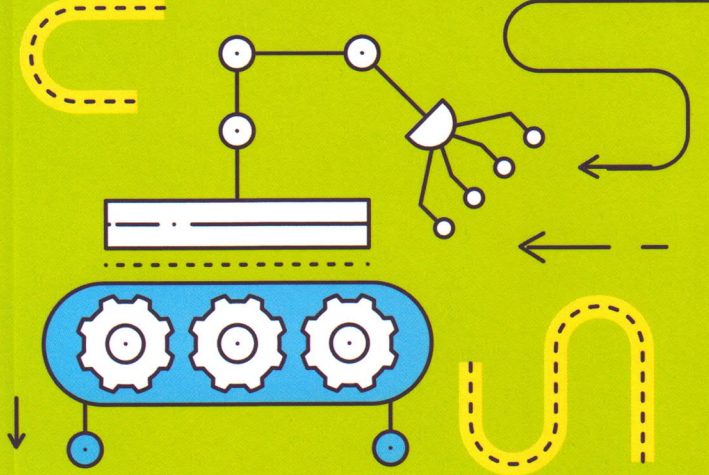


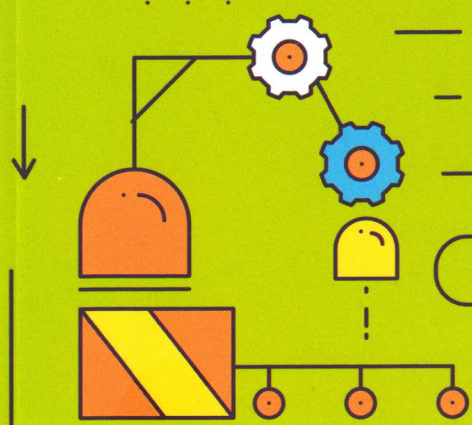
Основы языка
Си на практике

Показательные
примеры
(от простейшей
программы до
многопоточности на Си)
с разбором кода

Основывается на
последних стандар-
тах (C99, C11)

Си
НА ПРИМЕРАХ

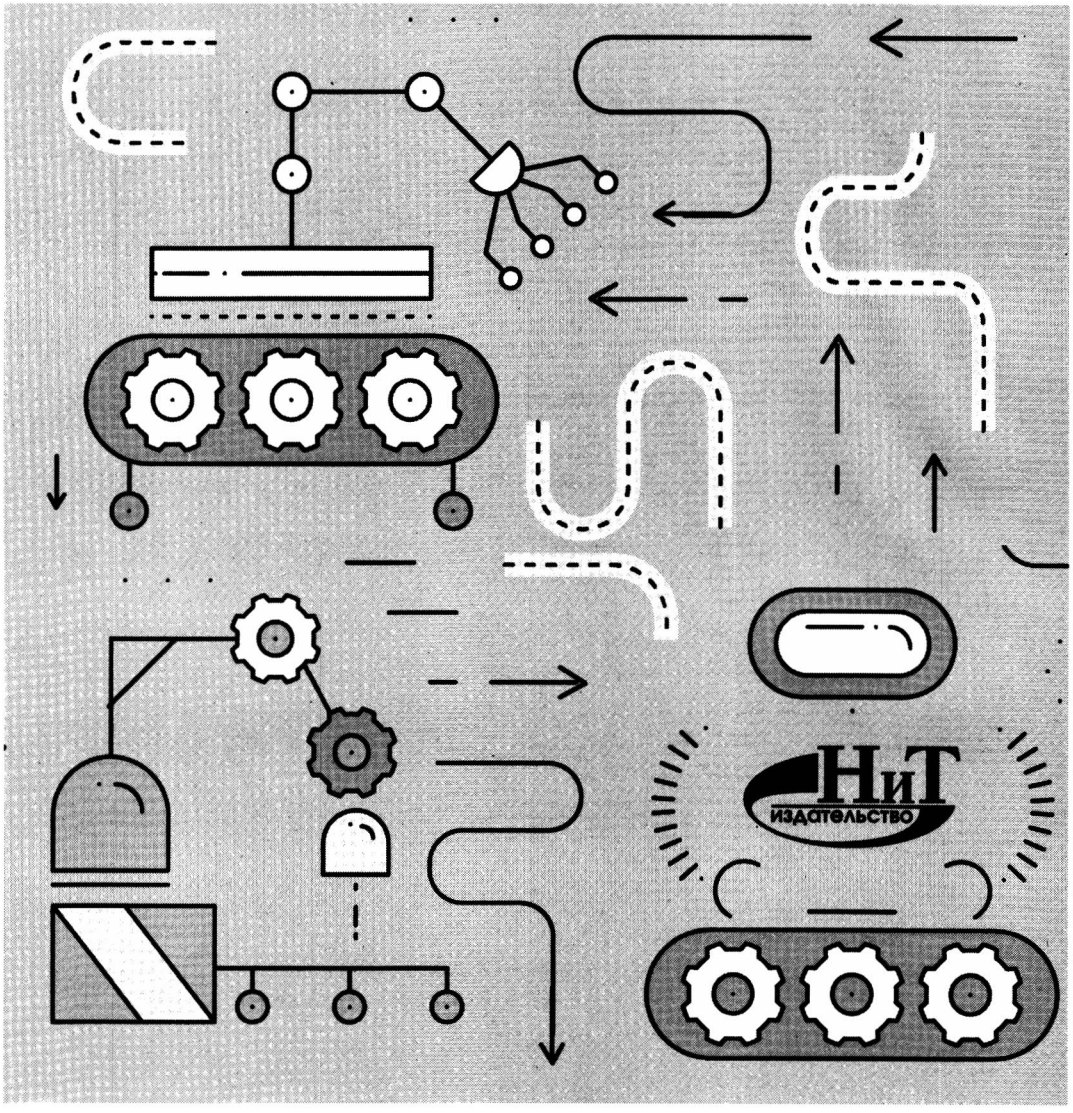
ПРАКТИКА, ПРАКТИКА
И ТОЛЬКО ПРАКТИКА



НиТ
издательство

"Наука и Техника"

Санкт-Петербург


НиТ
издательство

Кольцов Д. М.

Си

на примерах

ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА

"Наука и Техника"
Санкт-Петербург

УДК 004.43 ; ББК 32.973

ISBN 978-5-94387-776-6

Кольцов Д. М.

СИ НА ПРИМЕРАХ. ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА —
СПб.: Наука и Техника, 2019. — 288 с., ил.

Серия "На примерах"

Эта книга является превосходным учебным пособием для изучения языка программирования Си на примерах.

В книге рассмотрена базовая теоретическая часть языка Си, позволяющая ориентироваться в языке и создавать свои программы: операторы, логические конструкции, массивы, связанные списки и деревья, очереди и стеки, работа с файлами. Отдельное внимание уделено программированию различных алгоритмов, а также рассмотрению нововведений языка Си на момент 2019 года (стандарты C99, C11, современные практики использования, многопоточность). В книге используется большое количество примеров с подробным анализом кода.

Будет полезна как начинающим программистам, студентам, так и всем, кто хочет быстро начать программировать на Си.

Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

Содержание

Введение

Язык Си изначально разрабатывался программистами и для программистов. Как может язык программирования быть разработан не для программистов? Увы, но далеко не все языки программирования учитывают потребности программистов.

Не будем далеко ходить – всем известный язык Basic, многие изучали его в школе. Он был разработан для пользователей, которые могли бы решать на компьютере какие-либо простые задачи.

Язык COBOL был создан для людей, которые не являются программистами. Основная задача – сделать так, чтобы ничего не понимающие в программировании люди, могли бы понимать написанные на COBOL программы. У разработчиков ничего не вышло, поэтому COBOL особого распространения не получил. Однако он и не умер. Язык появился в 1959 году, а последняя его версия вышла в 2014-ом году, а за это время появилось более 15 версий его диалекта. Впрочем, эта книга не о нем, а я о языке Си.

Язык Си изначально разрабатывался с учетом потребностей именно программистов, каждый новый стандарт этого языка учитывает интересы программистов. В результате программисты получили то, что хотели – универсальный язык, позволяющий разрабатывать самые разные приложения.

В рядах не совсем профессионалов бытует мнение, что язык Си – очень древний и уже мертвый, что-то вроде латыни в обычном мире. А вместо него нужно использовать молодые языки программирования вроде C++ и C#. Мол, ничего серьезного на Си сегодня не напишешь.

Во-первых, это не так. Язык Си не мертв и благодаря стандарту C11 – очень даже перспективный язык. Во-вторых, какой бы язык вы ни выбрали – Си, C++ или C#, вам все равно нужно знать синтаксис базового Си. Данная книга знакомит вас с синтаксисом языка, предоставляет множество полезных примеров. А пока, во введении, мы рассмотрим историю языка Си и имеющиеся стандарты (спецификации) языка.

История языка Си

Язык Си был разработан программистом Деннисом Ритчи (Dennis Ritchie) в начале 70-ых годов. Тогда язык разрабатывался для компьютера DEC PDP-11, работающего под управлением UNIX. Впрочем, язык Си разрабатывался не с нуля, а на базе языка B (вот это и есть действительно мертвый язык – что-то вроде латыни в мире программирования), который в свою очередь был разработан ранее Кеном Томпсоном

Многие годы язык Си считался языком программирования только для UNIX-систем. Впрочем, других особо тогда и не было. Если вы помните, первые персональные ПК в виде IBM PC появились значительно позже – в 1981 году и основным языком программирования для них был Basic. А уже с 1983 года Американский институт национальных стандартов (ANSI) начал работу над стандартом языка Си. Работа над стандартом длилась долго – целых 6 лет, в результате появился стандарт ANSI C.

В 1989 году к стандарту ANSI C была добавлена первая поправка – несколько новых библиотечных функций, и кое-что еще. Версия языка ANSI C с первой поправкой стала называться C89 – это уже новый стандарт.

Кстати, разработка языка C++ началась в 1989 году, так что этот язык тоже немолодой. Хотя и не такой "древний", как Си, перекочевавший с PDP-11. Не смотря на то, что в 1998 году появился язык C++, работу над Си никто не прекращал. В 1995-ом году вышло приложение 1 (сокращенно оно назы-

вается NA1), в котором описывались дополнения к языку Си. В частности, были добавлены заголовочные файлы <iso646.h>, <wchar.h> и <wctype.h>.

В 1999-ом году был принят новый стандарт языка Си – C99. Версия C99 сохранила все свойства стандарта C89, не изменив основных аспектов языка. Таким образом, язык Си, описанный стандартом C99, практически совпадает с языком, соответствующим стандарту C89. Стандарт C99 сосредоточился на некоторых специфических и довольно сложных свойствах – массивах с переменной длиной, квалификатор указателей restrict. Также в состав языка были включены математические библиотеки. Поскольку разработка C++ завершилась до разработки стандарта C99, то ни одно из новшеств C99 не вошло в состав C++ - вот вам и передовой язык. Он еще в 1999-ом году отставал от Си по функционалу.

Рассмотрим некоторые особенности стандарта C99:

- Объявление локальных переменных в любом операторе программного текста (как в C++)
- Подставляемые inline-функции
- Массивы переменной длины
- Поддержка ограниченных указателей restrict
- Именованная инициализация структур
- Поддержка однострочных комментариев (//), как в C++
- Несколько новых типов данных (long long int, complex и т.д.), библиотечных функций и заголовочных файлов.

Многие, кто поспешил перейти на язык C++, даже не знают о стандарте C11 (он же C1X), появившемся в 2011 году (8 декабря, так что почти 2012 год). К основным особенностям этого стандарта относятся:

- Поддержка многопоточности.
- Поддержка Юникода (теперь не должно быть проблем с символами национальных алфавитов).
- Анонимные структуры и объединения.
- Обобщенные макросы.

- Управление выравниванием объектов (спецификатор `_Alignas`, оператор `alignof`, функция `aligned_alloc` – в заголовочном файле stdalign.h).
- Удаление опасной функции `gets` (вместо нее нужно использовать `gets_s`)
- Новая функция `quick_exit`
- Спецификатор функции `_Noreturn`.
- Новый режим эксклюзивного открытия файла.
- Статичные утверждения.

Это далеко не все нововведения стандарта C11, а только самые значимые.

Рабочая группа WG14

Скептики спросят: "И на этом все?". Действительно, после 2011 года не выходили какие-либо новые стандарты языка Си, но это не означает, что о нем забыли. Перерыв между стандартами C99 и C11 составил 12 лет, так что время у разработчиков еще есть. Для работы над новым стандартом сформирована рабочая группа WG14. Наработки этой группы можно просмотреть по адресу:

http://www.open-std.org/jtc1/sc22/wg14/

В данный момент собирается информация о необходимом функционале, который войдет в состав нового стандарта. В группе есть наработки с 2013 до 2018 год включительно. Так что новый стандарт языка Си – не за горами.

Сам же стандарт C11 пересматривается каждые 5 лет. На данный момент последней ревизией является ISO/IEC 9899:2018, приводим соответствующие ссылки:

https://www.iso.org/standard/57853.html

https://www.iso.org/standard/74528.html

Как читать эту книгу?

Материал в этой книге построен по принципу учебника – от простого к сложному. Поэтому начинайте чтение книги, как говорят, "от корки до корки". Если вы пропустите какую-то главу, может получиться, что материал следующей главы будет вам непонятен.

Структура программы на Си

Данное введение будет нетривиальным. Хотя бы потому, что в нем мы объясним структуру программы на Си. Структура типичной программы выглядит так:

```
<подключение заголовочных файлов>

<объявление глобальных переменных>

// точка входа
int main()
{
  // код программы
}
```

Заголовочные файлы содержат различные библиотечные функции. Вряд ли у вас получится создать какую-то полезную программу без подключения заголовочных файлов. Подключаются они с помощью инструкции `include`. Как минимум, нужно подключить заголовочных файл stdio.h, содержащий средства для работы со стандартным вводом/выводом:

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

Глобальные переменные можно не объявлять – это необязательно.

Точка входа – это функция, с которой начинается выполнение программы. В языке Си такая функция называется `main`.

Рассмотрим самую простую программу, которая запрашивает у пользователя ввести число и выводит введенное число на экран:

```
#include <stdio.h>
int main()
{
    int k;

    // printf() отображает приглашение
    printf("Введите число: ");

    // scanf() читает форматированный ввод пользователя и
    // записывает его в переменную
    scanf("%d", &k);

    // printf() отображает отформатированный вывод
    printf("Вы ввели: %d\n", k);
    return 0;
}
```

После того, как программа написана, ее нужно откомпилировать, чтобы получить исполнимый файл. Для компиляции С-программы мы будем использовать компилятор `gcc` (GNU C Compiler). Компилятору нужно указать имя С-файла программы и (желательно) имя результирующего файла:

```
gcc program.c -o program
```

После этого нашу программу нужно запустить:

```
./program
```

Разберемся, как работает программа:

- Первая строка `#include <stdio.h>` - это команда препроцессора. Она говорит компилятору добавить содержимое заголовочного файла stdio.h (название расшифровывается как Standard Input and Output -

стандартный ввод/вывод) в нашу программу. Файл `stdio.h` содержит функции вроде `scanf()` и `printf()` для работы с вводом и выводом соответственно.

- Выполнение программы на Си начинается с функции `main()` - это точка входа в программу, как уже было отмечено.
- Функция `prinf()` - это библиотечная функция, отправляющая отформатированный вывод на экран. В нашем случае данная функция выводит строку "Введите число" (без кавычек) на экран.
- Далее функцией `scanf()` мы читаем ввод пользователя.
- Далее мы функцией `printf()` выводим то, что ввел пользователь. Символ \n обеспечивает перевод строки, иначе вывод командного интерпретатора (приглашения) будет начинаться сразу после нашей строки, а это выглядит очень некрасиво.
- Выражение `return 0;` означает код возврата (exit code) программы. С помощью кода возврата вы можете дать другим программа, запустившим вашу программу, как прошло ее выполнение. Код возврата 0 обычно соответствует отсутствию ошибок. Любое другое значение означает код ошибки.

Библиотечные заголовочные файлы Си

Таблица B.1 содержит список библиотечных заголовочных файлов ANSI C с дополнениями C99 и C11.

Таблица B.1. Заголовочные файлы Си

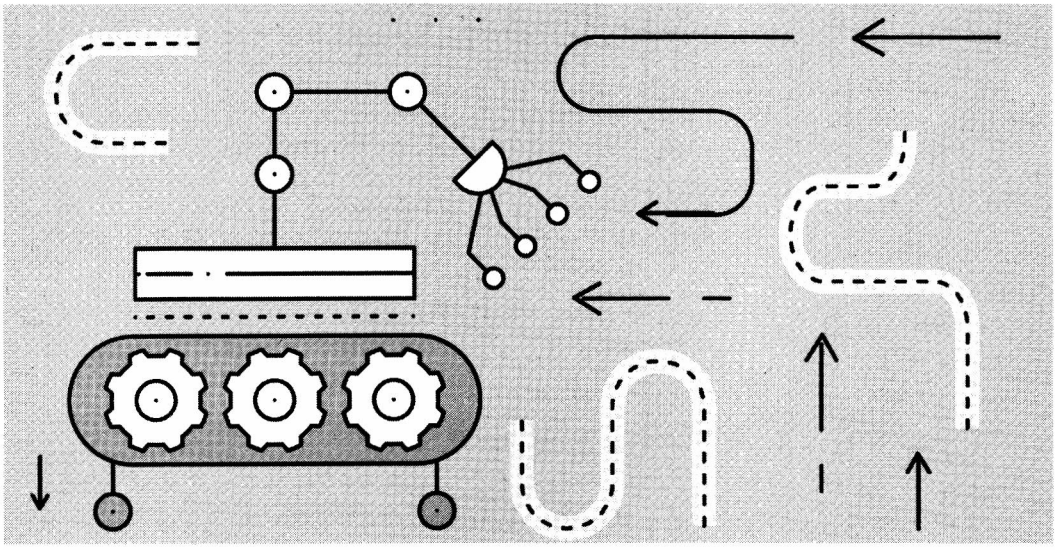
Файл	Назначение
`assert.h`	Содержит макрос утверждений, используемый для обнаружения логических и некоторых других типов ошибок в отлаживаемой версии программы.
`complex.h`	Набор функций для работы с комплексными числами (C99)

Файл	Назначение
ctype.h	Содержит функции для классификации символов по их типам или для конвертации между верхним и нижним регистрами независимо от используемой кодировки (обычно ASCII или одно из ее расширений, хотя есть и реализации, использующие EBCDIC).
errno.h	Для проверки кодов ошибок, возвращаемых библиотечными функциями.
fenv.h	Для управления средой, использующей числа с плавающей точкой (C99)
float.h	Содержит заранее определенные константы, описывающие специфику реализации свойств библиотеки для работы с числами с плавающей точкой, как, например, минимальная разница между двумя различными числами с плавающей точкой (_EPSILON), максимальное число цифр точности (_DIG) и область допустимых чисел (_MIN, _MAX).
inttypes.h	Для точной конвертации целых типов (C99)
iso646.h	Для программирования в кодировке ISO 646. (Появилось в NA1)
limits.h	Содержит заранее заданные константы, определяющие специфику реализации свойств целых типов, как, например, область допустимых значений (_MIN, _MAX).
locale.h	Для setlocale() и связанных констант. Используется для выбора соответствующего языка.
math.h	Для вычисления основных математических функций. Для компиляции требует опции -lm
setjmp.h	Объявляет макросы setjmp и longjmp, используемые для нелокальных переходов
signal.h	Для управления обработкой сигналов
stdarg.h	Для доступа к различному числу аргументов, переданных функциям.
stdbool.h	Для булевых типов данных (C99)
stdint.h	Для определения различных типов целых чисел (C99)

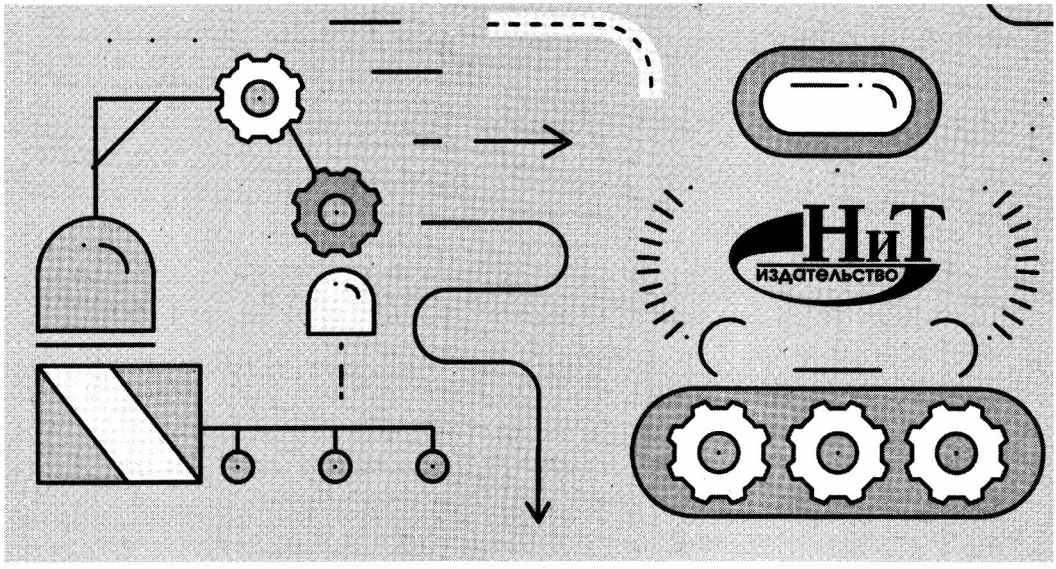
Файл	Назначение
stddef.h	Для определения нескольких стандартных типов и макросов.
stdio.h	Реализует основные возможности ввода и вывода в языке Си. Этот файл содержит весьма важную функцию printf().
stdlib.h	Для выполнения множества операций, включая конвертацию, генерацию псевдослучайных чисел, выделение памяти, контроль процессов, окружения, сигналов, поиска и сортировки.
string.h	Для работы с различными видами строк.
tgmath.h	Для типовых математических функций (C99)
threads.h	Заголовочный файл <threads.h> наряду с <stdatomic.h> предоставляет поддержку для параллельного программирования (C11)
time.h	Для конвертации между различными форматами времени и даты.
wchar.h	Для обработки "широких" потоков и нескольких видов строк при помощи "широких" символов (поддержка набора языков) (NA1)
wctype.h	Для классификации "широких" символов (NA1)

Онлайн-компиляторы

Пока вы учитесь, вы можете вообще не устанавливать на свой компьютер компилятор Си. Для выполнения большинства несложных примеров (например, чтобы попробует, как работает тот или иной цикл), достаточно использовать любой из онлайн-компиляторов, например, https://www.onlinegdb.com/online_c_compiler. Для работы с таким компилятором вам нужен только браузер – вводите код программы, нажимаете кнопку Run и получаете результат выполнения программы или же ошибки компилятора.



Часть I. Основные конструкции языка Си



В этой книге не будет банальщины вроде программы "Привет, мир!". Начнем книгу сразу с выражений – эдаких кирпичиков, из которых состоит любая мало-мальски полезная (в отличие от "Привет, мир!") программа.

Глава 1.
Выражения в Си

Выражения (expressions) – самые важные элементы языка Си. Выражения состоят из атомарных элементов – данных и операторов. Данные можно представить либо в виде переменных, либо в виде констант. Операторы позволяют выполнить некоторые операции над данными. Какие операции будут выполнены, зависит от типа данных. Например, оператор + может выполнить сложение чисел, а может использоваться для конкатенации строк. Когда вы пытаетесь вычислить результат выражения 2 + 3, то получите 5, а вот результатом выражения "2" + "3" будет "23", то есть объединение двух строк.

1.1. Основные типы данных

В языке Си существует пять элементарных типов данных: символ (`char`), целое число (`int`), число с плавающей точкой (`float`), число с плавающей точкой удвоенной точности (`double`) и переменная, не имеющая значений (`void`).

Остальные типы данных в языке Си создаются на основе этих элементарных типов. Размер переменных и диапазон их значений зависит от типа процессора и компилятора. Однако во всех случаях размер символа равен 1 байт. Размер целочисленной переменной обычно равен длине машинного слова, принятой в конкретной операционной системе.

Реальное представление чисел с плавающей точкой зависит от их конкретной реализации. Размер целого числа обычно равен длине машинного слова, принятой в операционной системе. Значения переменных типа `char`, как правило, используются для представления символов, предусмотренных в системе кодирования ASCII. Значения, выходящие за пределы допустимого диапазона, на разных компьютерах обрабатываются по-разному.

Диапазон изменения переменных типа `float` и `double` зависит от способа представлениячисел с плавающей точкой. Нужно отметить, что данный диапазон очень широк и его хватит для большинства поставленных задач (особенно диапазона типа `double`).

Тип `void` используется для определения функции, не возвращающей никаких значений, либо для создания обобщенного указателя (generic pointer).

Таблица 1.1 содержит общую информацию об элементарных типах данных, определенных в стандарте ANSI C. Повторюсь, это базовые типы данных, а остальные типы на данный момент нам пока не нужны.

Таблица 1.1. Типы данных, определенные в стандарте ANSI C

Тип	Обычный размер, бит	Минимальный диапазон
int	16 или 32	-32768..32767

unsigned int	16 или 32	0..65535
signed int	16 или 32	-32768..32767
short int	16	-32768..32767
unsigned short int	16	0..65535
signed short int	16	-32768..32767
long int	32	-2147483648..2147483647
signed long int	32	-2147483648..2147483647
unsigned long int	32	0..4294967295
float	32	6 значащих цифр
double	64	10 значащих цифр
long double	80	10 значащих цифр
char	8	-128..127
unsigned char	8	0..255
signed char	8	-128..127

Как вы уже заметили, основные типы данных (кроме `void`) могут иметь различные модификаторы типа: `signed`, `unsigned`, `long`, `short`.

Целочисленные типы можно модифицировать с помощью ключевых слов `signed`, `unsigned`, `short`, `long`. Символьные типы можно уточнять с помощью модификаторов `unsigned` и `signed`. Тип `double` можно модифицировать ключевым словом `long`.

Нужно отметить, что применение модификатора `signed` не имеет смысла, поскольку все целые числа по умолчанию имеют знак. А сам модификатор существует только потому, что есть `unsigned`, значит должен быть его антипод. Также в некоторых вариантах языка Си по умолчанию тип `char` не имеет знака, поэтому есть смысл в использовании `signed`.

Разница между целыми числами, имеющими и не имеющими знак, заключается в интерпретации бита в старшем разряде. Если в программе определено целое число со знаком, то компилятор генерирует код, в котором старший бит интерпретируется как признак знака (sign flag). Если признак знака равен 0, число считается положительным, если он равен 1 — отрицательным.

1.2. Идентификаторы

Имена переменных, меток, функций и других определенных пользователем объектов называются идентификаторами. Идентификатор может состоять из одного или более символов. Первый символ идентификатора должен быть буквой или символом подчеркивания (но в этом случае сам идентификатор должен состоять из более чем одного символа). Следующие символы могут быть буквами, цифрами и знаками подчеркивания.

Рассмотрим имена правильных идентификаторов:

- `a`
- `Counter`
- `test_car`

А вот примеры неправильных имен:

- `1c` (начинается с цифры)
- `hello!word` (содержит недопустимый символ)

Длина идентификатора в языке Си – произвольная. Однако будьте разумными и создавайте читабельные идентификаторы, чтобы можно было сразу понять, для чего используется та или иная переменная или функция.

Не все символы в имени идентификатора являются значащими (именно поэтому нет смысла создавать очень длинные идентификаторы). Если идентификатор используется в процессе редактирования внешних связей, то значащими считаются, по крайней мере, шесть его первых символов. Эти идентификаторы называются *внешними именами* (external names). К ним относятся имена функций и глобальных переменных, используемых несколькими файлами. Если идентификатор не используется в процессе редактирования внешних связей, то значащими считаются, по крайней мере, 31 первый символ. Такие идентификаторы называются *внутренними именами* (internal names).

Одно из отличий языка C++ - там значащими являются первые 1024 символов и это нужно учитывать, если вы переходите с одного языка на другой.

Имена идентификаторов являются чувствительными к регистру, поэтому `counter` и `Counter` – это разные идентификаторы.

1.3. Переменные

Переменная – это поименованная область памяти, которую можно использовать для хранения модифицируемого значения. Доступ к хранимому в переменной значению осуществляется через идентификатор – имя переменной.

В языке Си все переменные должны быть объявлены до первого использования. Общий синтаксис объявления переменной выглядит так:

```
тип список_переменных;
```

Примеры:

```
int count, j, k;
double total;
short int s;
```

Переменные объявляются в трех местах: внутри функций (локальные переменные), в определении параметров функции (формальные параметры) и за пределами всех функций (глобальные переменные).

1.3.1. Локальные переменные

Локальными называют переменные, объявленные внутри функций. Локальные переменные на то и локальные, что их можно использовать только внутри функции, в которых они объявлены. Они не видимы за пределами функций. Рассмотрим примеры:

```
int f1(void)
{
  int z;
  z = 1;
  return z;
}

int f2(void)
{
  int z;
  z = 2;
  return z;
}
```

Посмотрим на наш фрагмент кода. Переменная `z` объявлена дважды. Один раз в функции `f1()`, второй раз – в функции `f2()`. При этом переменная `z` из функции f1() не имеет никакого отношения к переменной `z` из функции `f2()`. Если функция `f1()` изменит переменную `z`, то это никак не повлияет на переменную `z`, объявленную в функции `f2()` и наоборот.

> **Примечание.** Язык Си содержит ключевое слово auto, которое можно использовать для объявления локальных переменных. Однако, поскольку все локальные переменные по умолчанию считаются автоматическими, это ключевое слово на практике никогда не используется. По этой причине мы также не будем его применять. Данное ключевое слово используется для обеспечения обратной совместимости со старыми версиями Си.

По традиции и для собственного удобства большинство программистов объявляют функции в самом начале функции (сразу после открывающейся фигуркой скобки), как было показано ранее. Однако переменную можно объявить в любом месте кода функции, но до первого использования переменной:

```
int f3(void)
{
   int randomNum;
```

```
    randomNum = rand() % 10;
    if (randomNum < 6) {
      int secondRandomNum;
      secondRandomNum = rand() % 10;
      return secondRandomNum;
    }
    return randomNum;
}
```

Наша функция сначала генерирует случайное число `randomNum`: если сгенерированное число меньше 6, то мы генерируем еще раз и возвращаем второе сгенерированное число. Сейчас нет задачи вернуть случайное число больше 6, сейчас задача продемонстрировать, что переменная `secondRandomNum` объявлена не в начале функции.

Переменная `secondRandomNum` будет доступна не по всей функции, а только внутри блока, в котором она объявлена. Позже мы еще поговорим об областях видимости функции.

Нельзя не сказать о стандарте C89, где есть существенные ограничения. В C89 переменная должна быть объявлена в начале блока кода, то есть сразу после открывающейся фигуркой скобки:

```
void f(void)
{
 int k;
 k = 0;
 int m;
}
```

Объявление переменной `m` сгенерирует ошибку, поскольку до этого уже выполнялся оператор действия (в данном случае оператор присваивания). Однако данный момент исправлен в стандарте C99 (соответственно и в C11), поэтому в более новых версиях Си такое объявление переменной не станет причиной ошибки.

Примечание. Поскольку локальная переменная создается при входе в блок, где она объявлена, и разрушается при выходе из него, ее значение теряется. Данный факт особенно важно учитывать при вызове функции.

Локальные переменные функции создаются при ее вызове и уничтожаются при возврате управления в вызывающий модуль. Это значит, что между двумя вызовами функции значения ее локальных переменных не сохраняются. Заставить их сохранять свои значения можно с помощью модификатора `static`, о котором мы еще поговорим.

Локальные переменные хранятся в стеке. Поскольку стек является динамической структурой, и объем занимаемой им памяти постоянно изменяется, становится понятным, почему локальные переменные в принципе не могут сохранять свои значения между двумя вызовами функции, внутри которой они объявлены.

При объявлении переменной ее сразу можно инициализировать, то есть присвоить какое-то начальное значение:

```
void f(void)
{
  int j = 5;
  printf("%d", j);
}
```

1.3.2. Формальные параметры

Передать аргументы функции можно с помощью так называемых формальных параметров. Внутри функции формальные параметры можно использовать так же, как и локальные переменные, то есть присваивать значения, использовать в выражениях и т.д.

Объявляются формальные параметры в круглых скобках после названия функции:

```
int summa(int a, int b) {
  int z;
  z = a + b;
  return z;
}
```

Тип формальных параметров задается при их объявлении. После этого их можно использовать как обычные локальные переменные. Как и локальные переменные, формальные параметры являются динамическими и разрушаются при выходе из функции.

Формальные параметры можно использовать в любых конструкциях и выражениях.

Несмотря на то, что свои значения они получают от аргументов, передаваемых извне функции, во всем остальном они ничем не отличаются от локальных переменных.

1.3.3. Глобальные переменные

Глобальные переменные на то и являются глобальными, что доступны из любой части программы и могут быть использованы, где угодно. Рассмотрим объявление глобальной переменной `counter`, которая доступна для всех функций программы, поскольку объявлена вне всех этих функций:

```
#include <stdio.h>

int counter = 0;

void f1(void);
void f2(void);

int main(void)
{
  counter = 100;
  f1();
  f2();
  return 0;
}

void f1(void)
{
  int z;
```

```
  z = counter;
  printf("counter = %d\n", counter);
}

void f2(void)
{
  int z;
  z = counter;
  printf("counter = %d\n", counter);
}
```

Данная программа выведет два раза число 100. Первый раз его выведет функция `f1()`, во второй раз – `f2()`.

Посмотрите внимательно на нашу программу. Хотя ни функция `main()`, ни функции `f1()` и `f2()` не содержат объявления переменной `counter`, все они ее успешно используют. Это потому, что переменная `counter` является глобальной. Мы можем объявить локальную переменную `counter` в любой из функций:

```
#include <stdio.h>

int counter = 0;

void f1(void);
void f2(void);

int main(void)
{
  counter = 100;
  f1();
  f2();
  return 0;
}

void f1(void)
{
  int z;
  z = counter;
  printf("counter = %d\n", counter);
```

```
}

void f2(void)
{
  int counter = 200;
  printf("counter = %d\n", counter);
}
```

В данном случае в функции `f2()` объявлена локальная переменная counter. Наша программа отобразит числа 100 и 200.

Если глобальная и локальная переменные имеют одинаковые имена, то все ссылки на имя переменной внутри блока будут относиться к локальной переменной и не влиять на ее глобальную тезку. Возможно, это удобно, однако об этой особенности легко забыть, и тогда действия программы могут стать непредсказуемыми.

Глобальные переменные хранятся в специально отведенном для них месте. Они весьма полезны, если разные функции в программе используют одни и те же данные. Однако, если в них нет особой необходимости, глобальных переменных следует избегать. Дело в том, что они занимают память во время всего выполнения программы, даже когда уже не нужны.

