main`, в которой объявляется указатель на файл.

```
int main()

{

    FILE *file_ptr;

}
```

3. Далее в блоке функции `main` попробуйте создать файл для записи.

```
file_ptr = fopen("data.txt", "w");
```

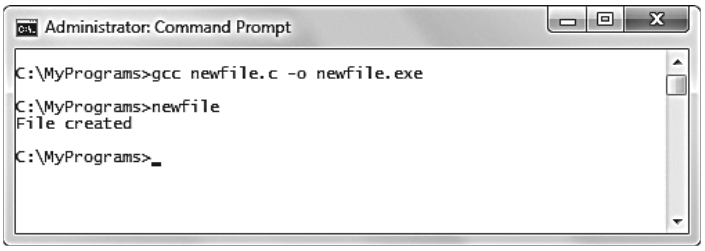
4. Теперь выведите сообщение, подтверждающее успешность попытки создания файла, а затем закройте файл и верните значение 0, как того требует объявление функции.

```
if(file_ptr != NULL)

{

    printf("File created\n");

    fclose(file_ptr);

    return 0;

}
```

5. Выведите альтернативное сообщение, если попытка была безуспешной.

```
else

{

    printf( "Unable to create file\n" ) ; return 1 ;

}
```

6. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы попытаться создать текстовый файл.



`newfile.c`

Внимание

Обратите внимание на то, что имя файла и режим его открытия должны быть заключены в двойные кавычки.

Совет

Обратите внимание на то, что эта программа возвращает значение 1 в случае неудачной попытки открытия файла — это говорит системе, что все прошло не очень гладко.

Чтение и запись символов

Стандартный ввод

Функция `scanf()`, использованная в предыдущих примерах, является упрощенной версией функции `fscanf()`, которая требует в качестве первого аргумента входной файловый поток. Он указывает источник, из которого последовательность символов попадет в программу.

В программировании на языке C файловый поток с именем `stdin` представляет собой клавиатуру и является источником данных по умолчанию для функции `scanf()`. Вызов функции `scanf(…)` аналогичен вызову функции `fscanf(stdin, …)`.

Стандартный вывод

Таким же образом функция `printf()` является упрощенной версией функции `fprintf()`, которая в качестве первого аргумента требует выходной поток. Он указывает место, в которое последовательность символов попадет из программы.

Файловый поток с именем `stdout` представляет монитор и является источником по умолчанию для функции `printf()`. Вызов функции `printf(…)` аналогичен вызову функции `fprintf(stdout, …)`.

Другие стандартные функции, имеющие в качестве потоков по умолчанию потоки `stdin` и `stdout`, также имеют эквиваленты, позволяющие указать альтернативный файловый поток. Они могут быть использованы для чтения файлов путем передачи им указателя на файл в качестве альтернативы чтению из потока `stdin`. Они также могут быть использованы для записи файлов путем указания потока, альтернативного потоку `stdout`:

- Функция `fputc()` может использоваться для записи в файловый поток одного символа за раз — обычно путем прохода в цикле по массиву символов. Ее компаньон `fgetc()` может использоваться для чтения из файлового потока по одному символу за раз.
- Функция `fputs()` может использоваться для записи в файловый поток одной строки за раз, а ее компаньон `fgets()` — для чтения одной строки за раз.
- Функция `fread()` может использоваться для чтения файлового потока полностью, а функция `fwrite()` — для записи файлового потока полностью.
- Функции `fscanf()` и `fprintf()` могут использоваться для чтения и записи в файловые потоки строк и чисел.

Совет

Еще одним стандартным файловым потоком является поток `stderr`, который используется для вывода сообщений об ошибках.

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляется указатель на файл, переменная-счетчик цикла и инициализируется массив символов.

```
int main()
{
    FILE *file_ptr; int i;
    char text[50] = { "Text, one character at a time." } ;
}
```

3. Далее в блоке функции **main** попробуйте создать файл для записи.

```
file_ptr = fopen("chars.txt", "w");
```

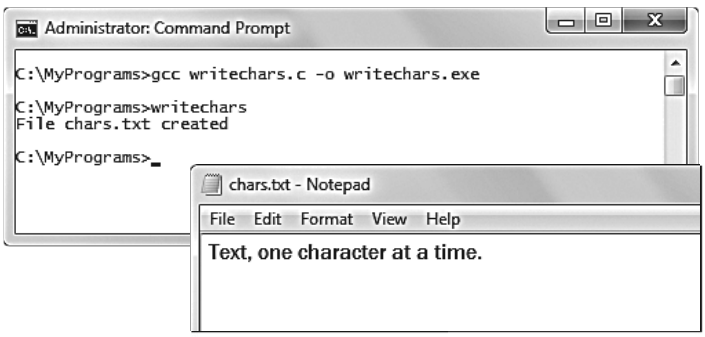
4. Теперь выведите сообщение, подтверждающее успешность попытки создания файла, запишите в файл каждый символ массива, а затем закройте файл и верните значение 0, как того требует объявление функции.

```
if(file_ptr != NULL)
{
    printf( "File chars.txt created\n" ) ;
    for(i = 0; text[i]; i++) { fputc(text[i], file_ptr); }
    fclose(file_ptr);
    return 0;
}
```

5. Выведите альтернативное сообщение, если попытка была безуспешной.

```
else
{
    printf( "Unable to create file\n" ) ; return 1 ;
}
```

6. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы попытаться записать данные в текстовый файл.



writechars.c

Совет

Функция `fputc()` возвращает ASCII-код текущего символа или константу EOF, которая говорит о том, что достигнут конец файла.

Чтение и запись строк

Использование функции `fgetc()` — не самый эффективный способ считывания текста в программе, поскольку эту функцию необходимо вызывать большое количество раз подряд, чтобы считать все символы. Лучше использовать функцию `fgets()`, которая считывает текст по одной строке за раз.

Функция `fgets()` принимает три аргумента. Первый аргумент определяет символьный указатель или массив символов, куда будет записан текст. Второй аргумент — это число, которое определяет максимальное количество символов в считываемой строке. Третий аргумент — файловый указатель, указывающий, из какого файла следует производить чтение.

Аналогично, функция `fputc()`, которая записывает текст в файл по одному символу за раз, менее эффективна, чем функция `fputs()`, которая записывает текст в файл по одной строке за раз. Функция `fputs()` принимает два аргумента, определяющих текст, который нужно записать, и файл, куда следует записать текст.

Функция `fputs()` добавляет символ новой строки всякий раз, когда записывает строку. Эта функция возвращает 0 в случае успеха или константу EOF, когда происходит ошибка или функция достигает конца файла.

`lines.c`

1. Начните новую программу с инструкций препроцессора, позволяющую включить функции стандартной библиотеки ввода/вывода и функции работы со строками.

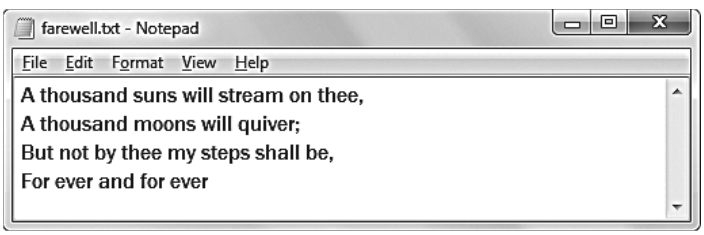
```
#include <stdio.h>
#include <string.h>
```

2. Добавьте функцию `main`, в которой объявляются указатель на файл и неинициализированный массив символов.

```
int main()
{
    FILE *file_ptr;
    char text[50];
}
```

3. Далее в блоке функции `main` попробуйте открыть текстовый файл для чтения и записи данных в его конец.

```
file_ptr = fopen("farewell.txt", "r+a");
```



4. Теперь выведите сообщение, подтверждающее успешность попытки создания файла.

```
if(file_ptr != NULL)
{
    printf( "File farewell.txt opened\n" ) ;
}
```

5. Теперь в блоке `if` считайте и выведите все строки файла.

```
while( fgets(text, 50, file_ptr) ) { printf("%s", text); }
```

6. В блоке `if` скопируйте новую строку в массив, а затем добавьте ее в текстовый файл.

```
strcpy( text , "...by Lord Alfred Tennyson" ) ;
fputs(text, file_ptr);
```

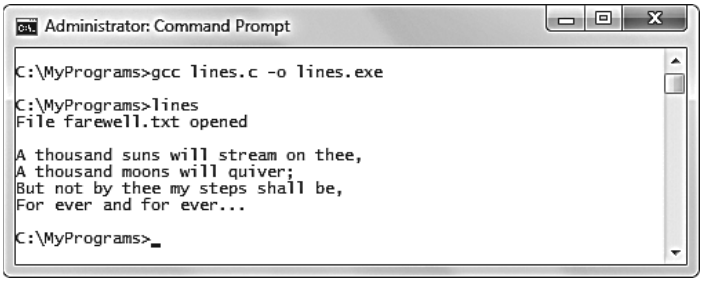
7. В конце блока `if` закройте файл и верните требуемое определением функции число 0.

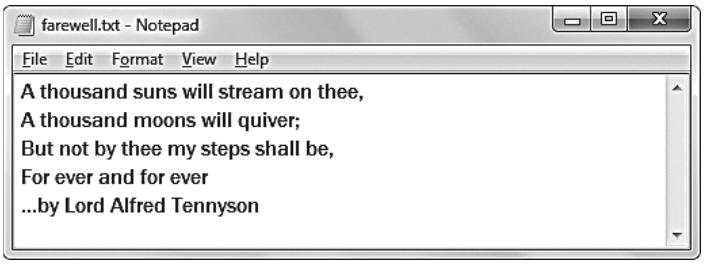
```
fclose(file_ptr);
return 0;
```

8. Добавьте альтернативное сообщение на случай, если файл открыть не удастся.

```
else { printf( "Unable to open file\n" ) ; return 1 ; }
```

9. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы попытаться открыть файл и добавить к нему текстовую строку.





Совет

Обратите внимание на то, как функция `strcpy()` используется для присвоения новой строки массиву символов.