1.3.4. Области видимости

Язык Си предусматривает четыре области видимости переменных (а не только локальные и глобальные переменные):

- **Файл**. Идентификатор объявляется вне каких-либо блоков и классов. Объявленные таким образом переменные считаются глобальными.
- **Блок**. Идентификатор, объявленный после {. Область видимости начинается с момента объявления идентификатора и до закрывающейся скобки }.
- **Прототип функции**. Идентификаторы, объявленные в прототипе функции, видимы только внутри этого прототипа.
- **Функция**. Переменные, объявленные в функции, доступны только в этой функции. Такие переменные называются *локальными*.

1.3.5. Квалификаторы типа

К квалификаторам типа относят `const`, `volatile` и `restrict`. Квалификатор типа указывает на доступность и модифицируемость переменной.

Переменные с квалификатором `const` не могут быть изменены, но они могут быть инициализированы, то есть можно установить значение такой переменной при ее объявлении:

```
const port = 4850;
```

Это объявление создает целочисленную переменную с именем `port` и начальным значением 4850, которое в остальной части программы изменить невозможно. Однако переменную `port` можно использовать в других выражениях. Переменная типа `const` может получить свое значение либо во время инициализации, либо с помощью других машинно-зависимых средств.

Квалификатор `const` можно использовать для защиты объектов, передаваемых в функцию в качестве аргументов. Иными словами, если в функцию передается указатель, она может модифицировать фактическое значение, на которое он ссылается. Но если указатель уточнить квалификатором `const`, функция не сможет изменить значение в соответствующей ячейке.

Многие функции из стандартной библиотеки используют квалификатор `const` в объявлениях своих параметров. Например, прототип функции `strlen()` выглядит следующим образом:

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

Делается это для того, чтобы внутри функции невозможно было изменить переданный ей аргумент.

Квалификатор `volatile` сообщает компилятору, что значение переменной может изменяться неявно. Большинство компиляторов языка Си автоматически оптимизируют некоторые выражения, считая, что значения переменных не изменяются, если они не указаны в левой части оператора присваивания.

Таким образом, их значения нет смысла перепроверять при каждом обращении. Кроме того, некоторые компиляторы изменяют порядок вычисления выражений в ходе компиляции.

Стандарт C99 дополнительно вводит новый квалификатор типа `restrict`, применимый только для указателей. Если указатель объявлен с квалификатором `restrict`, то к объекту, на который он ссылается, можно обратиться только с помощью этого указателя. Обращение к объекту с помощью другого указателя возможно только в том случае, если другой указатель основан на первом. Таким образом, доступ к объекту можно получить только с помощью выражений, основанных на указателе с квалификатором `restrict`. Указатели `restrict` используются главным образом как параметры функции или совместно с `malloc()`. Если указатель объявлен с квалификатором `restrict`, компилятор способен лучше оптимизировать некоторые процедуры. Используется довольно редко, начинающим программистам мало чем полезен.

1.3.6. Спецификаторы хранения

В языке Си есть еще и спецификаторы хранения, указывающие где должна храниться переменная. Спецификатор `extern` указывает, что идентификатор хранится в другом файле. Общая практика такова: в одном файле вы объявляете все глобальные переменные, а во всех остальных – используете их с помощью спецификатора `extern`.

Рассмотрим пример первого файла:

```
int port;
char data;
```

Второй файл:

```
extern int port;
extern char data;
```

Спецификатор `static` позволяет сделать переменную статической. Статические переменные, в отличие от глобальных, неизвестны вне своей функции или файла, и сохраняют свои значения между вызовами. Это очень полезно при создании обобщенных функций и библиотек функций, которые могут использоваться другими программистами. Спецификатор `static` можно использовать, как для локальных, так и для глобальных переменных:

```
static int port;
```

Если локальная переменная объявлена как статическая, компилятор выделит для нее постоянное место хранения, как и для глобальной переменной. **Принципиальное отличие локальной статической переменной от глобальной состоит в том, что первая остается доступной лишь внутри своего блока.** Проще говоря, локальная статическая переменная — это локальная переменная, сохраняющая свои значения между вызовами функции.

Рассмотрим пример:

```
int number_series(void)
{
  static int num = 0;
  num = num + 5;
  return num;
}
```

Данная функция формирует числовой ряд 5, 10, 15 и т.д. При каждом вызове функции возвращается следующее число числового ряда. Переменная num инициализируется один только раз – при первом вызове функции `number_series()`, а не при каждом запуске функции, как это происходит с локальными переменными.

Спецификатор `register` изначально применялся лишь к переменным типа `int`, `char` или указателям. Однако впоследствии его толкование было расширено, и теперь его можно применять к переменным любого типа.

Спецификатор `register` был разработан для того, чтобы компилятор сохранял значение переменной в регистре центрального процессора, а не

памяти, как обычно. Это означает, что операции над регистровыми переменными выполняются намного быстрее.

Данный спецификатор может применяться только с локальными переменными и глобальными параметрами. К глобальным переменным он не применяется.

1.3.7. Инициализация переменных. Оператор присваивания

При объявлении переменной ее можно инициализировать, то есть присвоить начальное значение. Общий синтаксис выглядит так:

```
тип имя_переменной = значение;
```

Рассмотрим примеры:

```
int counter = 0;
float total = 0;
```

Далее вы можете изменять значение переменной с помощью оператора присваивания (=). Переменной можно присвоить, как константу, так и результат вычисления выражения:

```
counter = 1;   // присваиваем константу
counter = counter + 1;   // результат вычисления выражения
```

1.4. Константы

Константы – это фиксированные значения, которые программа не может изменить. Способ объявления константы зависит от типа значения. Символьные константы заключаются в одинарные кавычки. Например, символы 'D' и '?' являются константами.

```
int c = 0;
char ch = 'A';
```

В первой строчке переменной с присваивается константа 0, во втором – символьная константа 'A'.

> **Внимание!** Особое внимание обратите на используемые кавычки. Если кавычки одинарные 'A' – перед нами один символ (`char`). Если кавычки двойные ("A"), перед нами – строка, состоящая из одного символа!

Целочисленные константы считаются числами, не имеющими дробной части. Например, числа 1 и 10 являются целочисленными константами. Константы с плавающей точкой содержат дробную часть, которая отделяется десятичной точкой. Например, число 1.234 представляет собой константу с плавающей точкой.

В некоторых случаях удобно использовать восьмеричную и шестнадцатеричную системы счисления. Перед шестнадцатеричным числом ставится префикс 0x, перед восьмеричной – 0:

```
int hex_value = 0xff;
into ct_value = 011;
```

Особого внимания заслуживают управляющие символьные константы. Они также называются Escape-последовательностями (табл. 1.2).

Таблица 1.2. Часто используемые управляющие Escape-последовательности

Константа	Значение
`\n`	Новая строка
`\r`	Перевод каретки (актуально в Windows)
`\t`	Горизонтальная табуляция
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\0`	Нуль

\\	Обратный слэш
\a	Звуковой сигнал
\N	Восьмеричная константа, N – восьмеричное число
\xN	Шестнадцатеричная константа, N – шестнадцатеричное

Пример:

```
void beep(void)
{
  printf("\a");
}
```

Функция выведет на стандартный вывод символ \a, что соответствует звуковому сигналу. Издаст современный компьютер этот звуковой сигнал или нет, зависит от самого компьютера. Ранее все PC-совместимые компьютеры при выводе символа \a издавали гудок через системный динамик. Сейчас некоторые компьютеры вообще не оснащаются таким динамиком. На некоторых системный звук воспроизводится через звуковую плату. В общем, можете поэкспериментировать прямо сейчас и проверить, будет ли издавать гудок ваш компьютер или нет.

1.5. Операторы

В языке Си очень много самых разных операторов – логические операторы, арифметические операторы, операторы сравнения, побитовые. Также есть оператор присваивания, с которым мы уже знакомы.

1.5.1. Оператор присваивания

Несмотря на то, что о нем мы уже говорили, рассмотрим несколько важных моментов. Общий синтаксис оператора присваивания выглядит так:

```
имя_переменной = выражение;
```

Здесь выражение может быть, как константой, так и результатом вычисления выражения. При этом выражение может содержать вызовы функций, имена переменных, константы и другие операторы. Например:

```
double total = 0;
int s = 100;
total = get_sum() * 0.82 + s;
```

Здесь для вычисления значения переменной `total` используется и вызов функции `get_sum()` и константа 0.82 и переменная `s`.

Выражение может содержать операнды разных типов. Не всегда типы операндов совпадают с типом переменной. Самый тривиальный пример:

```
int a = 2, b = 2;
float c;

c = a * b;
```

Здесь тип операндов – `int`, а тип результирующей переменной – `float`. В этом случае компилятор будет использовать преобразование типов. Не всегда типы можно преобразовать – в этом случае вы получите ошибку компилятора. В случаях, когда преобразование возможно, в силу вступают правила преобразования типов, описанные в таблице 1.3.

Таблица 1.3. Правила преобразования типов

К какому типу приводится значение	**Тип значения**	**Потери**
float	double	Точность, округление результата
double	long double	Точность, округление результата
int	float	Дробная часть
int	long int	Нет

int (16 бит)	long int	Старшие 16 бит
short int	int (32 бит)	Старшие 16 бит
short int	int (16 бит)	Нет
signed char	char	Если значение > 125, результат – отрицательное число
char	short int	Старшие 8 бит
char	int (16 бит)	Старшие 8 бит
char	int (32 бит)	Старшие 24 бит

В языке Си также возможны множественные присваивания, которые удобно использовать для инициализации нескольких переменных:

```
a = b = c = 0;
```

1.5.2. Арифметические операторы

Арифметические операторы в языке Си такие же, как и в большинстве других языков программирования:

+ сложение;

- вычитание и унарный минус;

* умножение;

/ деление;

% деление по модулю;

-- декремент;

++ инкремент.

Оператор деления при применения к числам с плавающей запятой. Если его операнды – целые числа, то дробная часть отбрасывается (7 / 2 = 3). Оператор % возвращает остаток целочисленного деления.

Пример:

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

printf("%d ", a/b);
printf("%d ",a%b);
```

Программа выведет числа 2 и 1. Первое число – результат целочисленного деления, второе – остаток от деления.

Напишем программу, вычисляющую частное и остаток при делении двух чисел. Для тех, кто в школе пропустил урок математики, давайте разберемся, что есть что:

- Делимое - это число, стоящее слева от знака деления.
- Делитель - это число, стоящее справа от знака деления, по сути, это число, на которое делим.
- Частное - это число, стоящее после знака равно, результат деления.
- Остаток - это число, оставшееся не делимым, которое меньше делителя.

Листинг 1.1. Вычисляем частное и остаток от деления

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

    int dividend, divisor, quotient, remainder;

    printf("Введите делимое: ");
    scanf("%d", &dividend);

    printf("Введите делитель: ");
    scanf("%d", &divisor);

    // Вычисляем частное
    quotient = dividend / divisor;

    // Вычисляем остаток
```

```
    remainder = dividend % divisor;

    printf("Частное = %d\n", quotient);
    printf("Остаток = %d\n", remainder);

    return 0;
}
```

Работает программа так:

- Мы вводим делимое и делитель - `dividend` и `divisor` соответственно.
- Используя оператор `/`, мы вычисляем частное (выражаясь математическим языком - неполное частное во многих случаях)
- Используя оператор `%`, мы вычисляем остаток от деления.
- Наконец, вы выводим частное и остаток с помощью функции `printf()`.

Особое место занимают операции инкремента и декремента. Для этого в Си есть очень полезные операторы `++` и `--`. Оператор `++` добавляет к своему операнду 1, а `--` – вычитает. Примеры:

`x = x + 1`	`++x`
`x = x - 1`	`--x`

Операторы инкрементации и декрементации имеют две формы: префиксную (`++x`) и постфиксную (`x++`). Между префиксной и постфиксной формами существует важное отличие, когда они используются внутри выражений. Если используется префиксная форма, операторы инкрементации и декрементации применяются к старому значению операнда, если постфиксная — к новому.

Рассмотрим примеры:

```
x = 5;
y = ++x;
z = x++;
```

Переменной y будет присвоено значение 6, а переменной z – значение 5.

1.5.3. Операторы сравнения и логические операторы

В основе и операторов сравнения, и логических операторов лежат понятия "истина" (true) и "ложь" (false). В языке Си истинным считается любое значение, не равное нулю. Ложное значение всегда равно 0. Выражения, использующие операторы сравнения и логические операторы, возвращают 0, если результат ложен, и 1, если результат истинен.

К операторам сравнения относят следующие операторы:

> больше;

>= больше или равно;

< меньше;

<= меньше или равно;

!= не равно.

Логические операторы:

&& - И;

|| - ИЛИ;

! – НЕ.

Таблица истинности логических операторов приведена ниже.

Таблица 1.4. Таблица истинности логических операторов

x	y	x&&y	x||y	!x
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

В языке Си, в отличие от некоторых других языков программирования, нет оператора XOR (исключительное ИЛИ). При желании его можно реализовать в виде функции, которая возвращает true, если один операндов (но не оба) истинный:

```
int xor(int x, int y)
{
  return (x || y) && !(x && y );
}
```

1.5.4. Побитовые операторы

Язык Си разрабатывался в качестве замены языка ассемблера, поэтому он предусматривает многие операции низкого уровня, в частности, побитовые операции (bitwise operation), предназначенные для проверки, установки и сдвига битов, из которых состоят байты и машинные слова, образующие переменные типа `char` или `int`. Побитовые операции нельзя применять к переменным типа `float`, `double`, `bool` и другим более сложным типам.

Рассмотрим список побитовых операторов:

`&` И

`|` ИЛИ

`^` исключающее ИЛИ (XOR)

`~` дополнение до единицы (НЕ)

`>>` сдвиг вправо

`<<` сдвиг влево

Ранее мы приводили таблицу истинности для логических операторов. Для побитовых операторов `&`, `|` и `~` она такая же. Для оператора `^` (XOR) таблица истинности представлена ниже.

Таблица 1.5. Таблица истинности для оператора ^

x	y	x^y
0	0	0
1	0	1
0	1	1
1	1	0

Операторы побитового сдвига имеют синтаксис:

```
значение >> количество разрядов
значение << количество разрядов
```

Например, возьмем число 7 и его двоичное представление 00000111.

```
x = 7;     // 00000111
x = x<<1;    // 0001110 или 14
x = 7;
x = x>>1;    // 00000011 или 3
```

Как правило, побитовые операторы используются для разработки драйверов и других низкоуровневых программ. Также они часто применяются в шифровальных программах.

1.5.5. Тернарный оператор

В языке Си есть так называемый тернарный оператор, похожий на оператор if-then-else. Тернарный оператор имеет следующий вид:

```
выражение_1 ? выражение_2 : выражение_3;
```

Оператор ? выполняется следующим образом: сначала вычисляется `Выражение_1`. Если оно истинно, вычисляется `Выражение_2`, и его значение становится результатом всего оператора. Если `Выражение_1` ложно,

вычисляется Выражение_3, и его значение становится результатом оператора. Рассмотрим фрагмент программы.

```
x = 1;
y = x>9 ? 500 : 700;
```

Здесь переменной y будет присвоено значение 700, поскольку значение переменной x меньше 9.

Тернарный оператор позволяет сделать код программ компактнее.

1.5.6. Оператор получения адреса (&) и разыменования ссылки (*)

Указатель (pointer value) — это переменная, объявленная особым образом, в которой хранится адрес объекта определенного типа. Для чего они нужны?

С помощью указателей легко индексировать элементы массивов. Также указатели позволяют функциям модифицировать свои параметры. На основе указателей можно создавать связанные списки и другие динамические структуры данных.

Как видите, указатели – штука полезная. Но и очень много ошибок начинающие программисты допускают именно при использовании указателей. Указатели – это такая себе группа риска в языке Си. Из-за непонимания, как работают указатели, можно допустить серьезные ошибки при написании программы. При этом компилятор генерировать ошибку не будет, но программа при этом будет работать неправильно.

Подробно об указателях мы поговорим далее в этой книге, а сейчас лишь рассмотрим, зачем нужны операторы & и *.

Оператор получения адреса & возвращает адрес операнда, например:

```
counter = 1;
addr = &counter;
```

В переменную addr будет помещен адрес переменной count. Адрес – это ничто иное, как адрес ячейки памяти, в которой находится переменная counter. Адрес никак не связан со значением самой переменной. Изменение значения переменной не изменяет ее адрес. В данном случае в переменную addr будет записано значение адреса – то есть, что угодно, но только не 1.

Оператор разыменовывания указателя (*) является противоположностью оператора &. Этот унарный оператор возвращает значение объекта, расположенного по указанному адресу:

```
value = *addr;
```

В данном случае переменная addr содержит **адрес** переменной counter. При разыменовывании адреса в переменную value будет записано **значение** переменной counter, то есть value будет равно 1.

1.5.7. Статический оператор sizeof

Статический унарный оператор sizeof вычисляет длину операнда в байтах. Операндом может быть как отдельная переменная, так и имя типа, заключенное в скобки. Если размер типа double составляет 8 байт, то следующий код выведет две восьмерки:

```
double v = 0;
printf("%d\n", sizeof v);
printf("%d\n", sizeof(double));
```

Следует помнить, что для вычисления размера типа его имя должно быть заключено в скобки. Имя переменной в скобки заключать не обязательно, хотя и ошибки от этого не будет.

Листинг 1.2. Вычисление размера *int*, *float* и *char*

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

```
    int integerType;
    float floatType;
    double doubleType;
    char charType;

    printf("Размер int: %ld байт\n",sizeof(integerType));
    printf("Размер float: %ld байт\n",sizeof(floatType));
    printf("Размер double: %ld байт\n",sizeof(doubleType));
    printf("Размер char: %ld байт\n",sizeof(charType));

    return 0;
}
```

Если вы запустите эту программу, то увидите, что размер типа `int` составляет 4 байта, `float` и `double` - 8 байтов, а `char` - 1 байт.

1.5.8. Оператор последовательного вычисления или запятая

В Си есть так называемый оператор последовательного вычисления, связывающий в единое целое несколько выражений. Символ этого оператора – запятая. При этом подразумевается, что левая часть оператора всегда имеет тип `void`. Выражение, стоящее в правой части оператора, становится значением всего выражения. Рассмотрим пример:

```
z = (x=2, x+1)
```

Сначала мы присваиваем переменной `x` значение 2, а затем прибавляем к ней 1 и результат (3) присваиваем переменной `z`. Скобки здесь нужны, поскольку приоритет оператора последовательного вычисления меньше, чем приоритет оператора присваивания.

По существу, выполнение оператора последовательного вычисления сводится к выполнению нескольких операторов подряд. Если этот оператор стоит в правой части оператора присваивания, то его результатом всегда будет результат выражения, стоящего последним в списке.

1.5.9. Оператор доступа к члену структуры

Структуры – тема для отдельного разговора. Пока вы должны знать, что для доступу к члену структуры используются операторы `.` и `->`.

Пусть у структуры `car` есть член `speed`. Присвоить ему значение можно так:

```
struct car
{
  char name[50];
  int year;
  float speed;
} priora;

priora.speed = 55;
```

или так:

```
struct car *c = &priora; // записываем в с адрес priora
c->speed = 55;
```

1.5.10. Операторы [] и ()

Круглые скобки используются для изменения приоритета операций. Операции, заключенные в круглые скобки, будут выполнены в первую очередь:

```
x = 2 * 2 + 2;    // результат 6
x = 2 * (2 + 2);    // результат 8
```

Квадратные скобки используются для доступа к элементам массива. В квадратные скобки заключается индекс массива, доступ к которому вы хотите получить.

1.5.11. Приоритеты операторов

Таблица 1.6 содержит приоритеты операторов. Все операторы, кроме унарных и тернарного, выполняются слева направо. Операторы `*`, `&` и `~` выполняются справа налево. Операторы перечислены сверху вниз в порядке уменьшения приоритета.

Таблица 1.6. Приоритеты операторов в языке Си

Приоритет	Оператор	Описание	Ассоциативность
1	`++ --`	Суффиксные/постфиксные инкремент и декремент	Слева направо
	`()`	Вызов функции	
	`[]`	Обращение к элементу массива	
	`.`	Обращение к члену структуры или объединения	
	`->`	Обращение к члену структуры или объединения через указатель	
	`(type) {list}`	Составной литерал (C99)	
2	`++ --`	Префиксные инкремент и декремент	Справа налево
	`+ -`	Унарные плюс и минус	
	`! ~`	Логическое НЕ и побитовое НЕ	
	`(type)`	Приведение типа	
	`*`	Разыменование	
	`&`	Взятие адреса	
	`sizeof`	Размер	
	`_Alignof`	Выравнивание (C11)	

3	* / %	Умножение, деление и остаток	Слева направо
4	+ -	Сложение и вычитание	
5	<< >>	Побитовые левый сдвиг и правый сдвиг	
6	< <=	Для операторов сравнения < и ≤ соответственно	
	> >=	Для операторов сравнения > и ≥ соответственно	
7	== !=	Для сравнений = и ≠ соответственно	
8	&	Побитовое И	
9	^	Побитовое XOR (исключающее или)	
10	|	Побитовое ИЛИ (включающее или)	
11	&&	Логическое И	
12	||	Логическое ИЛИ	
13	?:	Тернарное условие	Справа налево
14	=	Простое присваивание	
	+= -=	Присваивание через сумму и разность	
	*= /= %=	Присваивание через произведение, частное и остаток	
	<<= >>=	Присваивание через левый сдвиг и правый сдвиг	
	&= ^= |=	Присваивание через побитовые И, исключающее ИЛИ и ИЛИ	
15	,	Запятая	Слева направо

1.6. Практический пример: меняем местами два числа

Задача проста: пользователь должен ввести два числа, а наша программа - поменять числа местами. Для этого мы будем использовать временную переменную `temp`.

Листинг 1.3. Меняем местами два числа

```
#include <stdio.h>
int main()
{
      double A, B, temp;

      printf("Введите A: ");
      scanf("%lf", &A);

      printf("Введите B: ");
      scanf("%lf",&B);

      // Значение A будет присвоено переменной temp
      temp = A;

      // Значение B будет назначено переменной A
      A = B;

      // Значение temp будет присвоено B
      B = temp;

      printf("\nПосле замены, A = %.2lf\n", A);
      printf("После замены, B = %.2lf", B);

      return 0;
}
```

Но теперь усложним немного задачу - как мы можем сделать то же самое, но без использования временной переменной? Ведь еще одна переменная типа `double` - это еще 4 байта памяти. Сейчас лишние 4 байта кажутся смешными. Но это только сейчас. Привыкайте программировать правильно и не используйте лишние ресурсы. Если бы все программисты эффективно использовали ресурсы, то современные телефонами не были бы оснащены 2 Гб оперативной памяти, а внутренняя память не забивалась бы на 80% в течение двух дней использования телефона.

Оказывается, с помощью математики можно легко решить эту задачу. Нужно от первого числа отнять второе число и присвоить результат первому (A). Ко второму числу нужно добавить новое значение первого числа. За-

тем от второго отнять новое значение первого и присвоить первому. Код приведен в листинге 1.4.

Листинг 1.4. Меняем местами два числа без использования временной переменной

```
#include <stdio.h>
int main()
{
      double A, B;

      printf("Введите A: ");
      scanf("%lf", &A);

      printf("Введите B: ");
      scanf("%lf",&B);

   A = A - B;
   B = A + B;
A = B - A;

      printf("\nПосле замены, A = %.2lf\n", A);
      printf("После замены, B = %.2lf", B);

      return 0;
}
```

Глава 2.
Операторы в Си

Оператор – это фрагмент программы, который может быть выполнен отдельно. Оператор может быть записан в одну строчку, а может – в несколько. Друг от друга операторы отделяются точками с запятой, как вы уже заметили.

Операторы языка Си можно разделить на следующие группы:

- Условные операторы (`if` и `switch`)
- Операторы цикла (`while`, `for`, `do while`)
- Операторы перехода и метки (`break`, `continue`, `goto`, `return`, `case`, `default`)
- Операторы-выражения
- Блоки

Для последних двух групп не зарезервированы какие-либо ключевые слова. Оператор-выражение – это оператор, состоящий из допустимых выражений, а блок – это любой фрагмент программы, заключенный в фигурные скобки.

2.1. Условные операторы

Прежде, чем рассматривать непосредственно условные операторы, нужно поговорить об истинных и ложных значениях. Ведь цель любого условного оператора – проверить истинность выражения и выполнить какие-либо действия в зависимости от того, является выражение истинным или ложным.

В языке Си истинным считается любое ненулевое значение, в том числе отрицательное. Ложное значение всегда равно нулю. Такое представление истинных и ложных значений позволяет создавать чрезвычайно эффективные программы.

К условным операторам в языке Си относят два оператора – `if` и `switch`. Также в качестве альтернативы `if` иногда можно использовать тернарный оператор `?`.

2.1.1. Оператор if

Синтаксис оператора `if` следующий:

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

В выражении можно использовать как операторы сравнения, так и любые другие. Как уже было отмечено, любое ненулевое значение считается истинным. Примеры выражений:

```
k < 10        // значение переменной k должно быть меньше 10
k >= 5        // значение переменной k должно быть больше или
равно 5
k == 0        // значение переменной k должно быть равно 0
k != 0        // значение k не равно 0
```

Нужно отметить, что оператор может быть, как одним оператором, как и целым блоком, заключенным в фигурные скобки. Часть `else` вовсе необязательна, поэтому она заключена в квадратные скобки.

Работает он так: если выражение истинно, то будет выполнен `оператор_1`, в противном случае – `оператор_2`, если часть `else` указана. Если части `else` нет, а выражение – ложное, то оператор `if` не будет выполнять никаких действий.

Продемонстрируем работу оператора `if` на примере программы, определяющей, является ли число четным.

Листинг 2.1. Пример использования оператора *if*

```
#include <stdio.h>
int main()
{
    int number;

    printf("Введите целое число: ");
    scanf("%d", &number);


    if(number % 2 == 0)
        printf("%d - четное\n", number);
    else
        printf("%d - нечетное\n", number);

    return 0;
}
```

Вложенным (nested) называется оператор `if`, находящийся внутри другого оператора `if` или `else`. Вложенные операторы `if` встречаются довольно часто. Во вложенном условном операторе раздел `else` всегда связан с ближайшим оператором `if`, находящимся с ним в одном блоке и не связанным с другим оператором `else`.

Рассмотрим пример:

```
if (k < 10)
{
  if (j>5) оператор_1;
  if (i>5) оператор_2;
  else оператор_3;  // данный оператор связан с оператором
if(i>5)
}
else оператор_4; // данный else связан с if (k<10)
```

Последний `else` связан с оператором `if(j>5)`, а с оператором `if (k<10)`. Оператор `if(k<10)` вообще находится в другом блоке. Оператор `else оператор_3` связан с оператором `if(i>5)`, поскольку он является ближайшим к нему.

Язык Си допускает до 15 уровней вложенности условных операторов. На практике многие компиляторы предусматривают намного большую глубину.

В качестве примера использования вложенных операторов давайте рассмотрим приложение, определяющее, является ли год високосным.

Високосные года (в которых есть 29 февраля и соответственно - 366 дней) - это те, которые делятся на 4 без остатка: 2004, 2008, 2012, 2016, 2020, 2024...

Однако в григорианском католическом календаре, по которому мы ныне живем ("новый стиль") есть еще редкое и малоизвестное правило: те года, которые нацело делятся на 100 (т.е. оканчиваются на -00) и которые делятся нацело на 400 - високосные, а которые делятся с остатком - не високосные.

Поэтому 1700, 1800, 1900 года - были невисокосными (хотя и делятся нацело на 4), 2000 - был високосным как обычно (делится нацело на 400), 2100,2200, 2300 - также будут невисокосными.

Листинг 2.2. Приложение, определяющее является ли год високосным

```
#include <stdio.h>

int main()
{
```

```
    int year;

    printf("Введите год: ");
    scanf("%d",&year);

    if(year%4 == 0)
    {
        if( year%100 == 0)
        {
            // year делится на 400, поэтому високосный
            if ( year%400 == 0)
                printf("%d - високосный\n", year);
            else
                printf("%d - невисокосный\n", year);
        }
        else
            printf("%d - високосный\n", year );
    }
    else
        printf("%d - невисокосный\n", year);

    return 0;
}
```

Как видите, здесь мы использовали не только вложенные операторы, но даже построили целую цепочку из операторов `if-else`. Общий вид такой конструкции следующий:

```
if (выражение) оператор;
else
  if (выражение) оператор;
  else
    if (выражение) оператор;
...
else (оператор);
```

Эти условия вычисляются сверху вниз. Как только значение условного выражения становится истинным, выполняется связанный с ним оператор, и оставшаяся часть конструкции игнорируется. Если все условные выраже-

ния оказались ложными, выполняется оператор, указанный в последнем разделе `else`. Если этого раздела нет, то не выполняется ни один оператор.

Чтобы не запутаться самому, лучше отдельные части заключать в фигурные скобки (блоки). Так код будет более понятным, что и показано в листинге 2.2.

2.1.2. Оператор switch

В языке Си предусмотрен оператор многовариантного ветвления `switch`, который последовательно сравнивает значение выражения со списком целых чисел или символьных констант. Если обнаруживается совпадение, выполняется оператор, связанный с соответствующей константой. Синтаксис оператора `switch` следующий:

```
switch (выражение)
{
 case константа1:
   последовательность_операторов1;
   break;
 case константа2:
   последовательность_операторов2;
   break;
...
 default:
   последовательность операторов
}
```

Значением выражения должен быть символ или целое число. Например, выражения, результатом которых является число с плавающей точкой, не допускаются. Значение выражения последовательно сравнивается с константами, указанными в операторах `case`. Если обнаруживается совпадение, выполняется последовательность операторов, связанных с данным оператором `case`, пока не встретится оператор `break` или не будет достигнут конец оператора `switch`. Если значение выражения не совпадает ни с одной из констант, выполняется оператор `default`. Этот раздел оператора

switch является необязательным. Если он не предусмотрен, в отсутствие совпадений не будет выполнен ни один оператор.

Язык Си поддерживает до 257 операторов case. На практике это очень много и вряд ли у вас получится превысить это ограничение, если не делать, конечно, это намеренно.

Оператор break относится к группе операторов перехода. Его можно использовать как в операторе switch, так и в циклах (см. далее). Когда поток управления достигает оператора break, программа выполняет переход к оператору, следующему за оператором switch.

При использовании switch обратите внимание на следующие моменты:

1. Две константы в разных разделах case не могут иметь одинаковых значений, за исключением случая, когда один switch вложен в другой (такое тоже допускается).
2. Оператор switch отличается от if тем, что значение его выражения сравнивается исключительно с константами, а в операторе if можно выполнять любые сравнения и производить вычисление значений выражений.
3. Если в операторе switch используются символьные константы, они автоматически преобразовываются в целочисленные.

Рассмотрим оператор switch на примере простейшего калькулятора. Программа просит пользователя ввести арифметический оператор, его операнды, а затем с помощью switch производит необходимое вычисление.

Листинг 2.3. Простейший калькулятор

```
# include <stdio.h>

int main() {

    char operator;
    double firstNumber,secondNumber;

    printf("Введите оператор (+, -, *,): ");
    scanf("%c", &operator);
```

```
    printf("Введите два операнда: ");
    scanf("%lf %lf",&firstNumber, &secondNumber);

    switch(operator)
    {
        case '+':
            printf("%.1lf + %.1lf = %.1lf",firstNumber,
                     secondNumber, firstNumber + secondNumber);
            break;

        case '-':
            printf("%.1lf - %.1lf = %.1lf",firstNumber,
                   secondNumber, firstNumber - secondNumber);
            break;

        case '*':
            printf("%.1lf * %.1lf = %.1lf",firstNumber,
                     secondNumber, firstNumber * secondNumber);
            break;

        case '/':
             if (secondNumber != 0)
               printf("%.1lf / %.1lf = %.1lf",firstNumber,
                      secondNumber, firstNumber / firstNumber);
             else printf("На ноль делить нельзя!");
             break;

        // оператор неизвестен (+, -, *, /)
        default:
            printf("Ошибка! Неправильный оператор");
    }

    printf("\n");
    return 0;
}
```

2.2. Операторы цикла

Язык Си поддерживает несколько операторов цикла – `for` (он же цикл со счетчиком), `while`, `do-while`. Циклы предназначены для выполнения повторяющихся инструкций, пока действует определенное правило – пока истинно указанное в цикле условие.

2.2.1. Цикл for

Первый цикл называют циклом-счетчиком. Его удобно использовать, когда мы знаем, сколько итераций (повторений) должно быть. В нашем случае удобно использовать именно его. Общий синтаксис выглядит так:

```
for (оператор_1; условие; оператор_2)
{
  тело_цикла
}
```

Здесь `оператор_1` выполняется в самом начале и один раз, перед первым выполнения `тела цикла`. Его удобно использовать для инициализации переменной-счетчика. `Оператор 2` выполняется после каждой итерации, в нем удобно увеличивать счетчик. Наш цикл будет выполнять операции в `теле цикла` до тех пор, пока истинно условие, заданное при объявлении цикла.

Код программы, использующий цикл `for`, приведен в листинге 2.4.

Листинг 2.4. Вычисляем сумму натуральных чисел с помощью цикла *for*

```
#include <stdio.h>
int main()
{
    int n, i, sum = 0;

    printf("Введите положительное целое число: ");
    scanf("%d",&n);
```

```
    for(i=1; i <= n; ++i)
    {
        sum += i;    // sum = sum+i;
    }

    printf("Сумма = %d\n",sum);

    return 0;
}
```

Первым делом цикл присваивает переменной `i` значение 1 - он инициализирует счетчик (нам нужно вычислить сумму от 1 до n). Далее цикл проверяет, истинно ли условие - если пользователь ввел 3 в качестве n, да, условие истинно, поскольку `i` (1) меньше 3. Если условие истинно, выполняется тело цикла, а именно к переменной `sum` добавляется значение переменной `i`. И так до тех пор, пока `i` не станет больше n (у нас условие выполнения цикла меньше или равно).

Организация бесконечного цикла

Хотя в качестве бесконечного можно использовать любой цикл, традиционно для этой цели применяется оператор `for`. Поскольку все разделы оператора `for` являются необязательными, его легко сделать бесконечным, не задав никакого условного выражения.

```
for ( ; ; ) printf("Этот цикл будет выполняться бесконечно\n");
```

Если условное выражение не указано, оно считается истинным. Разумеется, в этом случае можно по-прежнему выполнять инициализацию и приращение счетчика, однако программисты в качестве бесконечного цикла чаще всего используют конструкцию `for( ; ; )`.