Считывание и запись файлов целиком

Файл может быть считан целиком с помощью функции **fread()** и записан целиком с помощью функции **fwrite()**. Обе эти функции принимают четыре одинаковых аргумента. Первый из них — символьная переменная, где текст может храниться. Во втором аргументе указывается размер считываемых или записываемых за раз фрагментов текста — обычно он равен 1. Третий аргумент позволяет указать общее количество символов для чтения или записи, а четвертый аргумент является файловым указателем, указывающим на файл, с которым нужно работать.

Функция **fread()** возвращает количество считанных объектов, в котором учитываются символы, пробелы переводы каретки. Аналогично, функция **fwrite()** возвращает количество записанных объектов.

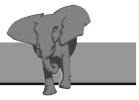
readwrite.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляются два указателя на файл, массив символов и целочисленная переменная.

```
int main()
{
    FILE *orig_ptr;
    FILE *copy_ptr;
    char buffer[1000];
    int num;
}
```

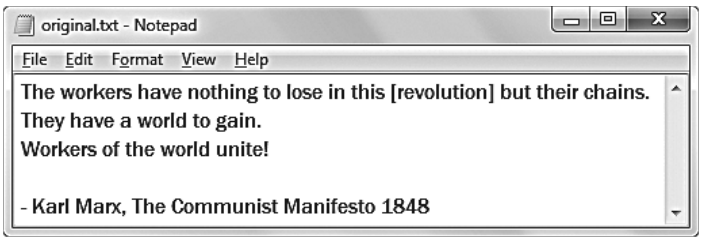


На заметку

Массив символов, в котором будет храниться текст, должен быть достаточно большим, чтобы разместить содержимое всего файла.

3. Далее в блоке функции **main** попробуйте открыть существующий файл для чтения и другой файл для записи.

```
orig_ptr = fopen("original.txt", "r");
copy_ptr = fopen("copy.txt", "w");
```



4. Теперь проверьте, что оба файла были успешно открыты.

```
if((orig_ptr != NULL) && (copy_ptr != NULL))
{
}
```

5. Внутри блока **if** считайте содержимое оригинального файла в массив символов, подсчитывая каждый считанный объект, а затем запишите содержимое массива во второй файл.

```
num = fread(buffer, 1, 1000, orig_ptr);

fwrite(buffer, 1, num, copy_ptr);
```

6. Не забудьте закрыть оба текстовых файла по завершении.

```
fclose(orig_ptr);

fclose(copy_ptr);
```

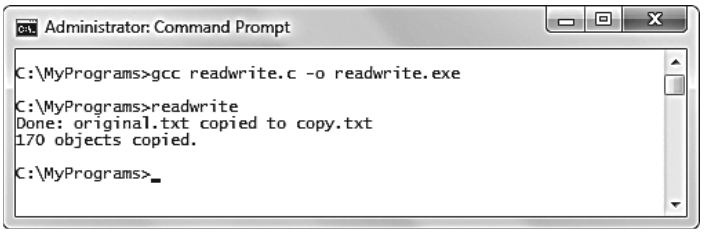
7. В заключение в блоке **if** выведите сообщение, подтверждающее успех операции, которое содержит количество объектов, и верните значение 0.

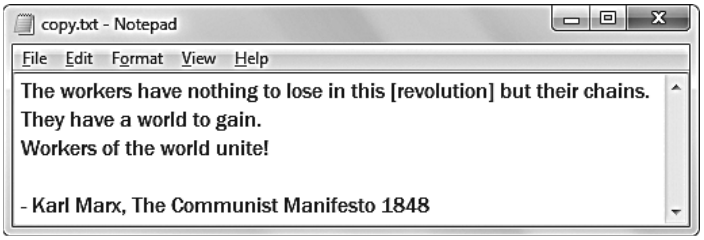
```
printf( "Done: original.txt copied to copy.txt" ) ;

printf( "\n%d objects copied.\n" , num ) ;

return 0;
```

8. Добавьте альтернативное сообщение, если попытка была безуспешной.

```
else

{

    if(orig_ptr == NULL) printf( "Unable to open original.txt\n") ;

    if(copy_ptr == NULL) printf( "Unable to open copy.txt\n" ) ;

    return 1;

}
```

9. Сохраните файл программы, а затем скомпилируйте и запустите ее, чтобы открыть файл и скопировать его содержимое в новый.





Совет

Используйте количество объектов, возвращенное функцией `fread()` как третий аргумент для функции `fwrite()`, чтобы гарантировать, что количество записанных объектов совпадет с количеством считанных объектов.

Внимание

Убедитесь, что третий аргумент функции `fread()` достаточно велик, чтобы вместить все копируемое содержимое. Если в этом примере изменить значение данного аргумента на 100, будут скопированы первые 100 объектов. При этом последние 70 объектов окажутся опущены.

Сканирование файловых потоков

Функция **scanf()**, использовавшаяся для получения данных, введенных пользователем, является упрощенной версией функции **fscanf()**, которая всегда выполняет чтение из потока **stdin**. Функция **fscanf()** позволяет вам указать файловый поток для чтения в качестве ее первого аргумента. Также она имеет преимущество при чтении файлов, содержащих только числа — числа в текстовом файле видны как простой набор символов, но если они будут считаны функцией **fscanf()**, они окажутся преобразованы к их числовому типу.

Аналогично, функция **printf()**, использованная для вывода информации на экран, является упрощенной версией функции **fprintf()**, которая всегда выполняет запись в поток **stdout**. Функция **fprintf()** позволяет вам указать файловый поток для записи в качестве ее первого аргумента.

Функции **fscanf()** и **fprintf()** предлагают высокий уровень гибкости, позволяя выбрать поток, с которым будет производиться чтение или запись.

fscanprint.c

1. Начните новую программу с инструкции препроцессора, позволяющей включить функции стандартной библиотеки ввода/вывода.

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

2. Добавьте функцию **main**, в которой объявляются два указателя на файл, массив целых чисел и две обычные целочисленные переменные.

```
int main()
{
    FILE *nums_ptr, *hint_ptr;
    int nums[20], i, j;
}
```

3. Далее в блоке функции **main** попробуйте открыть один существующий текстовый файл для чтения и второй для записи.

```
nums_ptr = fopen("nums.txt", "r");
hint_ptr = fopen("hint.txt", "w");
```

4. Теперь проверьте, что оба файла были успешно открыты.

```
if((nums_ptr != NULL) && (hint_ptr != NULL))
{
}
```

5. Внутри блока **if** считайте числа из файла в целочисленный массив.

```
for(i = 0; !feof(nums_ptr); i++)
{
    fscanf(nums_ptr, "%d", &nums[i]);
}
```

6. Далее в блоке **if** выведите значения элементов массива.

```
fprintf( stdout , "\nTotal numbers found: %d\n" , i ) ;
for (j = 0; j < i; j++) { fprintf(stdout, '%d", nums[j]); }
```

7. Теперь запишите значения элементов массива в файл.

```
fprintf( hint_ptr , "fscanf and fprintf are flexible\n" ) ;
for (j = 0; j < i; j++) { fprintf(hint_ptr, "%d", nums[j]); }
```

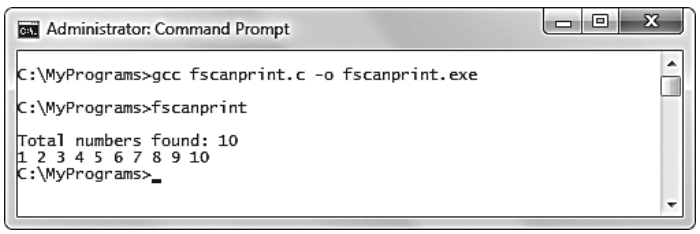
8. В заключение в блоке **if** закройте оба текстовых файла по завершении.

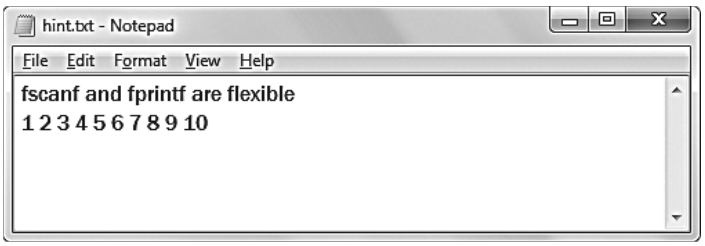
```
fclose(nums_ptr);
fclose(hint_ptr);
```

9. Добавьте альтернативное сообщение, если попытка была безуспешной.

```
else
{
    fprintf( stdout , "Unable to open a file\n" ) ; return 1 ;
}
```

10. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы открыть файл, а затем вывести на экран и записать в другой файл его содержимое.





Совет

Обратите внимание на то, что функция `feof()` используется в этом примере для того, чтобы проверить, достигнут ли конец файла, и выйти из цикла, когда это случится.

На заметку

Функции `fscanf()` и `fprintf()` принимают те же аргументы, что и функции `scanf()` и `printf()` плюс дополнительный аргумент, позволяющий указать поток.

Сообщение об ошибках

Язык программирования C предоставляет функцию с именем **perror()**, которая находится в заголовочном файле **stdio.h** и может быть использована для вывода на экран подробных сообщений об ошибках. В качестве аргумента она принимает строку, к которой добавляет двоеточие, а за ним — описание текущей ошибки.