На самом деле конструкция `for( ; ; )` не гарантирует бесконечное выполнение цикла, поскольку его тело может содержать оператор `break`, приводящий к немедленному выходу. В этом случае программа передаст управ-

ление следующему оператору, находящемуся за пределами тела цикла `for`, как показано ниже.

```
int k = 0;
for ( ; ; ) {
  printf("\nВведите цифру: ");
  scanf("%d", &k);
  if (k==5) break;  // выход из бесконечного цикла
}
```

Данная программа будет бесконечно запрашивать у пользователя ввести цифру. Однако, если пользователь введет 5, то выполнение бесконечного цикла будет прервано.

2.2.2. Цикл while

Как уже было отмечено, поскольку мы знаем количество итераций (n), нашу задачу удобнее всего решить именно с циклом `for`. Однако ее можно решить и с помощью других циклов. Цикл `while()` выполняется, пока истинно его условие:

```
while (условие)
{
  тело_цикла;
}
```

Такой цикл удобно использовать, когда мы не знаем заранее количество итераций, например:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int k = 0, i = 0;
```

```
  while (k < 7) {
    k = 1 + rand() % 10;
    printf("%d ", k);
    i++;
  }

  printf("Количество итераций %d\n", i);

  return 0;
}
```

Программа вызывает функцию `rand()`, которая возвращает случайное число до 10. В цикле `while` у нас условие: работать, пока `k` < 7, если полученное значение будет больше 7, например 8, то цикл прекратит свою работу. Программа выведет счетчик итераций - сугубо для информации и все сгенерированные случайным образом числа. Например:

```
4 7 Количество итераций 2
```

В листинге 2.4 было показано, как вычислить сумму натуральных чисел с помощью цикла `for`. Перепишем программу с использованием цикла `while` (лист. 2.5).

Листинг 2.5. Сумма натуральных чисел (цикл *while*)

```
#include <stdio.h>
int main()
{
    int n, i, sum = 0;

    printf("Введите n: ");
    scanf("%d",&n);

    i = 1;
    while ( i <=n )
    {
        sum += i;
        ++i;
```

```
    }

    printf("Сумма = %d",sum);

    return 0;
}
```

В отличие от цикла `for`, мы инициализируем счетчик `i` до цикла, затем увеличиваем его в теле цикла. При написании подобных циклов главное не забыть увеличить переменную-счетчик в теле цикла, иначе цикл будет выполняться вечно.

Кстати, организовать бесконечный цикл с помощью `while()` еще проще:

```
while (1) {
  // цикл будет выполняться бесконечно
}
```

Как и в случае с `for()`, нужно предусмотреть условие выхода из цикла, например:

```
while (1) {
  // что-то делаем, например, получаем значение s
  s = getS();
  if (s == 10) break;  // прерываем цикл
}
```

Бесконечные циклы используются на практике чаще, чем может показаться. Например, сервер (как будет показано в главе 12) в бесконечном цикле обрабатывает подключения клиентов:

```
      while (1) {
            ans_len = sizeof(client);
            sock2 = accept (sock1, &client, &ans_len);
            write (sock2, MSG_TO_SEND, sizeof(MSG_TO_SEND));
            total+=1;
            ans_len = read (sock2, buffer, BUF_SIZE);
            write (1, buffer, ans_len);
```

```
            printf("Client no %d\n",total);
            shutdown (sock2, 0);
            close (sock2);
        };
```

Здесь мы даже не предусмотрели условие выхода из цикла – оно не нужно, так как в нашей демо-программе сервер должен обрабатывать запросы постоянно. На практике можно предусмотреть какое-то условие выхода, например, читать значение из какого-то контрольного файла и проверять его. Если в нем записано EXIT (или что-то другое, например, 0), то прерываем цикл.

Впрочем, прервать цикл можно и в самой консоли. Для этого достаточно нажать Ctrl + C в окне (или в терминале), в котором вы запустили программу с бесконечным циклом.

2.2.3. Цикл do-while

Цикл `do..while` в отличие от цикла `while` сначала выполняет тело цикла, а затем уже проверяет условие. Его удобно использовать, когда нужно, чтобы тело цикла выполнилось хотя бы один раз.

Синтаксис следующий:

```
do {
  оператор;
} while (условие);
```

Давайте переделаем программу из листинга 2.4 так, чтобы она запрашивала у пользователя ввод, пока он не введет целое положительное число. Далее вычислим сумму с помощью цикла `for`. Код программы в листинге 2.6.

Листинг 2.6. Демонстрация цикла *do..while*

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

```
    int n, i, sum = 0;

    do {
        printf("Введите целое положительное число > 0: ");
        scanf("%d",&n);
    }
    while (n <= 0);

    for(i=1; i <= n; ++i)
    {
        sum += i;    // sum = sum+i;
    }

    printf("Сумма = %d",sum);

    return 0;
}
```

2.3. Операторы перехода

В языке Си предусмотрены четыре оператора безусловного перехода – `return` (используется в функциях), `goto`, `break`, `continue`. Операторы `break` и `continue` связаны с операторами циклов, поэтому их использование подразумевает применение в циклах. Также `break` можно применять внутри оператора `switch`.

2.3.1. Оператор return

Оператор `return` – это оператор возврата управления из функции. Он относится к операторам безусловного перехода (jump operators), поскольку выполняет возврат в точку вызова функции. С ним может быть связано определенное значение, хотя это и не обязательно. Если оператор `return` связан с определенным значением, оно становится результатом функции. В стандарте C89 все функции, возвращающие значения, формально не обязаны были содержать оператор `return`. Если программист забывал указать

его, функция возвращала какое-то случайное значение, так называемый "мусор". Однако все изменилось в стандарте C99, где все функции обязаны содержать оператор `return`. Исключение делается только для функций типа `void`, которые не возвращают каких-либо значений.

Синтаксис оператора следующий:

```
return выражение;
```

Выражение указывается лишь тогда, когда в соответствии со своим объявлением функция возвращает какое-то значение. В этом случае результатом функции является значение данного выражения.

Внутри функции можно использовать сколько угодно операторов `return`. Однако функция прекратит свои вычисления, как только достигнет первого оператора `return`. Закрывающая фигурная скобка, ограничивающая тело функции, также приводит к прекращению ее выполнения. Она интерпретируется как оператор `return`, не связанный ни с каким значением. Если программист не укажет оператор `return` в функции, возвращающей некое значение, то ее результат останется неопределенным.

2.3.2. Оператор goto

Оператор `goto` – настоящий архаизм и в современных программах нет места для этих операторов. В этой книге он приводится только потому, что он все еще не исключен из состава языка Си. Возможно, в каком-то следующем стандарте языка этот оператор будет упразднен.

Синтаксис оператора:

```
goto метка;
...
метка:
```

Оператор производил безусловный переход к указанной метке. Далее выполнение программы продолжается с метки, на которую выполнен переход.

Пример кода с использованием goto:

```
z = 1;
loop:
  z++;
  if (z<10) goto loop;
```

2.3.3. Операторы break и continue

Оператор break может использоваться в циклах и операторе switch. В операторе switch он необходим для обеспечения нормальной работы оператора. Рассмотрим пример:

```
switch (n) {
 case 1:
   op1(n);
 case 2:
   op2(n);
 case 3:
   op3(n);
 case 4:
   op4(n);
}
```

Представим, что n равно два. Поскольку мы не использовали break, то будут выполнены операторы op2(n), op3(n) и op4(n). Это явно не то, что мы хотели. Если же переписать наш оператор switch так:

```
switch (n) {
 case 1:
   op1(n);
   break;
 case 2:
   op2(n);
   break;
 case 3:
   op3(n);
   break;
```

```
 case 4:
   op4(n);
   break;
}
```

то теперь он будет работать так, как нам нужно. Если n = 2, то будет выполнен только оператор `op2(n)`.

В теле цикла оператор `break` выполняет прерывание цикла. Полезно использовать этот оператор в бесконечных циклах, чтобы контролировать выход из такого цикла. Также можно прервать выполнение цикла, если цель уже достигнута, например, найден нужный нам элемент в массиве – чтобы вхолостую программа не работала.

Оператор `continue` осуществляет прерывание не всего цикла, а только текущей его итерации. Например, следующий цикл выведет только четные числа от 0 до 10:

```
for (c = 1; c<11; c++) {
  if (c%2!=0) continue;
  printf("%d ", c);
}
```

Программа выведет числа 2, 4, 6, 8 и 10. Здесь все просто, если у нас есть остаток от деления (то есть результат c%2 не равен 0), тогда мы переходим к следующей итерации цикла (оператор continue).

2.4. Операторы-выражения

Выражения были подробно рассмотрены в предыдущей главе. Но нужно знать некоторые моменты. Во-первых, любое выражение, завершающееся точкой с запятой, считается оператором. Во-вторых, вызов функции – это тоже оператор. Рассмотрим несколько примеров:

```
a = b + c;      // самый обычный оператор
```

```
func1(a);    // вызов функции - тоже оператор
;       // пустой оператор
b + c;       // оператор, правда, его значение будет потеряно,
// поскольку не присвоено ни одной из переменных
```

2.5. Блоки

Блок – это группа связанных операторов, заключенных в фигурные скобки. Операторы в блоке логически связаны друг с другом. Иногда блок также называют составным оператором. Блоки часто используются как составная часть другого оператора, например, оператора `if`, если при достижении какого-то условия нужно выполнить несколько действий – тогда эти действия принято объединить в блок.

Например:

```
if (low > high) {
        temp = low;
        low = high;
        high = temp;
    }
```

2.6. Функция exit()

Функция `exit()` не относится к управляющим операторам, но любой программист обязан знать о ней. Вызов данной функции приводит к завершению работы программы и передаче управления операционной системе. Сразу после вызова `exit()` производится немедленное прекращение работы программы, а оболочке, вызвавшей вашу программу, передается значение, указанное программистом в качестве параметра функции `exit()`:

```
void exit(int код_возврата);
```

Значение переменной `код_возврата` передается вызывающему процессу, в роли которого чаще всего выступает операционная система. Нулевое

значение кода возврата соответствует нормальному завершению работы. Другие значения аргумента указывают на вид ошибки. В качестве кода возврата можно применять макросы EXIT_SUCCESS (успешный выход) и EXIT_FAILURE (ошибка). Для вызова функции `exit()` необходим заголовочный файл stdlib.h.

Часто функцию используют, если нужно прервать работу программы в результате возникшей ошибки, например:

```
if (bind(sock1, (struct sockaddr *)&sin, sizeof(sin)) ==
SOCKET_ERROR)
{
   printf("Error bind socket\n");
   exit(EXIT_FAILURE);
}
```

Глава 3.
Массивы и строки в Си

Массив – это упорядоченный набор данных, имеющих одинаковый тип и объединенных под одним именем. Доступ к отдельному элементу массива осуществляется через его индекс. Все массивы состоят из смежных ячеек памяти. Младший адрес принадлежит первому элементу массива, старший – последнему. Массивы могут быть одномерными и многомерными. Наиболее распространенным массивом является строка, завершающаяся нулевым байтом. Она представляет собой обычный массив символов, последним элементом которого является нулевой байт. Получается, что ваша программа редко обойдется без использования массивов – даже если вы непосредственно не используете массив, то вы будете в любом случае использовать строки, которые в свою очередь являются массивами символов.

3.1. Одномерные массивы

Синтаксис объявления массива выглядит так:

```
тип имя[размер];
```

Например:

```
double total[50];
```

Нумерация элементов массива начинается с нуля. Получить доступ к элементам массива можно так:

```
total[0] = 50.99;   // первый элемент равен 50.99
total[1] = 88.11;   // второй элемент равен 88.11
..
total[49] = 77.56;  // последний элемент равен 77.56
```

Одномерный массив представляет собой список переменных, имеющих одинаковый тип и хранящихся в смежных ячейках памяти в порядке возрастания их индексов. Если у нашего массива `total` адрес первого элемента был бы 1000, тогда весь массив будет выглядеть так:

Элемент	total[0]	total[1]	..	total[49]
Адрес	1000	1001	..	1049

Объем памяти, необходимый для хранения массива, зависит от его типа и размера.

Размер одномерного массива в байтах вычисляется по формуле:

```
размер = sizeof(базовый_тип) * к-во_элементов
```

В языке Си не предусмотрена проверка выхода индекса массива за пределы допустимого диапазона. Иными словами, во время выполнения программы можно по ошибке выйти за пределы памяти, отведенной для массива, и записать данные в соседние ячейки, в которых могут храниться другие переменные и даже программный код. Ответственность за предотвращение подобных ошибок лежит на программисте.

Рассмотрим практический пример. Пусть у нас есть массив элементов `arr[100]`. Мы пройдемся по элементам массива от 1 до 100, точнее до `n` (`n` вводит пользователь, а максимум `n` = 100) и попробуем найти максимум в массиве. При этом максимум мы будем хранить не в отдельной переменной, а в самом массиве - в элементе `arr[0]`.

```
int i, n;
float arr[100];

printf("Введите количество элементов массива (1-100): ");
scanf("%d", &n);
printf("\n");

// Читаем ввод пользователя
for(i = 0; i < n; ++i)
{
   printf("Введите элемент %d: ", i+1);
   scanf("%f", &arr[i]);
}

// Проходимся по массиву, сохраняя наибольший элемент в
arr[0]
for(i = 1; i < n; ++i)
{
   // Если arr[0] < текущего элемента, значит у нас есть
новый максимум
   if(arr[0] < arr[i])
       arr[0] = arr[i];
}
printf("Максимум = %.2f\n", arr[0]);
```

Аналогичным образом можно вычислить среднее значение массива:

```
// Заполняем массив и параллельно подсчитываем среднее
арифметическое
for(i = 0; i < n; ++i)
{
    printf("%d. Введите число: ", i+1);
    scanf("%f", &num[i]);
```

```
        sum += num[i];
    }

    average = sum / n;
    printf("Среднее = %.2f\n", average);
```

В процессе заполнения массива пользователем мы вычисляем сумму всех элементов. Так мы экономим один проход по массиву – нам уже не придется перебирать все элементы массива, чтобы вычислить общую сумму. Вот она вся прелесть "параллельных" вычислений.

Далее мы делим сумму на количество элементов и таким образом находим среднее значение.

3.2. Указатель на массив

Имя массива является указателем на первый его элемент. Допустим, массив объявлен с помощью оператора:

```
double total[50];
```

Указателем на его первый элемент при этом является имя `total`. Таким образом, в следующем фрагменте указателю присваивается адрес первого элемента массива `total`:

```
double *p;
double total[50];
p = total;
```

Адрес первого элемента массива можно также вычислить с помощью оператора `&`. Например, выражения `total` и `&total[0]` эквивалентны. Однако в профессионально написанных программах на языке Си вы никогда не встретите выражение `&total[0]`.

В языке Си вы не можете передать в функцию весь массив. Вместо него можно передать указатель на массив. Собственно, поэтому данный раздел

и появился в этой главе. Представим, что у нас есть функция `max()`, вычисляющая максимум в массиве. Разберемся, как правильно ее объявить:

```
int j[10];
...
max(j);
```

Если аргументом функции должен быть одномерный массив, ее формальный параметр можно объявить тремя способами: как указатель, как массив фиксированного размера и как массив неопределенного размера. Например, чтобы предоставить функции `max()` доступ к массиву `j`, можно использовать три варианта. **Первый** способ: в качестве ее аргумента можно объявить указатель.

```
int max(int *x)
{
...
}
```

Второй способ: передать указатель на массив фиксированного размера:

```
int max(int x[10])
{
...
}
```

Третий: использовать массив неопределенного размера:

```
int max(int x[])
{
...
}
```

Эти три объявления эквивалентны друг другу, поскольку их суть одинакова: в функцию передается указатель на целочисленную переменную. В первом случае действительно используется указатель. Во втором применяется стандартное объявление массива. В последнем варианте объявляется, что

в функцию будет передан целочисленный массив неопределенной длины. Размер массива, передаваемого в функцию, не имеет никакого значения, поскольку проверка выхода индекса за пределы допустимого диапазона в языке Си не предусмотрена. При указании размера массива можно написать любое число — это ничего не изменит, поскольку в функцию будет передан не массив, состоящий из 99 элементов, а лишь указатель на его первый элемент.

В качестве примера передачи массива функции давайте рассмотрим функцию `calculateSD()`, которая вычисляет среднеквадратическое отклонение. В программе мы просим пользователя ввести 10 элементов массива `data`. Далее мы передаем этот массив функции `calculateSD()` и выводим ее результат:

```
    printf("Введите 10 элементов: ");
    for(i=0; i < 10; ++i)
        scanf("%f", &data[i]);

    printf("\nСреднеквадратическое отклонение = %.6f\n",
calculateSD(data));
```

Код функции выглядит так:

```
float calculateSD(float data[])
{
    float sum = 0.0, mean, standardDeviation = 0.0;

    int i;

    for(i=0; i<10; ++i)
    {
        sum += data[i];
    }

    mean = sum/10;

    for(i=0; i<10; ++i)
        standardDeviation += pow(data[i] - mean, 2);
```

```
    return sqrt(standardDeviation/10);
}
```

3.3. Работа со строками

3.3.1. Объявление строки

Очень часто для представления строк используется массив, если быть предельно точным, то одномерный массив символов. В языке Си предусмотрен только один вид строк – строки, завершающиеся нулевым байтом, их также называют С-строками. Такие строки представляют собой массив символов, последним элементом которых является нулевой байт. В C++ есть еще класс `string`, который реализует объектно-ориентированный подход при работе со строками.

Вот как можно объявить массив символов:

```
char first_name[21];
```

Объявляя массив символов помните, что последняя его ячейка должна быть зарезервирована для нулевого символа. Поэтому если реально у вас должно храниться 20 символов, то добавьте еще один – для нулевого символа.

Второй способ объявления строки выглядит так:

```
char s[] = "Это строка";
```

Справа от знака присваивания записана строковая константа. В конце строки автоматически добавляется ноль ('\0'). Константы символьных строк помещаются в класс статической памяти. Длину строки компилятор вычислит автоматически.

Третий способ заключается в использовании указателей:

```
char *s="Третий вариант инициализации";
```

Переменная s будет указателем на то место в оперативной памяти, где располагается строковая константа. В такой форме записи кроется потенциальная ошибка, заключающаяся в том, что указатель на символ часто называют строкой.

3.3.2. Правильное выделение памяти под строку

При работе со строками в Си вы всегда должны помнить о том, что они заканчиваются всегда символом '\0', а это еще один байт при выделении памяти и об этом нужно помнить. В частности, рассмотрим следующий код:

```
char *str = (char *)malloc(sizeof(char) * strlen(buffer));
```

Здесь как раз программист допустил ошибку. Правильно выделять память под строку так:

```
char *str = (char *)malloc(sizeof(char) * (strlen(buffer)+1));
```

Сейчас мы учли как раз тот самый нулевой символ. Поверьте, программ с такой ошибкой очень много на практике и до поры до времени данная ошибка никак себя не проявляет. Пользователь просто вводит строку меньше, чем может поместиться. А вот если он введет строку максимального размера, то окажется, что памяти выделено недостаточно, поскольку о нулевом символе программист забыл.

3.3.3. Функции для работы со строками

Рассмотрим часто используемые функции для работы со строками (табл. 3.1).

Таблица 3.1. Функции для работы со строками

Функция	Описание
`char *strcat(char *s1, char *s2)`	Присоединяет `s2` к `s1`, возвращает `s1`
`char *strncat(char *s1, char *s2, int n)`	Присоединяет не более `n` символов `s2` к `s1`, завершает строку символом '\0', возвращает `s1`
`char *strcpy(char *s1, char *s2)`	Копирует строку `s2` в строку `s1`, включая '\0', возвращает `s1`
`char *strncpy(char *s1, char *s2, int n)`	Копирует не более `n` символов строки `s2` в строку `s1`, возвращает `s1`;
`int strcmp(char *s1, char *s2)`	Сравнивает `s1` и `s2`, возвращает значение 0, если строки эквивалентны
`int strncmp(char *s1, char *s2, int n)`	Сравнивает не более n символов строк `s1` и `s2`, возвращает значение 0, если начальные `n` символов строк эквивалентны
`int strlen(char *s)`	Возвращает количество символов в строке `s`
`char *strset(char *s, char c)`	Заполняет строку `s` символами, код которых равен значению `c`, возвращает указатель на строку `s`
`char *strnset(char *s, char c, int n)`	Заменяет первые n символов строки `s` символами, код которых равен c, возвращает указатель на строку s

Все эти строки объявлены в заголовочном файле string.h.

3.3.4. Работа со строкой как с массивом символов

О строках мы еще поговорим в этой книге, а пока рассмотрим пример программы, которая удаляет из заданной строки все символы, кроме цифр:

```
char line[101];
int i, j;
gets(line);

for(i = 0; line[i] != '\0'; ++i)
{
    while (!( (line[i]>='0' && line[i]<='9') || line[i] == '\0') )
    {
        for(j = i; line[j] != '\0'; ++j)
        {
            line[j] = line[j+1];
        }
        line[j] = '\0';
    }
}

printf("%s", line);
```

Работает программа так:

- Пользователь вводит строку, которую мы записываем в переменную `line`
- В цикле мы проверяем, является ли символ цифровым.
- Если нет, то все символы после него, включая нулевой символ, смещаются на 1 позицию влево.
- С помощью `printf()` выводим результат на экран.

3.3.5. Пример использования библиотечных функций

Рассмотрим программу, демонстрирующую работу с библиотечными строковыми функциями.

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

```
int main() {
  // Не забываем о последнем нулевом символе!
  char m1[80] = "First string";
  char m2[80] = "Second string";
  char m3[80];
  strncpy(m3, m1, 6);  // не добавляет '\0' в конце строки
  puts("Результат strncpy(m3, m1, 6)");
  puts(m3);
  strcpy(m3, m1);
  puts("Результат strcpy(m3, m1)");
  puts(m3);
  puts("Результат strcmp(m3, m1) равен");
  printf("%d", strcmp(m3, m1));
  strncat(m3, m2, 5);
  puts("Результат strncat(m3, m2, 5)");
  puts(m3);
  strcat(m3, m2);
  puts("Результат strcat(m3, m2)");
  puts(m3);
  puts("Количество символов в строке m1 равно  strlen(m1): ");
  printf("%d\n", strlen(m1));
  return 0;
}
```

Вывод программы будет таким:

```
Результат strncpy(m3, m1, 6)
First
Результат strcpy(m3, m1)
First string
Результат strcmp(m3, m1) равен
0
Результат strncat(m3, m2, 5)
First stringSecon
Результат strcat(m3, m2)
First stringSeconSecond string
Количество символов в строке m1 равно  strlen(m1):
12
```

3.3.6. Многобайтовые строки. Работа с UTF-8

Все сказанное ранее правильно при работе с английским языком. Как правило, один символ занимает один байт. Это правильно для многих кодировок, в том числе CP-1251, KOI8-R. Но в настоящее время все современные операционные системы, в том числе Linux, перешли на UTF-8. В этой кодировке латиница по-прежнему занимает 1 байт для обеспечения обратной совместимости, а вот символы национальных алфавитов, в том числе кириллица – 2 байта.

В некоторых случаях код работает с UTF-8 нормально:

```
char *slov = NULL;
size_t bufslov = 0;
getline(&slov, &bufslov, stdin);
printf("%s", slov);
if(slov)
{
  free(slov);
  slov = NULL;
}
```

А такой код уже не будет нормально работать, вместо русской буквы вы получите вопросительные знаки:

```
char *slov = NULL;
size_t bufslov = 0;
getline(&slov, &bufslov, stdin);
for( ; slov != '\n'; slov++) printf("%c", *slov);
if(slov)
{
  free(slov);
  slov = NULL;
}
```

Нужно использовать широкие символы wchar_t и функции для работы с ними, объявленные в заголовочном файле wchar.h

Вот как можно работать с многобайтовыми строками:

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

int main(int argc, char *argv[])
{
    wchar_t buf[BUFSIZ];
    wchar_t *pch;
    size_t count;

    setlocale(LC_ALL, "");

    fputws(L"Введите свое имя: ", stdout);
    fflush(stdout);

    fgetws(buf, BUFSIZ, stdin);

    fputws(L"Ваше имя (по буквам):\n", stdout);

    /* Убираем символ перевода строки */
    if((pch = wcschr(buf, L'\n')) != NULL)
        *pch = L'\0';

    for(pch = buf, count = 0; *pch != L'\0'; ++pch, ++count)
        putwchar(*pch);

    putwchar(L'\n');

    wprintf(L"Введено букв (без символа перевода строки): %zd\n",
count);

    exit(0);
}
```

Вывод программы может быть таким:

```
Введите свое имя: Денис
Ваше имя (по буквам):
```

```
Денис
Введено букв (без символа перевода строки): 5
```

Как видите, код работает нормально.

3.4. Многомерные массивы

Язык Си поддерживает многомерные массивы. На практике наиболее часто используются двухмерные массивы, которые представляют собой массив одномерных массивов. Синтаксис объявления многомерного массива выглядит так:

```
тип имя [размер1][размер2]...[размерN]
```

В случае с двухмерных массивов размеров будет два:

```
тип имя [размер1][размер2]
```

Вот как можно объявить массив, состоящий из 100 строк и 200 столбцов:

```
int c[100][200];
```

Обратиться к элементу массива можно так:

```
c[1][5]
```

Двухмерный массив хранится в виде матрицы, в которой первый индекс задает номер строки, а второй — номер столбца. Таким образом, при обходе элементов в порядке их размещения в памяти правый индекс изменяется быстрее, чем левый.

Объем памяти, занимаемый двухмерным массивом, считается так:

```
размер = к-во_строк * к-во_столбцов * sizeof(базовый тип)
```

Если размер типа `int` составляет 4 байта, то наш массив с займет 80000 байтов.

Если двухмерный массив используется в качестве аргумента функции, то в нее передается только указатель на его первый элемент. Однако при этом необходимо указать, по крайней мере, количество столбцов.

```
void process(int x[][20])
{
...
}
```

Рассмотрим довольно сложный пример - сложение двух матриц с использованием многомерных массивов. У нас будет три матрицы: a[100][100], b[100][100] и sum[100][100]. Последняя, как вы уже догадались, будет содержать сумму матриц `a` и `b`.

Пользователь вводит количество строк `r` и количество колонок `c`. Значения `r` и `c` в этой программе должны быть меньше 100.

После ввода `r` и `c`, пользователь должен будет ввести элементы обеих матриц. Далее программа выполнит сложение матриц и отобразит результат.

```
int r, c, a[100][100], b[100][100], sum[100][100], i, j;

    printf("Введите количество строк от 1 до 100: ");
    scanf("%d", &r);
    printf("Введите количество колонок от 1 до 100: ");
    scanf("%d", &c);

    printf("\nВведите элементы первой матрицы:\n");

    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("Введите элемент a%d%d: ",i+1,j+1);
            scanf("%d",&a[i][j]);
        }
```

```
printf("Введите элементы второй матрицы:\n");
for(i=0; i<r; ++i)
    for(j=0; j<c; ++j)
    {
        printf("Введите элемент b%d%d: ",i+1, j+1);
        scanf("%d", &b[i][j]);
    }

// Сложение 2 матриц

for(i=0;i<r;++i)
    for(j=0;j<c;++j)
    {
        sum[i][j]=a[i][j]+b[i][j];
    }

// Отображаем результат
printf("\nСумма 2 матриц: \n\n");

for(i=0;i<r;++i)
    for(j=0;j<c;++j)
    {

        printf("%d   ",sum[i][j]);

        if(j==c-1)
        {
            printf("\n\n");
        }
    }
```

Данная программа демонстрирует ввод (заполнение) многомерного массива, проход по многомерному массиву с целью обработки данных и вывода элементов.

Раз мы уже начали работать с матрицами, то эта книга была бы неполной, если бы мы не рассмотрели умножение двух матриц.

Чтобы умножить две матрицы, нужно каждую строку первой матрицы умножить на каждый столбец второй матрицы. Умножая первую строку первой матрицы на каждый столбец второй матрицы, мы получим все элементы

первой строки матрицы произведения, затем делаем то же самое для второй строки первой матрицы и т.д.

Чтобы умножить две матрицы, число колонок первой матрицы должно быть равно количество строк второй матрицы. Программа отобразит ошибку, если это не так.

```
int a[10][10], b[10][10], result[10][10], r1, c1, r2, c2, i, j, k;

printf("Введите количество строк и колонок первой матрицы: ");
scanf("%d %d", &r1, &c1);

printf("Введите количество строк и колонок второй матрицы: ");
scanf("%d %d",&r2, &c2);

// Проверяем, можем ли мы умножить две матрицы
while (c1 != r2)
{
    printf("Ошибка! К-во колонок первой матрицы не равно
            количеству строк второй\n\n");
    printf("Введите количество строк и колонок первой
            матрицы: ");
    scanf("%d %d", &r1, &c1);
    printf("Введите количество строк и колонок второй
            матрицы: ");
    scanf("%d %d",&r2, &c2);
}

// Вводим элементы 1 матрицы
printf("\nВведите элементы 1 матрицы:\n");
for(i=0; i<r1; ++i)
    for(j=0; j<c1; ++j)
    {
        printf("Введите элемент a%d%d: ",i+1, j+1);
        scanf("%d", &a[i][j]);
    }

// Вводим элементы 2 матрицы
printf("\nВведите элементы 2 матрицы:\n");
for(i=0; i<r2; ++i)
```

```
    for(j=0; j<c2; ++j)
    {
        printf("Введите элемент b%d%d: ",i+1, j+1);
        scanf("%d",&b[i][j]);
    }

// Заполняем все элементы результирующей матрицы нулями
for(i=0; i<r1; ++i)
    for(j=0; j<c2; ++j)
    {
        result[i][j] = 0;
    }
```

Код умножения матриц выглядит так:

```
// Умножаем матрицы a и b
// Результат сохраняем в матрице result
for(i=0; i<r1; ++i)
    for(j=0; j<c2; ++j)
        for(k=0; k<c1; ++k)
        {
            result[i][j]+=a[i][k]*b[k][j];
        }
```

Затем нам останется лишь вывести матрицу `result`:

```
printf("\nРезультат умножения матриц:\n");
for(i=0; i<r1; ++i)
    for(j=0; j<c2; ++j)
    {
        printf("%d  ", result[i][j]);
        if(j == c2-1)
            printf("\n\n");
    }
```

3.5. Индексация указателей

Как уже отмечалось ранее, имя массива без индекса представляет собой указатель на его первый элемент. Пусть у нас есть массив:

```
char c[20];
```

Следующие выражения полностью идентичны:

```
c
&c[0]
```

Также является истинным следующее выражение (поскольку адрес первого элемента массива совпадает с адресом всего массива):

```
c == &c[0]
```

Указатель можно проиндексировать, как массив. Например:

```
int *c, j[10];
c = j;

c[5] = 10;      // присвоение значения по индексу
*(c+5) = 10; // присвоение значения по адресу, результат - тот
же
```

Второй способ называется способом *адресной арифметики*. Его можно применять с успехом и к многомерным массивам. Правила адресной арифметики требуют приведения типа указателя на массив к его базовому типу. Обращение к элементам массива с помощью указателей используется довольно широко, поскольку операции адресной арифметики выполняются быстрее, чем индексация.

3.6. Инициализация массива

Ранее мы заполняли массив путем ввода с клавиатуры. В языке Си допускается инициализация массива при его объявлении. Делается это так:

```
тип имя [размер1]…[размерN] = {значения}
```

Пример:

```
int j[5] = {1, 2, 3, 4, 5}
```

Инициализировать строку можно так:

```
char s[20] = "Hello!";
```

Конечно, можно инициализировать строку и посимвольно:

```
char s[20] = {'H', 'e', 'l', 'l', 'o'};
```

При инициализации многомерного массива можно заключать элементы списка в фигурные скобки, что называется *субагрегатной* группировкой. Так проще и нагляднее для самого программиста:

```
int s[5][2] = {
 {1, 1},
 {1, 2},
 {1, 3},
 {1, 4},
 {1, 5}
};
```

Глава 4.
Указатели в Си

Указатели – это очень мощный инструмент языка Си. Одновременно – это одно из наиболее сложных понятий языка. При работе с указателями начинающие программисты очень часто допускают ошибки, в результате программа работает не так как нужно.

4.1. Что такое указатели и для чего они нужны?

Указатель (pointer) – это переменная, содержащая адрес другой переменной. Если одна переменная содержит адрес другой, говорят, что она указывает (point) на ту переменную. Посмотрим на рисунок 4.1. На нем изображены ячейки в памяти. Под ячейками указаны их адреса. Первая ячейка является указателем. Она хранит в себе не конкретное значение, а адрес другой переменной (1004). Переменная с адресом 1004 хранит в себе значение 100. Через указатель с адресом 1004 можно добраться к значению 100.

Рис. 4.1. Что такое указатель

Зачем нужны указатели, если к заданному значению и так можно добраться через идентификатор переменной (через имя переменной)? Есть как минимум три причины использовать указатели:

- Указатели позволяют функциям изменять свои аргументы.
- С помощью указателей осуществляется динамическое распределение памяти, вследствие чего программа использует память более рационально.
- Указатели повышают эффективность многих процедур.

4.2. Объявление указателей и операторы для работы с ними

Синтаксис объявления указателя следующий:

```
тип *имя_указателя;
```

Другими словами, указатель объявляется так же, как и переменная, только перед именем ставится `*`. `Тип` – это базовый тип указателя, то есть тип переменной, на которую будет ссылаться указатель. С формальной точки зрения указатель любого типа может ссылаться на любое место в памяти. Однако операции адресной арифметики тесно связаны с базовым типом указателей, поэтому очень важно правильно их объявить.

Ранее мы уже рассматривали операторы для работы с указателями - * и &. Первый указатель называется оператором разыменования, а второй – оператор получения адреса. Оператор & возвращает адрес своего операнда. Пример:

```
a = &b;
```

В результате в указатель **a** будет записан адрес переменной **b**. Заметьте – адрес, а не значение. Адрес и значение переменной никак не связаны друг с другом. Адрес – это фактическое расположение переменной в памяти, а значение она может содержать абсолютно любое, соответствующее ее типу. Итак, запомните, & - это адрес. Следовательно, предыдущий оператор может быть прочитан как "присвоить указателю **a** адрес переменной **b**". Далее мы будем считать, что переменная **b** хранится в ячейке с адресом 1004 и содержит значение 100.

Оператор * - антипод оператора &. Данный оператор возвращает значение, хранящееся по указанному адресу. Пример:

```
c = *a;
```

В данном случае в переменную c будет записано значение 100, поскольку по адресу 1004 хранится именно это значение (рис. 4.1).

Учитывайте, что приоритет операторов * и & выше, чем приоритет всех арифметических операторов, за исключением унарного минуса.

> **Примечание.** Необходимо убедиться, что указатель всегда ссылается на переменную правильного типа. Например, если в программе объявлен указатель на целочисленную переменную, компилятор полагает, что адрес, который в нем содержится, относится к переменной типа `int`, независимо от того, так ли это на самом деле.