В дополнение в заголовочном файле **errno.h** определено целочисленное выражение с именем **errno**, которому присваивается числовой код ошибки, когда случается ошибка. Этот код может быть передан в качестве аргумента функции **strerror()**, располагающейся в заголовочном файле **string.h**, которая выводит связанное с этим кодом сообщение об ошибке.

errno.c

1. Начните новую программу с инструкций препроцессора, позволяющих включить функции стандартной библиотеки ввода/вывода, функции обработки сообщений об ошибках и функции работы со строками.

```
#include <stdio.h>

#include <errno.h>

#include <string.h>
```

2. Добавьте функцию **main**, в которой объявляется указатель на файл и целочисленная переменная.

```
int main()

{

    FILE *f_ptr;

    int i;

}
```

3. Далее в блоке функции **main** попробуйте открыть несуществующий текстовый файл.

```
f_ptr = fopen("nosuch.file", "r");
```

4. Выведите подтверждение или сообщение об ошибке, если попытка провалится.

```
if( f_ptr != NULL ) { printf( "File opened\n" ) ; }

else { perror( "Error" ) ; }
```

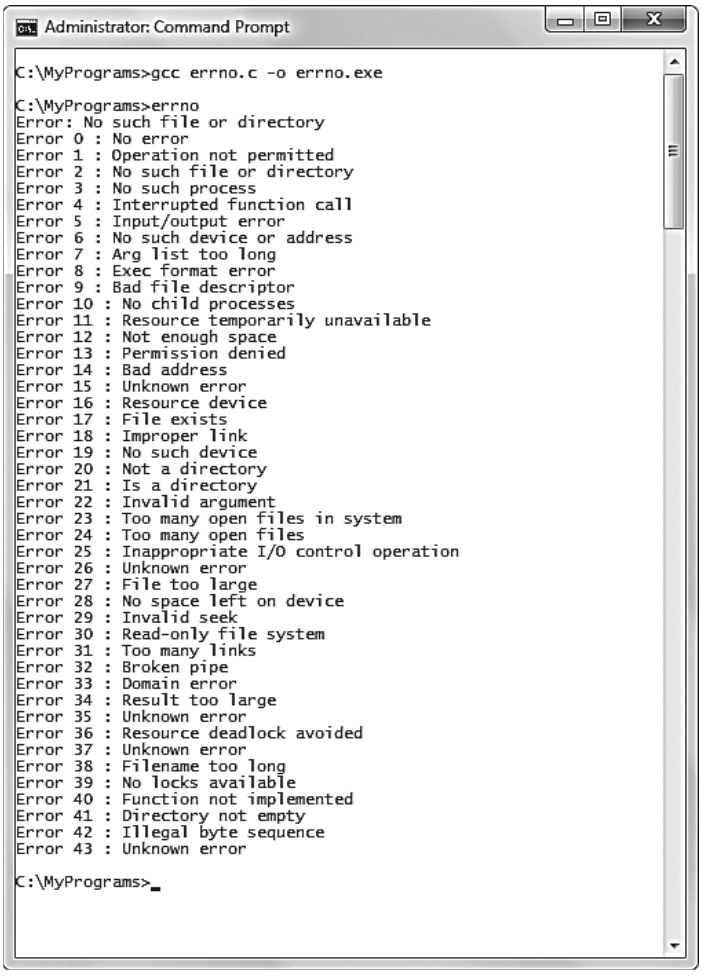
5. Теперь добавьте цикл, позволяющий вывести сообщение об ошибке, связанное с каждым цифровым кодом ошибки.

```
for(i = 0; i < 44; i++)

{

    printf( "Error %d : %s\n" , i , strerror(i) ) ;

}
```

6. В конце блока функции **main** верните значение 0, как того требует объявление функции.

```
return 0;
```

7. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть сообщения об ошибках.



Совет

Диапазон значений цифровых кодов зависит от реализации. В системах на базе Linux он больше, чем на платформе Windows.

Получение даты и времени

Совет

Компонент `tm_isdst` имеет положительное значение, если действует летнее время, 0 — если нет, отрицательное значение — если информация недоступна.

Текущее системное время обычно определяется как количество секунд, прошедшее с момента наступления эпохи Unix (00:00:00 GMT 1 января 1970 года. Оно представляет собой текущую дату и время в соответствии с Григорианским календарем и называется «Календарное время». Специальные функции, предназначенные для работы с датой и временем, располагаются в файле `time.h` наряду со структурой `tm`, в которой хранятся компоненты даты и времени, как показано ниже.

Компонент	Описание
`int tm_sec`	Количество секунд, как правило, в диапазоне 0-59
`int tm_min`	Количество минут, 0-59
`int tm_hour`	Количество часов, 0-23
`int tm_mday`	День месяца, 1-31
`int tm_mon`	Количество месяцев, прошедших с января, 0-11
`int tm_year`	Количество лет, прошедших с 1900 года
`int tm_wday`	Количество дней, прошедших с воскресенья, 0-6
`int tm_yday`	Количество дней, прошедших с 1 января, 0-365
`int tm_isdst`	Признак летнего времени

На заметку

Полный список всех спецификаторов формата включен в разделе справочной информации в конце книги.

Текущее количество прошедших секунд возвращает функция `time(NULL)` как тип данных `time_t`. Этот параметр может быть передан как аргумент в функцию `localtime()` для преобразования к формату компонентов структуры `tm`.

Компоненты структуры разрешается вывести в стандартном формате даты и времени с помощью функции `asctime()`. В качестве альтернативы отдельные компоненты можно вывести с использованием специальных спецификаторов формата времени с помощью функции `strftime()`. Она принимает четыре аргумента, позволяющих указать массив символов, в котором будет храниться отформатированная строка даты, максимальная длина строки, текст и спецификаторы формата, необходимые для извлечения требуемых компонентов для структуры `tm`.

`datetime.c`

1. Начните новую программу с инструкций препроцессора, позволяющих включить функции стандартной библиотеки ввода/вывода и функции работы со временем.

```
#include <stdio.h>

#include <time.h>
```

2. Добавьте функцию **main**, в которой объявляется массив символов, переменная типа **time_t** и структура **tm**.

```
int main()

{

    char buffer[100];

    time_t elapsed;

    struct tm *now;

}
```

3. Далее в блоке функции **main** получите текущее количество секунд, прошедшее с момента наступления эпохи Unix.

```
elapsed = time(NULL);
```

4. Теперь преобразуйте это число к формату компонентов структуры **tm**.

```
now = localtime(&elapsed);
```

5. Выведите дату и время в стандартном формате.

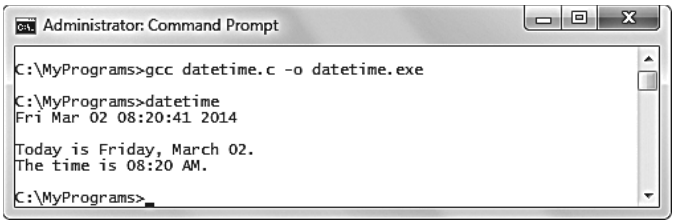
```
printf("%s\n", asctime(now));
```

6. Далее выведите отдельные компоненты даты и времени.

```
strftime ( buffer, 100, "Today is %A, %B %d.\n", now ) ;

printf( "%s", buffer ) ;

strftime ( buffer, 100, "The time is %I:%M %p.\n", now ) ;

printf( "%s", buffer ) ;
```

7. В конце блока функции **main** верните значение 0, как того требует объявление функции, а затем сохраните файл программы, скомпилируйте и запустите программу, чтобы увидеть текущую информацию о дате и времени.

```
return 0;
```



Внимание

Обратите внимание на то, что операция адресации, **&**, используется в этом преобразовании к местному времени для того, чтобы получить Календарное Время из типа данных **time_t**.

Запуск таймера

Возможность получить текущее время до и после какого-нибудь события означает, что можно определить продолжительность события, найдя их разность. В заголовочном файле `time.h` содержится функция `difftime()`, отвечающая именно за это. Эта функция принимает два аргумента, они оба имеют тип данных `time_t`. Она вычитает второй аргумент из первого и возвращает разницу, выраженную в целом количестве секундах, которая имеет формат `double`.

Еще одним способом работы с временем является функция `clock()`, которая располагается в заголовочном файле `time.h`. Она возвращает время процессора, использованное с момента начала программы, выраженное в «тиках». Эта функция может быть использована для того, чтобы приостановить выполнение программы путем запуска пустого цикла, который будет выполняться до тех пор, пока не окажется достигнут определенный момент в будущем.

write.c

1. Начните новую программу с инструкций препроцессора, позволяющих включить функции стандартной библиотеки ввода/вывода и функции работы с временем.

```
#include <stdio.h>

#include <time.h>
```

2. Далее объявите прототип функции, приостанавливающей программу.

```
void wait(int seconds);
```

3. Добавьте функцию `main`, в которой объявляются две переменные типа `time_t` и целочисленная переменная.

```
int main()

{

    time_t go, stop;

    int i;

}
```

4. Далее в блоке функции `main` получите текущее время и выведите сообщение.

```
go = time(NULL);

printf ( "\nStarting countdown...\n\n" ) ;
```

5. Добавьте цикл, позволяющий вывести значение счетчика, и вызывайте другую функцию на каждой итерации.

```
for(i = 10; i > -1; i--)
{
    printf(" - %d", i);
    wait(1);
}
```

6. Далее в блоке функции `main` еще раз получите текущее время, а затем выведите время, потребовавшееся для работы цикла.

```
stop = time(NULL);
printf("\nRuntime: %.0f seconds\n" , difftime( stop, go ) ) ;
```

7. В заключение в блоке функции `main` верните значение 0, как того требует описание функции.

```
return 0;
```

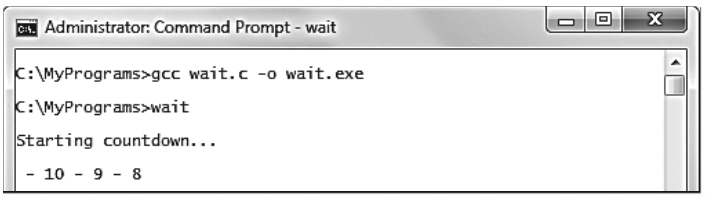
8. После блока функции `main` начните описание функции с инициализации переменной `clock_t` как момента в будущем.

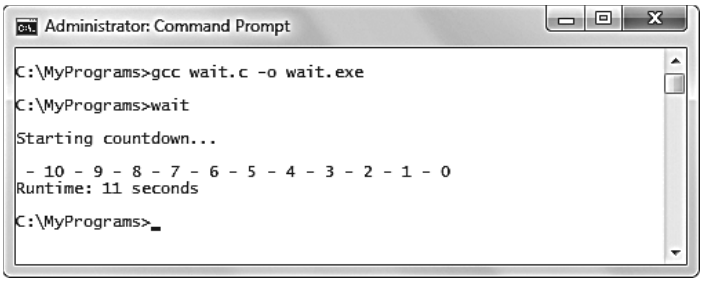
```
void wait(int seconds)
{
    clock_t end_wait = (clock() + (seconds * CLOCKS_PER_SEC));
}
```

9. В заключение в блоке функции `wait` добавьте пустой цикл, который прекращает свою работу, когда текущее время достигает заданной точки в будущем, и верните управление в цикл функции `main`.

```
while(clock() < end_wait) {}
```

10. Сохраните файл программы, а затем скомпилируйте и запустите программу, чтобы увидеть таймер и время ее работы.





Внимание

Количество `clock ticks` зависит от реализации, оно определено в файле `time.h` как константа `CLOCKS_PER_SEC`.

Совет

Вы можете найти время в секундах, прошедшее с начала работы программы, разделив значение, возвращенное функцией `clock()` на значение `CLOCKS_PER_SEC`.

Генерация случайных чисел

В заголовочном файле `stdlib.h` имеется функция `rand()`, которая генерирует псевдослучайное положительное целое число. По умолчанию она вернет число в диапазоне от 0 до очень большого числа (как минимум 32,767). Можно задать определенную верхнюю границу с помощью операции целочисленного деления. Например, чтобы получить число от 0 до 9, следует использовать выражение `rand() % 9`.

Чтобы установить минимальную нижнюю границу диапазона, нужно добавить некоторое значение к результату выражения. Например, чтобы получить число от 1 до 10, следует использовать выражение `(rand() % 9) + 1`.

Числа, генерируемые функцией `rand()`, не являются полностью случайными, поскольку она успешно создает одну и ту же последовательность чисел всякий раз, когда выполняется программа, использующая эту функцию. Для того чтобы сгенерировать другую последовательность чисел, необходимо указать *зерно*, с которого начинается последовательность. По умолчанию, исходное зерно равно 1, но его можно изменить, передав в качестве аргумента функции `srand()` другое целое число. Например, `srand(8)`. Это приведет к тому, что функция `rand()` сгенерирует другую последовательность чисел с использованием нового зерна, но последовательность все равно будет повторяться при каждом запуске программы.

Чтобы каждый раз генерировать другую последовательность случайных чисел, аргумент, передаваемый функции `srand()`, должен быть чем-то другим, нежели статичным целым числом. Распространенное решение — использовать в качестве зерна текущее время в секундах, `srand(time(NULL))`.

Теперь последовательность случайных чисел, сгенерированная с помощью функции `rand()`, будет отличаться при каждом запуске программы, использующей функцию `rand()`.

lotto.c

1. Начните новую программу с инструкций препроцессора, позволяющих включить функции стандартной библиотеки ввода/вывода, функции работы со случайными числами, временем и строками.

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

2. Добавьте функцию `main`, в которой объявляются целочисленные и символьные переменные.

```
int main()
{
    int i, r, temp, nums[50];
    char buf[4], str[50]= { "Your Six Lucky Numbers Are: " } ;
}
```

3. В блоке функции **main** с помощью количества прошедших секунд задайте зерно генератора случайных чисел.

```
srand(time(NULL));
```

4. Далее заполните массив числами от 0 до 49 по порядку.

```
for(i = 0; i < 50; i++) { nums[i] = i; }
```

5. Теперь перемешайте последовательность.

```
for(i = 1; i < 50; i++)
{
    r = (rand() % 49) + 1;
    temp = nums[i]; nums[i] = nums[r]; nums[r] = temp;
}
```

6. Добавьте числа из шести элементов массива в строку.

```
for(i = 1; i < 7; i++)
{
    sprintf(buf, "%d", nums[i]);
    strcat(buf, " "); strcat(str, buf);
}
```

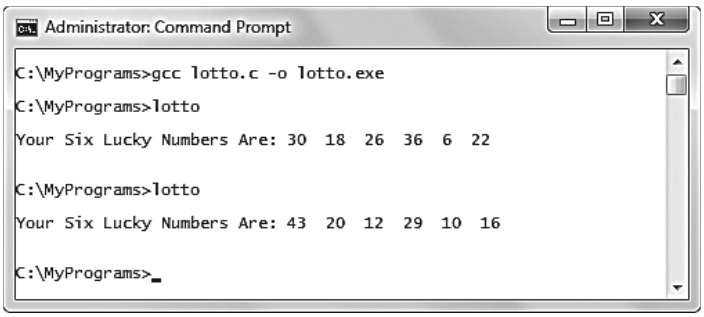
7. Далее выведите строку.

```
printf("\n%s\n\n", str);
```

8. В конце блока функции **main** верните значение 0, как того требует описание функции.

```
return 0;
```

9. Сохраните файл программы, а затем скомпилируйте и запустите программу несколько раз, чтобы увидеть разные последовательности чисел в диапазоне 1-49 при каждом запуске.



На заметку

Элементу массива `nums[0]` присваивается значение 0, но в перемешивании он не участвует, поскольку для выбора доступны только элементы в диапазоне 1-49.

Совет

Обратите внимание на то, как функция `sprintf()` используется для преобразования целочисленных значений к символам, которые впоследствии могут быть сконкатенированы в более крупную строку.

Отображение диалогового окна

В операционной системе Windows программы, написанные на языке C, могут создавать графические компоненты с помощью специальных функций, предоставляемых интерфейсом Windows Application Programming Interface (WINAPI). Эти функции можно использовать в программе, добавив характерный для операционной системы Windows заголовочный файл `windows.h`.

На заметку

Обратите внимание на то, что функция `WinMain()` требует возвращения целого числа по завершении, как и функция `main()`.

В программе, написанной на языке C, для операционной системы Windows, обычная точка входа — функция `main()` — заменяется на специальную функцию `WinMain()`:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow)
```

Эти аргументы всегда необходимы для того, чтобы программа осуществляла коммуникацию с операционной системой. Аргумент `hInstance` — это *дескриптор*, ссылка на программу, аргумент `hPrevInstance`, использовавшийся ранее в программировании для Windows, в наши дни может быть проигнорирован. Аргумент `lpCmdLine` — это строка, которая представляет собой все элементы, используемые в командной строке для компилирования приложения, а аргумент `nCmdShow` управляет способом отображения окна.

Наиболее простым в создании графическим компонентом является простое диалоговое окно, имеющее всего одну кнопку «OK». Его можно создать, вызвав функцию `MessageBox()`:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Совет

Другие константы иконок диалоговых окон — `MB_ICONERROR`, `MB_ICONQUESTION`, `MB_ICONINFORMATION`.

Аргумент `hWnd` — это *дескриптор*, ссылка на родительское окно, если таковое существует — если его нет, этот аргумент будет иметь значение `NULL`. Аргумент `lpText` — это строка с сообщением, которое будет отображено, а аргумент `lpCaption` представляет собой заголовок отображаемого диалогового окна. Наконец, аргумент `uType` может определять иконки и кнопки с помощью специальных константных значений, содержащихся в списке, разделенном вертикальной чертой. Например, константа `MB_OK` добавляет кнопку OK, а иконку с восклицательным знаком можно добавить с помощью константы `MB_ICONEXCLAMATION`.

После компилирования программа, написанная на языке C, которая создает диалоговые окна, обычно запускается с помощью командной строки. Помимо этого ее можно запустить двойным щелчком на иконке исполняемого файла, но это откроет и окно командной строки, и диало-

говое окно. Компилятор Gnu C имеет специальную опцию —**mwindows**, которая может быть добавлена в самом конце команды компиляции, чтобы предотвратить появление окна командной строки в случае, если программа запускается по двойному щелчку.

1. Создайте копию предыдущего примера, **lotto.c**, и переименуйте ее как **winlotto.c**.

winlotto.c

2. В самом начале переименованного файла программы добавьте еще одну инструкцию препроцессора, чтобы сделать доступными функции WINAPI.

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

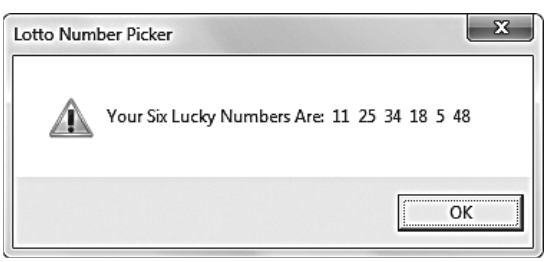
3. Далее замените строку **int main()** на другую, которая осуществляет коммуникацию с операционной системой Windows.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
 lpCmdLine, int nCmdShow)
```

4. Теперь замените вызов функции **printf()**, который выводит на экран выбранные числа, на вызов, который отображает их в диалоговом окне.

```
MessageBox(NULL, str, "Lotto Number Picker", MB_OK | MB_ICONEXCLAMATION);
```

5. Сохраните файл программы, скомпилируйте его с добавлением специальной опции компилятора, а затем запустите программу двумя способами — из командной строки и по двойному щелчку.



Заключение

- Указатель на файл имеет синтаксис `FILE *fp` и может быть использован для открытия, чтения, записи и закрытия файлов.
- Функция `fopen()`, которая открывает файлы, должна знать о расположении файла и о режиме его открытия.
- После завершения операций с файлом этот файл должен быть закрыт с помощью функции `fclose()`.
- Отдельные символы могут быть считаны и записаны с помощью функций `fgetc()` и `fputs()`, а функции `fgets()` и `fputs()` могут считывать целые строки.
- Функции `fread()` и `fwrite()` могут считывать и записывать файловые потоки целиком в символьные буферы и из них.
- Уровень гибкости, предоставляемый функциями `fscanf()` и `fprintf()`, позволяет считывать и записывать данные в файловые потоки, `stdin` или `stdout`.
- Сообщения об ошибках могут быть выведены на экран с помощью функции `perror()`, функция `strerror()` способна вывести сообщение по связанному с ним коду `errno`.
- Функция `time(NULL)` возвращает количество секунд, прошедшее с момента Эры в полночь 1 января 1970 года.
- Члены структуры `tm struct` инициализируются путем преобразования количества прошедших секунд с помощью функции `localtime()`.
- Дата и время в стандартном формате предоставляются функцией `asctime()`, но отдельные компоненты могут быть отформатированы с помощью спецификаторов формата и функции `strftime()`.
- Временные отсечки во время выполнения программы можно получить, вызвав функции `difftime()` и `clock()`.
- Для последовательности псевдослучайных чисел, генерируемой функцией `rand()`, следует предварительно задать зерно с помощью функций `srand()` и `time(NULL)`.
- Включение заголовочного файла `windows.h` в программу делает доступным Windows API, позволяющий создавать графические компоненты.
- Точкой входа в программу Windows является функция `WinMain()`, а функция `MessageBox()` позволяет создать простое диалоговое окно.