Указатель можно присвоить другому указателю. Рассмотрим пример:

```
int z;
int *p1, *p2;
```

```
p1 = &z;
p2 = p1;

printf("Переменная по адресу %p содержит значение %d", p2, z);
```

4.3. Адресная арифметика

К указателям можно применять две арифметических операции – сложение и вычитание. Представим, что `p1` ссылается на целую переменную, которая занимает в памяти 2 байта и находится по адресу 1004. Тогда после выражения `p1++` переменная `p1` будет равна не 1005, а 1006, поскольку при увеличении указателя на единицу он будет ссылаться на следующее целое число. Все сказанное верно и для оператора декремента. То есть оператор `p1--` присвоит в `p1` значение 1002.

Сформулируем следующие правила адресной арифметики:

- При увеличении указатель ссылается на ячейку, в которой хранится следующий элемент базового типа.
- При уменьшении он ссылается на предыдущий элемент.

Для указателей на символы сохраняются правила "обычной" арифметики, поскольку размер символов равен 1 байт. Все остальные указатели увеличиваются или уменьшаются на длину соответствующих переменных, на которые они ссылаются. Это гарантирует, что указатели всегда будут ссылаться на элемент базового типа.

Действия над указателями не ограничиваются инкрементом и декрементом. К указателям можно добавлять (и, соответственно, из указателей – вычитать) целые числа:

```
p1 = p1 + 10;
```

Это позволяет определить количество объектов базового типа, расположенных между двумя указателями. Все другие арифметические операции запрещены. В частности, указатели нельзя складывать, умножать и делить. К

ним нельзя применять побитовые операции, суммировать их со значениями типа float и т.п., а также вычитать такие значения из них.

В качестве примера рассмотрим доступ к элементам массива с использованием указателей:

```
int data[5], i;
printf("Введите 5 элементов: ");

for(i = 0; i < 5; ++i)
  scanf("%d", data + i);

printf("Содержимое массива: \n");
for(i = 0; i < 5; ++i)
   printf("%d\n", *(data + i));
```

Как видите, мы вывели содержимое массива без использования квадратных скобок.

Указатели можно сравнивать друг с другом, например:

```
if (p1 < p2)
  printf("Указатель p1 содержит меньший адрес, чем p2\n");
```

Как правило, указатели сравниваются между собой, когда они ссылаются на один

и тот же объект, например, массив. Далее, при рассмотрении динамического выделения памяти, мы рассмотрим сравнение указателей на практике. А пока рассмотрим еще один пример. Пользователь вводит три числа, которые мы сохраняем в переменные a, b, c соответственно. Затем, эти переменные передаются функции `cyclicSwap()`. Вместо передачи значений переменных мы передаем в функцию адреса переменных - посмотрите, что возле имени переменной есть *, а функцию мы передаем переменные с помощью &.

После того как `cyclicSwap()` закончит работу, переменные будут поменяны местами и в основной программе. Код основной программы выглядит так:

```
    printf("Введите a b и c: ");
    scanf("%d %d %d",&a,&b,&c);

    printf("До замены:\n");
    printf("a = %d \nb = %d \nc = %d\n",a,b,c);

    cyclicSwap(&a, &b, &c);

    printf("После:\n");
    printf("a = %d \nb = %d \nc = %d\n",a, b, c);
```

А вот код функции `cyclicSwap()`:

```
void cyclicSwap(int *a,int *b,int *c)
{

    int temp;

    // меняем местами переменные
    temp = *b;
    *b = *a;
    *a = *c;
    *c = temp;
}
```

4.4. Массивы указателей

Ранее было показано, как получить доступ к элементам массива с помощью указателей. Язык Си позволяет формировать также и массивы указателей:

```
int *x[5];
```

Чтобы присвоить адрес целой переменной с третьему элементу массива, используйте оператор:

```
x[2] = &c;
```

Чтобы извлечь значение переменной с с помощью указателя `x[2]`, используйте выражение `*x[2]`.

Массив указателей можно передать в обычную функцию:

```
void output(int *a[])
{
  int z;
  for (z=0; z<5; z++) printf("%d ", *a[z]);
}
```

4.5. Инициализация указателей

Если нестатический локальный указатель объявлен, но не инициализирован, его значение остается неопределенным. (Глобальные и статические локальные указатели автоматически инициализируются нулем.) При попытке применить указатель, содержащий неопределенный адрес, ваша программа может разрушиться. В Windows вы получите сообщение об ошибке, в Linux – аналогично, но еще и будет сформирован так называемый coredump-файл – профессионалы используют этот "черный ящик", чтобы понять, что пошло не так. К счастью, современные операционные системы содержат надежные механизмы защиты памяти и крах вашей программы не должен повлиять ни на другие программы, ни на операционную систему.

При работе с указателями большинство профессиональных программистов придерживаются следующего соглашения: указатель, не ссылающийся на конкретную ячейку памяти, должен быть равен нулю. По определению любой нулевой указатель ни на что не ссылается и не должен использоваться. Однако это еще не гарантирует безопасности. Использование нулевого указателя – всего лишь общепринятое соглашение.

Пример:

```
struct lnode *new = NULL;
```

Если указатель должен указывать на строку, то можно инициализировать его так:

```
char *s = "Hello";
```

Все компиляторы языка Си создают так называемую таблицу строк (string table), в которой хранятся строковые константы, используемые в программе. Следовательно, предыдущий оператор присвоит указателю p адрес строковой константы "Hello", записанной в таблицу строк. Указатель p используется в программе как обычная строка (однако изменять его нежелательно).

4.6. Нулевой указатель (NULL)

Указатель до инициализации хранит мусор, как и любая другая переменная. Но в то же время, этот "мусор" вполне может оказаться валидным адресом. Пусть, к примеру, у нас есть указатель. Каким образом узнать, инициализирован он или нет? В общем случае никак. Для решения этой проблемы был введен макрос NULL библиотеки `stdlib`.

Принято при определении указателя, если он не инициализируется конкретным значением, делать его равным NULL.

```
int *ptr = NULL;
```

По стандарту гарантировано, что в этом случае указатель равен NULL, и равен нулю, и может быть использован как булево значение false. Хотя в зависимости от реализации NULL может и не быть равным 0 (в смысле, не равен нулю в побитовом представлении, как например, `int` или `float`).

Это значит, что в данном случае

```
int *ptr = NULL;
if (ptr == 0) {
...
```

```
}
```

вполне корректная операция, а в случае

```
int a = 0;
if (a == NULL) {
...
}
```

поведение не определено. То есть указатель можно сравнивать с нулем, или с NULL, но нельзя NULL сравнивать с переменной целого типа или типа с плавающей точкой.

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

void main() {
    int *a = NULL;
    unsigned length, i;

    printf("Введите длину массива: ");
    scanf("%d", &length);

    if (length > 0) {
        //При выделении памяти возвращается указатель.
        //Если память не была выделена, то возвращается NULL
        if ((a = (int*) malloc(length * sizeof(int))) != NULL)
{
            for (i = 0; i < length; i++) {
                a[i] = i * i;
            }
        } else {
            printf("Не могу выделить память");
        }
    }
     //Если переменная была инициализирована, то очищаем её
    if (a != NULL) {
        free(a);
```

```
    }
    getch();
}
```

4.7. Указатели на функции

Несмотря на то что функция не является переменной, она располагается в памяти, и, следовательно, ее адрес можно присваивать указателю. Этот адрес считается точкой входа в функцию. Именно он используется при ее вызове. Поскольку указатель может ссылаться на функцию, ее можно вызывать с помощью этого указателя. Это позволяет также передавать функции другим функциям в качестве аргументов

Адрес функции задается ее именем, указанным без скобок и аргументов. Чтобы разобраться в этом механизме, рассмотрим небольшой пример.

```
void test(char *x, *y, int(*cmp)(const char *, const char *))
{
  printf("Сравнение строк\n");
  if(!(*cmp(x,y)) printf("Равны");
  else printf("Не равны");
}
```

Использовать нашу функцию можно так:

```
char s1[50], s2[50];
// объявляем указатель на функцию
int (*p)(const char *, const char *);

// инициализируем указатель на функцию
p = strcmp;

// читаем строки
gets(s1);
gets(s2);

// вызываем нашу функцию test
test(s1, s2, p);
```

При вызове функция `test()` получает два указателя на символьные переменные и указатель на функцию. Соответствующие аргументы объявлены в ее заголовке. Обратите внимание на то, как объявлен указатель на функцию. Эту форму объявления следует применять для любых указателей на функции, независимо от того, какой тип имеют их аргументы и возвращаемые значения. Объявление *cmp заключено в скобки для того, чтобы компилятор правильно его интерпретировал.

Выражение `(*cmp)(x,y)` внутри функции `test()` означает вызов функции `strcmp()`, на которую ссылается указатель cmp. Скобки нужны, поскольку это один из способов вызвать функцию с помощью указателя. Конечно, можно вызвать функцию так: `cmp(x, y)`, но такой способ является менее информативным, поскольку непонятно, либо это какая-то пользовательская функция или библиотечная. А при использовании скобок сразу понятно, что вызывается функция через указатель.

Обратите внимание, что нашу функцию `test()` можно вызвать и без указателей:

```
test(s1, s2, strcmp);
```

В этом случае можно обойтись без лишнего указателя.

4.8. Динамическое распределение памяти

Одна из тесно связанных с указателями тематик – динамическое распределение памяти (dynamic allocation). Благодаря этому механизму программа можем получать необходимую ей память в ходе выполнения, а не на этапе компиляции.

Память для глобальных переменных выделяется па этапе компиляции. Локальные переменные хранятся в стеке. Следовательно, в ходе выполнения программы невозможно объявить новые глобальные или локальные переменные. И все же иногда в ходе выполнения программы возникает необходимость выделить дополнительную память, размер которой заранее не

известен. Наиболее частый пример – динамические структуры, такие как связные списки или деревья (будут рассмотрены во второй части книги). Такие структуры являются динамическими по своей природе и могут увеличиваться или уменьшаться по мере необходимости.

В языке Си есть функции динамического распределения памяти. Точнее две функции распределения памяти – `malloc()` и `calloc()` и одна функция освобождения памяти – `free()`. В некоторых компиляторах есть и другие функции, но эти стандартные функции есть всегда, независимо от стандарта, компилятора, платформы и т.д.

Память, выделяемая функциями динамического распределения, находится в куче (heap), которая представляет собой область свободной памяти, расположенную между кодом программы, сегментом данных и стеком.

При каждом вызове `malloc()` выделяется дополнительный участок памяти, который можно вернуть обратно в кучу с помощью функции `free()`.

Прототип функции `malloc()`:

```
void *malloc(size_t количество_байтов)
```

Параметр количество байтов задает размер памяти, которую необходимо выделит. Тип `size_t` определен в заголовочном файле `stdlib.h` как целое число без знака. Функция `malloc()` возвращает указатель типа `void *`. Это означает, что его можно присваивать указателю любого типа. В случае успеха функция `malloc()` возвращает указатель на первый байт памяти, выделенной в куче. Если размера кучи недостаточно для успешного выделения памяти, функция `malloc()` возвращает нулевой указатель.Также полезна функция `calloc()`, позволяющая выделить память под данные конкретного типа данных, ее прототип выглядит так:

```
void *calloc(size_t num, size_t size)
```

Размер выделенной памяти будет равен величине num*size, где `size` задается в байтах.

Следующий пример выделяет 2000 байт непрерывной памяти:

```
char *s;
s = malloc(2000);
```

После этого указатель s будет ссылаться на первый из 2000 байт выделенной памяти.

Прототип функции `free()` выглядит так:

```
void free(void *p)
```

Здесь параметр p является указателем на участок памяти, ранее выделенный функцией `malloc()`.

В качестве примера работы с динамически выделяемой памятью рассмотрим следующий пример. Представим, что нам нужно вычислить максимум в массиве, размер которого мы заранее не знаем.

```
int i, num;
float *data;

printf("Введите количество элементов: ");
scanf("%d", &num);

// Выделяем память под 'num' элементов
data = (float*) calloc(num, sizeof(float));

if(data == NULL)
{
    printf("Ошибка выделения памяти\n.");
    exit(1);
}

printf("\n");

// Вводим элементы
for(i = 0; i < num; ++i)
{
   printf("Введите элемент %d: ", i + 1);
   scanf("%f", data + i);
}
```

```
// Ищем максимальный элемент
for(i = 1; i < num; ++i)
{
   // Сохраняем максимальный элемент
   if(*data < *(data + i))
       *data = *(data + i);
}

printf("Максимум = %.2f\n", *data);
```

Мы выделяем память функцией `calloc()` – ее удобно, поскольку сразу можем выделить нужное количество памяти, чем высчитывать это количество и передавать в `malloc()`. Далее мы вводим элементы массива и вычисляем максимум. Подобный пример уже был приведен ранее. Вот только тогда у нас массив `data` состоял из заданного количества элементов – 5, а сейчас его размер неизвестен до запуска программы. Пользователь вводит размер массива, а потом уже мы выделяем память.

Глава 5.
Функции в Си

Практически в любом языке программирования есть подпрограммы. Подпрограммы позволяют экономить время программиста, поскольку ему не приходится писать один и тот же код несколько раз, но для разных данных. В языке Си подпрограммы называются функциями.

5.1. Синтаксис объявления функции

Синтаксис объявления функции выглядит так:

```
<тип> имя_функции (список_параметров)
{
 тело функции;
}
```

Здесь тип – это тип возвращаемого функцией значения. Функция может возвращать переменные любого типа, кроме массива. В списке параметров

перечисляются типы аргументов и их имена, разделенные запятыми. Если функция не имеет аргументов, ее список параметров пуст. В этом случае скобки все равно необходимы.

Рассмотрим функцию `checkPrimeNumber()`, которая возвращает 1, если переданное ей число является простым или 0 – в противном случае:

```
int checkPrimeNumber(int n)
{
    int i, flag = 1;

    for(i=2; i<=n/2; ++i)
    {

    // Условие для не простого числа
        if(n%i == 0)
        {
            flag = 0;
            break;
        }
    }
    return flag;
}
```

Наша функция называется `checkPrimeNumber()` и возвращает значение типа `int`. В качестве параметра передается целое число с идентификатором n. Функция проверяет, является ли число простым и возвращает результат проверки с помощью оператора `return` (далее он будет рассмотрен подробнее).

5.2. Область видимости функции

Каждая функция – это отдельный блок операторов. Код функции является закрытым и на него нельзя повлиять извне – из другой функции или из основной программы (которая, впрочем, в Си тоже является функцией). Код закрыт двухстороннее: вы не можете из функции как-либо изменить код

другой функции ли же перейти в другую функцию посредством оператора `goto`.

Код, образующий тело функции, скрыт от остальной части программы, если он не использует глобальные переменные. Этот код не может влиять на другие функции, а они, в свою очередь, не могут влиять на него. Иначе говоря, код и данные, определенные внутри некой функции, никак не взаимодействуют с кодом и данными, определенными в другой функции, поскольку их области видимости не перекрываются.

Переменные, определенные внутри функции, называются локальными (local variables). Они создаются при входе в функцию и уничтожаются при выходе из нее. Иными словами, локальные переменные не сохраняют свои значения между вызовами функции. Единственное исключение из этого правила составляют статические локальные переменные. Они хранятся в памяти вместе с глобальными переменными, однако область их видимости ограничена функцией, в которой они объявлены.

Если в функции объявлена локальная переменная с таким же именем, как и глобальная, то функция будет использовать свою локальную версию.

5.3. Формальные параметры функции

5.3.1. Список параметров

Аргументы функции объявляются в ее заголовке в виде переменных, принимающих значения. Такие переменные называются формальными параметрами. Как и локальные переменные, они создаются при входе в функцию и уничтожаются при выходе из нее. Ранее был показан пример функции с одним параметром – `int n`. Если нужно указать несколько параметров, то они перечисляются через запятую, например:

```
int gcd(int n1, int n2)
{
    if (n2 != 0)
       return gcd(n2, n1%n2);
```

```
    else
       return n1;
}
```

Функция `gcd()` вычисляет НОД двух чисел n1 и n2. Оба параметра типа int, перечисление параметров осуществляется с помощью запятой.

Несмотря на то, что эти переменные должны, в основном, получать значения аргументов, передаваемых функции, их можно использовать наравне с другими локальными переменными, в частности, присваивать им значения или использовать в любых выражениях.

5.3.2. Параметры по ссылке и по значению

Параметры можно передавать по ссылке (call by reference) и по значению (call by value). Обычно используется передача параметра по значению. В этом случае формальному параметру функцию присваивается копия значения аргумента и любые изменения копии никак не отражаются на оригинале.

Ранее мы как раз и использовали передачу параметров по значению. Рассмотрим небольшой пример:

```
int change(int n) {
    n = 2;
    printf("Change = %d\n", n);
    return n;
}
```

Функция принимает параметр n, далее изменяет его значение, выводит его на стандартный вывод и возвращает оператором `return`.

Далее следует код основной программы:

```
    int n = 1;

    printf("n = %d\n", n);
    change(n);
    printf("n = %d\n", n);
```

Мы выводим значение n, вызываем функцию change(), а затем снова выводим значение n. Вывод будет таким:

```
n = 1
Change = 2
n = 1
```

Как видите, функция change() никак не изменила значение параметра. В большинстве случаев этого и не нужно, ведь она должна принять данные, обработать их и вернуть какой-то результат. А что уже будет делать с результатом основная программа – ее проблемы. В нашем случае функция change() результат то вернула, но основная программа его не сохранила. Можно было бы переписать оператор так:

```
n = change(n);
```

И тогда бы в переменную (объявленную в функции main()) было бы записано значение 2.

При передаче параметров по ссылке у функции есть возможность изменить значение переданного аргумента, поскольку в функцию передается не значение, а адрес аргумента. Внутри функции этот адрес открывает доступ к фактическому аргументу и все изменения будут отображены на аргументе. Если нужно передавать параметры по ссылке, то указывают * перед именем параметра. А при вызове функции нужно указать & перед именем параметра. Рассмотрим код нашей функции change():

```
int change(int *n) {
    *n = 2;
    printf("Change = %d\n", *n);
    return n;
}
```

Внутри функции нужно использовать оператор * при каждом обращении к аргументу – независимо от того, хотите ли вы прочитать или записать его значение.

А в основной программе используйте & для передачи аргументов функции:

```
int n = 1;

printf("n = %d\n", n);
change(&n);
printf("n = %d\n", n);
```

Вывод программы будет такой:

```
n = 1
Change = 2
n = 2
```

5.3.3. Передача массива в качестве параметра

Если аргументом функции является массив, ей передается его адрес. Эта ситуация представляет собой исключение из общепринятого правила передачи параметров по значению. Функция, получившая массив, получает доступ ко всем его элементам и может его модифицировать.

Рассмотрим функцию, которая ничего не модифицирует, а просто выводит содержимое массива:

```
void display(int num[10])
{
  int i;
  for (i=0; i<10; i++) printf ("%d", num[i]);
}
```

Использовать ее нужно так:

```
int  t [10], i;
for (i=0; i<10; ++i) t[i]=i;
display(t);
```

Сначала мы заполняем массив данными (цикл `for`), а затем выводим массив на стандартный вывод функцией `display()`.

Теперь рассмотрим пример изменения массива. Для разнообразия будем работать с символьными, а не числовыми данными:

```
void char_upper(char *string)
{
  register int t;
  for(t=0; string[t]; ++t) {
      string[t] = toupper(string[t]);
      // для контроля функции можно включить вывод символов
      //putchar(string[t]);
  }
}
```

Использовать функцию можно так:

```
char s[80];

// Рекомендуется использовать функцию gets_s() - безопасный
// вариант gets(),
// но некоторые старые компиляторы могут ее не понимать,
// поэтому используем
// небезопасный вариант
gets(s);
// изменяем регистр букв
char_upper(s);
// выводим массив символов
printf("%s\n", s);
```

Ранее, в главе 3, мы рассмотрели программу умножения двух матриц. Давайте ее перепишем с использованием функций. Функций у нас будет три:

- `enterData()` - ввод данных от пользователя.
- `multiplyMatrices()` - умножение двух матриц.
- `display()` - отображение результата матрицы после умножения.

Благодаря использованию функций код основной программы будет очень компактным:

```
  int firstMatrix[10][10], secondMatrix[10][10], mult[10][10],
rowFirst, columnFirst, rowSecond, columnSecond, i, j, k;

  printf("Введите количество строк и колонок матрицы 1: ");
  scanf("%d %d", &rowFirst, &columnFirst);

  printf("Введите количество строк и колонок матрицы 2: ");
  scanf("%d %d", &rowSecond, &columnSecond);

  // Проверяем, можем ли мы умножить 2 матрицы
  while (columnFirst != rowSecond)
  {
     printf("Ошибка! К-во колонок первой матрицы не равно
            количеству строк второй\n");
     printf("Введите количество строк и колонок матрицы 1: ");
     scanf("%d%d", &rowFirst, &columnFirst);
     printf("Введите количество строк и колонок матрицы 2:");
     scanf("%d%d", &rowSecond, &columnSecond);
  }

  // Вводим данные
        enterData(firstMatrix, secondMatrix, rowFirst,
                  columnFirst, rowSecond, columnSecond);

        // Умножаем 2 матрицы
        multiplyMatrices(firstMatrix, secondMatrix, mult,
           rowFirst, columnFirst, rowSecond, columnSecond);

        // Выводим результат
        display(mult, rowFirst, columnSecond);
```

Функция ввода данных будет выглядеть так:

```
void enterData(int firstMatrix[][10], int secondMatrix[]
[10], int rowFirst, int columnFirst, int rowSecond, int
columnSecond)
{
   int i, j;
   printf("\nВведите элементы матрицы 1:\n");
   for(i = 0; i < rowFirst; ++i)
   {
      for(j = 0; j < columnFirst; ++j)
      {
         printf("Введите элемент a%d%d: ", i + 1, j + 1);
         scanf("%d", &firstMatrix[i][j]);
      }
   }

   printf("\nВведите элементы матрицы 2:\n");
   for(i = 0; i < rowSecond; ++i)
   {
      for(j = 0; j < columnSecond; ++j)
      {
         printf("Введите элемент b%d%d: ", i + 1, j + 1);
         scanf("%d", &secondMatrix[i][j]);
      }
   }
}
```

Первый параметр – это первая матрица, второй – вторая, третий и четвертый – количество строк и колонок первой матрицы, 5-ый и 6-ой – количество строк и колонок второй матрицы.

Основная функция – это функция умножения матриц. Она принимает практические такие параметры, как и функция ввода данных, но третий параметр у нее – матрица, в которой функция будет хранить результат умножения.

```
void multiplyMatrices(int firstMatrix[][10], int secondMatrix[]
[10], int mult[][10], int rowFirst, int columnFirst, int
rowSecond, int columnSecond)
{
   int i, j, k;
```

```
   // Заполняем результирующую матрицу нулями
   for(i = 0; i < rowFirst; ++i)
   {
      for(j = 0; j < columnSecond; ++j)
      {
         mult[i][j] = 0;
      }
   }

   // Умножаем 2 матрицы и сохраняем результат в mult
   for(i = 0; i < rowFirst; ++i)
   {
      for(j = 0; j < columnSecond; ++j)
      {
         for(k=0; k<columnFirst; ++k)
         {
            mult[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
         }
      }
   }
}
```

Самая простая – это функция вывода матрицы. Ей нужно передать матрицу, а также количество строк и колонок:

```
void display(int mult[][10], int rowFirst, int columnSecond)
{
   int i, j;
   printf("\nРезультат:\n");
   for(i = 0; i < rowFirst; ++i)
   {
      for(j = 0; j < columnSecond; ++j)
      {
         printf("%d  ", mult[i][j]);
         if(j == columnSecond - 1)
            printf("\n\n");
      }
   }
}
```

5.3.4. Аргументы функции main()

Функция `main()` является точкой входа в программу, как уже отмечалось ранее. Этой функции также можно передать параметры, но параметры вполне определенные – `argv` и `argc`. С помощью этих параметров можно передать данные в нашу программу извне, то есть из командной строки.

Целочисленный параметр `argc` содержит количество аргументов командной строки. Его значение не может быть меньше 1, поскольку имя программы считается первым аргументом. Параметр `argv` представляет собой указатель на массив символьных указателей. Каждый элемент этого массива ссылается на аргумент командной строки. Все аргументы командной строки являются строками — введенные числа должны конвертироваться в соответствующее внутреннее представление.

Рассмотрим простую программу, которая выводит на экран строку, переданную программе в качестве параметров или же сообщение о том, что обязательный параметр не указан.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  if(argc!=2) {
    printf("Вы забыли указать обязательный параметр!\n");
    exit(1);
   }
   printf("%s\n", argv[1]);
   return 0;
}
```

Если аргументов несколько, то вывести все аргументы позволяет следующий код:

```
  int i;
  for( i = 0 ; i < argc; i++) {
      printf("Аргумент %d: %s\n", i, argv[i]);
  }
  if(argc == 1) {
      printf("Аргументы командной строки не заданы\n");
  }
```

Напишем программу, которая открывает указанные пользователем в командной строке файлы на запись или добавление и записывает (добавляет) туда одну и туже информацию, которую пользователь вводит с клавиатуры в процессе выполнения программы:

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

main (int argc, char **argv) {
   int i, ch;
   FILE *f[5];

   if (argc < 3 || argc > 7) {
      puts("Неверное количество параметров");
      return 1;
   }

   if (strcmp(argv[1], "-w") != 0 && strcmp(argv[1], "-a") != 0)
{
      puts("Первый параметр может быть либо -w, либо -a");
      return 2;
   }

   for (i=0; i < argc-2; i++){
      f[i] = fopen(argv[i+2], argv[1]+1);
      if (f[i] == NULL) {
         printf("Файл %s нельзя открыть\n", argv[i+2]);
         return 3;
      }
   }
       // Читаем ввод и пишем в файл
   while ((ch = getchar()) != EOF)
      for (i=0; i < argc-2; i++)
         putc(ch,f[i]);
       // закрываем файлы
   for (i=0; i < argc-2; i++)
      fclose(f[i]);

   return 0;
}
```

5.4. Оператор return

Оператор `return` используется для немедленного выхода из функции. Также он передает значение, вычисленное функцией.

Существует два способа прекратить выполнение функции и вернуть управление вызывающему модулю. В первом случае функция полностью выполняет свои операторы,

и управление достигает закрывающей фигурной скобки }. Такой способ используется редко по следующим причинам:

1. Функция для того и пишется, чтобы вернуть какой-то результат. А фигурная скобка сама по себе никакого результата не возвращает.
2. Стандарт C99 предусматривает обязательное наличие оператора `return`. Исключение из этого правила – функции с типом возвращаемого значения `void`.
3. Снижается наглядность функции.

Функция может иметь несколько операторов return, однако, как только будет выполнен первый из них, выполнение функции будет завершено.

Все функции, за исключением функций, объявленных со спецификатором `void`, с помощью оператора `return` возвращают некий результат. В соответствии со стандартом C89, если функция без оператора `return` возвращает некий "мусор" – случайное и ни с чем не связанное значение. Стандарт C99 предусматривает, что все функции должны использовать оператор `return`.

Иными словами, если в объявлении указано, что функция возвращает некое значение, любой оператор `return`, расположенный в ее теле, должен быть связан с определенной переменной. Однако если поток управления достигнет конца функции и при этом там не будет оператора `return`, то тоже будет возвращен "мусор". Эта ошибка не является синтаксической, но должна быть исключена полностью – так программы никто не пишет.

Если функция не объявлена со спецификатором void, ее можно использовать в качестве операнда в любом выражении. Таким образом, фрагмент программы, приведенный ниже, является совершенно правильным:

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

y = 10;
x = y * sum(20, 30);
```

Функция main() возвращает целое число вызвавшему ее процессу, как правило, операционной системе. Вызвав функцию exi() с этим аргументом, вы получите тот же самый эффект. С формальной точки зрения, если функция main() ничего не возвращает явным образом, то вызывающему процессу передается неопределенное значение. На практике многие компиляторы языка Си по умолчанию возвращают число 0, но полагаться на это не стоит, поскольку это снизит машино-независимость вашей программы. Обычно статус 0 обозначает нормальное завершение программы – без ошибок. Отличное от 0 состояние означает, что произошла ошибка, например, не найден один из файлов, переданных программе в качестве параметра. Возвращаемые программой значения обычно описаны в документации по программе.

5.5. Тип void

В языке Pascal есть два типа подпрограмм – функции и процедуры. Первые возвращают какое-то значение, вторые – не возвращают никаких значений. В Си такого разнообразия нет, но для функций, которые не возвращают ничего предусмотрен тип void.

Например, иногда нужна функция, которая выводит на экран набор разделителей, в этом случае она не должна ничего возвращать:

```
void print_stars(int n)
{
  int j;
  for (j=0; j<n; j++)
    printf("*");
  printf("\n");
}
```

В самых первых вариантах языка Си тип `void` не поддерживался, вместо него у функций был тип `int`, если функция не возвращала значение какого-то другого типа.

5.6. Рекурсия

Рекурсия - это явление, когда функция вызывает саму себя. Начинающим программистам не рекомендуется использовать рекурсию, поскольку она может привести к переполнению стека программы, если не предусмотреть корректного условия выхода из рекурсии, то есть завершения функции.

Когда функция вызывает саму себя, в стеке размещается новый набор се локальных переменных и параметров, и функция выполняется с начала. Рекурсивный вызов не создает новую копию функции. Копируются лишь переменные, с которыми она работает. При каждом рекурсивном возврате управления копии переменных и параметров удаляются из стека, а выполнение программы возобновляется с места вызова функции.

Обычно рекурсивные функции незначительно уменьшают размер кода и ненамного повышают эффективность использования памяти по сравнению с ее итеративными аналогами. Кроме того, рекурсивные версии большинства функций выполняются несколько медленнее, чем их итеративные эквиваленты. Большое количество рекурсивных вызовов может вызвать переполнение стека, поскольку при каждом вызове функции в стеке размещается новый набор ее локальных переменных и параметров.

При этом программист может пребывать в полном неведении, пока не произойдет аварийное прекращение работы программы. Основное преимущество рекурсивных функций заключается в том, что они упрощают и делают

нагляднее некоторые алгоритмы. Например, алгоритм быстрой сортировки трудно реализовать итеративным способом.

Для знакомства с рекурсией давайте рассмотрим простую программу, вычисляющую сумму натуральных чисел.

```
#include <stdio.h>
int addNumbers(int n);

int main()
{
    int num;
    printf("Введите целое положительное число: ");
    scanf("%d", &num);
    printf("Сумма = %d\n",addNumbers(num));
    return 0;
}

int addNumbers(int n)
{
    if(n != 0)
        return n + addNumbers(n-1);
    else
        return n;
}
```

Давайте разберемся, что происходит. Допустим, пользователь ввел цифру 2. Функция `addNumbers()` вызывается с параметром 2. Функция проверяет, что переданное ей значение не равно 0 и поэтому возвращает значение:

```
2 + addNumbers(1);
```

Снова вызывается функция, но уже с параметром 1. Так как 1 - не равно 0, то функция возвращает значение:

```
1 + addNumbers(0);
```

Функция проверяет, что переданное ей значение равно 0 и возвращает его. Теперь что у нас есть:

- 0
- 1
- 2

Сумма всех этих значений равна 3.

Очень важно предусмотреть условие выхода из рекурсии. В нашем случае условием выхода является n = 0: функция просто возвращает 0 и не вызывает снова саму себя.

5.7. Прототипы функций

В языке Си все функции должны быть объявлены до своего первого вызова. Следовательно, мы получим длинную простыню всех функций, предшествующих функции `main()` – основной функции программы, что делает код программы не очень удобным для чтения.

Для повышения удобства чтения кода в языке Си используются прототипы функций. Работают они так – сначала вы приводите только прототипы функций (описание, содержащее тип функции, ее имя и список параметров), далее идет функция `main()`, а после нее – реальные описания функций, то есть прототип функции и ее код.

Указывать имена параметров не обязательно. Однако они дают компилятору возможность

обнаружить имя параметра, тип которого не совпадает с именем аргумента, поэтому рекомендуется использовать имена аргументов в прототипах.

Рассмотрим программу, выполняющую преобразование между двоичной и десятичной системами счисления. Для преобразования двоичного числа в десятичное используется функция `convertBinaryToDecimal()`. Для обратного преобразования – функция `convertDecimalToBinary()`. Теперь посмотрим на нашу программу:

- Сначала мы подключаем необходимые заголовочные файлы

- Затем – приводим прототипы функций
- Описываем функцию `main()` – основной код программы
- После `main()` – описываем наши функции.

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

int convertBinaryToDecimal(long long n);
long long convertDecimalToBinary(int n);

int main()
{
    long long n;
    printf("Введите двоичное число: ");
    scanf("%lld", &n);
    printf("%lld (binary) = %d (decimal)\n", n,
           convertBinaryToDecimal(n));

    printf("Введите десятичное число: ");
    scanf("%d", &n);
    printf("%d (decimal) = %lld (binary)\n", n,
           convertDecimalToBinary(n));
    return 0;
}

int convertBinaryToDecimal(long long n)
{
    int decimalNumber = 0, i = 0, remainder;
    while (n!=0)
    {
        remainder = n%10;
        n /= 10;
        decimalNumber += remainder*pow(2,i);
        ++i;
    }
    return decimalNumber;
}
```

```
long long convertDecimalToBinary(int n)
{
    long long binaryNumber = 0;
    int remainder, i = 1, step = 1;

    while (n!=0)
    {
        remainder = n%2;
        n /= 2;
        binaryNumber += remainder*i;
        i *= 10;
    }
    return binaryNumber;
}
```

Для компиляции программы нужно использовать опцию `-lm`:

```
gcc program.c -o program -lm
```

Опция `-lm` обеспечивает подключение библиотеки math (заголовочный файл `math.h`).

Глава 6.
Пользовательские типы в Си

Язык Си содержит средства объявления пользовательских типов. К таковым относятся: структуры, объединения, перечисления и ключевое слово `typedef`. В отличие от C++ язык Си не поддерживает объектно-ориентированный подход к программированию, поэтому объекты и классы не поддерживаются.

6.1. Структуры

6.1.1. Объявление структуры

Структура – это набор переменных (часто разных типов), объединенных под одним именем. Структура обеспечивает удобный способ организации связанных данных. Например, можно создать структуру, содержащую

данные о студенте, а потом создать массив таких структур, чтобы хранить данные о группе или даже о целом курсе.

Синтаксис объявления структуры следующий:

```
struct имя_типа_структуры {
тип имя_члена;
...
тип имя_члена;
} имена_экземпляров_структуры;
```

Члены структуры тесно связаны друг с другом логически. Возьмем нашу структуру `student`: у нее будет три члена - `name` (строка), `ticket` (целое), `group` (целое).