Справочная информация

В этой главе книги перечислены стандартные ASCII-коды и все функции, содержащиеся в стандартной библиотеке языка C, сгруппированные по файлам, в которых они находятся, а также описания всех стандартных констант.

- **ASCII-коды символов**
- **Функции ввода и вывода** `<stdio.h>`
- **Функции проверки символов** `<ctype.h>`
- **Арифметические функции** `<math.h>`
- **Функции работы со строками** `<string.h>`
- **Вспомогательные функции** `<stdlib.h>`
- **Диагностические функции** `<assert.h>`
- **Функции для работы с аргументами** `<stdarg.h>`
- **Функции для работы с датой и временем** `<time.h>`
- **Функции переходов** `<setjmp.h>`
- **Сигнальные функции** `<signal.h>`
- **Константы пределов** `<limits.h>`
- **Константы с плавающей точкой** `<float.h>`
- **Основы программирования на языке C**

ASCII-коды символов

ASCII (расшифровывается как American Standard Code for Information Interchange, американская стандартная кодировочная таблица) является стандартным представлением символов с помощью цифровых кодов. Коды для непечатаемых символов, первоначально разработанные для телетайпов, в наше время редко используются для первоначальных целей.

Существуют также расширенные наборы ASCII-кодов, лежащие в диапазоне 128–255 (не перечислены), которые содержат символы с акцентами и многое другое, но эти наборы отличаются друг от друга.

Обратите внимание на то, что символ, представленный кодом 32, на самом деле присутствует — это непечатаемый символ пробела. Символ, представленный кодом 127, — это символ `delete`.

Код	Символ	Описание	Код	Символ	Описание
0	NUL	null	16	DLE	
1	SOH	Начало заголовка	17	DC1	
2	STX	Начало текста	18	DC2	
3	ETX	Конец текста	19	DC3	
4	EOT	Конец передачи	20	DC4	
5	ENQ	Прошу подтверждения	21	NAK	Не подтверждаю
6	ACK	Подтверждаю	22	SYN	Синхронизация
7	BEL	Звонок, звуковой сигнал	23	ETB	Конец блока текста
8	BS	Возврат на один символ	24	CAN	Отмена
9	TAB	Горизонтальная табуляция	25	EM	Конец носителя
10	NL	Новая строка	26	SUB	Подставить
11	VT	Вертикальная табуляция	27	ESC	Начало управляющей последовательности
12	FF	Прогон страницы, новая страница	28	FS	Разделитель файлов
13	CR	Возврат каретки	29	GS	Разделитель групп
14	SO	Изменить цвет ленты	30	RS	Разделитель записей
15	SI	Обратно к предыдущему коду	31	US	Разделитель юнитов

Код	Символ	Код	Символ	Код	Символ	Код	Символ
32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	del

Совет

Чтобы узнать больше о расширенных наборах ASCII-кодов посетите веб-сайт **www.asciitable.com** или выполните поиск по запросу ASCII во Всемирной паутине.

Совет

Термин «ASCII-файл» означает пустой текстовый файл, похожий на те, что можно создать в приложении Блокнот (Notepad) операционной системы Windows.

stdio.h

Функции ввода и вывода

Функции и типы, определенные в заголовочном файле `stdio.h` составляют примерно одну треть всей библиотеки языка C. Они используются для передачи данных в программу и из нее.

Поток — это источник данных, который завершается символом новой строки `\n`. Он может быть считан или записан путем *открытия* потока и освобожден путем его *закрытия*. Открытие потока возвращает указатель типа `FILE`, в котором хранится информация, необходимая для управления потоком.

Функции, перечисленные в следующей таблице, выполняют операции над файлами.

Функции работы с файлами

`FILE fopen(const char *filename, const char mode)`
Функция `fopen()` возвращает указатель типа `FILE` или значение `NULL`, если файл не может быть открыт. При вызове функции следует указать один из следующих режимов:
`r` — открыть файл только для чтения,
`w` — создать текстовый файл для записи и стереть все его предыдущее содержимое,
`a` — открыть или создать текстовый файл для записи в конец файла,
`r+` — открыть текстовый файл для обновления данных (чтение и запись),
`w+` — открыть текстовый файл для обновления данных (чтение и запись, стереть все предыдущее содержимое),
`a+` — открыть или создать текстовый файл для обновления, запись будет производиться в конец файла.
Если нужно открыть бинарный файл, к режиму следует добавить символ `b`. Например, `wb+`.

`FILE freopen(const char *filename, const char mode, FILE *stream)`
Функция `freopen()` открывает файл в заданном режиме и привязывает его к потоку. Она возвращает поток или значение `NULL`, если происходит ошибка. Обычно эта функция используется для изменения файлов, связанных с потоками `stdin`, `stdout` или `stderr`.

int fflush(FILE *stream)
Вызов этой функции приводит к тому, что буферизованные данные, находящиеся в выходном потоке, немедленно записываются. При ошибке записи функция возвращает константу **EOF**, в противном случае — значение **NULL**. Вызов **fflush(NULL)** очищает все выходные потоки.

int fclose(FILE *stream)
Вызов функции **fclose()** очищает любые незаписанные данные из потока, а затем закрывает поток. Эта функция возвращает константу **EOF** в случае ошибки или, в противном случае, 0.

int remove(const char *filename)
Эта функция удаляет указанный файл — все последующие попытки открыть его обернутся неудачей. Функция возвращает ненулевое значение, если файл удалить не удается.

int rename(const char *old-name, const char *new-name)
Функция **rename()** изменяет имя указанного файла или возвращает ненулевое значение, если файл переименовать не удается.

FILE *tmpfile(void)
Вызов функции **tmpfile()** создает временный файл, открытый в режиме **wb+**, который будет удален по окончании работы программы. Эта функция возвращает поток или значение **NULL**, если файл создать не удается.

char *tmpnam(char arr[L_tmpnam])
Эта функция хранит строку, располагающуюся в массиве, и возвращает указатель с уникальным именем, указывающим на этот массив. Массив **arr** должен содержать как минимум **L_tmpnam** символов. Функция **tmpnam()** генерирует новое имя при каждом вызове.

int setvbuff(FILE *stream, char *buffer, int mode, size_t size)
Вызов функции **setvbuff()** начинает буферизацию указанного потока. Эта функция должна быть вызвана после того, как поток был открыт, но перед тем, как с ним будет выполнена хотя бы одна операция. Корректными режимами для этой функции являются **_IOFBF** (указывает проводить полную буферизацию), **_IOLBF** (указывает проводить буферизацию строки) и **_IONBF** (отключает буферизацию). Значение переменной **size** указывает размер буфера. Функция возвращает ненулевое значение, если происходит ошибка.

void setbuf(FILE *stream, char *buffer)
Функция **setbuf()** определяет, как будет буферизован поток. Эта функция должна быть вызвана после того, как поток был открыт, но перед тем, как с ним будет выполнена хотя бы одна операция. Аргумент **buffer** указывает на массив, который будет использован в качестве буфера.

stdio.h

Функции для форматирования выходных данных

int fprintf(FILE *stream, const char *format, …)
Функция **fprintf()** преобразует и записывает данные в указанный файловый поток под управлением спецификатора формата. Эта функция возвращает количество записанных символов или отрицательное значение, если происходит ошибка.

int printf(const char *format, …)
Функция **printf()** преобразует и записывает данные в поток **stdout**. Ее вызов эквивалентен вызову **fprintf(stdout, const char *format)**.

int sprintf(char *s, const char *format)
Функция **sprintf()** аналогична функции **printf()** за исключением того, что данные записываются в строку, которая завершается символом \0.

vprintf(const char *format va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
Эти три функции эквивалентны соответствующим функциям **printf()** за исключением того, что их переменное число параметров заменяется аргументом типа **va_list type**. Обратитесь к списку функций **stdarg.h** (страница 180) для получения более подробной информации.

Функции для форматирования входных данных

int fscanf(FILE *stream, const char *format, …)
Функция **fscanf()** считывает данные из указанного потока под управлением спецификатора формата и присваивает преобразованные значения указанным аргументам. Эта функция возвращает количество преобразованных символов или константу **EOF** если достигнут конец файла или произошла ошибка.

int scanf(const char *format, …)
Функция **scanf()** преобразовывает и считывает данные из потока **stdin**, Ее вызов эквивалентен вызову **fscanf(stdin, const char *format)**

int sscanf(char *s, const char *format, …)
Функция **sscanf()** аналогична функции **scanf()** за исключением того, что входные данные считываются из указанной строки.

Спецификаторы формата выходных данных

stdio.h

В языке программирования C префикс % указывает на спецификатор формата. На этой странице перечислены все спецификаторы для функций **printf()**, на следующей — для функций **scanf()**.