```
struct student
{
    char name[50];
    int ticket;
    int group;
};
```

Структура хранит информацию об имени студента, выбранном билете и номере группы студента.

В приведенном выше фрагменте еще не создана ни одна переменная. В нем лишь определен составной тип данных, а не сама переменная. Для того чтобы возникла реальная переменная данного типа, ее нужно объявить отдельно.

Делается это или при объявлении структуры, вот так:

```
struct student
{
    char name[50];
    int ticket;
    int group;
} s;
```

Если структура уже была объявлена ранее, тогда так:

```
struct student s;
```

После объявления переменной, компилятор выделяет память для ее членов. Всего для нашей структуры будет выделено 50 + 4 + 4 = 58 байтов памяти.

6.1.2. Доступ к членам структуры

Для доступа к членам структуры используется следующий синтаксис:

имя_экземпляра_структуры.имя_члена

Например:

```
s.group = 45;
```

Вот как можно заполнить нашу структуру путем ввода с клавиатуры:

```
printf("ФИО: ");
scanf("%s", s.name);

printf("Группа: ");
scanf("%d", &s.group);

printf("Билет: ");
scanf("%d", &s.ticket);
```

6.1.3. Массивы структур и динамическое выделение памяти

Как уже было отмечено, можно создать массив структур, в котором будет хранить информацию о целой группе студентов:

```
struct student
{
    char name[50];
    int ticket;
    int group;
} s[30];
```

Далее можно заполнить массив структур в цикле так:

```
for(i=0; i<30; ++i)
    {
        printf("\nВведите номер билета: ");
        scanf("%s",s[i].ticket);

        printf("Введите имя: ");
        scanf("%s",s[i].name);

        printf("Введите группу: ");
        scanf("%d",&s[i].group);

        printf("\n");
    }
```

Все хорошо, но, что делать, если мы не знаем точного количества структур? Выделить память под очень большое количество структур - неправильно. Можно, например, объявить массив структур так:

```
struct student
{
    char name[50];
    int ticket;
    int group;
} s[1024];
```

Но что делать, если придется заполнить информацию о 1025-ом студенте? Мы получим ошибку. А до этого, когда количество студентов будет меньше, мы будем просто нерационально использовать память компьютера.

Выход есть, и он заключается в динамическом выделении памяти. Мы можем выделить ровно столько памяти, столько нам будет нужно, с помощью функции `malloc()`:

```
ptr = (struct course*) malloc (noOfRecords * sizeof(struct
course));
```

Здесь `course` - это наша структура (мы должны знать ее тип), `noOfRecords` - количество записей, которое введет пользователь (мы его не знаем), а функция `sizeof()` возвращает объем памяти, необходимый для хранения одной структуры.

Рассмотрим пример:

```
struct course
{
   int mark;
   char subject[30];
};

struct course *ptr;
int i, noOfRecords;

printf("Введите к-во записей: ");
scanf("%d", &noOfRecords);

// Выделяем память
ptr = (struct course*) malloc (noOfRecords * sizeof(struct course));

for(i = 0; i < noOfRecords; ++i)
{
       printf("Введите название предмета и оценку: ");
       scanf("%s %d", &(ptr+i)->subject, &(ptr+i)->mark);
}

printf("\nРезультат:\n");

for(i = 0; i < noOfRecords ; ++i)
   printf("%s\t%d\n", (ptr+i)->subject, (ptr+i)->mark);
```

6.1.4. Передача структур функциям

Структуры можно передавать функциям. При этом можно как передавать члены структур, так и все структуры целиком.

Пусть у нас есть структура:

```
structure user
{
  char login[50];
  char pswd[50];
  int active;
} frenk;
```

Вот так можно передать члены структуры в функцию:

```
f(frenk.active);     // передаем целое число
f(frenk.login);   // передаем адрес строки login
f(frenk.login[2]);   // передаем символ строки login
```

Передать всю структуру можно так:

```
int is_active(struct user u){
  return u.active;
}
```

Данная функция возвращает значение члена `active`. В реальном мире можно обойтись, конечно же, без этой функции, но мы ее создали для примера передачи структуры. В коде основной программы вызвать функцию можно так:

```
if (is_active(frenk))
  printf("Пользователь активен\n");
```

6.1.5. Указатели на структуры

На структуры можно ссылаться, как и на любой другой тип данных. Объявить указатель на структуру можно так:

```
struct user *uptr;
```

Теперь можем работать так:

```
uptr = frenk;     // создаем указатель на структуру
uptr->active = 1;   // получаем доступ к члену структуры
```

Указатели на структуры используются в двух ситуациях: для передачи структуры в функцию по ссылке и для создания структур данных, основанных на динамическом распределении памяти (например, связанных списков).

Передача структур в качестве аргументов функции имеет один существенный недостаток: в стек функции приходится копировать целую структуру. Аргументы, передаваемые по значению, копируются в стек.

Если структура невелика, дополнительные затраты памяти будут относительно небольшими. Однако, если структура состоит из большого количества членов или ее членами являются большие массивы, затраты ресурсов могут оказаться чрезмерными. Этого можно избежать, если передавать функции не сами структуры, а лишь указатели на них.

Если функции передается указатель на структуру, в стек заталкивается только ее адрес. В результате вызовы функции выполняются намного быстрее. Второе преимущество, которое проявляется в некоторых случаях, заключается в том, что, получив адрес, функция может модифицировать структуру, являющуюся ее фактическим параметром

6.2. Объединения

Объединение – это область памяти, которая используется для хранения переменных разных типов. Объявление объединения напоминает объявление структуры. Объединение тоже содержит несколько типов данных, однако эти данные занимают одну и ту же область памяти. Вот его общий вид:

```
union имя_типа
{
  тип_элемента1 имя_элемента1;
  тип_элемента2 имя_элемента2;
  тип_элемента3 имя_элемента3;
...
};
```

К членам объединения можно обращаться так же, как и к членам структур, либо через операцию "точка", либо через операцию "->" – для указателей:

```
union user
(
char login[50];
char pswd[50];
int active;
} mark;

mark.login = "mark";
printf("%s", mark.login);
```

Структуры удобно использовать при работе со встраиваемыми системами, поскольку можно задать структуру пакета в битах. Представим, что у нас есть один байт, разбитый на биты (например, настройки интерфейса RS232):

- Биты скорости 3 – 0 (4 бита)
- Стоп-бит
- Бит управления потоком
- Биты четности (всего 2 бита)

Мы можем создать следующее объединение:

```
union
{
    struct
    {
        unsigned char speed:4;
        unsigned char stop: 1;
        unsigned char control:1;
        unsigned char parity:2;
    };
    unsigned char  byte;
} control;
```

После этого – заполнить объединение данными:

```
control.speed = SPEED_2400;    // define равный 0001
control.stop = 1;
control.control  = 1;
control.parity = 1;
```

А дальше отправить так:

```
send(control.byte);
```

Все настройки будут отправлены куда надо. В этом и есть прелесть объединений.

6.3. Перечисления и typedef

С помощью оператора `typedef` можно связать новые типы данных с существующими:

```
typedef double real;
```

После такого описания можно использовать `real` (этот тип данных привычен всем, кто программировал на Pascal) вместо `double`.

Использовать `typedef` необходимо с осторожностью. Слишком много новых типов могут ухудшить читаемость программы.

Перечисляемый тип данных `enum` (или перечисление) позволяет определить список последовательных целых чисел, каждое из которых имеет собственное имя. Объявление перечисляемого типа выглядит следующим образом:

```
enum имя_типа {имя1=знач1, имя2=знач2, имя3=знач3, ...}
переменная;
```

Здесь имя1, имя2, – это имена констант. Им можно присваивать целочисленные значения. Если значения отсутствуют, то предполагается, что они последовательно увеличиваются на единицу, начинаясь с нуля. Память под эти константы во время выполнения не выделяется, поэтому удобно использовать этот оператор для создания констант, если не указывать имя_типа и переменную:

```
enum (c28=28, c29,c30,c31);
```

Тип enum часто используется, когда информацию можно представить списком целых значений, подобно номерам дней недели или месяцев в году:

```
enum months
{Jan=1,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}
current_month;

current_month=Dec;  //используется константа 12

int diff=(int)current_month-2;
//тип enum автоматически не преобразуется в int
```

Поскольку имена эквивалентны последовательным целым значениям, то с ними можно выполнять арифметические операции. Фактически в данном примере переменной `current_month` присваивается целочисленное значение 12.

Глава 7.
Ввод/вывод в Си

В языке Си ввод и вывод данных осуществляются с помощью библиотечных функций, которые работают как с консолью, так и с файлами. Именно поэтому в одной главе будет рассмотрен, как консольный, так и файловый ввод/вывод. С технической точки зрения консоль и файлы мало отличаются друг от друга, но концептуально они совершенно разные.

Прежде, чем мы приступим вам нужно знать кое-что о вводе/выводе в Си. Ни один из стандартов (даже самый современный) не предусматривает каких-либо функций для работы с графическим интерфейсом. Другими словами, используя стандартные функции языка Си, у вас не получится создать графическое приложение, содержащее окно, кнопки, меню и т.д. Также нет каких-либо графических функций, например, вывод строк, линий и т.д. Перед языком Си ставилась задача быть универсальным и работать на самых разных платформах. Графика – вещь специфическая для каждой платформы, именно поэтому графических функций нет в Си – сложно реализовать их для всех платформ.

Все функции ввода/вывода определены в заголовочном файле stdio.h. Говоря о консольных функциях, мы имеем в виду функции для ввода данных с

клавиатуры и для вывода данных на экран. Однако на самом деле эти функции работают со стандартными потоками, которые можно переназначать. Например, программа может выводить данные на стандартный вывод, то есть на консоль, а вы уже можете перенаправить их в файл или на стандартный ввод другой программы.

7.1. Консольный ввод/вывод

7.1.1. Чтение и запись символов

Начнем с самого малого – с символов. Простейшими консольными функциями ввода и вывода являются функция `getchar()`, считывающая символ с клавиатуры, и функция `puychar()`, выводящая символ на экран. Функция `getchar()` ожидает нажатия клавиши и возвращает ее значение, которое автоматически выводится на экран. Функция `putchar()` выводит символ на экран в точку, определенную текущим положением курсора. Прототипы функций выглядят так.

```
int getchar(void);
int putchar(inc c);
```

В случае ошибки `getchar()` возвращает значение EOF (End-Of-File, конец файла). Если все хорошо, она возвращает целое значение – код символа. Это значение можно присвоить переменной типа `char` (обычно так и делают). Код символа находится в младшем байте, старший, как правило, обнуляется, поэтому мы можем использовать данное целое значение как код символа.

Функция `puchar()`, хоть и объявлена с целым аргументом, обычно она вызывается с символом. На экран выводится лишь младший байт аргумента (в случае с целым значением). Функция возвращает или символ, выведенный ею на консоль, или EOF (в случае ошибки).

Типичный код использования `getchar()` выглядит так:

```
#include <stdio.h>
...
while ((c = getchar()) != EOF) {
     // делаем что-то с символом c
}
```

Что можно сделать с символом? Например, преобразовать его к верхнему регистру и вывести с помощью `putchar()`:

```
c = toupper(c);
putchar(c);
```

Аналогично, можно преобразовать к нижнему регистру – функция `tolower()`.

Прежде, чем продолжить, вы должны знать о некоторых особенностях функции `gechar()`. Обычно эта функция помещает входные данные в буфер, пока не будет нажата клавиша Enter. Такой способ называется буферизованным вводом (line-buffered input). Для того чтобы данные, которые вы ввели, действительно были переданы программе, следует нажать клавишу Enter. Кроме того, при каждом вызове функция `getchar()` вводит символы по одному, последовательно размещая их в очереди. Если программа использует интерактивный диалог, такое торможение становится раздражающим фактором. С другой стороны, эту особенность можно использовать в "мирных целях", а именно при перенаправлении ввода/вывода. Рассмотрим программу (лист. 7.1), подсчитывающую количество символов, строк и слов.

Листинг 7.1. Подсчет символов, строк и слов

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

int main(void) {
    int tot_chars = 0;      /* к-во символов */
    int tot_lines = 0;      /* к-во строк */
    int tot_words = 0;      /* к-во слов */
    int in_space = 1;
    int c, last = '\n';
```

```
    while ((c = getchar()) != EOF) {
        last = c;
        tot_chars++;
        if (isspace(c)) {
            in_space = 1;
            if (c == '\n') {
                tot_lines++;
            }
        } else {
            tot_words += in_space;
            in_space = 0;
        }
    }
    if (last != '\n') {
        /* учитываем последню строку */
        tot_lines++;
    }

    printf("Строк, Слов, Символов\n");
    printf(" %3d %3d %3d\n", tot_lines, tot_words, tot_chars);

    return 0;
}
```

Представим, что программа откомпилирована в файл `count`. Тогда ее можно использовать так:

```
cat file.txt | ./count
```

Мы выводим файл `file.txt` и перенаправляем вывод на ввод программы `count`.

С другой стороны функция `getchar()` используется довольно редко по двум причинам. Мало кто читает символ, обычно читается строка, а для этого есть гораздо более удобная функция `scanf()`. Во-вторых, если и стоит задача прочитать символ, например, при программировании какой-то игры нужно считать символ нажатой клавиши, то явно ненужно ждать нажатия Enter. Вы только представьте, что после каждого нажатия стрелок влево или вправо для перемещения объекта в игре нужно было бы нажимать Enter.

Именно поэтому программисты часто используют функцию `getch()`. Вот реальный пример использования этой функции для считывания нажатой клавиши:

```
// функция, считывающая нажатую клавишу.
void change_direction() {
  // Считываем нажатую клавишу с помощью функции getch().
  symbol = getch();
  switch (symbol) {
    // Управление у нас через wasd.
    case 'w': if (change_x != 1 || change_y != 0) {
                change_x = -1; change_y = 0;
              }
              break;
    case 'a': if (change_x != 0 || change_y != 1) {
                change_x = 0; change_y = -1;
              }
              break;
    case 's': if (change_x != -1 || change_y != 0) {
                change_x = 1; change_y = 0;
              }
              break;
    case 'd': if (change_x != 0 || change_y != -1) {
                change_x = 0; change_y = 1;
              }
              break;
#ifndef WINDOWS
    case 'q': nonblock(NB_DISABLE); std::exit(0);
#endif
    default: break;
  }
}
```

Прототип функции выглядит так:

```
int getch(void);
```

Вам нужно знать, что функция `getch()` не является стандартной для языка Си. Но, тем не менее, многие программисты ее используют. Ее особен-

ность в том, что она сразу же возвращает код нажатой клавиши – как только клавиша нажата. При этом введенный символ не отображается на экране. При желании можно использовать функцию `getche()`, которая отображает символ, введенный пользователем, на экране.

В компиляторе от компании Microsoft данная функция находится в заголовочном файле `conio.h`. В других компиляторах ее может вовсе не быть. Но при желании ее можно реализовать самостоятельно:

```
int getch() {
  return fgetc(stdin);
}
```

При рассмотрении работы с файлами вы поймете, что и к чему.

7.1.2. Чтение и вывод строк

Аналогично функциям чтения и вывода символов в языке Си предусмотрены функции ввода и вывода целых строк. Функция `gets()` читает строку, а функция `puts()` – выводит ее на экран.

Функция `gets()` считывает строку символов, введенных с клавиатуры, и размещает их по адресу, указанному в аргументе. Символы на клавиатуре набираются до тех пор, пока не будет нажата клавиша Enter. В конец строки ставится не символ перехода на новую строку, а нулевой байт. После этого функция завершает свою работу. Ошибки, допущенные при наборе строки, можно исправить с помощью клавиши Backspace.

Прототип функции `gets()` выглядит так:

```
char *gets(char *строка);
```

Пример:

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

```
...
char s[100];
gets(s);
printf("Пользователь ввел строку длиной %d символов",
strlen(s));
```

Функция `gets()` считается небезопасной, поскольку она не проверяет выход индекса массива за пределы допустимого диапазона. Это означает, что если пользователь (в нашем примере) введет строку длиннее, чем 100 символов, функция вызовет переполнение массива. Именно поэтому в стандарте C11 вместо этой функции предлагают использовать функцию `gets_s()` прототип которой выглядит так:

```
char *gets_s( char *buffer, size_t n );
```

Функция `gets_s` считывает строку из стандартного потока ввода stdin и сохраняет ее в буфере buffer. Второй параметр задает размер буфера. Строка состоит из всех символов до первого символа новой строки ("\n"). Затем перед возвратом строки функция `gets_s` заменяет символ новой строки нуль-символом ("\0"). Напротив, функция `fgets_s` (аналогичный вариант, но для работы с файлами) сохраняет символ новой строки.

Если первый символ, считанный символ конечного файла, символ `null` хранится в начале `buffer` и NULL возвращается.

Рассмотрим пример:

```
   char line[21];         // храним место для 20 символов +
'\0'
   gets_s( line, 20 );          // здесь указываем 20 символов
   printf( "Введенная строка: %s\n", line );
```

Функция `puts()` выводит на экран строку символов и переводит курсор на следующую строку экрана. Ее прототип выглядит так.

```
int puts (const char *s)
```

Функция `puts()`, как и функция `printf()`, распознает Esc-последовательности, например '\n' (новая строка). На вызов функции `puts()` тратится меньше ресурсов, чем на функцию `printf()`, поскольку она может выводить лишь строки, но не числа. Кроме того, она не форматирует вывод. Именно поэтому функцию `puts()` часто используют для оптимизации кода.

Использовать ее просто:

```
puts("Привет");
```

В случае ошибки функция возвращает EOF. Однако, сами понимаете, при выводе на консоль вероятность ошибки стремится к 0, поэтому программисты редко проверяют возвращаемое функцией значение.

Таблица 7.1 содержит основные консольные функции ввода/вывода языка Си.

Таблица 7.1. Основные функции консольного ввода/вывода

Функция	**Назначение**
getchar()	Считывает символ с клавиатуры, ждет нажатия Enter для чтения символа
getch()	Считывает символ с клавиатуры и немедленно передает управление вызвавшей ее функции. Считанный символ не выводится на экран.
getche()	Аналогична предыдущей, но считанный символ выводится на экран.
putchar()	Выводит символ на экран.
gets()	Считывает строку с клавиатуры, не проверяет размер буфера. Рекомендуется вместо нее использовать `get_s()` или же задавать очень большой буфер, значительно превышающий возможный ввод пользователя (но это не рационально с точки зрения использования ресурсов).

gets_s()	Безопасная версия gets(), проверяет размер буфера.
puts()	Выводит строку на консоль.

7.1.3. Форматированный вывод: функция printf()

Язык Си содержит два мощнейших инструмента форматированного ввода/вывода – это функции `printf()` и `scanf()`. Первая осуществляет вывод данных в указанном программистом формате, вторая – ввод данных. Обе функции могут работать с любыми встроенными типами данных.

Прототип функции `printf()` выглядит так:

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

Первый аргумент функции `printf()` – это строка формата или управляющая строка. В ней кроме обычных символов могут содержаться спецификаторы формата, указывающие данные какого формата будут выведены. После первого параметра выводятся аргументы разных типов. Количество спецификаторов должно соответствовать количеству аргументов после строки формата. Спецификаторы формата функции `printf()` приведены в таблице 7.2.

Таблица 7.2. Спецификаторы формата

Спецификатор	Назначение
%c	Символ
%d	Десятичное знаковое целое число (d = digital)
%i	То же самое, что и %d (i = int)
%e	Экспоненциальный формат (с буквой e, при этом e – строчная)
%E	Экспоненциальный формат (E – прописная)
%f	Число с плавающей запятой

%g	Будет применен спецификатор или %f или %e – в зависимости от того, какой будет короче
%G	Или %f или %E, смотря что короче
%o	Восьмеричное число без знака
%s	Строка
%u	Целое число без знака (u – unsigned)
%x	Шестнадцатеричное число (x – от анг. heX), строчные буквы
%X	Шестнадцатеричное число, прописные буквы
%p	Указатель
%n	Указатель на целую переменную
%%	Используется для вывода знака %

Рассмотрим несколько примеров:

```
// Вывод символа
char c = 'S';
printf("Символ: %c", c);

// Вывод строки и числа
char s[81];
gets_s(s, 80);
printf("Строка %s содержит %d символов\n", s, strlen(s));
```

Первым делом нужно отметить, что управляющая строка функции `printf()` может вовсе не содержать спецификаторов. Как правило, такой вариант используется для вывода строковых констант, например:

```
printf('********************************************');
```

В первом примере управляющая строка содержит один спецификатор. После первого аргумента `printf()` указан всего один дополнительный аргумент – идентификатор переменной, значение которой нужно вывести.

Второй пример уже сложнее. Управляющая строка содержит два спецификатора. Первый используется для вывода строки, второй – числа. Мы выводим саму строку, введенную пользователем, и ее длину. Обратите внимание, что строка формата также содержит символ перевода новой строки.

Вывод чисел

Особое внимание заслуживает вывод цифр. Вывод целых чисел можно осуществить с помощью спецификаторов %d и %i. По сути, это одно и тоже. Поддержка этих двух спецификаторов осуществляется по историческим причинам, на практике чаще используется %d. Пример использования %d уже был приведен и в этой главе и ранее по всей книге. Вывести целое число без знака можно спецификатором %u.

Для вывода чисел с плавающей запятой используется формат %f. Для вывода double можно использовать модификаторы %e или %E. Они обеспечивают вывод числа в так называемом экспоненциальном формате:

```
x.dddddE+/-yy
```

Если используется формат %e, то буква E будет строчной, а если %E – прописной. При желании можно использовать формат %g, когда функция printf() сама будет выбирать формат – или %f или %e – в зависимости от того, какой вид числа будет короче. Формат %G аналогичен %g, только выбор происходит между форматами %f и %E.

В некоторых других языках программирования для перевода числа в другую систему счисления нужно использовать специальные функции. В Си все гораздо проще:

```
int num = 251;

printf("%d %x %o", num, num, num);
```

Здесь мы выводим три представления числа 251 – десятичное, шестнадцатеричное и восьмеричное.

Если перед спецификаторами f, g, e, E стоит модификатор #, это означает, что число будет содержать десятичную точку, даже если оно не имеет дробной части. Если этот модификатор стоит перед спецификаторами x или X, шестнадцатеричные числа выводятся с префиксом 0x. Если же символ # указан перед спецификатором o, число будет дополнено ведущими нулями. К другим спецификаторам модификатор # применять нельзя.

Пример:

```
int k = 124;
printf("%x %#x", k, k);
```

Вывод адресов

При желании можно вывести адрес переменной на экран. Делается это так:

```
int x = 10;

int main(void)
{
  printf("%p", &x);
  return 0;
}
```

Модификатор минимальной ширины поля

Целое число, размещенное между символом процента и кодом формата, задает минимальную ширину тля. Если строка вывода короче, чем нужно, она дополняется пробелами, если длиннее, строка все равно выводится полностью. Строку можно дополнять не только пробелами, но и нулями. Для этого достаточно поставить 0 перед модификатором ширины поля. Например, спецификатор %04d дополнит число, количество цифр которого меньше 4, ведущими нулями, так что в результате оно будет состоять из 4 цифр.

Модификатор минимальной ширины поля чаще всего используется для форматирования таблиц. Рассмотрим вывод таблицы умножения.

Листинг 7.2. Вывод таблицы умножения с помощью модификатора ширины поля

```
#include <stdio.h>
int main()
{
  int i, j;

  for (i = 1; i <= 9; i++) {

     for (j = 1; j <= 9; j++)
      printf("%2d  ", i*j);

     printf("\n");

  }

  return 0;
}
```

Вывод программы будет следующий:

```
1   2   3   4   5   6   7   8   9
2   4   6   8  10  12  14  16  18
3   6   9  12  15  18  21  24  27
4   8  12  16  20  24  28  32  36
5  10  15  20  25  30  35  40  45
6  12  18  24  30  36  42  48  54
7  14  21  28  35  42  49  56  63
8  16  24  32  40  48  56  64  72
9  18  27  36  45  54  63  72  81
```

Модификатор точности

Модификатор точности указывается после модификатора ширины поля (если он есть). Этот модификатор состоит из точки, за которой следует це-

лое число. Точный смысл модификатора зависит от типа данных, к которым он применяется.

Если модификатор точности применяется к числам с плавающей точкой с форматами %f, %e, %E, он означает количество десятичных цифр после точки. Например, спецификатор формата %8.2f означает, что на экран будет выведено число, состоящее из 8 цифр, две из которых расположены после точки.

Пример:

```
float f = 123.123123;
printf("%f\n", f);
printf("%5.2f\n", f);
printf("%10.2f\n", f);
```

Вывод будет таким:

```
123.123123
123.12
    123.12
```

Выравнивание вывода

По умолчанию вывод выравнивается по правому краю. Иначе говоря, если ширина поля больше, чем выводимые данные, результаты "прижимаются" к правому краю. Вывод можно выровнять по левому краю, поставив перед символом % знак "минус". Например, спецификатор %-8.2f выравнивает число с двумя знаками после точки по левому краю поля, состоящего из 8 позиций.

Рассмотрим пример:

```
printf("По правому краю %8d\n", 1000);
printf("По левому краю %-8d\n", 1000);
```

7.1.3. Форматированный ввод: функция scanf()

Функция `scanf()` представляет собой процедуру ввода. Она может считывать данные всех встроенных типов и автоматически преобразовывать числа в соответствующий внутренний формат. Данная функция выглядит полной противоположностью функции `printf()`. Прототип функции:

```
int scanf ( const char * format, ... );
```

Аргументы нужно передавать по ссылке, то есть аргументами должны быть указатели. Поэтому перед именем переменной нужно указывать `&`. Исключение – строки. Строки считываются в символьные массивы, имена которых сами являются адресами, поэтому `&` перед именем строки указывать не нужно.

Таблица 7.3. Спецификаторы формата для `scanf()`

Спецификатор	Назначение
%a	Читает значение с плавающей точкой (только C99 и C11)
%c	Символ
%d	Читает целое значение
%i	Читает целое значение
%e	Читает значение с плавающей точкой
%f	Читает значение с плавающей точкой
%g	Читает значение с плавающей точкой
%o	Восьмеричное число
%s	Читает строку
%u	Целое число без знака (u – unsigned)
%x	Читает hex-число
%p	Указатель
%n	Принимает целое значение, равное количеству прочитанных символов

%%	Используется для вывода знака %
%[]	Набор символов

Спецификаторы формата у функции `scanf()` практически такие же, как у `printf()`, но к ним добавляется еще один - %[] – набор сканируемых символов.

Рассмотрим несколько примеров:

```
// ввод целого числа
int k;
scanf("%d", &k);

// ввод отдельных символов
char a, b, c;
scanf("%c%c%c", &a, &b, &c);

// ввод строки
char s[80];
// & перед именем строки не нужен
scanf("%s", s);
```

Функцию `scanf()` можно применять для ввода строк из входного потока. Для этого используется спецификатор %s. Он заставляет функцию `scanf()` считывать символы, пока не обнаружится разделитель. Символы, считанные из входного потока, записываются в массив, на который ссылается соответствующий аргумент, а в конец этого массива записывается нулевой байт. Функция `scanf()` считает разделителем пробел, символ перехода на новую строку, символ табуляции, символ вертикальной табуляции, а также символ прогона бумаги. Проблема в том, что с помощью `scanf()` нельзя ввести строку, содержащую пробелы. В этом и есть ее недостаток, поэтому для ввода строк лучше используйте функцию `gets_s()` или `gets()`, но помните, что она небезопасна.

Используя спецификатор [], можно задать набор сканируемых символов. При использовании набора сканируемых символов функция продолжает считывать символы, помещая их в соответствующий массив, пока не обна-

ружится символ, не входящий в заданный набор. По возвращении из функции `scanf()` массив будет содержать строку, состоящую из считанных элементов и завершающуюся нулевым байтом.

Пример:

```
char s[100];
scanf("%[abcdef]", s);
printf("%s", s);
```

Спецификаторы формата могут содержать модификатор максимальной ширины поля. Он представляет собой целое число, указанное между знаком процента и спецификатором формата. Это число ограничивает количество символов, которое можно ввести. Например, чтобы в строку `str` нельзя было ввести больше 30 символов, следует выполнить следующий вызов.

```
scanf("%20s", str);
```

7.2. Файловый ввод/вывод

Настало время рассмотреть файловый ввод/вывод. Функции файлового вывода имеют дело не с консолью, а с файлами на носителе данных.

7.2.1. Файлы и потоки

Прежде, чем приступить к работе с файлами, нужно поговорить о потоках. Система ввода/вывода языка Си предлагает программисту универсальный интерфейс, который не зависит от физического устройства – такую себе абстракцию, с которой работает программист, а именно записывает в нее данные или читает из нее данные. Эта абстракция называется потоком, а файл – это уже физическое устройство.

Все потоки (будь то вывод в файл, на консоль, или на принтер) работают одинаково. Существует только два видов потоков: текстовые и бинарные.

Текстовый поток – это последовательность символов. Стандарт позволяет организовать потоки в виде строк, которые заканчиваются символов перехода, в последней строке символ перехода (\n) указывать не обязательно. Большинство компиляторов Си не завершают текстовые потоки символом перехода. В зависимости от окружения некоторые символы в текстовых потоках могут подвергаться трансформации. Например, при выводе на принтер символ \n может заменяться парой символов \r\n, символ \r используется для перевода каретки. Поэтому между символами, записанными в текстовом потоке, и символами, выведенными на внешние устройства, нет взаимно однозначного соответствия. По этой же причине количество символов в текстовом потоке и на внешнем устройстве может быть разным.

Бинарный поток – это последовательность байтов (не символов!). Эта последовательность однозначно соответствует последовательности, которая будет записана на внешнее устройство. Также если считать последовательность байтов из бинарного потока и записать ее на физическое устройство, то количество записанных байтов будет равно количеству считанных.

В Си файлом считается все – от файла на диске до дисплея. Это не ноухау. Такой принцип использует файловая система UNIX, где в каталоге /dev хранятся файлы устройств. Для UNIX/Linux-пользователей – это норма, для Windows-пользователей – это новшество.

Не все файлы ведут себя одинаково. Например, из файла на жестком диске можно прочитать данные, а вот из файла принтера – нет, в такой файл можно только записать данные.

Если файл поддерживает запрос позиции (position request), при его закрытии курсор файла устанавливается на его начало. При чтении/записи очередного символа курсор смещается на одну позицию вперед.

При закрытии файла связь с потоком разрывается. Если файл был открыт для записи, его содержимое будет записано на внешнее устройство. В принципе, можно выполнить запись содержимого в любой момент с помощью функции `fflush()`, не закрывая файл – на случай, если вы хотите убедиться, что данные записаны на диск.

При нормальном завершении программы все файлы будут закрыты автоматически. Если произошла ошибка и аварийный останов программы или же программа остановлена функцией `abort()`, файлы останутся открытыми со всеми вытекающими обстоятельствами.

7.2.2. Основные функции файлового ввода/вывода

В таблице 7.4 приведены основные функции файлового ввода/вывода. Их список больше, чем функций консольного ввода/вывода, поскольку в него добавлены функции открытия/закрытия файла, сброса, перемещения позиции внутреннего курсора файла и т.д.

Таблица 7.4. Функции файлового ввода/вывода

Функция	Операция
`fopen()`	Открывает файл
`fclose()`	Закрывает файл
`putc()`	Записывает символ в файл
`fputc()`	То же самое, что и putc()
`getc()`	Считывает символ из файла
`fgetc()`	То же, что и getc()
`fgets()`	Читает строку из файла
`fputs()`	Записывает строку в файл
`fseek()`	Устанавливает курсор файла в заданную позицию
`ftell()`	Возвращает позицию курсора
`fprintf()`	То же, что и printf(), но для файлов
`fscanf()`	То же, что и scanf(), но для файлов
`feof()`	Возвращает истинное значение, если достигнут конец файла
`ferror()`	Возвращает истинное значение, если произошла ошибка
`rewind()`	Устанавливает курсор в начало файла
`remove()`	Удаляет файл
`fflush()`	Сбрасывает буфер записи на устройство, то есть немедленно записывает данные из буфера на диск

Прототипы файловых функций определены в заголовочном файле stdio.h. Также в нем определены типы `size_t`, `fops_t`, `FILE` и макросы NULL, EOF (конец файла), FOPEN_MAX, SEEK_SET, SEEK_CUR и SEEK_END. Первый макрос – это нулевой указатель, EOF обычно равен -1 и означает конец файла. Макрос FOPEN_MAX содержит число файлов, которые можно открыть одновременно. Остальные макросы используются функцией fseek() для перемещения курсора файла.

Тип FILE – особый. Это звено, которое связывает между собой все компоненты системы ввода/вывода. Указатель файла представляет собой указатель на структуру типа FILE. В этой структуре хранится информация о файле, в частности, его имя, статус и текущее положение курсора. По существу, указатель файла описывает конкретный файл именно поэтому в некоторой литературе можно встретить определение дескриптор файла (а не указатель файла). Объявить указатель файла можно так:

```
FILE *my_file;
```

Далее будут рассмотрены функции файлового ввода/вывода, кроме функции `fseek()`, которая в современном программировании используется довольно редко.

7.2.3. Открытие и закрытие файла

Прежде, чем начать работу с файлом, его нужно открыть. Делается это функцией `fopen()`:

```
FILE *fopen( const char *имя_файла, const char * режим)
```

Функция принимает два параметра - имя файла и режим работы с файлом. В нашем случае режим будет "w" - запись. Другие режимы приводятся в таблице 7.5.

Таблица 7.5. Режимы открытия файла (функция `fopen`)

Режим открытия	Описание
`r`	Режим открытия файла для чтения. Файл должен существовать.
`w`	Режим создания пустого файла для записи. Если файл с таким именем уже существует его содержимое стирается, и файл рассматривается как новый пустой файл.
`a`	Дописать в файл. Операция добавления данных в конец файла. Файл создается, если он не существует.
`r+`	Режим открытия фала для обновления чтения и записи. Этот файл должен существовать.
`w+`	Создаёт пустой файл для чтения и записи. Если файл с таким именем уже существует его содержимое стирается, и файл рассматривается как новый пустой файл.
`a+`	Открыть файл для чтения и добавления данных. Все операции записи выполняются в конец файла, защищая предыдущее содержания файла от случайного изменения. Вы можете изменить позицию (FSEEK, перемотка назад) внутреннего указателя на любое место файла только для чтения, операции записи будет перемещать указатель в конец файла, и только после этого дописывать новую информацию. Файл создается, если он не существует.
`rb`	Открывает бинарный файл для чтения
`wb`	Создать бинарный файл для записи
`ab`	Добавляет информацию в конец бинарного файла
`r+b`	Открывает бинарный файл для чтения и записи
`w+b`	Создать бинарный файл для чтения и записи
`a+b`	Добавить информацию в конец бинарного файла или создать бинарный файл для чтения и записи

Функция `fopen()` возвращает указатель файла, в собственной программе программист не должен изменять значение этого указателя. Если во время открытия файла произойдет программа, функция вернет NULL.

Пример использования:

```
FILE *fptr;
fptr = fopen("text.txt", "w");
```

Однако после этого я бы рекомендовал проверить fptr на NULL:

```
if(fptr == NULL)
{
   printf("Error!\n");
   exit(1);
}
```

Количество одновременно открытых файлов определяется константой FOPEN_MAX. Как правило, она равна 8, но ее точное значение следует искать в документации, сопровождающей компилятор.

Функция `fclose()` закрывает поток, открытый ранее функцией `fopen()`. Она записывает все оставшиеся в буфере данные в файл и закрывает его, используя команды операционной системы. Ошибка, возникшая при закрытии файла, может породить множество проблем, начиная с потери данных и разрушения файлов и заканчивая непредсказуемыми последствиями для программы. Кроме того, функция `fopen()` освобождает управляющий блок файла, связанного с потоком, позволяя использовать этот блок повторно. Помните, что поскольку система ограничивает количество одновременно открытых файлов, поэтому иногда вам придется закрыть один из открытых файлов, чтобы открыть другой.

Прототип функции:

```
int fclose(FILE *fp)
```

Здесь fp – указатель файла, полученный из функции `fopen()`:

```
fclose(fptr);
```

7.2.4. Чтение и запись символов

Для записи символа используется функция `fputc()`. Функция `putc()` – это всего лишь макрос, при желании ее можно использовать вместо `fputc()`. Наличие двух функций необходимо для сохранения совместимости со старыми версиями Си. Прототипы функций выглядят одинаково:

```
int fputc(int символ, FILE *fp)
int putc(int символ, FILE *fp)
```

Для чтения файла можно использовать функции `getc()` и `fgetc()`. Как и в прошлом случае, наличие двух функций необходимо для совместимости со старыми версиями. Функция `getc()` считывает символ из файла, который открыт функцией `fopen()`:

```
int fgetc(FILE *fp)
int getc(FILE *fp)
```

При чтении символов из файла нужно контролировать достижение конца файла – как только он будет достигнут, функция `getc()` вернет значение EOF:

```
FILE *fp;
char c;
fp = fopen(__FILE__,"r");
do {
     c = getc(fp);
     putchar(c);
}
while(c != EOF);
fclose(fp);
```

Данная программа открывает файл с собственным исходным кодом – для этого используется макрос `__FILE__`. Вместо него можно использовать любую строковую константу с именем файла (например, "file.txt"). Далее в цикле `do` мы читаем символы из файла функцией `getc()`, выводим эти символы на экран функцией `putchar()`. Цикл `do` читает символы, пока

не будет достигнут EOF. После этого программа закрывает файл функцией `fclose()`.

У нашей программы есть один недостаток. Для определения конца файла мы используем константу EOF. Однако это не самый лучший способ для распознавания конца файла. Во-первых, операционная система может работать как с текстовыми, так и с бинарными файлами. Если файл открыт для бинарного ввода, из него может быть считано целое число, равное константе EOF. Следовательно, конец файла, распознаваемый функцией `getc()`, может не совпадать с реальным концом файла. Во-вторых, функция `getc()` возвращает константу EOF не только по достижении конца файла, но и при возникновении любой другой ошибки. По этой причине константу EOF невозможно интерпретировать однозначно. Для решения этой проблемы в языке Си предусмотрена функция `feof()`, которая распознает конец файла. Ее прототип выглядит так:

```
int feof(FILE *fp)
```

Если `feof()` обнаружит конец файла, она вернет истинное значение, поэтому процедура чтения бинарного файла может выглядеть так:

```
while(!feof(fptr)) ch = getc(fptr);
```

7.2.5. Чтение и запись строк

Для чтения и записи строк используются функции `fputs()` и `fgets()`:

```
int fputs(const char *строка, FILE *fp);
int *fgets(char *строка, int длина, FILE *fp)
```

Функция `fputs()` записывает в заданный поток строку, на которую ссылается указатель строка. При возникновении ошибки она возвращает константу EOF. Функция `fgets()` считывает строку из указанного потока, пока не обнаружит символ перехода или не прочитает длина – 1 символов. В отличие от функции `getc()` символ перехода считается составной частью строки. Результирующая строка должна завершаться нулем. В случае

успеха функция возвращает указатель на введенную строку, в противном случае она возвращает нулевой указатель.

Рассмотрим, как записать введенное пользователем предложение в файл с использованием функции `fprintf()`:

```
char sentence[1000];
FILE *fptr;

fptr = fopen("text.txt", "w");
if(fptr == NULL)
{
   printf("Ошибка!\n");
   exit(1);
}

printf("Введите предложение:\n");
gets(sentence);

fprintf(fptr,"%s", sentence);
fclose(fptr);
```

Программа запускается, запрашивает ввод пользователя, читает его с помощью функции `gets()` в переменную sentence. Далее программа записывает введенное пользователем предложение в файл text.txt функцией `fprintf()`.

7.2.6. Функция ferror()

Данная функция определяет, возникла ли ошибка при работе с файлом. Прототип функции выглядит так:

```
int ferror(FILE *fp)
```

Здесь параметр `fp` является допустимым указателем файла. Функция возвращает истинное значение, если при выполнении операции возникла

ошибка, в противном случае она возвращает ложное значение. Поскольку с каждой операцией над файлом связан определенный набор ошибок, функцию `ferror()` следует вызывать сразу же после выполнения операции, в противном случае ошибка может быть упущена.

Рассмотрим пример использования этой функции:

```
do {
     c = getc(fp);
  if (ferror(fp)) {
 printf("Ошибка при чтении файла\n");
 break;  // прерываем цикл
     }
     putchar(c);
}
while(c != EOF);
```

7.2.7. Сброс буфера – функция fflush()

Функция `fflush()` сбрасывает содержимое буфера в файл, связанный с указателем `fp`:

```
int fflush(FILE *fp);
```

Сброс буфера приводит к немедленной записи информации в файл. Обычно ее можно вызвать при записи важной информации, если хотите, чтобы система сразу записала ее на диск, не дожидаясь завершения программы.

7.2.8. Удаление файла

С помощью функций Си можно не только создать файл, но и удалить его. Для этого используется функция `remove()`:

```
int remove(const char *имя_файла);
```

Функции нужно передать имя файла, а не указатель. Пример использования:

```
if (remove("file.txt")) {
   printf("Не могу удалить файл\n");
}
else printf("Файл успешно удален\n");
```

В случае успеха функция возвращает 0, а в случае ошибки – отличное от нуля значения (код ошибки). Именно поэтому условный оператор выглядит так не логично

7.2.9. Функции fread() и fwrite()

Функции `fread()` и `fwrite()` могут использоваться для чтения и записи данных, размер которых превышает 1 байт. Прототипы функций выглядят так:

```
size_t fread(void *буфер, size_t к-во_байтов, size_t к-во_блоков, FILE *fp);
size_t fwrite(const void *буфер, size_t к-во_байтов, size_t к-во_блоков, FILE *fp);
```

Здесь буфер – это указатель на область памяти, содержащую данные, которые должны быть записаны в файл или прочитаны из него (в зависимости от функции). Параметр количество_блоков – определяет, сколько блоков нужно прочитать или записать. Размер блока задается параметром количество_байтов.

Функция `fread()` возвращает количество считанных блоков. Если количество считанных блоков меньше заданного количества блоков, то или обнаружен конец файла или возникла ошибка. Функция `fwrite()` возвращает количество реально записанных блоков. Если не произошло никаких ошибок (а может случиться, что закончилось место на диске), то это количество равно количеству блоков, указанных при вызове функции.

Рассмотрим, как использовать эти функции:

```
FILE *fp;
double a = 33.33;
int b = 300;

// открываем файл
fp = fopen("test.dat", "wb+");
if (fp == NULL) {
  printf("Невозможно открыть файл\n");
  exit(1);
}

// записываем один блок размера sizeof(double) в файл
fwrite(&a, sizeof(double), 1, fp);
// записываем один блок размера sizeof(int) в файл
fwrite(&b, sizeof(int), 1, fp);

// на всякий случай сбрасываем буфер на диск
fflush(fp);

// "перематываем" файл на начало
rewind(fp);
// читаем данные из файла
fread(&a, sizeof(double), 1, fp);
fread(&b, sizeof(int), 1, fp);

// Выводим то, что прочитали
printf("%f %d\n", a, b);

// Закрываем файл
fclose(fp);
```

Как правило, функции `fread()` и `fwrite()` используются для чтения и записи структур. Например, следующую структуру можно записать так:

```
struct car_type {
   char model[80];
   int year;
} car;
fwrite(&car, sizeof(struct car_type), 1, fp);
```

7.2.10. Функции fprintf() и fscanf()

Обе эти функции полностью аналогичны своим консольным вариантам, вот только первым параметром является указатель файла, а управляющая строка (строка формата) следует вторым параметром:

```
int fprintf(FILE *fp, const char* строка_формата, ...);
int fscanf(FILE *fp, const char* строка_формата, ...);
```

Работают они так же, как и консольные варианты:

```
char c[1000];
FILE *fptr;

if ((fptr = fopen("text.txt", "r")) == NULL)
{
    printf("Ошибка при открытии файла");
    exit(1);
}

// Читаем текст, пока не достигнем символа \n
fscanf(fptr,"%[^\n]", c);

printf("Строка из файла:\n%s\n", c);
fclose(fptr);
```

7.3. Работа с каталогами

В предыдущем разделе была рассмотрена работы с файлами, в этом мы поговорим о работе с каталогами. В частности, будут рассмотрены следующие операции:

- Создание каталога
- Чтение каталога

- Удаление каталога
- Закрытие каталога
- Определение текущего рабочего каталога

7.3.1. Создание каталога

Для создания каталога мы будем использовать системный вызов Linux с названием `mkdir()`:

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
```

Первый параметр – это название каталога. Второй параметр – это режим, задающий права доступа к каталогу. Права доступа задаются с помощью битов режима:

- S_IRWXU – чтение, запись, выполнение/поиск для владельца.
- S_IRUSR – только чтение для владельца.
- S_IWUSR – только запись для владельца.
- S_IXUSR – выполнение/поиск для владельца.
- S_IRWXG – чтение, запись, выполнение/поиск для группы.
- S_IRGRP – чтение для группы.
- S_IWGRP – запись для группы.
- S_IXGRP – выполнение/поиск для группы.
- S_IRWXO – чтение, запись, выполнение/поиск для остальных пользователей.
- S_IROTH – только чтение для остальных.
- S_IWOTH – запись для остальных.
- S_IXOTH – выполнение/поиск для остальных.
- S_ISUID – SUID при выполнении.
- S_ISGID – SGID при выполнении.

Данные биты можно комбинировать, например, S_IRUSR|S_IWUSR.

В среде Windows можно использовать функцию `mkdir()` из `dir.h`:

```
#include<dir.h>
int mkdir(const char *path);
```

Этой функции нужно просто передать имя каталога. В случае успеха функция вернет 0, в противном случае – отличное от нуля значение.

7.3.2. Чтение каталога

Первым делом нужно открыть поток:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

После этого нужно произвести чтение из потока функцией `readdir()`:

```
#include
struct dirent *readdir(DIR *dirp);
```

Функция возвращает указатель на структуру `dirent`, определенную так:

```
struct dirent
{
    ino_t          d_ino;     /* номер инода */
    off_t          d_off;       /* смещение на след. dirent */
    unsigned short d_reclen;  /* длина записи */
    unsigned char  d_type;    /* тип файла, поддерживается не
                                 всеми ФС */
    char           d_name[256]; /* имя файла */
};
```

Каждый вызов `readdir()` возвращает `dirent` для следующего файла/каталога. Если достигнут конец каталога или произошла ошибка, функция возвращает NULL.

По окончанию работы с каталогом его нужно закрыть. Для этого используется вызов `closedir()`:

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

7.3.3. Удаление каталога

Для удаления каталога используется системный вызов `rmdir()`:

```
#include <unistd.h>
int rmdir(const char *pathname);
```

Функции `rmdir()` нужно передать только имя каталога. Каталог будет удален только, если он пуст и не является текущим рабочим каталогом. В противном случае нужно сначала его очистить и установить другой рабочий каталог.

В Windows функция удаления называется так же, но определена в заголовочном `dir.h`.

7.3.4. Получение рабочего каталога

Linux предоставляет системный вызов `getcwd()`:

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

Функция `getcwd()` копирует абсолютный путь к текущему рабочему каталогу в массиве, на который указывает `buf`, имеющий длину `size`.

Если текущий абсолютный путь требует буфера, длина которого превышает size, то возвращается NULL, а errno принимает значение ERANGE; приложение должно проверить, возникла эта ошибка или нет и, если необходимо, выделить буфер большего размера.

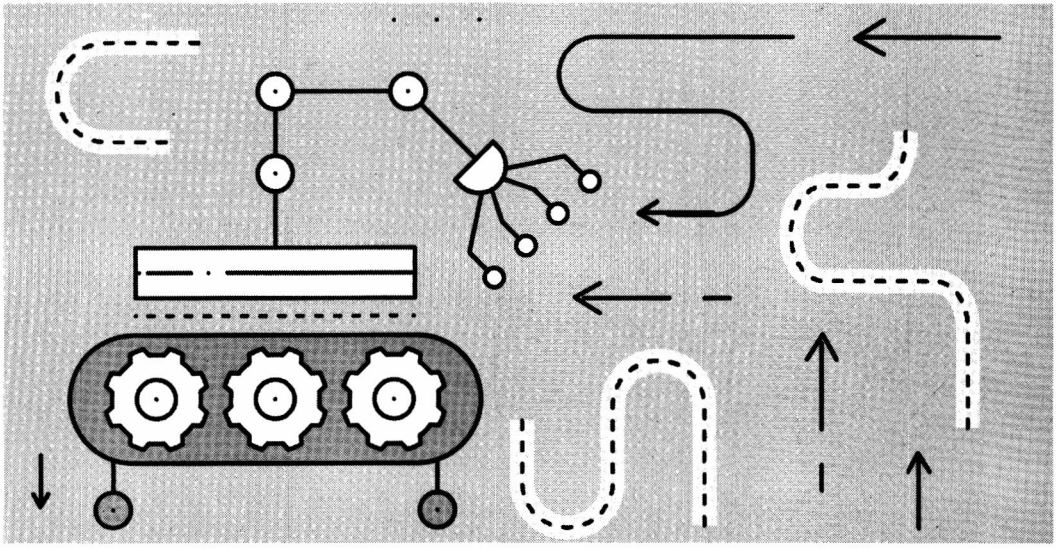
7.3.5. Пример работы с каталогом

Напишем программу, которая принимает имя каталога в качестве своего первого аргумента. Она создаст в этом каталоге подкаталог newDir и выведет содержимое каталога.

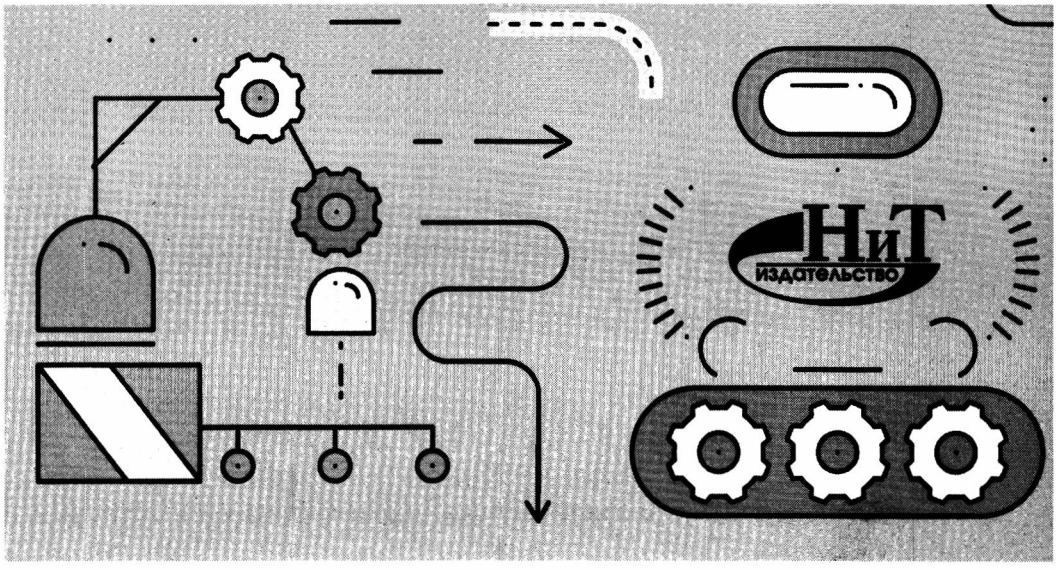
```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    if(2 != argc)
    {
        printf("\n Укажите имя каталога в качестве первого
                параметра \n");
        return 1;
    }

    DIR *dp = NULL;
    struct dirent *dptr = NULL;
    // Буфер для хранения имени каталога
    char buff[128];
    memset(buff,0,sizeof(buff));

    // копируем путь, указанный пользователем
    strcpy(buff,argv[1]);

    // Открываем поток каталога
    if(NULL == (dp = opendir(argv[1])) )
```

```
    {
        printf("\n Не могу открыть каталог [%s]\n",argv[1]);
        exit(1);
    }
    else
    {
        // Проверяем, указал ли пользователь '/' в конце имени каталога.
        // Добавляем к имени строку newDir (что именно мы
        // добавим, зависит от
        // того, указал ли пользователь слеш или нет)
        if(buff[strlen(buff)-1]=='/')
        {
            strncpy(buff+strlen(buff),"newDir/",7);
        }
        else
        {
            strncpy(buff+strlen(buff),"/newDir/",8);
        }

        printf("\n Создаем новый каталог [%s]\n",buff);
        // Создаем новый каталог
        mkdir(buff,S_IRWXU|S_IRWXG|S_IRWXO);
        printf("\n Содержимое каталога [%s] \n",argv[1]);
        // Читаем содержимое каталога
        while(NULL != (dptr = readdir(dp)) )
        {
            printf(" [%s] ",dptr->d_name);
        }
        // Закрываем поток каталога
        closedir(dp);
        // Удаляем каталог, созданный нами
        rmdir(buff);
        printf("\n");
    }

    return 0;
}
```

Программа демонстрирует создание каталога, вывод содержимого каталога и удаление каталога.



Часть II. Алгоритмизация в Си



Данная часть книги посвящена различным алгоритмам обработки данных. Сначала мы рассмотрим динамические структуры, такие как очереди, стеки, списки, затем – алгоритмы поиска и сортировки данных, а напоследок попробуем разобраться с многопоточной обработкой данных – новшеством, появившемся в стандарте C11.

Глава 8.
Очереди и стеки в Си

Когда мы говорим об обработке данных, то должны подразумевать, как минимум один из механизмов доступу к ним. Данные механизмы обеспечивают возможность сохранения и получения информации. Существует четыре механизма доступа: очередь, стек, связанный список и двоичное дерево. В этой главе мы рассмотрим первые два механизма, в следующей - поговорим о списках и деревьях.

8.1. Очереди

Очередь (queue) – это линейный список данных, работа с которым осуществляется по принципу FIFO (First In, First Out, первый пришел, первый вышел). Другими словами, первый помещенный в очередь элемент будет получен из нее первым. Это единственный вариант работы с очередью. Вы не можете обратиться к произвольным элементам очереди, как в случае с массивом.

В реальной жизни очереди встречаются везде и всюду – загляните в тот же McDonalds, банк, магазин и т.д.

Сейчас мы попробуем реализовать очередь с помощью функций `qin()` и `qout()`. Первая будет помещать элемент в очередь, вторая – извлекать элемент из очереди. При помещении элемента в очередь он добавляется в конец очереди, а при извлечении элемента из очереди он удаляется из начала очереди, а функция `qout()` возвращает его значение. Чтобы понять, как работают эти функции, посмотрим на табл. 8.1.

Таблица 8.1. Работа очереди

Вызов функции	Возвращаемое значение	Содержимое очереди
qin(1)	-	1
qin(2)	-	12
qin(3)	-	123
qout()	1	23
qin(4)	-	234
qout()	2	34
qout()	3	4

Теперь напишем код этих функций. Для большей наглядности будем работать со строками (сугубо для демонстрации передачи строк функциям), но на практике вы частенько будете работать именно с числами, например, приложение организации электронной очереди работает именно с числами

– клиенту сообщается его номер в очереди, а затем – номер оператора, когда его очередь подойдет.

```
// Размер очереди
#define MAX[10];

char *p[MAX];
int in_pos;
int out_pos;

void qin(char *q)
{
  if (in_pos == MAX) {
     printf("Очередь переполнена\n");
     return;
  }
  p[in_pos] = q;
  in_pos++;
}

char *qout()
{
  if (in_pos == out_pos) {
    printf("Очередь пуста\n");
    return '\0';
  }
  out_pos++;
  return p[outpost-1];
}
```

Нашим функциям нужны две глобальные переменные `in_pos` и `out_pos` – это индексы следующего свободного места в списке и индекс следующего элемента, подлежащего выборке. Такая реализация очереди позволяет легко адаптировать ее к работе с элементами любого типа, просто изменив базовый тип массива.

Функция `qin()` помещает новый элемент в конец списка, а `qout()` должна удалить элемент из списка и вернуть его значение. В нашем случае вместо удаления функция просто увеличивает значение следующего элемента, подлежащего выборке – принцип тот же. Когда оба значения встретятся (`in_pos == out_pos`), очередь считается пустой.

Кроме этих двух функций нам понадобятся функции отображения очереди – `display()` и удаления элемента из очереди-списка – `remove_el()`:

```
void display(void) {
  register int j;
  for(j=out_pos; j < in_pos; j++)
   printf("%d. %s\n", j+1, p[j]);
}

void remove_el(void) {
  char *p;
  if((p=qout())==NULL) return;
  printf("%s\n", p);
}
```

Вставкой нового элемента будет заниматься функция `new_el()`. Она будет принимать ввод от пользователя и добавлять введенную строку в очередь:

```
void new_el(void)
{
  char s[100], *p;
  do {
    printf("Введите строку для вставки в очередь: \n");
    gets(s);
    if(*s==0) break;
    p = (char *)malloc(strlen(s)+1);
    if(!p) {
      printf("Нет памяти!\n");
      return;
    }
    strcpy(p, s);
    if(*s) qin(p);
  } while(*s);
}
```

Функция читает ввод пользователя, выделяет память и вставляет элемент в очередь. Функция будет запрашивать ввод, пока пользователь не введет пустую строку.

Код основной программы, учитывая, что все необходимые функции уже созданы, будет очень простым:

```
char s[80];
register int j;

// инициализируем массив пустыми указателями
for (j=0;j<MAX;++j) p[j] = NULL;

for (;;) {
 printf("N - ввести, D - вывести, R - удалить, Q - выход: ");
 gets(s);
 *s = toupper(*s);
 switch (*s) {
  case 'E': new_el(); break;
  case 'D': display(); break;
  case 'R': remove_el(); break;
  case 'Q': return 0;
 }
}
```

Наша реализация очереди хороша тем, что ее легко адаптировать к работе с данными любого типа. Однако многие программисты предпочитают реализовывать очередь на базе односвязного линейного списка. Подобная реализация будет рассмотрена в следующей главе.

8.2. Стеки

Стеки (англ. stack) – противоположность очереди, работающая по принципу LIFO (Last In First Out, последним пришел, первым вышел). Традиционно принцип работы стека демонстрируется на примере стопки тарелок. Тарелка, которая попала в стопку первой (она лежит на столе), будет использована последней. А последняя тарелка, которую вы положили последней, будет использована первой.

При работе со стеком, также по традиции, используются два термина – `push` (затолкать в стек) и `pop` (вытолкать из стека). Именно поэтому мы

создадим две функции `push()` и `pop()`. Первая будет помещать элемент в стек, вторая – извлекать из него. Далее приводится код этих функций. В качестве "среды хранения" мы будем использовать целочисленный массив. При желании вы можете изменить тип данных, например, сделать массив строк, как мы это делали в примере с очередью.

Таблица 8.2. Принцип работы стека

Вызов функции	Возвращаемое значение	Содержимое стека
push(1)	-	1
push(2)	-	2 1
push(3)	-	3 2 1
pop()	3	2 1
push(4)	-	4 2 1
pop()	4	2 1
pop()	2	1

```
#define MAX 80
int stack[MAX];
int top;     // вершина стека

void push(int j)
{
  if (top >= MAX) {
    printf("Стек заполнен!\n");
    return;
  }
  stack[top] = j;
  top++;
}
int pop(void)
{
  top--;
  if (top < 0) {
   printf("Стек пуст\n");
   return 0;
```

```
    }
    return stack[top];
}
```

Обе функции используют вспомогательную переменную `top`, которая содержит индекс вершины стека. Нам нужно учитывать случаи, когда стек пуст или заполнен. Ведь мы используем не связные списки, а массив, поэтому важно не выйти за пределы массива.

Попробуем использовать эти функции:

```
push(1);
push(2);
push(3);
printf("%d\n", pop());
printf("%d\n", pop());
printf("%d\n", pop());
```

Вывод будет таким:

```
3
2
1
```

Глава 9.
Связанные списки и деревья в Си

В прошлой главе мы не зря реализовывали очереди и стеки с помощью массивов. Это потому, что мы еще не знали о том, что такое связанные списки. Часто очередь, стек и другие динамические структуры реализуют немного иначе – в виде связанных списков. Связанный список – это базовая динамическая структура данных, состоящая из узлов, где каждый узел (элемент) имеет ссылку на другой элемент в цепочке данных. В отличие от очереди или стека при получении доступа к элементу, он не удаляется из списка. Но это при условии, что вам нужен традиционный связанный список. Если же вы хотите реализовать очередь или стек посредством связанного списка, есть возможность удаления элемента из списка. При этом нужно предусмотреть, чтобы у предыдущего (перед удаляемым) элемента ссылка на следующей элемент была установлена так, чтобы он указывал на элемент, идущий за удаляемым, иначе список будет поврежден, и вы не сможете получить доступ к элементам, которые идут после удаленного.

Связанные списки бывают односвязными и двусвязными. В односвязном списке каждый элемент имеет одну ссылку на другой элемент списка – как

правило, это ссылка на следующий элемент. Обход односвязного списка возможен только в одном направлении.

В двусвязном списке есть две ссылки – на предыдущий и следующий элемент. Обход такого элемента возможен в двух направлениях. Выбор типа списка осуществляется в зависимости от поставленной задачи – нельзя сказать, что какой-то из вариантов списка лучше или хуже.

9.1. Односвязные списки

Как уже отмечалось, в односвязных списках каждый элемент (кроме последнего) имеет ссылку на следующий элемент. Типичный односвязный список изображен на рис. 9.1.

Рис. 9.1. Односвязный список

Первым делом определим две структуры – элемент списка и сам список:

```
// Определяем элемент списка
typedef struct list_node {
    struct list_node *next;
    void *data;
} list_node_t;

// Определяем сам список
typedef struct list {
    /*
     * Размер списка хранить не обязательно,
     * он нужен для упрощения работы
```

```
     */
    int size;
    // начало списка
    list_node_t *head;
    // конец списка
    list_node_t *tail;
} list_t;
```

Первая структура – это структура, описывающая узел списка. Она содержит указатель на следующий узел списка и указатель на данные.

Вторая структура описывает сам список. Элемент `size` – это размер списка, а элементы `head` и `tail` – это указатели, соответственно, на голову и хвост списка.

9.1.1. Инициализация списка

Для инициализации списка используется функция `create_list()`. Рассмотрим ее подробнее. Первым делом функция выделяет необходимую память, устанавливает размер списка (0), а также указатели head и tail. Первый указывает на NULL (поскольку головы списка у нас еще нет), а второй – указывает на head (по сути, тоже на NULL).

```
// Инициализация массива
list_t * create_list(void)
{
    list_t *lt = malloc(sizeof(list_t));

    lt->size = 0;
    lt->head = NULL;
    lt->tail = lt->head;

    return lt;
}
```

9.1.2. Добавление узла

Далее создадим функцию добавления элемента в список. По аналогии со стеком мы назовем ее `list_push()`. Функция будет принимать два параметра – указатель на список и на данные.

```
// Добавляем элемент в начало списка
void list_push(list_t *lt, void * data)
{
    list_node_t * node = malloc(sizeof(list_node_t));
    node->data = data;
    node->next = lt->head;

    lt->head = node;
    lt->size += 1;
}
```

Функция выделяет память под новый элемент списка, добавляет данные в элемент, создает указатель на голову списка (`head`), то есть элемент `next` будет указывать на `lt->head`. Конечно же, функция увеличивает количество элементов списка.

В зависимости от поставленной задачи может понадобится добавление элемента не в начало, а в конец списка. Для этого мы напишем функцию `list_push_back()`.

```
// Добавляем элемент в конец списка
void list_push_back(list_t *lt, void * data)
{
    list_node_t * node = malloc(sizeof(list_node_t));
    node->data = data;
    if(lt->tail != NULL)
        lt->tail->next = node;
    else {
        lt->head = node;
    }

    lt->tail = node;
    lt->size += 1;
}
```

Функция добавления элемента в конец списка чуть сложнее. Она выделяет память для будущего узла, устанавливает данные узла, а затем ей нужно проверить, что хвост списка не равен NULL. Если это так, то указатель next хвоста списка (последнего элемента) теперь будет указывать на наш новый

элемент. Если `tail` указывает на NULL, тогда новый элемент будет головой списка (потому что список пуст). По окончанию операций функция увеличивает размер списка.

9.1.3. Извлечение узла

Функция `list_pop()` извлекает элемент из списка. Если список пуст, функция возвращает NULL. Если список не пуст, функция возвращает значение удаленного элемента, устанавливает должным образом указатели и уменьшает размер списка. Элемент удаляется функцией `free()` – мы освобождаем ранее выделенную память. Если функция удалила последний элемент, то указатели `head` и `tail` списка будут установлены в NULL.

```
// Извлекаем элемент из начала списка
void * list_pop(list_t *lt)
{
    if(lt->size == 0){
        // Список пуст
        return NULL;
    }

    list_node_t *node = lt->head;
    void * ret_val = node->data;

    lt->size -= 1;
    lt->head = node->next;

    free(node);

    if(lt->size == 0){
        // Это был последний элемент
        lt->head = NULL;
        lt->tail = NULL;
    }

    return ret_val;
}
```

9.1.4. Реализация стека и очереди

Созданные нами функции позволяют использовать наш связанный список и как очередь, и как стек.

Использовать список как очередь можно так:

```
list_t *st = create_list();          // создаем список
int data = 0, *ptr;

// Добавляем в начало списка
list_push(st, &data);
// Извлекаем из начала списка
ptr = (int *)list_pop(st);
```

Для использования списка как стека нужно добавлять элементы в конец списка, поэтому нам нужно использовать функцию `list_push_back()`:

```
list_t *queue = create_list();
int data = 0, *ptr;

// Добавляем в конец списка
list_push_back(queue, &data);
// Извлекаем из начала списка
ptr = (int *)list_pop(queue);
```

9.1.5. Практический пример: реверс односвязного списка

Задача заключается в том, чтобы выполнить реверс списка. Но не просто отобразить элементы списка в обратном порядке, а именно поменять их в списке местами так, чтобы они следовали в обратном порядке. Алгоритм довольно простой и интуитивно понятный, однако его реализация требует осторожной работы с указателями, чтобы не повредить исходный список.

Реверс выполняется с помощью следующего цикла:

```
while(a != NULL) {
  c = b, b = a, a = a->next;
  b->next = c;
 }
```

Полный код программы приведен в листинге 9.1. В этой программе мы разработали три вспомогательных функции:

```
/* добавляет lnode в начало списка */
void llist_add_begin(struct lnode **n, int val);
/* реверс списка */
void llist_reverse(struct lnode **n);
/* отображение списка */
void llist_display(struct lnode *n);
```

Первая добавляет узел в начало списка, вторая, собственно, выполняет реверс списка, а третья - выводит список на экран.

Листинг 9.1. Реверс односвязного списка

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

#define MAX 10     /* максимум 10 элементов */

struct lnode {
 int number;
 struct lnode *next;
};

/* добавляет lnode в начало списка */
void llist_add_begin(struct lnode **n, int val);
/* реверс списка */
void llist_reverse(struct lnode **n);
/* отображение списка */
void llist_display(struct lnode *n);

int main(void) {
```

```
 struct lnode *new = NULL;
 int i = 0;

 /* вставляем числа в список от 0 до 10 */
 for(i = 0; i <= MAX; i++)
  llist_add_begin(&new, i);

 printf("Исходный список:");
 llist_display(new);
 llist_reverse(&new);
 printf("В обратном порядке:");
 llist_display(new);

 return 0;
}

/* добавляет lnode в начало списка */
void llist_add_begin(struct lnode **n, int val)  {
 struct lnode *temp = NULL;

 /* добавляет новый узел */
 temp = malloc(sizeof(struct lnode));
 temp->number = val;
 temp->next = *n;
 *n = temp;
}

/* реверс списка */
void llist_reverse(struct lnode **n)  {
 struct lnode *a = NULL;
 struct lnode *b = NULL;
 struct lnode *c = NULL;
 a = *n, b = NULL;

 while(a != NULL) {
  c = b, b = a, a = a->next;
  b->next = c;
 }

 *n = b;
}
```

```
/* отображение списка */
void llist_display(struct lnode *n) {
 while(n != NULL)
  printf(" %d", n->number), n = n->next;

 printf("\n");
}
```

Поскольку мы добавляем элементы в начало списка, то изначально список будет в обратном порядке: 10, 9, 8 и т.д. После того, как мы выполняем реверс, список будет выводиться в прямом порядке.

9.2. Двусвязный список

Двусвязный список состоит из узлов, каждый из которых содержит ссылки на следующий и предыдущий узлы (см.рис. 9.2). Наличие двух ссылок дает несколько преимуществ. Самое важное из них – возможность перемещения в двух направлениях. Второе – немного упрощается работа со списком, в частности, вставка и удаление элементов. Третье – поскольку список можно пройти по двум направлениям, то в случае, если какая-то из ссылок станет неверной, целостность списка можно восстановить по другой ссылке.

Рис. 9.2. Двусвязный список

Двусвязный список можно представить в виде структуры:

```
struct list
{
  int field; // поле данных
  struct list *next; // указатель на следующий элемент
  struct list *prev; // указатель на предыдущий элемент
};
```

Структура содержит поле данных, а также указатели на следующий и предыдущий элементы – next и prev соответственно.

Далее будет рассмотрен ряд действий над двусвязным списком.