Символ	Функция printf() выполняет преобразование
d, i	Целочисленный тип данных, знаковое десятичное число
o	Целочисленный тип данных, беззнаковое восьмеричное число без нуля в начале
x, X	Целочисленный тип данных, беззнаковое шестнадцатеричное число. 0x для отображения в нижнем регистре (например, 0xff), 0X для отображения в верхнем регистре (например, 0XFF)
u	Целочисленный тип данных, беззнаковое десятичное число
c	Целочисленный тип данных, один символ после преобразования к типу char
s	Символьный указатель на строку, которая закончится символом \0
f	Тип данных double в формате xxx.yyyyyy, где количество знаков после десятичной точки определяется заданной точностью. По умолчанию точность равна шести знакам
e, E	Тип данных double в формате xx.yyyyyye±zz или xx.yyyyyyE±zz, где количество знаков после десятичной точки определяется заданной точностью. По умолчанию точность равна шести знакам
g, G	Тип данных double, выводится как преобразование %e, %E или %f, выбирается самый короткий вариант
p	Адрес указателя в памяти
n	Не является преобразованием, сохраняет в целочисленном указателе количество символов, записанных к моменту вызова функции printf()
%	Не является преобразованием, выводит символ «%»

stdio.h

Спецификаторы формата входных данных

Символ	Функция scanf() выполняет преобразование
d	Целочисленный тип данных, знаковое десятичное число
i	Целочисленный тип данных, число может быть восьмеричным (оно будет начинаться с 0) или шестнадцатеричным (оно будет начинаться с 0x или 0X)
o	Целочисленный тип данных, восьмеричное число, которое может начинаться с 0
u	Целочисленный тип данных, беззнаковое десятичное число
x	Целочисленный тип данных, шестнадцатеричное число, которое может начинаться с 0
c	Символы, которые будут помещены в указанный массив. Считывает количество символов, указанное как ширина строки (по умолчанию 1), не добавляя символ \0 к концу строки. Считывание прекратится, если встретится пробел
s	Строка, не содержащая пробелов, которая будет помещена в указанный массив. Он должен быть достаточно большим, чтобы вместить все символы плюс завершающий символ \0
e, f, g	Тип данных с плавающей точкой, число может иметь знак. После знака будут располагаться цифры в формате строки. Выражение может иметь десятичную точку, а также экспоненту ("e" или "E"), после которой будет следовать число, которое может иметь знак
p	Адрес в памяти, имеет тот же формат, что и выводимое функцией printf() значение под управлением преобразования %p
n	Не является преобразованием. Хранит количество символов, считанных к моменту вызова функции scanf(), которое будет сохранено в целочисленном указателе
[…]	Выполняет сравнение строки из потока со строкой, указанной в квадратных скобках, и добавляет символ \0
[^…]	Выполняет сравнение всех символов ASCII из потока, исключая символы, указанные в квадратных скобках, и добавляет символ \0

Функции для ввода и вывода символов

stdio.h

`int fgetc(FILE *stream)` Возвращает следующий символ указанного потока как переменную типа `char` или константу `EOF`, если достигнут конец файла или произошла ошибка
`char *fgets(char *s, int n, FILE *stream)` Считывает следующие *n*-1 символов указанного потока, затем добавляет символ `\0` в конец массива. Эта функция возвращает указатель `s` или значение `NULL`, если достигнут конец файла или произошла ошибка
`int fputc(int c, FILE *stream)` Записывает символ `c` в указанный поток и возвращает записанный символ или константу `EOF`, если произошла ошибка
`int fputs(const char *s, FILE *stream)` Записывает строку `s` в указанный поток и возвращает неотрицательное значение или константу `EOF`, если произошла ошибка
`int getc(FILE *stream)` Функция `getc()` эквивалентна функции `fgetc()`
`int getchar(void)` Функция `getchar()` эквивалентна функции `getc(stdin)`
`char *gets(char *s)` Считывает следующую введенную строку в массив, заменяя ее завершающий символ перевода каретки символом `\0`. Эта функция возвращает указатель `s` или значение `NULL`, если достигнут конец файла или произошла ошибка
`int putc(int c, FILE *stream)` Функция `putc()` эквивалентна функции `fputc()`
`int putchar(int c)` эквивалентна функции `putc(c, stdout)`
`int puts(const char *s)` Записывает строку `s` и символ новой строки в поток `stdout`. Эта функция возвращает неотрицательное значение или константу `EOF`, если произошла ошибка
`int ungetc(int c, FILE *stream)` Помещает символ `c` обратно в указанный поток, он будет считан при следующем обращении. Гарантированно можно выполнить только один возврат символа, константу `EOF` вернуть нельзя. Эта функция возвращает отправленный назад символ или константу `EOF`, если произошла ошибка

stdio.h

Функции ввода и вывода данных напрямую из потоков

Функции **fread()** и **fwrite()**, содержащиеся в заголовочном файле **stdio.h**, наиболее эффективны для чтения и записи текстовых файлов целиком:

size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream) Функция **fread()** считывает данные из указанного потока в указанный массив *ptr*. Может быть считано максимум *nobj* объектов размера *size*. Эта функция возвращает количество считанных объектов, которое может быть меньше запрошенного количества объектов. Состояние функции **fread()** можно узнать во время ее выполнения с помощью функций **feof()** и **ferror()** — информацию о них вы найдете ниже на странице
size_t fwrite(const void *ptr, size_t, size, size_t nobj, FILE *stream) Функция **fwrite()** записывает в поток *nobj* объектов размера *size* из указателя *ptr*. Эта функция возвращает количество записанных объектов. Если произойдет ошибка, возвращенное значение будет меньше, чем количество запрошенных объектов *nobj*.

Функции для работы с ошибками

Множество стандартных функций библиотеки языка программирования C устанавливают индикаторы, когда происходит ошибка или когда они достигают конца файла. Эти индикаторы разрешается проверить с помощью перечисленных ниже функций. Также целочисленное выражение **errno**, определенное в заголовочном файле **errno.h**, может содержать код, с помощью которого можно узнать более подробную информацию об ошибке, произошедшей последней.

void clearer(FILE *stream) Функция **clearer()** очищает индикаторы конца файла и ошибок для заданного потока
int feof(FILE *stream) Функция **foef()** возвращает ненулевое значение, если для заданного потока был установлен индикатор конца файла
int ferror(FILE *stream) Функция **ferror()** возвращает ненулевое значение, если для заданного потока был установлен индикатор ошибки
void perror(const char *s) Функция **perror()** выводит определяемое реализацией сообщение об ошибке, соответствующее целочисленному значению, содержащемуся в выражении **errno**. Смотрите также информацию о функции **strerror()**, которая располагается в заголовочном файле **string.h**

Функции позиционирования в файлах

stdio.h

Файловый поток обрабатывается по одному символу за раз. Функции, приведенные в следующей таблице, могут использоваться для управления позицией в файловом потоке.

`int fseek(FILE *stream, long offset, int original)` Функция `fseek()` устанавливает позицию файла в заданном потоке. Все последующие операции чтения и записи будут производиться в новой позиции. Новая позиция определяется путем указания, насколько следует сместиться (*offset*) от оригинальной позиции (*original*). Опционально третий аргумент может иметь значение `SEEK_SET` (начало файла), `SEEK_CUR` (текущая позиция) или `SEEK_END` (конец файла). Для текстового потока смещение может быть равно либо нулю, либо значению, возвращенному функцией `ftell()`, — при этом значение оригинальной позиции должно быть равно `SEEK_SET`. Функция `fseek()` возвращает ненулевое значение, если происходит ошибка
`long ftell(FILE *stream)` Функция `ftell()` возвращает позицию в текущем файле указанного потока или значение -1, если происходит ошибка
`int fgetpost(FILE *stream, fpos_t *ptr)` Функция `fgetpos()` записывает текущую позицию файла указанного потока в заданный указатель *ptr*, который имеет специальный тип `fpos_t`. Эта функция возвращает ненулевое значение, если происходит ошибка
`int fsetpos(FILE *stream, const fpos_t *ptr)` Функция `fsetpos()` позиционирует файловый указатель в заданном потоке в позиции, записанной функцией `fgetpos()` в указатель *ptr*. Эта функция возвращает ненулевое значение, если происходит ошибка

ctype.h

Функции проверки символов

Заголовочный файл **ctype.h** содержит функции для проверки символов. В каждом случае символ должен быть передан как аргумент функции. Она возвращает ненулевое значение (**true**), если символ соответствует заданным условиям, или 0 (**false**) — в противном случае. В дополнение, этот заголовочный файл содержит две функции, предназначенные для преобразования регистра букв. Все функции заголовочного файла **ctype.h**, а также их описание, перечислены в таблице ниже:

Функция	Описание
`isalpha(c)`	Является ли символ буквой?
`isalnum(c)`	Является ли символ буквой или цифрой?
`iscntrl(c)`	Является ли символ управляющим?
`isdigit(c)`	Является ли символ десятичной цифрой?
`isgraph(c)`	Является ли символ печатаемым (не включая пробел)?
`islower(c)`	Является ли символ буквой в нижнем регистре?
`isprintf(c)`	Является ли символ печатаемым (включая пробел)?
`ispunct(c)`	Является ли символ печатаемым (исключая пробелы, буквы и цифры)?
`isspace(c)`	Является ли символ пробелом или символом отправки формы, новой строки, возврата каретки, горизонтальной или вертикальной табуляции?
`isupper(c)`	Является ли символ буквой в верхнем регистре?
`isxdigit(c)`	Является ли символ шестнадцатеричной цифрой?
`int tolower(int c)`	Преобразует символ к нижнему регистру
`int toupper(int c)`	Преобразует символ к верхнему регистру

Арифметические функции

math.h

В заголовочном файле `math.h` содержатся функции, которые выполняют математические расчеты. Все функции и их описания перечислены в таблице ниже. В этой таблице переменные `x` и `y` имеют тип `double`, а переменная `n` — тип `int`. Все функции возвращают значение типа `double`:

Функция	Описание
`sin(x)`	Возвращает синус переменной `x`
`cos(x)`	Возвращает косинус переменной `x`
`tan(x)`	Возвращает тангенс переменной `x`
`asin(x)`	Возвращает арксинус переменной `x`
`acos(x)`	Возвращает арккосинус переменной `x`
`atan(x)`	Возвращает арктангенс переменной `x`
`atan2(y, x)`	Возвращает угол (в радианах) между осью `x` и точкой `y`
`sinh(x)`	Возвращает гиперболический синус переменной `x`
`cosh(x)`	Возвращает гиперболический косинус переменной `x`
`tanh(x)`	Возвращает гиперболический тангенс переменной `x`
`exp(x)`	Возвращает значение `e` (основание натурального логарифма) в степени `x`
`log(x)`	Возвращает натуральный логарифм переменной `x`
`log10(x)`	Возвращает десятичный логарифм переменной `x`
`pow(x, y)`	Возвращает значение `x` в степени `y`
`sqrt(x)`	Возвращает квадратный корень из `x`
`ceil(x)`	Возвращает минимальное целое число, не меньше `x`, имеющее тип `double`
`floor(x)`	Возвращает максимальное целое число, не большее `x`, имеющее тип `double`
`fabs(x)`	Возвращает модуль `x`
`ldexp(x, n)`	Возвращает значение `x`, умноженное на 2 и возведенное в степень `n`
`frexp(x, int *exp)`	Разбивает значение `x` на две части — возвращает мантиссу (лежащую в диапазоне 0.5 и 1) и сохраняет экспоненту в `exp`
`modf(x, double *ip)`	Разбивает `x` на целую и дробную части — дробную часть возвращает, а целую сохраняет в `ip`
`fmod(x, y)`	Возвращает остаток от деления `x` на `y`

string.h

Функции работы со строками

Заголовочный файл string.h содержит следующие функции, которые могут быть использованы для сравнения или манипулирования текстовыми строками:

Функция	Описание
`char *strcpy(s1, s2)`	Копирует строку s1 в строку s2, затем возвращает строку s1
`char *strncpy(s1, s2, n)`	Копирует n символов строки s1 в строку s2, затем возвращает строку s1
`char *strcat(s1, s2)`	Добавляет строку s2 к концу строки s1, затем возвращает строку s1
`char *strncat(s1, s2, n)`	Добавляет n символов строки s2 к концу строки s1, затем возвращает строку s1
`char *strncmp(s1, s2)`	Сравнивает строки s1 и s2, затем возвращает <0, если s1 < s2 или 0, если s1 == s2 или >0, если s1 > s2
`char *strncmp(s1, s2, n)`	Сравнивает n символов строк s1 и s2, затем возвращает <0, если s1 < s2 или 0, если s1 == s2 или >0, если s1 > s2
`char *strchr(s, c)`	Возвращает указатель на первое включение символа c в строке s или значение NULL, если символ не найден
`char *strrchr(s, c)`	Возвращает указатель на последнее включение символа c в строке s или значение NULL, если символ не найден
`size_t strspn(s1, s2)`	Возвращает длину префикса строки s1, содержащего символы строки s2
`size_t strcspn(s1, s2)`	Возвращает длину префикса строки s1, содержащего символы, отсутствующие в строке s2
`char *strpbrk(s1, s2)`	Возвращает указатель на первое включение любого символа строки s2 в строке s1 или значение NULL, если ничего не найдено
`char *strstr(s1, s2)`	Возвращает указатель на первое включение строки s2 в строке s1 или значение NULL, если ничего не найдено

string.h

Функция	Описание
`size_t strlen(s)`	Возвращает длину строки `s`
`char *strerror(n)`	Возвращает указатель на определяемую реализацией строку, связанную с кодом ошибки `n`
`char *strok(s1, s2)`	Разбивает строку `s1` на фрагменты, используя разделители, содержащиеся в строке `s2`
`void *memcpy(s1, s2, n)`	Копирует `n` символов из строки `s2` в строку `s1`, затем возвращает строку `s1`
`void *memmove(s1, s2, n)`	Аналогична функции `memcpy()`, но она также работает в случае, если объекты пересекаются
`int memcmp(s1, s2, n)`	Сравнивает первые `n` символов строк `s1` и `s2`, возвращает значения, аналогичные значениям, которые возвращает функция `strcmp()`
`void *memchr(s, c, n)`	Возвращает указатель на первое включение символа `c` в строке `s` или значение `NULL`, если символ не найден в первых `n` символах строки
`void *memset(s, c, n)`	Заполняет первые `n` символов строки `s` указанным символом `c`

stdlib.h

Вспомогательные функции

`double atof(const char *s)` преобразовывает строку `s` к типу `double`
`int atoi(const char *s)` преобразовывает строку `s` к типу `int`
`long atol(const char *s)` преобразовывает строку `s` к типу `double`
`double strtod(const char *s, char **endp)` Преобразовывает начальный фрагмент строки `s` к переменной типа `double`, игнорируя пробелы в начале этой строки. Указатель на остальную часть строки хранится в `*endp`.
`long strtol(const char *s, char **endp, int b)` Преобразовывает начальный фрагмент строки `s` к переменной типа `long` с основанием `b`, игнорируя пробелы в начале этой строки. Указатель на остальную часть строки хранится в `*endp`.
`unsigned long strtoul(const char *s, char **endp, int b)` Функция аналогична функции `strtol()`, за исключением того, что она выполняет преобразование к типу `unsigned long`
`int rand(void)` Возвращает псевдослучайное число, лежащее в диапазоне между 0 и зависящим от реализации максимумом (не меньше 32.767)
`void srand(unsigned int seed)` Устанавливает зерно для новой последовательности случайных чисел, генерируемой функцией `rand()`. Исходное зерно равно 1
`void *calloc(size_t nobj, size_t size)` Возвращает указатель на фрагмент в памяти, выделенный для массива объектов *nobj*, имеющих размер *size*, или значение `NULL`, если требование не может быть выполнено. Выделенная память заполняется нулями
`void *malloc(size_t size)` Возвращает указатель на фрагмент памяти, выделенный для объекта, имеющего размер *size*, или значение `NULL`, если требование не может быть выполнено. Выделенная память никак не инициализируется
`void *realloc(void *p, size_t size)` Изменяет размер объекта, на который ссылается указатель *p*, на размер, указанный в переменной *size*. Эта функция возвращает указатель на новый фрагмент памяти или значение `NULL`, если операцию нельзя выполнить

`void *free(void *p)` Функция `free()` освобождает ранее выделенную память, но которую ссылается указатель `p`. Обратите внимание на то, что *`p`* должен быть указателем, содержащим результат вызова функций `calloc()`, `malloc()` или `realloc()`.
`void abort(void)` заставляет программу аварийно завершиться
`void exit(int status)` Заставляет программу завершиться обычным способом. Значение переменной *`status`* будет возвращено системе. Опционально переменная *`status`* может иметь значение `EXIT_SUCCESS` или `EXIT_FAILURE`
`int atexit(voif (*fcn) (void))` Указывает, что функция *`fcn`* должна быть вызвана по завершении программы. Функция `atexit()` возвращает ненулевое значение, если произошла ошибка
`int system(const char *s)` Передает строку *`s`* операционной системе для обработки. Возвращаемое значение зависит от реализации
`char *getenv(const char name)` Возвращает строку переменной окружения с именем *`name`* или `NULL`, если строки, связанной с именем *`name`* не существует. Детали зависят от реализации
`void *bsearch(const void *key, const void *base, size_t n, size_t size, int (*cmp), (const void *keyval, const void *datum))` Выполняет поиск элемента, соответствующего значению *`key`*, в диапазоне `base[0]…base[n-1]`. Эта функция возвращает указатель на найденный элемент. Если элемент не найден, будет возвращено значение `NULL`. Элементы в массиве `base` должны располагаться в порядке возрастания
`void qsort(void *base, size_t n, size_t size, int (*cmp) (const void*, const void *))` Функция выполняет сортировку диапазона *`base`*`[0]…`*`base`*`[n-1]` объектов, имеющих размер *`size`*, в порядке возрастания. Функция сравнения `cmp()` является аналогом функции `bsearch()`
`int abs(int n)` Возвращает модуль числа `int n`
`long labs(long n)` Возвращает модуль числа `long n`
`div_t ldiv(long num, long demon)` Делит число `num` на число `denom` и сохраняет результаты как члены структуры, имеющей тип `ldiv_t`. Член этой структуры `quot` содержит частное, а `rem` — остаток

assert.h

Диагностические функции

Функция **assert()**, расположенная в файле **assert.h**, может быть использована для диагностирования программы:

```
void assert(int expression);
```

Если значение выражения **expression** равно нулю во время вызова **assert(expression)**, функция выведет сообщение об ошибке в поток **stderr**, например, такое:

```
Утверждение не выполнено: выражение, файл имяфайла, строка номерстроки
```

Затем функция **assert()** попытается завершить программу

stdarg.h

Функции для работы с аргументами

Заголовочный файл **stdarg.h** содержит функции, используемые для прохождения по списку аргументов функции, если вы не знаете их количество и тип. По своей природе эти функции должны быть реализованы как макросы внутри тела функции.

Список аргументов присваивается переменной, имеющей специальный тип данных **va_list**. Функции, перечисленные в следующей таблице, работают с переменной, имеющей имя **args** и тип **va_list**:

va_start(va_list ***args*****,** ***lastarg*****)** Эта функция должна быть вызвана один раз, чтобы инициализировать переменную ***args***, имеющую тип **va_list**, в позиции последнего известного аргумента ***lastarg***.
va_arg(va_list ***args*****,** ***data-type*****)** После того, как переменная **va_list** ***args*** была инициализирована путем вызова функции **va_start**, каждый успешный вызов функции **va_arg()** вернет значение следующего аргумента в списке **args** как значение типа ***data-type***
va_end(va_list ***args*****)** Эта функция должна быть вызвана только один раз после того, как был обработан список **va_list** ***args***, но до того, как функция завершит работу

Функции для работы с датой и временем

time.h

Заголовочный файл time.h содержит функции для работы с датой и временем. Некоторые из них работают с «календарным временем», которое основано на Григорианском календаре. Оно выражается в секундах, прошедших с момента начала эры UNIX (00:00:00 GMT 1 января 1970 года).

Другие функции, содержащиеся в файле time.h, работают с «местным временем», которое является преобразованием календарного времени согласно часовому поясу.

Тип данных time_t используется для описания как календарного, так и местного времени.

Структура с именем tm содержит компоненты календарного времени, которые перечислены в следующей таблице:

Имя	Описание
int tm_sec	Секунды от начала минуты, 0-60
int tm_min	Минуты от начала часа, 0-61
int tm_hour	Часы от полуночи, 0-23
int tm_mday	Число месяца, 1-31
int tm_mon	Месяцы после января, 0-11
int tm_year	Года с 1900
int tm_wday	Дни с воскресенья, 0-6
int tm_yday	Дни с 1 января, 0-365
int tm_isdst	Признак летнего времени

Компонент tm_isdst положителен, если действует летнее время, равен нулю, если действует зимнее время, или отрицателен, если информация недоступна.

Функции, содержащиеся в заголовочном файле time.h, перечислены в таблице, приведенной на следующей странице.

time.h

Заголовочный файл time.h содержит функции, перечисленные ниже, которые могут быть использованы для работы с датой и временем:

`clock_t clock(void)` Возвращает процессорное время, использованное программой с момента ее запуска, или -1, если эти данные недоступны
`time_t time(time_t *tp)` Возвращает текущее календарное время или -1, если эти данные недоступны
`double difftime(time_t time2, time_t time1)` Возвращает разницу между *time2* и *time1*, выраженную в секундах
`time_t mktime(struct tm *tp)` Преобразует местное время, содержащееся в структуре **tp*, в календарное
`char *asctime(const struct tm *tp)` Преобразует время, содержащееся в структуре **tp*, в стандартную строку
`char *ctime(const time_t *tp)` Преобразует календарное время в местное
`struct tm *gmtime(const time_t *tp)` Преобразует календарное время во всемирное координированное время (UTC или GMT)
`struct tm *localtime(const time_t *tp)` Преобразует календарное время, хранящееся в структуре **tp*, в местное
`size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)` Форматирует время, хранящееся в структуре **tp*, в выбранный формат **fmt*

Функция `strftime()` форматирует выбранные компоненты структуры `tm` в соответствии с указанным спецификатором формата. Все возможные спецификаторы формата перечислены в таблице, расположенной на соседней странице.