9.2.1. Инициализация списка

Инициализация списка предназначена для создания корневого узла списка, у которого поля указателей на следующий и предыдущий узлы содержат нулевое значение. Для инициализации мы будем использовать следующую функцию:

```
struct list * init(int a)  // a- значение первого узла
{
  struct list *lst;
  // выделение памяти под корень списка
  lst = (struct list*)malloc(sizeof(struct list));
  lst->field = a;
  lst->next = NULL; // указатель на следующий узел
  lst->prev = NULL; // указатель на предыдущий узел
  return(lst);
}
```

Функция выделяет память под список, устанавливает поле данных, а также указатели – next и prev. При создании узла они равны NULL.

9.2.2. Добавление узла

Для добавления узла в список функции нужно передать указатель на узел, после которого происходит добавление и данные для добавляемого узла.

Добавление узла в двусвязный список включает в себя следующие операции:

- Создание узла добавляемого элемента и заполнение его поля данных;
- Переустановка указателя "следующий" узла, предшествующего добавляемому, на добавляемый узел;
- Переустановка указателя "предыдущий" узла, следующего за добавляемым, на добавляемый узел;
- Установка указателя "следующий" добавляемого узла на следующий узел (тот, на который указывал предшествующий узел);
- Установка указателя "предыдущий" добавляемого узла на узел, предшествующий добавляемому (узел, переданный в функцию).

Код функции добавления узла в двусвязный список выглядит так:

```
struct list * add_node(list *lst, int number)
{
  struct list *temp, *p;
  temp = (struct list*)malloc(sizeof(list));
  p = lst->next;    // сохранение указателя на следующий узел
  lst->next = temp; // предыдущий узел указывает на создаваемый
  temp->field = number; // сохранение поля данных добавляемого узла
  temp->next = p; // созданный узел указывает на следующий узел
  temp->prev = lst; // созданный узел указывает на предыдущий узел
  if (p != NULL)
    p->prev = temp;
  return(temp);
}
```

9.2.3. Удаление узла

В качестве аргументов функции удаления узла передается указатель на удаляемый узел. Поскольку узел списка имеет поле указателя на предыдущий узел, нет необходимости передавать указатель на корень списка. Функция возвращает указатель на узел, следующий за удаляемым.

Удаление узла включает в себя следующие операции:

- установка указателя "следующий" предыдущего узла на узел, следующий за удаляемым;
- установка указателя "предыдущий" следующего узла на узел, предшествующий удаляемому;
- освобождение памяти удаляемого узла.

Код функции выглядит так:

```
struct list * delet_node(list *lst)
{
  struct list *prev, *next;
  prev = lst->prev;            // узел, предшествующий lst
  next = lst->next;            // узел, следующий за lst
  if (prev != NULL)
    prev->next = lst->next;    // переставляем указатель
  if (next != NULL)
    next->prev = lst->prev;    // переставляем указатель
  free(lst);        // освобождаем память удаляемого элемента
  return(prev);
}
```

Для удаления головы списка используется функция `delete_head()`:

```
struct list * delete_head(list *root)
{
  struct list *temp;
  temp = root->next;
  temp->prev = NULL;
  free(root);        // освобождение памяти текущего корня
  return(temp);      // новый корень списка
}
```

Функция удаления головы списка в качестве аргумента получает указатель на текущий корень списка. Возвращаемым значением будет новый корень списка — тот узел, на который указывает удаляемый корень.

9.2.4. Вывод списка

В качестве аргумента в функцию вывода элементов передается указатель на корень списка. Функция осуществляет последовательный обход всех узлов с выводом их значений.

```
void list_print(list *lst)
{
  struct list *p;
  p = lst;
  do {
    printf("%d ", p->field);     // вывод значения элемента p
    p = p->next;                 // переход к следующему узлу
  } while (p != NULL);           // условие окончания обхода
}
```

Поскольку наш список двусвязный можно вывести его элементы в обратном порядке:

```
void list_reverse_print(list *lst)
{
  struct list *p;
  p = lst;
  while (p->next != NULL)
    p = p->next;                 // переход к концу списка
  do {
    printf("%d ", p->field); // вывод значения элемента p
    p = p->prev;          // переход к предыдущему узлу
  } while (p != NULL);           // условие окончания обхода
}
```

Функция вывода элементов двусвязного списка в обратном порядке принимает в качестве аргумента указатель на корень списка. Функция перемещает указатель на последний узел списка и осуществляет последовательный обход всех узлов с выводом их значений в обратном порядке.

9.2.5. Замена местами двух элементов

Самая сложная операция при работе с двусвязным списком – это замена местами двух элементов. В качестве аргументов функция свопа узлов принимает два указателя на обмениваемые узлы, а также указатель на корень списка. Функция возвращает адрес корневого узла списка.

Взаимообмен узлов списка осуществляется путем переустановки указателей. Для этого необходимо определить предшествующий и последующий узлы для каждого заменяемого. При этом возможны две ситуации:

- заменяемые узлы являются соседями;
- заменяемые узлы не являются соседями, то есть между ними имеется хотя бы один узел.

Код функции свопа выглядит следующим образом:

```
struct list * swap(struct list *lst1, struct list *lst2, struct
list *head)
{
  // Возвращаем новый корень списка
  struct list *prev1, *prev2, *next1, *next2;
  prev1 = lst1->prev;            // узел предшествующий lst1
  prev2 = lst2->prev;            // узел предшествующий lst2
  next1 = lst1->next;            // узел следующий за lst1
  next2 = lst2->next;            // узел следующий за lst2
  if (lst2 == next1)             // обмениваются соседние узлы
  {
    lst2->next = lst1;
    lst2->prev = prev1;
    lst1->next = next2;
    lst1->prev = lst2;
    if(next2 != NULL)
      next2->prev = lst1;
    if (lst1 != head)
      prev1->next = lst2;
  }
  else if (lst1 == next2)        // обмениваются соседние узлы
```

```
  {
    lst1->next = lst2;
    lst1->prev = prev2;
    lst2->next = next1;
    lst2->prev = lst1;
    if(next1 != NULL)
      next1->prev = lst2;
    if (lst2 != head)
      prev2->next = lst1;
  }
  else  // обмениваются отстоящие узлы
  {
    // указатель prev можно установить только для элемента,
    // не являющегося головой списка
    if (lst1 != head)
prev1->next = lst2;
    lst2->next = next1;
    if (lst2 != head)
      prev2->next = lst1;
    lst1->next = next2;
    lst2->prev = prev1;
   // указатель next можно установить только для элемента,
   // не являющегося последним
    if (next2 != NULL)
       next2->prev = lst1;
    lst1->prev = prev2;
    if (next1 != NULL)
      next1->prev = lst2;
  }
  if (lst1 == head)
    return(lst2);
  if (lst2 == head)
    return(lst1);
  return(head);
}
```

Следующий код демонстрирует пример использования ранее приведенных функций:

```
list *head, *cur;
```

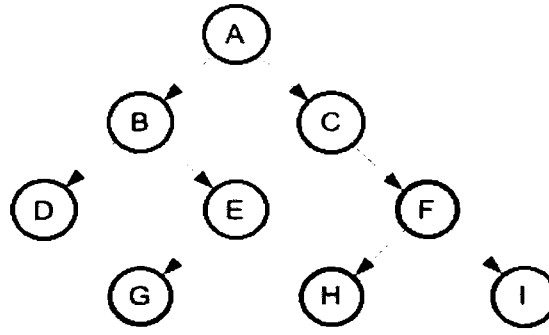
```
int num;
// Создаем список из 6 элементов
printf("a = ");
scanf("%d", &num);
head = init(num);
cur = head;
for (int i = 0; i < 5; i++) {
   printf("a = ");
   scanf("%d", &num);
   cur = add_node(cur, num);
}
list_print(head);
printf("\n");
// Меняем местами первый и второй элементы
cur = head->next;
head = swap(head, cur, head);
list_print(head);
printf("\n");
// Удаляем третий элемент списка
cur = head->next->next;
delet_node(cur);
list_print(head);
```

9.3. Деревья

Дерево – структура данных, представляющая собой древовидную структуру в виде набора связанных узлов.

Бинарное дерево — это конечное множество элементов, которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Каждый элемент бинарного дерева называется узлом. Связи между узлами дерева называются его ветвями.

Способ представления бинарного дерева:



Здесь:

- A — корень дерева
- B — корень левого поддерева
- C — корень правого поддерева

Корень дерева расположен на уровне с минимальным значением.

Узел D, который находится непосредственно под узлом B, называется потомком B. Если D находится на уровне i, то B – на уровне i-1. Узел B называется *предком* D.

Максимальный уровень какого-либо элемента дерева называется его *глубиной* или высотой.

Если элемент не имеет потомков, он называется листом или *терминальным узлом* дерева. Остальные элементы – внутренние узлы (узлы ветвления).

Число потомков внутреннего узла называется его степенью. Максимальная степень всех узлов есть степень дерева.

Число ветвей, которое нужно пройти от корня к узлу x, называется длиной пути к x. Корень имеет длину пути равную 0; узел на уровне i имеет длину пути равную i.

Бинарное дерево применяется в тех случаях, когда в каждой точке вычислительного процесса должно быть принято одно из двух возможных решений.

Имеется много задач, которые можно выполнять на дереве. Самая распространенная задача — выполнение заданной операции p с каждым элементом дерева. Здесь p рассматривается как параметр более общей задачи посещения всех узлов или задачи обхода дерева.

9.3.1. Способы обхода дерева

Представим дерево с тремя узлами. A – корень дерева, B и C – левые и правые поддеревья (рис 9.4).

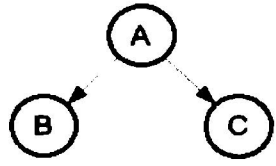


Рис. 9.4. Простое дерево

Существует три способа обойти это дерево:

1. A, B, C – префиксная форма или обход в прямом порядке (сверху вниз).
2. B, A, C – инфиксная форма или обход в симметричном порядке (слева направо).
3. B, C, A – постфиксная форма или обход в обратном порядке (снизу вверх).

9.3.2. Реализация дерева

Как и с другими динамическими структурами, начнем с описания структуры дерева:

```
struct tnode {
  int field;            // поле данных
```

```
  struct tnode *left;  // левый потомок
  struct tnode *right; // правый потомок
};
```

Учитывая, что мы объявили такую структуру, напишем три функции обхода дерева – в прямом, симметричном и обратном порядках:

```
// Функция обхода дерева в прямом порядке
void tree_print(tnode *tree) {
  if (tree!=NULL) { //Пока не встретится пустой узел
    cout << tree->field; //Отображаем корень дерева
    treeprint(tree->left); //Рекурсивная функция для левого поддерева
    treeprint(tree->right); //Рекурсивная функция для правого поддерева
  }
}
```

Обход дерева в инфиксной форме будет иметь следующий вид:

```
void tree_symmetric_print(tnode *tree) {
  if (tree!=NULL) {          //Пока не встретится пустой узел
    treeprint(tree->left);     //Рекурсивная функция для
                               // левого поддерева
    cout << tree->field;       //Отображаем корень дерева
    treeprint(tree->right);    //Рекурсивная функция для
                               //правого поддерева
  }
}
```

Обход дерева в постфиксной форме будет иметь вид:

```
void tree_reverse_print(tnode *tree) {
  if (tree!=NULL) {          //Пока не встретится пустой узел
    treeprint(tree->left);   //Рекурсивная функция для левого
                             // поддерева
    treeprint(tree->right); //Рекурсивная функция для правого
                             // поддерева
    cout << tree->field;  //Отображаем корень дерева
  }
}
```

9.3.3. Бинарное дерево

Бинарное (двоичное) дерево поиска – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- оба поддерева – левое и правое, являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных.

При добавлении узла в бинарное дерево нужно учитывать особенности дерева такого типа:

```
struct tnode * addnode(int x, tnode *tree) {
  if (tree == NULL) {     // Если дерева нет, то формируем
                          // корень
    tree =new tnode;      // память под узел
    tree->field = x;      // поле данных
    tree->left =  NULL;
    tree->right = NULL;   // ветви инициализируем пустотой
  }else  if (x < tree->field)    // условие добавление левого
                                 // потомка
    tree->left = addnode(x,tree->left);
  else                    // условие добавление правого потомка
    tree->right = addnode(x,tree->right);
  return(tree);
}
```

Функция удаления дерева будет выглядеть так:

```
void freemem(tnode *tree) {
  if(tree!=NULL) {
    freemem(tree->left);
    freemem(tree->right);
    delete tree;
  }
}
```

Все рассмотренные в данной главе динамические структуры только подготовка к их применению на практике. В следующей главе будет показано, как использовать списки и деревья для поиска и сортировки данных.

Глава 10.
Алгоритмы поиска и сортировки в Си

Какая же серьезная программа на Си обходится без поиска и сортировки данных? Огромное внимание алгоритмам поиска и сортировки уделяется в настоящем шедевре - книге Дональда Кнута "Искусство программирования". Вот только "Искусство программирования" - отличный выбор, когда есть время на его изучение. А вот когда времени нет, а программа нужна прямо здесь и сейчас, то приходится обращаться к другим источникам, например, к этой книге. В этой части мы рассмотрим множество примеров, которые, я надеюсь, помогут вам при написании лабораторной, курсовой работы или даже собственного реального приложения.

10.1. Сортировка вставкой связного списка

Сортировка вставками (Insertion Sort) — это простой алгоритм сортировки. Суть его заключается в том, что, на каждом шаге алгоритма мы берем один

из элементов массива, находим позицию для вставки и вставляем. Нужно отметить, что массив из 1-го элемента считается отсортированным.

Связный список — это базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки на следующий и/или предыдущий узел списка. Понятно, что первый узел списка содержит ссылку только на следующий элемент, а последний - только на предыдущий.

Для реализации связного списка мы используем структуру `node`, состоящую из двух членов: `number` - это число, которое несет в себе узел списка, и `node` - указатель на следующий узел. В нашем случае можно обойтись без указателя на предыдущий узел - для алгоритма сортировки вставками он не нужен.

```
struct node {
 int number;
 struct node *next;
};
struct node *head = NULL;
```

Первый узел списка называется `head` (голова списка). У последнего узла списка член `node` равен NULL. Сортировка вставками осуществляется функцией `insert_node()`, которая вставляет новый элемент в нужное место списка. Элементы берутся из массива `test`. Затем программа выводит массив тест и получившийся список, который уже является отсортированным.

```
void insert_node(int value) {
 struct node *temp = NULL;
 struct node *one = NULL;
 struct node *two = NULL;

 // если список пуст, нужно выделить память под голову списка
 if(head == NULL) {
  head = (struct node *)malloc(sizeof(struct node *));
  head->next = NULL;
 }
```

```
 // первый элемент - голова, второй - следующий элемент
 one = head;
 two = head->next;

 // временный узел
 temp = (struct node *)malloc(sizeof(struct node *));
 temp->number = value;

 // меняем one и two местами в случае необходимости
 while(two != NULL && temp->number < two->number) {
  one = one->next;
  two = two->next;
 }

 one->next = temp;
 temp->next = two;
}
```

Код основной программы может выглядеть так:

```
 struct node *current = NULL;
 struct node *next = NULL;
 int test[] = {8, 3, 2, 6, 1, 5, 4, 7, 9, 0};
 int i = 0;

 /* вставляем некоторые элементы в связный список */
 for(i = 0; i < 10; i++)
  insert_node(test[i]);

 /* выводим список */
 printf(" До  После\n"), i = 0;
 while(head->next != NULL) {
  printf("%4d\t%4d\n", test[i++], head->number);
  head = head->next;
 }

 /* очищаем список */
 for(current = head; current != NULL; current = next)
  next = current->next, free(current);
```

Рассмотрим еще один пример сортировки вставкой связного списка. Во-первых, здесь мы строки будем читать со стандартного ввода:

```
while((fgets(line, 1024, stdin)) != NULL)
  list = insert(line, list);
```

На секунду - это позволит использовать нашу программу не в образовательных, а в реальных целях - для сортировки файлов, и я далее покажу, как это сделать.

В остальном программа похожа на предыдущую - сортировка осуществляется при вставке, поэтому все, что нам нужно после вставки элементов в список - вывести его и освободить память. Для этого используются функции `print_list()` и `free_list()` соответственно. Всего, если не считать функции `main()`, в нашей программе будет три функции:

```
struct lnode *insert(char *data, struct lnode *list);
void free_list(struct lnode *list);
void print_list(struct lnode *list);
```

Что они делают, вы уже знаете. Также немного изменен код функции вставки элемента. Но в целом функция вставки осталась примерно такой же, если не считать, что теперь она возвращает список, а не просто производит манипуляции над ним (в прошлом примере наша функция ничего не возвращала - тип `void`). Итак, рассмотрим код, который приведен в листинге 10.1. Код снабжен комментариями, чтобы вам было понятнее.

Листинг 10.1. Сортировка файла методом вставки

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

struct lnode {
 char *str;
 struct lnode *next;
};

// вставка и сортировка
```

```
struct lnode *insert(char *data, struct lnode *list);
// освобождение памяти
void free_list(struct lnode *list);
// вывод списка
void print_list(struct lnode *list);

int main(void) {
 char line[1024];
 struct lnode *list;

 list = NULL;
 while((fgets(line, 1024, stdin)) != NULL)
  list = insert(line, list);    // сортировка осуществляется при
вставке

 print_list(list);    // выводим отсортированный список
 free_list(list);     // освобождаем память
 return 0;
}

struct lnode *insert(char *data, struct lnode *list) {
 struct lnode *p;
 struct lnode *q;

 /* Создаем новый узел */
 p = (struct lnode *)malloc(sizeof(struct lnode));
 /* сохраняем данные в новый узел */
 p->str = strdup(data);

 /* сначала мы обрабатываем случай, где данные (data) должны быть
первым элементом */
 if(list == NULL || strcmp(list->str, data) > 0) {
  /* по всей видимости, это первый элемент */
  /* теперь данные станут первым элементом */
  p->next = list;
  return p;
 } else {
  /* производим поиск по связному списку, определяя правильную
позицию */
  q = list;
  while(q->next != NULL && strcmp(q->next->str, data) < 0) {
```

```
    q = q->next;
   }
   p->next = q->next;
   q->next = p;
   return list;
  }
}

void free_list(struct lnode *list) {
 struct lnode *p;

 while(list != NULL) {
  p = list->next;
  free(list);
  list = p;
 }
}

void print_list(struct lnode *list) {
 struct lnode *p;

 for(p = list; p != NULL; p = p->next)
  printf("%s", p->str);
}
```

Я обещал показать, как можно программу использовать в реальных условиях для сортировки реального файла. Поскольку программа читает данные со стандартного ввода, мы можем перенаправить на нее файл, она его отсортирует и выведет на экран.

Аналогично, если вам нужно получить результат сортировки в файл, вы также можете использовать перенаправление ввода/вывода:

```
cat <файл_который_нужно_отсортировать> | программа >
<результат>
```

10.2. Пузырьковая сортировка

Еще один популярный в программировании метод сортировки - это сортировка пузырьком (buble sort в англ. литературе).

Алгоритм пузырьковой сортировки считается самым простым, но довольно неэффективным. Его можно использовать разве что для сортировки небольших массивов. Алгоритм считается учебным и практически не применяется вне учебной литературы, вместо него на практике применяются более эффективные алгоритмы сортировки. Но поскольку мы как раз учимся программировать, данный алгоритм - настоящая находка.

Суть алгоритма заключается в следующем. Программа несколько раз проходится по сортируемому массиву. При каждой итерации элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Получается, что элементы как бы выталкиваются вверх, как пузырьки в воде, отсюда и название алгоритма.

Проходы (итерации) по массиву повторяются N − 1 {\displaystyle N-1} N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован.

При каждой итерации очередной наибольший элемент массива ставится на свое место в конце массива - рядом с предыдущим наибольшим элементом, а наименьший элемент перемещается на одну позицию к началу массива - "всплывает".

Думаю, принцип понятен. Осталось все это закодировать. В нашей программе мы создадим функцию `bubble_sort()`, которой нужно передать массив элементов и его размер. Функция использует два цикла `for` - внутренний и внешний. Внешний проходится от 0 до `size`, а переменная `size` содержит количество элементов в массиве. Во внутреннем цикле функция проходится от 0 до `size` - i. Если a[j] > a[j+1], то элементы a[j] и a[j+1] меняются местами. Переменная `hold` используется для хранения временного значения при свопе элементов.

Листинг 10.2. Пузырьковая сортировка

```
#include <stdio.h>
     .
void bubble_sort(int a[], int size);

int main(void) {
 int arr[10] = {10, 2, 4, 1, 6, 5, 8, 7, 3, 9};
 int i = 0;
```

```
 printf("До сортировки:\n");
 for(i = 0; i < 10; i++) printf("%d ", arr[i]);
 printf("\n");

 bubble_sort(arr, 10);

 printf("После:\n");
 for(i = 0; i < 10; i++) printf("%d ", arr[i]);
 printf("\n");

 return 0;
}

void bubble_sort(int a[], int size) {
 int switched = 1;
 int hold = 0;
 int i = 0;
 int j = 0;

 size -= 1;

 for(i = 0; i < size && switched; i++) {
  switched = 0;
  for(j = 0; j < size - i; j++)
   if(a[j] > a[j+1]) {
    switched = 1;
    hold = a[j];
    a[j] = a[j + 1];
    a[j + 1] = hold;
   }
 }
}
```

Еще раз отмечу, что данный алгоритм очень неэффективный: общее число сравнений равно (N-1)N, то есть если массив состоит из 10 элементов, как у нас, то программа выполнила 90 сравнений, чтобы отсортировать массив. Это настоящее расточительство ресурсов: представьте, что будет, если элементов будет не 10, а один миллион?! Тем не менее, этот алгоритм часто используется при обучении программированию. Если вы так и не разобра-

лись, как он работает, на страничке в Википедии можно увидеть анимацию, демонстрирующую алгоритм в динамике: **https://goo.gl/KGE6yn**

10.3. Пузырьковая сортировка связного списка

Давайте усложним нашу предыдущую задачу и выполним пузырьковую сортировку связного списка. Алгоритм будет таким же, но работать мы будем не с массивом, а со связным списком. Подобная задача - хорошая практика по работе с указателями, а они играют в Си очень важную роль - ни одна серьезная программа на этом языке программирования не обходится без указателей. В то же время большинство ошибок, допускаемых начинающими программистами, связаны как раз с работой с указателями, поэтому чем больше практики по работе с указателями у вас будет, тем лучше.

Как уже было отмечено, сам алгоритм сортировки останется тем же (только мы его слегка модифицируем). Но кроме него нам нужно реализовать еще две вспомогательных функции:

```
/* добавляет новый узел в связный список */
void llist_add(struct lnode **q, int num);
/* выводит результат */
void llist_print(void);
```

Рассмотрим сначала функцию `llist_add()`. Ей передаются два параметра - указатель на список и число, которое нужно добавить в список. Если список пуст, то она создает первый узел - выделяет память с помощью `malloc()`:

```
if(*q == NULL) {
  *q = malloc(sizeof(struct lnode));
```

Функция "перематывает" список, чтобы добраться к последнему узлу:

```
  /* переходим к последнему узлу */
```

```
  while(tmp->next != NULL)
   tmp = tmp->next;

   /* добавляем узел в конец списка */
   tmp->next = malloc(sizeof(struct lnode));
   tmp = tmp->next;
 }
```

Напомню, последним считается узел, у которого указатель на следующий узел (next) равен NULL. Поэтому в самой "перемотке" нет ничего сложного - нужно двигаться, пока next не будет равен NULL.

Как только мы "перемотали" список и добрались до последнего элемента, нужно присвоить ему данные:

```
tmp->data = num;
tmp->next = NULL;
```

Функция вывода связного списка очень проста. Она похожа на перемотку списка, только при этой самой перемотке мы выводим значение текущего элемента списка:

```
void llist_print(void) {
 visit = head;

 while(visit != NULL) {
  printf("%d ", visit->data);
  visit = visit->next;
 }
 printf("\n");
}
```

При программировании связных списков очень важно не "потерять голову". Следите за указателем head - одно "неправильное движение" и вы можете потерять весь список. Именно поэтому везде нужно работать с указателем visit (можете назвать его temp - это уже как вам захочется). А указатель head должен оставаться неизменным.

Сортировка связного списка осуществляется функцией `llist_bubble_sort()`. В ней, как и в предыдущем случае, есть два цикла - внешний и внутренний, только для большего удобства циклы заменены на `while()`:

```
while(e != head->next) {
 c = a = head;
 b = a->next;
  while(a != e) {
```

В программе мы будем генерировать случайные числа, и ними же будем заполнять наш список - чтобы избавить вас от ввода чисел вручную.

Листинг 10.3. Пузырьковая сортировка связного списка

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

#define MAX 10

struct lnode {
 int data;
 struct lnode *next;
} *head, *visit;

/* добавляем новый узел в связный список */
void llist_add(struct lnode **q, int num);
/* осуществляем сортировку связного списка */
void llist_bubble_sort(void);
/* выводим результат */
void llist_print(void);

int main(void) {
 /* связный список */
 struct lnode *newnode = NULL;
 int i = 0;      /* общий счетчик */

 /* загружаем случайные числа в связный список */
 for(i = 0; i < MAX; i++) {
  llist_add(&newnode, (rand() % 100));
```

```
 }

 head = newnode;
 printf("До сортировки:\n");
 llist_print();
 printf("После сортировки:\n");
 llist_bubble_sort();
 llist_print();

 return 0;
}

/* добавляем узел в конец связного списка */
void llist_add(struct lnode **q, int num) {
 struct lnode *tmp;

 tmp = *q;

 /* если список пуст, создаем первый узел */
 if(*q == NULL) {
  *q = malloc(sizeof(struct lnode));
   tmp = *q;
 } else {
  /* переходим к последнему узлу */
  while(tmp->next != NULL)
   tmp = tmp->next;

   /* добавляем узел в конец списка */
   tmp->next = malloc(sizeof(struct lnode));
   tmp = tmp->next;
 }

 /* присваиваем данные последнему узлу */
 tmp->data = num;
 tmp->next = NULL;
}

/* выводим связный список */
void llist_print(void) {
 visit = head;
```

```
 while(visit != NULL) {
  printf("%d ", visit->data);
  visit = visit->next;
 }
 printf("\n");
}

/* пузырьковая сортировка связного списка */
void llist_bubble_sort(void) {
 struct lnode *a = NULL;
 struct lnode *b = NULL;
 struct lnode *c = NULL;
 struct lnode *e = NULL;
 struct lnode *tmp = NULL;

// Алгоритм пузырьковой сортировки, адаптированный
// под связный список
while(e != head->next) {
 c = a = head;
 b = a->next;
  while(a != e) {
   if(a->data > b->data) {
    if(a == head) {
     tmp = b -> next;
     b->next = a;
     a->next = tmp;
     head = b;
     c = b;
    } else {
     tmp = b->next;
     b->next = a;
     a->next = tmp;
     c->next = b;
     c = b;
    }
   } else {
    c = a;
    a = a->next;
   }
   b = a->next;
   if(b == e)
```

```
    e = a;
  }
 }
}
```

10.4. Пузырьковая сортировка массива строк

Ваша коллекция алгоритмов пузырьковой сортировки была бы неполной, если бы мы не рассмотрели программу для пузырьковой сортировки массива строк. Для большего разнообразия в этом случае мы будем вводить строки вручную.

Программа сортировки массива строк будет гораздо более простой, чем в случае со связным списком. Во-первых, нам не нужно постоянно следить за указателями. Во-вторых, не нужно бояться "потерять голову" (как свою, так и списка). В-третьих, на помощь нам приходит уже знакомая функция сравнения строк `strcmp()`, которая существенно упрощает процесс сортировки.

Вспомогательная функция `swap()` меняет местами два слова, а сортировка выполняется функцией `sort_words()`. При этом у нас есть две константы: MAX и N. Первая задает максимальную длину слова, вторая - максимальное количество слов. Обе константы равны 50, но вы можете указать другие значения. Для завершения ввода нажмите Ctrl + D - если вам не хочется вводить все 50 слов.

Листинг 10.4. Сортировка массива строк пузырьком

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

#define MAX 50     // максимальная длина слова
#define N 50       // максимальное количество слов

void sort_words(char *x[], int y);
void swap(char **, char **);
```

```
int main(void) {
 char word[MAX];
 char *x[N];
 int n = 0;
 int i = 0;

 printf("Введите слова, для завершения ввода нажмите Ctrl + D: \n");

 for(i = 0; scanf("%s", word) == 1; ++i) {
  if(i >= N)
   printf("Достигнут лимит: %d\n", N), exit(1);

  x[i] = calloc(strlen(word)+1, sizeof(char));
  strcpy(x[i], word);
 }

 n = i;
 printf("\nРезультат:\n");
 sort_words(x, n);
  for(i = 0; i < n; ++i)
   printf("%s\n", x[i]);

  return(0);
}

void sort_words(char *x[], int y) {
 int i = 0;
 int j = 0;

 for(i = 0; i < y; ++i)
  for(j = i + 1; j < y; ++j)
   if(strcmp(x[i], x[j]) > 0)
    swap(&x[i], &x[j]);
}

void swap(char **p, char **q) {
 char *tmp;

 tmp = *p;
 *p = *q;
```

```
 *q = tmp;
}
```

У нашей программы есть один очень полезный "побочный эффект". Чтобы вы не думали, что все эти примеры не имеют практической ценности, сообщаю прекрасную новость. Наша программа умеет читать со стандартного ввода. Следовательно, ее можно использовать для сортировки реальных файлов. Все, что нужно для этого сделать - перенаправить список, прочитанный из файла на нашу программу, например:

```
cat sort.txt | ./sorting
```

В данном случае `sorting` – название исполнимого файла нашей программы.

Чтобы превратить нашу учебную программу в "боевую", выполните следующие рекомендации:

1. Увеличьте максимальную длину слова, скажем до 100 символов. Тогда программу можно будет использовать для сортировки имен файлов, например, для списка воспроизведения.
2. Увеличьте количество элементов до 2000. Можно и больше, но 2000, думаю, будет достаточно.
3. Уберите операторы `printf()`, выводящие подсказу и слово "Результат", иначе эти строки также будут помещены в вывод программы, что нежелательно.

10.5. Сортировка кучей или пирамидальная сортировка

Данный алгоритм является модификацией пузырьковой сортировки и представляет собой что-то среднее между сортировкой выбором и пузырьковой сортировкой.

Идея алгоритма заключается в следующем: ищем максимальный элемент в неотсортированной части массива и ставим его в конец этого подмассива.

В поисках максимума подмассив перестраивается в так называемое сортирующее дерево (она же двоичная куча, она же пирамида), в результате чего максимум сам "всплывает" в начало массива.

После этого над оставшейся частью массива снова осуществляется процедура перестройки в сортирующее дерево с последующим перемещением максимума в конец подмассива.

Что такое сортирующее дерево? Это такое дерево, у которого любой родитель не меньше, чем каждый из его потомков - так называемое неубывающее дерево. Есть и невозрастающее дерево - это когда любой родитель не больше, чем каждый из его потомков.

Листинг 10.5. Пирамидальная сортировка

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

/* максимальная длина массива ... */
#define MAXARRAY 5

/* осуществляет пирамидальную сортировку */
void heapsort(int ar[], int len);
/* помогает heapsort() "выталкивать" элементы, начиная с позиции pos */
void heapbubble(int pos, int ar[], int len);

int main(void) {
 int array[MAXARRAY];
 int i = 0;

 /* загружаем случайные элементы в массив */
 for(i = 0; i < MAXARRAY; i++)
  array[i] = rand() % 100;

 /* выводим исходный массив */
 printf("До сортировки: ");
 for(i = 0; i < MAXARRAY; i++) {
  printf(" %d ", array[i]);
 }
 printf("\n");
```

```
 /* Сортировка */
 heapsort(array, MAXARRAY);

 /* результат */
 printf("ПОсле сортировки: ");
 for(i = 0; i < MAXARRAY; i++) {
  printf(" %d ", array[i]);
 }
 printf("\n");

 return 0;
}

void heapbubble(int pos, int array[], int len) {
 int z = 0;
 int max = 0;
 int tmp = 0;
 int left = 0;
 int right = 0;

 z = pos;
 for(;;) {
  left = 2 * z + 1;
  right = left + 1;

  if(left >= len)
   return;
  else if(right >= len)
   max = left;
  else if(array[left] > array[right])
   max = left;
  else
   max = right;

  if(array[z] > array[max])
   return;

  tmp = array[z];
  array[z] = array[max];
  array[max] = tmp;
  z = max;
```

```
 }
}

void heapsort(int array[], int len) {
 int i = 0;
 int tmp = 0;

 for(i = len / 2; i >= 0; --i)
  heapbubble(i, array, len);

 for(i = len - 1; i > 0; i--) {
  tmp = array[0];
  array[0] = array[i];
  array[i] = tmp;
  heapbubble(0, array, i);
 }
}
```

10.6. Сортировка слиянием связного списка

Рассмотрим еще один алгоритм сортировки - сортировка слиянием (merge sort в англ. литературе). Это, нужно отметить, довольно эффективный алгоритм..

Сортировка слиянием — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке.

Слияние означает объединение двух (или более) последовательностей в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент.

Алгоритм довольно непростой. Попробую объяснить все по-простому. У нас есть два списка (или массива - не важно). Мы будем брать поочередно по одному элементу из каждого массива, сравнивать их и "сливать" в один массив. Меньший элемент будем ставить первым, больший – вторым.

А что делать, если у нас есть только один список (массив)? Тогда его нужно разбирать на две части примерно одинакового размера. Далее каждая из получившихся частей сортируется отдельно, после чего два упорядоченных массива соединяются в один. Это и есть сортировка слиянием.

В процессе сортировки мы рекурсивно вызываем функцию сортировки, пока размер массива не достигнет единицы. Любой массив (список), состоящий из одного элемента, можно считать упорядоченным. За сортировку слиянием отвечает функция `mergesort()`, которая была реализована специально для этого примера:

```
struct node *mergesort(struct node *head) {
 struct node *head_one;
 struct node *head_two;

 if((head == NULL) || (head->next == NULL))
  return head;

 head_one = head;
 head_two = head->next;
 while((head_two != NULL) && (head_two->next != NULL)) {
  head = head->next;
  head_two = head->next->next;
 }
 head_two = head->next;
 head->next = NULL;

 return merge(mergesort(head_one), mergesort(head_two));
}
```

Поскольку мы используем рекурсию, то мы должны предусмотреть условие выхода из рекурсии. В нашем случае условие выхода будет таким:

```
if((head == NULL) || (head->next == NULL))
  return head;
```

То есть или список пуст или список состоит из одного элемента (нет следующего, поэтому next указывает на NULL). В этом случае мы возвраща-

ем head, во всех остальных мы возвращаем merge(mergesort(head_one), mergesort(head_two));

Функция merge() выполняет непосредственно слияние списков. Мы передаем ей две головы двух списков, она выполняет слияние и возвращает его результат.