`time.h`

Спецификатор	Описание
`%a`	Сокращенное название дня недели
`%A`	Полное название дня недели
`%b`	Сокращенное название месяца
`%B`	Полное название месяца
`%c`	Представление местного времени
`%d`	День месяца (01-31)
`%H`	Часы (00-23)
`%I`	Часы (01-12)
`%j`	День года (001-366)
`%m`	Месяц года (01-12)
`%M`	Минуты (00-59)
`%p`	Местный эквивалент времени до и после полудня
`%S`	Секунды (00-61, с учетом секунды координации)
`%U`	Номер недели в году (Воскресенье как первый день недели, 00-53)
`%w`	Номер для недели (0-6, где 0 — это воскресенье)
`%W`	Номер недели в году (Понедельник как первый день недели, 00-53)
`%x`	Представление местной даты
`%X`	Представление местного времени
`%y`	Год без указания столетия (00-99)
`%Y`	Год с указанием столетия
`%Z`	Имя часового пояса (если доступно)

setjmp.h

Функции переходов

Заголовочный файл **setjmp.h** используется для управления низкоуровневыми вызовами и предоставляет способы избежать обычной последовательности вызова и возврата.

int setjmp(jmp_buf *env*)
Низкоуровневая функция, использующаяся в условных проверках для сохранения окружения в переменной *env*, и затем возвращающая 0.

void longjmp(jmp_buf env, int *value*)
Восстанавливает окружение, которое было сохранено в переменной *env* с помощью функции **setjmp()**, как если бы функция **setjmp()** вернула значение **value()**

signal.h

Сигнальные функции

Заголовочный файл **signal.h** содержит функции, предназначенные для обработки исключительных состояний, которые могут возникнуть во время выполнения программы:

void (*signal (int *sig*, void (handler*) (int))) (int)**
Функция **signal()** определяет, как будут обработаны последующие вызовы. Обработчик может иметь значение **SIG_DFL** – значение по умолчанию, зависящее от реализации, или **SIG_IGN**, позволяющее проигнорировать сигнал. Переменная *sig* может иметь одно из следующих значений:
SIGABRT: аварийное прекращение работы
SIGFPE: арифметическая ошибка
SIGILL: некорректная инструкция
SIGINT: внешнее прерывание
SIGSEGV: попытка получить доступ к памяти за пределами выделенного участка
SIGTERM: программе выслан запрос на завершение работы
Функция возвращает предыдущее значение обработчика заданного сигнала или **SIG_ERR**, если происходит ошибка. Когда сигнал *sig* срабатывает в следующий раз, сигнал восстанавливает свое первоначальное поведение, а затем вызывается обработчик сигнала. Если обработчик сигнала возвращает управление, выполнение программы продолжается с точки, где возник сигнал

int raise(int *sig*)
Эта функция пытается отправить сигнал *sig* программе и возвращает ненулевое значение, если попытка не удалась

Константы пределов

limits.h

В следующей таблице перечислены константы, связанные с максимальными и минимальными числовыми пределами. Их значения могут варьироваться в зависимости от реализации. Если таковое указано в скобках, оно представляет собой минимальное значение данной константы, но она может иметь и более крупные значения. При написании программ вы не должны предполагать, что какая-либо константа, зависящая от реализации, будет иметь определенное значение.

Константа	Значение
CHAR_BIT	Количество бит в переменной типа char (8)
CHAR_MAX	Максимальное значение переменной типа char (UCHAR_MAX или SCHAR_MAX)
CHAR_MIN	Минимальное значение переменной типа char (0 или SCHAR_MIN)
INT_MAX	Максимальное значение переменной типа int (+32,767)
INT_MIN	Минимальное значение переменной типа int (-32,767)
LONG_MAX	Максимальное значение переменной типа long (+2,147,483,647)
LONG_MIN	Минимальное значение переменной типа long (-2,147,483,647)
SCHAR_MAX	Максимальное значение переменной типа signed char (+127)
SCHAR_MIN	Максимальное значение переменной типа signed char (-127)
SHRT_MAX	Максимальное значение переменной типа short (+32,767)
SHRT_MIN	Минимальное значение переменной типа short (-32,767)
UCHAR_MAX	Максимальное значение переменной типа unsigned char (+255)
UINT_MAX	Максимальное значение переменной типа unsigned int (+65,535)
ULONG_MAX	Максимальное значение переменной типа unsigned long (+4,294,967,295)
USHRT_MAX	Максимальное значение переменной типа unsigned short (+65,535)

float.h

Константы с плавающей точкой

В следующей таблице перечислены константы, связанные с арифметикой с плавающей точкой. Их значения могут варьирования в зависимости от реализации. Если значение указано в скобках, это говорит о том, что оно является минимальным для данной константы.

Константа	Значение
FLT_RADIX	Основание экспоненциального представления с плавающей точкой (2)
FLT_ROUNDS	Округление до ближайшего числа
FLT_DIG	Количество цифр точности (5)
FLT_EPSILON	Наименьшее число x, где 1.0 + x != 1.0 (1E-5)
FLT_MANT_DIG	Количество цифр мантиссы FLT_RADIX
FLT_MAX	Максимальное число с плавающей точкой (1E+37)
FLT_MAX_EXP	Максимальное число n, при котором выражение $FLT\_RADIX^n - 1$ корректно
FLT_MIN	Минимальное число с плавающей точкой (1E-37)
FLT_MIN_EXP	Минимальное число n, при котором выражение 10^n корректно
DBL_DIG	Число знаков для числа типа double (10)
DBL_EPSILON	Минимальное число x, при котором 1.0 + x != 1.0 (1E-9)
DBL_MANT_DIG	Количество цифр мантиссы FLT_RADIX
DBL_MAX	Максимальное число типа double (1E + 37)
DBL_MAX_EXP	Максимальное число n, при котором выражение $FLT\_RADIX^n - 1$ корректно
DBL_MIN	Минимальное число типа double (1E-37)
DBL_MIN_EXP	Минимальное число n, при котором выражение 10^n корректно

Основы программирования на языке C

32 слова, перечисленные в следующей таблице, являются ключевыми словами языка программирования C. Они имеют особое значение в программах, написанных на языке C, и не могут быть использованы для любых других целей.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

В следующей таблице приведены 14 управляющих последовательностей, которые могут быть использованы в программах, написанных на языке C, чтобы добавить специальные символы, которые обычно имеют особое значение для компилятора.

Последовательность	Значение	Последовательность	Значение
\n	Новая строка	\\	Обратный слеш
\t	Табуляция	\?	Вопросительный знак
\v	Вертикальная табуляция	\'	Одинарная кавычка
\b	Возврат на шаг	\"	Двойная кавычка
\r	Возврат каретки	\xhh	Шестнадцатеричное число
\f	Отправка формы	\000	Восьмеричное число
\a	Звонок	\0	Символ-терминатор

Предметный указатель

Начни программировать прямо сейчас!

САМОЕ ВАЖНОЕ:

- ТИПЫ ДАННЫХ, ОПЕРАТОРЫ И ВЫРАЖЕНИЯ
- УКАЗАТЕЛИ И ССЫЛКИ
- ФУНКЦИИ
- ЛОКАЛЬНЫЕ И ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
- ШАБЛОНЫ
- МАССИВЫ
- СТРУКТУРЫ И ПЕРЕЧИСЛЕНИЯ
- УПРАВЛЯЮЩИЕ ИНСТРУКЦИИ
- И МНОГОЕ ДРУГОЕ

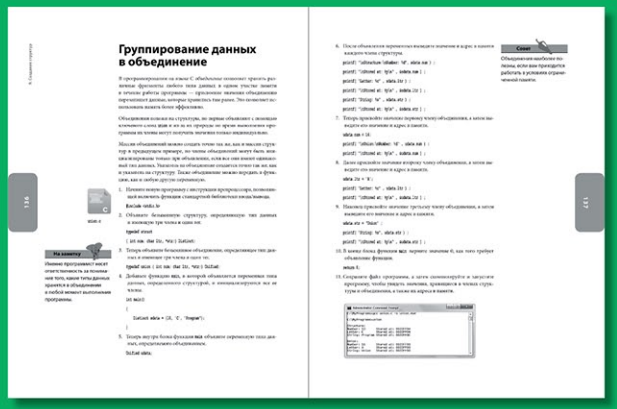
ПРОГРАММИРОВАНИЕ НА С ДЛЯ НАЧИНАЮЩИХ является исчерпывающим руководством для того, чтобы научиться программировать на языке С.

В этой книге с помощью примеров программ и иллюстраций, показывающих результаты работы кода, разбираются все ключевые аспекты языка С. В этой книге описано даже то, как установить бесплатный компилятор для языка С и работать в нем, — у вас просто не будет шансов ошибиться!

ПРОГРАММИРОВАНИЕ НА С ДЛЯ НАЧИНАЮЩИХ идеально подойдет программистам, переключающимся на работу с другим языком, студентам, изучающим язык С, а также тем, кто только начинает свою профессиональную деятельность и хочет научиться основам процедурного программирования.

ЧТО ВНУТРИ?

Эти значки сделают обучение еще проще. Каждый раз, когда при чтении книги вы встречаете один из этих значков, знайте — мы приготовили для вас какой-то полезный совет, придающий остроты процессу обучения, выделили нечто необходимое для запоминания или выносим предостережения держаться подальше от возможных проблем.



«Хорошее пособие для изучающих программирование. Позволяет начать создавать программы на С даже тем, кто раньше ни имел дела с программированием».

Александр Корчагин,
руководитель отдела финансовых информационных систем, Центральная Дистрибьюторская Компания

ISBN 978-5-699-79117-0