Дополнительную информацию об этом алгоритме вы можете получить на страничке Википедии: **https://goo.gl/natPWf**. На ней также вы найдете реализацию алгоритма на разных языках программирования - Си, C++. Не будет лишнем и просмотреть визуализацию алгоритма - как он работает. А я привожу собственную реализацию - см. листинг 10.6.

Листинг 10.6. Сортировка слиянием связного списка

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

struct node {
 int number;
 struct node *next;
};

/* добавляем узел в связный список  */
struct node *addnode(int number, struct node *next);
/* сортировка слиянием */
struct node *mergesort(struct node *head);
/* слияние списков */
struct node *merge(struct node *head_one, struct node *head_two);

int main(void) {
 struct node *head;
 struct node *current;
 struct node *next;
 int test[] = {8, 3, 2, 6, 1, 5, 4, 7, 9, 0};
 int i;

 head = NULL;
 /* вставляем числа в связный список */
 for(i = 0; i < 10; i++)
  head = addnode(test[i], head);
```

```
 /* сортируем список */
 head = mergesort(head);

 /* выводим результат */
 printf(" До  После\n"), i = 0;
 for(current = head; current != NULL; current = current->next)
  printf("%4d\t%4d\n", test[i++], current->number);

 /* освобождаем память */
 for(current = head; current != NULL; current = next)
  next = current->next, free(current);

 /* все */
 return 0;
}

/* добавляем узел в связный список */
struct node *addnode(int number, struct node *next) {
 struct node *tnode;

 tnode = (struct node*)malloc(sizeof(*tnode));

 if(tnode != NULL) {
  tnode->number = number;
  tnode->next = next;
 }

 return tnode;
}

/* сортировка слиянием связного списка */
struct node *mergesort(struct node *head) {
 struct node *head_one;
 struct node *head_two;

 if((head == NULL) || (head->next == NULL))
  return head;

 head_one = head;
 head_two = head->next;
```

```
 while((head_two != NULL) && (head_two->next != NULL)) {
  head = head->next;
  head_two = head->next->next;
 }
 head_two = head->next;
 head->next = NULL;

 return merge(mergesort(head_one), mergesort(head_two));
}

/* слияние списков */
struct node *merge(struct node *head_one, struct node *head_
two) {
 struct node *head_three;

 if(head_one == NULL)
  return head_two;

 if(head_two == NULL)
  return head_one;

 if(head_one->number < head_two->number) {
  head_three = head_one;
  head_three->next = merge(head_one->next, head_two);
 } else {
  head_three = head_two;
  head_three->next = merge(head_one, head_two->next);
 }

 return head_three;
}
```

10.7. Быстрая сортировка массива

Быстрая сортировка или сортировка Хоара (по имени разработчика алгоритма) - широко известный алгоритм сортировки, разработанный английским программистом Чарльзом Хоаром в 1960 году. Не удивляйтесь - большинство алгоритмов сортировки были разработаны очень давно, примерно

в 60-ых годах 20-го век, но они не потеряли свою актуальность до сих пор - пока никто ничего лучше не придумал.

Часто быструю сортировку называют `qsort` - по имени в стандартной библиотеке языка Си. Да, есть функция `qsort()`, можно использовать ее, но нам это не интересно. Гораздо интереснее написать собственную реализацию.

Быстрая сортировка - это улучшенный вариант пузырьковой сортировки, но эффективность этого алгоритма значительно выше. Принципиальное отличие заключается в том, что первым делом производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы. Интересно, что незначительное улучшение самого неэффективного алгоритма породило один из самых эффективных алгоритмов сортировки. Он эффективен до такой степени, что его включили в стандартную библиотеку функций Си.

Алгоритм заключается в следующем. Мы выбираем некоторый элемент - опорный элемент. Обычно это медиана - то есть элемент в середине массива.

Далее выполняется операция разделения: реорганизуем массив таким образом, чтобы все элементы со значением меньшим или равным опорному элементу, оказались слева от него, а все элементы, превышающие по значению опорный — справа от него.

Рекурсивно нужно упорядочить подмассивы, лежащие слева и справа от опорного элементы. Условие выхода из рекурсии - массив, состоящий из одного элемента (или пустой массив). Учитывая, что при каждой итерации длина обрабатываемого отрезка массива уменьшается как минимум на единицу, условие выхода из рекурсии обязательно будет достигнуто, и обработка массива гарантированно будет прекращена.

Программная реализация приведена в листинге 10.7.

Листинг 10.7. Быстрая сортировка массива

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

#define MAXARRAY 10
```

```
void quicksort(int arr[], int low, int high);

int main(void) {
 int array[MAXARRAY] = {0};
 int i = 0;

 /* загружаем в массив случайные числа */
 for(i = 0; i < MAXARRAY; i++)
  array[i] = rand() % 100;

 /* выводим массив */
 printf("До сортировки: ");
 for(i = 0; i < MAXARRAY; i++) {
  printf(" %d ", array[i]);
 }
 printf("\n");

 quicksort(array, 0, (MAXARRAY - 1));

 /* выводим результат */
 printf("После сортировки: ");
 for(i = 0; i < MAXARRAY; i++) {
  printf(" %d ", array[i]);
 }
 printf("\n");

 return 0;
}

/* сортируем все между `low' <-> `high' */
void quicksort(int arr[], int low, int high) {
 int i = low;
 int j = high;
 int y = 0;
 /* опорный элемент */
 int z = arr[(low + high) / 2];

 /* разделение */
 do {
  /* находим элемент левее */
  while(arr[i] < z) i++;
```

```
  /* находим элемент правее */
  while(arr[j] > z) j--;

  if(i <= j) {
   /* меняем местами 2 элемента */
   y = arr[i];
   arr[i] = arr[j];
   arr[j] = y;
   i++;
   j--;
  }
 } while(i <= j);

 /* рекурсия */
 if(low < j)
  quicksort(arr, low, j);

 if(i < high)
  quicksort(arr, i, high);
}
```

Язык Си содержит библиотечную функцию `qsort()` – уже готовую реализацию быстрой сортировки. Функции `qsort()` нужно передать следующие параметры:

- `base` - указатель на первый элемент массива, который нужно отсортировать.
- `nitems` - количество элементов в массиве.
- `size` - размер каждого элемента в байтах.
- `compare` - функция, которая будет сравнивать два элемента.

Прототип функции выглядит так:

```
void qsort(void *base, size_t nitems, size_t size, int
(*compar)(const void *, const void*))
```

Рассмотрим пример кода. Сначала определим пользовательские функции сравнения и сортировки:

```
// Наша пользовательская функция сравнения
// Мы просто сделали "обертку" для strcmp
static int cmpr(const void *a, const void *b) {
 return strcmp(*(char **)a, *(char **)b);
}

void sortstrarr(void *array, unsigned n) {
 // Вызываем функцию qsort и передаем ей все необходимое
 qsort(array, n, sizeof(char *), cmpr);
}
```

Первая функция просто вызывает `strcmp()` для сравнения, а вторая будет вызывать `qsort()`, просто нам будет так удобнее в коде основной программы. Далее прочитаем массив строк с клавиатуры и отсортируем его:

```
char line[1024];
 char *line_array[1024];
 int i = 0;
 int j = 0;

 // Читаем данные со стандартного ввода
 while((fgets(line, 1024, stdin)) != NULL)
  if(i < 1024)
   line_array[i++] = strdup(line);
  else
   break;

 // Сортируем массив
 sortstrarr(line_array, i);

 // выводим результат
 while(j < i)
  printf("%s", line_array[j++]);
```

10.8. Сортировка с помощью бинарного дерева

Еще один классический способ сортировки - это сортировка с помощью бинарного дерева (treesort). Этот же способ сортировки называют сортировкой двоичным деревом, просто сортировкой деревом, а также treesort в англоязычной литературе.

Алгоритм заключается в построении двоичного дерева поиска по ключам массива (списка) с последующей сборкой результирующего массива путем обхода узлов построенного дерева в необходимом порядке следования ключей. Данный алгоритм идеально подходит для сортировки данных, полученных путем непосредственного чтения с потока (из файла, со стандартного ввода, сокета и т.д.).

Собственно, алгоритм состоит из двух частей:

1. Построение двоичного дерева.
2. Сборка результирующего массива путем обхода узлов в необходимом порядке следования ключей.

В отличие от предыдущего алгоритма, здесь простое и описание, и реализация. Код программы, выполняющей сортировку стандартного ввода методом двоичного дерева, приведен в листинге 10.8.

Листинг 10.8. Сортировка двоичных деревом

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

struct tnode {
 char *str;
 struct tnode *left;
 struct tnode *right;
};

void insert(struct tnode **p, char *value);
```

```
void print(struct tnode *root);

int main(void) {
 char line[1024];
 struct tnode *root;

 root = NULL;
 while((fgets(line, 1024, stdin)) != NULL)
  insert(&root, line);

 print(root);
 return 0;
}

/* вызов по ссылке  */
void insert(struct tnode **p, char *value) {
 if(!*p) {
  *p = (struct tnode *)malloc(sizeof(struct tnode));
  (*p)->left = (*p)->right = NULL;
  (*p)->str = strdup(value);
  return;
 }

 if(strcmp(value, (*p)->str) < 0)
  insert(&(*p)->left, value);
 else
  insert(&(*p)->right, value);
}

/* выводим дерево в нужно порядке */
void print(struct tnode *root) {
 if(root != NULL) {
  print(root->left);
  printf("%s", root->str);
  print(root->right);
 }
}
```

10.9. Бинарный поиск в целочисленном массиве

Бинарный (он же двоичный) поиск — классический алгоритм поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины. Данный метод также известен как метод деления пополам.

Если у нас есть массив, содержащий упорядоченную последовательность данных, то очень эффективен двоичный поиск. Да, вы все правильно поняли, бинарный поиск работает только на уже отсортированных массивах, поэтому перед применением бинарного поиска к произвольному массиву (прочитанному из файла или введенному пользователем), его нужно отсортировать.

Переменные `left` и `right` содержат, соответственно, левую и правую границы отрезка массива, где находится нужный нам элемент. Мы начинаем всегда с исследования среднего элемента отрезка (`middle`). Если искомое значение меньше среднего элемента, мы переходим к поиску в верхней половине отрезка, где все элементы меньше только что проверенного. Другими словами, значением `right` становится (`middle - 1`) и на следующей итерации мы работаем с половиной массива. Таким образом, в результате каждой проверки мы вдвое сужаем область поиска. Так, в нашем примере, после первой итерации область поиска – всего лишь 5 элементов.

Двоичный поиск - очень мощный и эффективный метод. Если представить, что длина массива равна 1023, после первого сравнения область сужается до 511 элементов, а после второй - до 255. Легко посчитать, что для поиска в массиве из 1023 элементов достаточно 10 сравнений.

Листинг 10.9. Двоичный поиск в целом (int) массиве

```
#include <stdio.h>

#define TRUE  0
#define FALSE 1

int main(void) {
 int array[10] = {0, 1, 2, 3, 4, 6, 7, 8, 9, 10};
```

```
 int left = 0;
 int right = 10;
 int middle = 0;
 int number = 0;
 int bsearch = FALSE;
 int i = 0;

 printf("Массив: ");
 for(i = 0; i < 10; i++)
  printf("[%d] ", array[i]);

 printf("\nВведите искомый элемент: ");
 scanf("%d", &number);

 while(bsearch == FALSE && left <= right) {
  middle = (left + right) / 2;

  if(number == array[middle]) {
   bsearch = TRUE;
   printf("** Число есть в массиве! **\n");
  } else {
   if(number < array[middle]) right = middle - 1;
   if(number > array[middle]) left = middle + 1;
  }
 }

 if(bsearch == FALSE)
  printf("-- Элемент не найден --\n");

 return 0;
}
```

Дополнительную информацию по этому методу поиска и дополнительный пример кода вы можете получить в Википедии: **https://goo.gl/SKVJYx**

10.10. Бинарный поиск по массиву указателей строк

Ранее было показано, как выполнить поиск по упорядоченному массиву целых чисел. Но на практике чаще возникают задачи поиска определенной

строки, нежели определенного числа. Именно поэтому сейчас будет рассмотрен пример двоичного поиска по массиву указателей строк.

Принцип тот же. Исходный массив должен быть отсортирован. В функцию `binsearch` передается массив строк, размер массива и искомое значение. Функция возвращает 0, если значение не найдено или же позицию найденного значения. Учитывая, что массив отсортирован, средняя позиция определяется как сумма начальной и последней (begin + end), разделенная на 2. Далее нужно сравнить функцией `strcmp()` искомое слово со словом в получившейся позиции. Функция `strcmp()` возвращает значение

- < 0, если первый ее аргумент лексикографически меньше, чем второй;
- > 0, если первый аргумент лексикографически больше, чем второй
- 0, если аргументы равны.

Так вот, функция `strcmp()` не только сравнивает строки, но и еще и подсказывает нам в каком направлении двигаться - в соответствии с этим мы или увеличиваем позицию или уменьшаем ее. Если функция вернула 0, то мы можем вернуть позицию (переменная `position`), в которой это произошло.

Прототип функции `strcmp()` выглядит так:

```
int strcmp(const char *str1, const char *str2)
```

Код примера, реализующего бинарный поиск по массиву строк, приведен в листинге 10.10.

Листинг 10.10. Бинарный поиск по массиву строк

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

static int binsearch(char *str[], int max, char *value);

int main(void) {
 /* массив должен быть отсортирован... */
 char *strings[] = { "audi", "bentley", "bmw", "cadillac", "ford"
};
 int i, asize, result;
```

```
 i = asize = result = 0;

 asize = sizeof(strings) / sizeof(strings[0]);

 for(i = 0; i < asize; i++)
  printf("%d: %s\n", i, strings[i]);

 printf("\n");

 if((result = binsearch(strings, asize, "bmw")) != 0)
  printf("`bmw' найдено на позиции: %d\n", result);
 else
  printf("`bmw' не найдено..\n");

 if((result = binsearch(strings, asize, "mercedes")) != 0)
  printf("`mercedes' найдено на позиции %d\n", result);
 else
  printf("`mercedes' не найдено..\n");

 return 0;
}

static int binsearch(char *str[], int max, char *value) {
 int position;
 int begin = 0;
 int end = max - 1;
 int cond = 0;

 while(begin <= end) {
  position = (begin + end) / 2;
  if((cond = strcmp(str[position], value)) == 0)
   return position;
  else if(cond < 0)
   begin = position + 1;
  else
   end = position - 1;
 }

 return 0;
}
```

Глава 11.
Многопоточность в Си

Современные приложения редко бывают однопоточными, особенно, если речь идет о серьезном приложении, а не о программе-калькуляторе. Все достаточно просто – современные процессоры, даже самые простые – имеют два ядра, следовательно, одновременно, могут выполняться два потока. Современное приложение должно использовать возможности современных процессоров для более эффективной работы.

Можно привести следующие примеры многопоточных приложений:

- Серверы – многопоточный сервер может обслуживать несколько клиентов одновременно. Типичный пример – веб-сервер Apache, который запускает одновременно несколько потоков – по одному на клиента.
- Вычислительное приложение – на многопроцессорной/многоядерной системе такие приложения будут работать гораздо эффективнее при использовании многопоточности.
- Приложения реального времени – здесь без поддержки нескольких потоков не обойтись, поскольку такие приложения должны реагировать буквально мгновенно.

Поддержка многопоточности в Си появилась, начиная со стандарта C11. Поэтому если ваш компилятор до сих пор его не поддерживает, самое время обновиться. Впрочем, в стандарте C99 есть возможность использования POSIX-потоков библиотеки pthread. Оба варианта будут рассмотрены в этой главе.

11.1. Введение в потоки

Поток (анг. thread) – это независимое выполнение последовательности инструкций. Поток – это не процесс, поток работает внутри процесса и использует то же пространство в памяти, что и процесс. Можно сказать, что поток – это облегченная версия процесса.

Однако у каждого потока есть свое окружение времени выполнения – другими словами, у каждого потока есть свое собственное пространство для хранения переменных.

Нужно отметить, что многопоточные приложения будут быстрее работать даже однопроцессорных/одноядерных машинах – они лучше используют процессорное время. К недостаткам таких приложений можно отнести разве что сложность их разработки.

Есть множество ситуаций, где потоки могут быть полезны:

- Более эффективное использование доступных вычислительных ресурсов
  - Когда процесс ждет ресурсы (например, читает данные с периферийных устройств), он блокируется и контроль передается другому процессу.
  - Поток также ждет, но другой поток в этом же процессе может за это время сделать что-то полезное.
  - Многоядерные и многопоточные процессоры могут существенно ускорить вычислительный процесс, но для этого приложение должно уметь использовать все их преимущества, о чем было сказано.
- Обработка асинхронных событий:

  - Во время заблокированной операции ввода/вывода другой поток может использовать процессор для других вычислений.
  - Один поток можно выделить для операций ввода/вывода, например, для работы с каналом связи, остальные потоки будут работать для вычислений. Вы можете использовать несколько потоков, например, для загрузки больших файлов, как это делают торрент-клиенты.
- Взаимодействие с интерфейсом пользователя (GUI)
  - Здесь настоящий плацдарм для использования многопотоковых приложений. Примеров можно привести множество, например, один поток будет обрабатывать взаимодействие с пользователем, второй выполнять какую-то фоновую работу, например, воспроизведение музыки. Использование многопотоковых приложений позволяет более быстро реагировать на действия пользователя – пользователи будут довольны.

Таблица 11.1 описывает разницу между процессом и потоком.

Таблица 11.1. Разница между процессом и потоком

Процесс	**Поток**
Вычислительный поток	Вычислительный поток
Собственное пространство в памяти	Разделяет то же пространство, что и поток
Синхронизируется с использованием операционной системы (IPC)	Синхронизируется путем эксклюзивного доступа к переменным
На создание процесса необходимо больше времени	Создание потока требует меньше времени, чем процесса

Преимущества потоков выглядят так:

- Потоки довольно просто обмениваются данными по сравнению с процессами.
- Создавать потоки для ОС проще и быстрее, чем создавать процессы.

К недостаткам потоков можно отнести следующие вещи:

- При программировании приложения с множественными потоками необходимо обеспечить потоковую безопасность функций — т. н. thread safety. Приложения, выполняющиеся через множество процессов, не имеют таких требований. Об этом мы еще поговорим.
- Один поток с ошибкой (имеется в виду логическая ошибка) может повредить остальные, так как потоки делят общее адресное пространство. Процессы более изолированы друг от друга.
- Потоки конкурируют друг с другом в адресном пространстве. Стек и локальное хранилище потока, захватывая часть виртуального адресного пространства процесса, тем самым делает его недоступным для других потоков. Для встроенных устройств такое ограничение может иметь существенное значение.

11.2. Типы потоков

Потоки выполняются внутри процесса, но в зависимости от реализации они делятся на два типа:

- **Пользовательские** – используют пользовательское пространство процесса и реализованы с помощью какой-то пользовательской библиотеки. Таким потокам не нужна специальная поддержка от операционной системы, но они обычно не могут использовать все преимущества многоядерных процессоров. Обычно такие потоки управляются локальным планировщиком, предоставляемым библиотекой.
- **Сущности операционной системы** – управляют планировщиком операционной системы и могут использовать все преимущества многоядерных/многопроцессорных систем.

Разница между этими двумя типами приведена на рис. 11.1а и 11.1б.

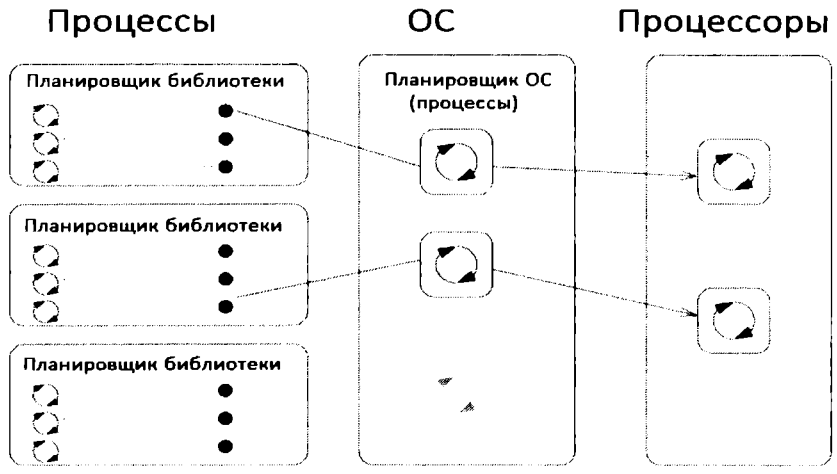


Рис. 11а. Потоки в пользовательском пространстве

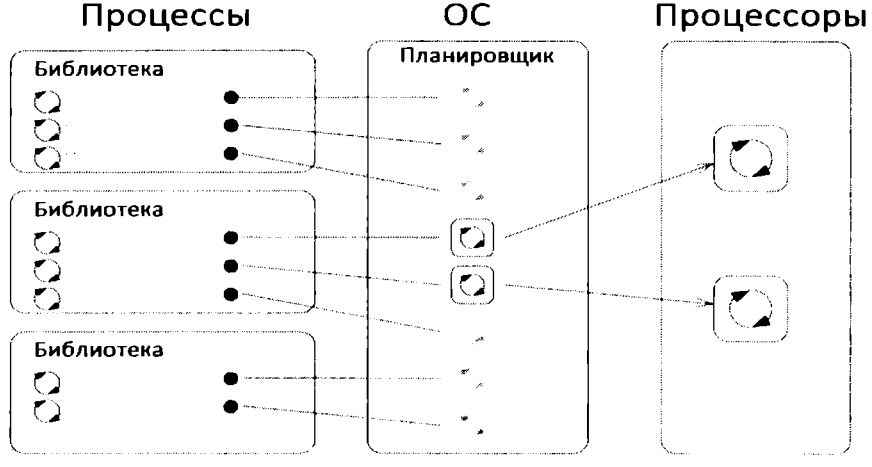


Рис. 11б. Потоки как сущности операционной системы

Таблица 11.2. Разница между пользовательскими потоками и потоками уровня ОС

Пользовательские потоки	Потоки уровня ОС
Не нуждаются в поддержке ОС	Потоки управляются планировщиком ОС вместе со всеми другими потоками в системе
Создание потока не требует системный вызов	Потоки могут быть запущены параллельно, на многоядерной/многопроцессорной системе достигается настоящий параллелизм
Потоки не могут выполняться по-настоящему параллельно (псевдо-параллелизм)	Создание потока более сложное (системный вызов)

11.3. Модели многопотоковых приложений

Существуют следующие модели многопоточных приложений:

- **Хозяин/рабочий (boss/worker)** – создается основной поток, контролирующий работу остальных потоков.
- **Одноранговая модель (peer)** – несколько потоков запускаются параллельно без выделенного потока управления.
- **Конвейерная модель (pipeline)** – здесь задание передается от одного потока к другому.

При проектировании многопоточных систем эти модели обладают целым рядом преимуществ, к числу которых можно отнести следующие:

- Большинство проблем многопоточного программирования могут быть разрешены с использованием одной из стандартных моделей, облегчающих проектирование, разработку и отладку программ.
- Применение понятных и проверенных моделей не только позволяет избежать многих ошибок, которые легко допустить при написании много-

поточных программ, но и способствует повышению производительности результирующих приложений.

- Эти модели естественным образом соответствуют структуре большинства обычных задач программирования.
- Программистам, сопровождающим программу, будет гораздо легче понять ее устройство, если она будет описана в документации на понятном языке.
- Находить неполадки в незнакомой программе гораздо легче, если ее можно анализировать в терминах моделей. Очень часто главную причину неполадок удается установить на основании видимых нарушений базовых принципов одной из моделей.

11.3.1. Модель "Хозяин/рабочий"

Рассмотрим подробнее модель "хозяин/рабочий" (рис. 11.3). В этой модели основной поток (хозяин) отвечает за управление запросами. Он работает в цикле:

- Получаем новый запрос.
- Создаем поток для обслуживания полученного запроса или передаем информацию уже работающему потоку, если тот запущен и простаивает (тот же Apache при своем запуске создает определенное количество потоков, которые всегда находятся "на подхвате" – в ожидании нового запроса).
- Ждем новый запрос.

Результаты (вывод) запроса могут контролироваться с использованием механизмов синхронизации основного потока, например, очередь событий.

А теперь немного псевдо-кода. Код основного потока:

```
// Boss
while (1) {
  switch(getRequest()) {
    case taskX:
       create_thread(taskX);
       break;
```

```
    case taskY:
       create_thread(taskY);
       break;
   }
}
```

Код рабочих может быть таким:

```
<тип> taskX()
{
  обрабатываем запрос
  // синхронизация осуществляется посредством общих ресурсов
}
<тип> taskY()
{
  обрабатываем запрос
```

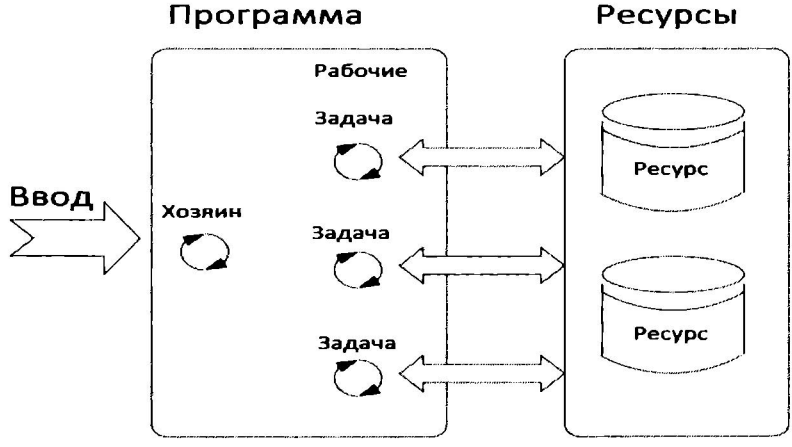


Рис. 11.3. Модель хозяин/рабочий

11.3.2. Одноранговая модель

Одноранговая модель изображена на рис. 11.4. Как видите, она похожа немного на модель хозяин/рабочий, но у нее нет основного потока. Первый поток в этой модели создает все остальные потоки и затем он становится

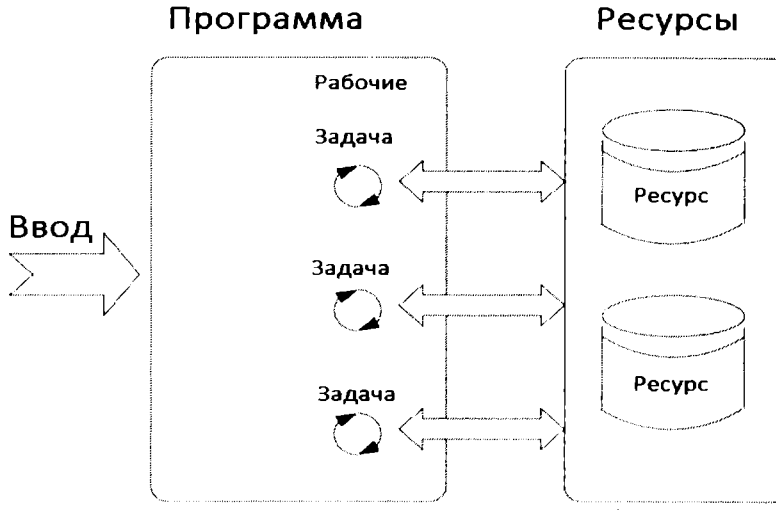


Рис. 11.4. Одноранговая модель

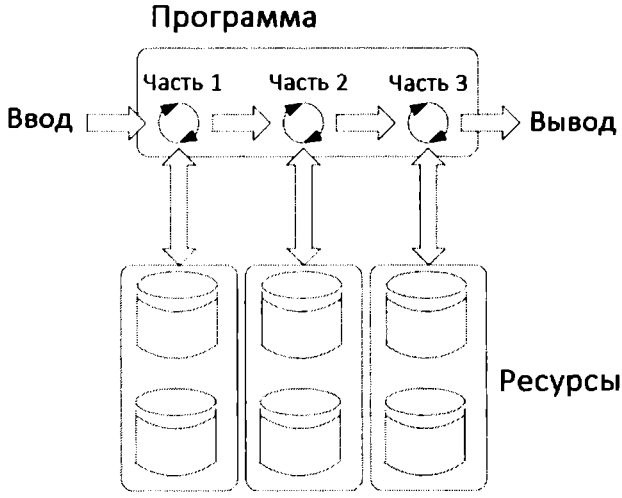
В этом случае код первого потока будет таким:

```
{
  create_thread(task1);
  create_thread(task2);
  …
  запускаем все потоки;
  ожидаем завершения всех потоков;
}

Код потоков может быть таким:

<тип> task1()
{
  ждем запуска
  решаем задачу
  // синхронизация осуществляется посредством общих ресурсов
}

<тип> task2()
{
  ждем запуска
```

```
  решаем задачу
  // синхронизация осуществляется посредством общих ресурсов
}
```

11.3.3. Конвейерная модель

Схема конвейерной модели изображена на рис. 11.5. Здесь у нас есть постоянный поток данных с последовательностью операций (часть обработки) – каждый блок входных данных должен быть обработан всеми частями операций обработки. В определенный момент времени разные блоки данных обрабатываются разными потоками.



Псевдокод будет выглядеть так:

```
int main()
{
  create_thread(stage1);
  create_thread(stage2);
  …
  ждем
}
stage1(void input)
{
  while(input) {
```

```
    получаем следующий ввод от программы;
    обрабатываем input;
    передаем результат следующей стадии;
  }
}
stage2(void input)
{
  while(input) {
    получаем следующий ввод от предыдущего потока;
    обрабатываем input;
    передаем результат следующей стадии;
  }
}
...
stageN(void input)
{
  while(input) {
    получаем следующий ввод от предыдущего потока;
    обрабатываем input;
    передаем результат на вывод;
  }
}
```

11.4. Механизмы синхронизации

Механизмы синхронизации потоков используют те же принципы, что и механизмы синхронизации процессов. Но поскольку потоки разделяют то же пространство памяти, что и процесс, основное взаимодействие между потоками осуществляется через память и глобальные переменные.

Основными примитивами синхронизации являются мьютексы и условные переменные.

Мьютекс (англ. mutex, от mutual exclusion — "взаимное исключение") — одноместный семафор, служащий в программировании для синхронизации одновременно выполняющихся потоков. Мьютекс отличается от семафора тем, что только владеющий им поток может его освободить, т.е. перевести в отмеченное состояние.

Мьютексы — это один из вариантов семафорных механизмов для организации взаимного исключения. Они реализованы во многих операционных

системах, их основное назначение — организация взаимного исключения для потоков из одного и того же или из разных процессов.

Мьютексы — это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт соответственно). Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта mutex, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищенным мьютексом. Если другому потоку будет нужен доступ к переменной, защищенной мьютексом, то этот поток засыпает до тех пор, пока мьютекс не будет освобожден.

Условная переменная — примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. Условные переменные используются вместе с ассоциированным мьютексом и являются элементом некоторых видов мониторов.

Рассмотрим опять псевдо-код, демонстрирующий использование мьютексов и условных переменных:

```
Mutex mtx;
CondVariable cond;

// Thread 1
Lock(mtx);
// Ждем поток 2
CondWait(cond, mtx);
// Критическая секция
...
UnLock(mtx);

// Thread 2
Lock(mtx);
// Критическая секция
// Условный сигнал
```

```
CondSignal(cond, mtx);
UnLock(mtx);
```

11.5. Параллелизм и функции

В многопоточной среде функции могут вызываться одновременно разными потоками. Учитывая параллельное вычисление, функции должны быть потоко-безопасными (thread safe).

Чтобы быть безопасной при параллельном вычислении функция должна жестко ограничить доступ к глобальным данным с использованием примитивов синхронизации.

Если функция не будет соответствовать этим требованиям, возможны различные неприятные эффекты, такие как *взаимная блокировка* (deadlock) – ситуация в многопоточной среде, когда несколько потоков находятся в состоянии ожидания ресурсов, занятых друг другом и ни один из потоков не может продолжить свое выполнение или же *состояние гонки*. Состояние гонки (второй термин – конкуренция) - ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

11.6. POSIX-функции для организации многопоточности

11.6.1. Подключение библиотеки и основные типы данных

Библиотека POSIX для работы с потоками – это набор функций для поддержки многопоточности. Подключается она путем подключения заголовочного файла pthread.h:

```
#include <pthread.h>
```

Откомпилировать программу с поддержкой pthread можно, указав опцию –lpthread:

```
gcc program.c -lpthread
```

Базовые типы для потоков, мьютексов и условных переменных следующие:

- pthread_t – тип, представляющий поток.
- pthread_mutex_t – тип для мьютекса.
- pthread_cond_t – тип для условной переменной.
- pthread_mutexattr_t — объект атрибутов мьютекса
- pthread_condattr_t — объект атрибута условной переменной;
- pthread_key_t — данные, специфичные для потока;
- pthread_once_t — контекст контроля динамической инициализации;
- pthread_attr_t — перечень атрибутов потока.

11.6.2. Запуск потока

Поток создается функций `pthread_create()`. Прототип этой функции следующий:

```
int pthread_create(pthread_t *thread, const pthread_attr_t
*attr,
    void *(*start_routine) (void *), void *arg);
```

Данная функция запускает указанную функцию, как новый поток. Первый параметр – это указатель потока, переменная типа `pthread_t`, второй – атрибуты потока, третий – та самая функция, которая будет выполняться в потоке, последний – аргументы функции `start_routine()`.

Рассмотрим пример многопоточной программы. Можете использовать ее как "болванку" для написания более сложных программ (листинг. 11.1).

Листинг 11.1. Пример многопоточной программы

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

int count;          /* общие данные для потоков */
```

```
int atoi(const char *nptr);
void *potok(void *param); /* потоковая функция */

int main(int argc, char *argv[])
{
  pthread_t tid;          /* идентификатор потока */
  pthread_attr_t attr;          /* атрибуты потока */

if (argc != 2) {
  fprintf(stderr,"usage: thread <integer value>\n");
  return -1;
}

if (atoi(argv[1]) < 0) {
  fprintf(stderr,"Аргумент %d не может быть отрицательным числом\
n",atoi(argv[1]));
  return -1;
}

/* получаем значения атрибутов по умолчанию */
  pthread_attr_init(&attr);

/* создаем новый поток */
  pthread_create(&tid,&attr,potok,argv[1]);

/* ждем завершения исполнения потока */
  pthread_join(tid,NULL);
  printf("count = %d\n",count);
}

/* Контроль переходит потоковой функции */
void *potok(void *param)
{
  int i, upper = atoi(param);
  count = 0;

  if (upper > 0) {
    for (i = 1; i <= upper; i++)
    count += i;
  }
```

```
  pthread_exit(0);
}
```

Откомпилировать программу можно так:

```
gcc -o thread -std=c99 -lpthread progtest.c
```

Программа подсчитывает сумму от 1 до переданного ей в качестве параметра целого числа. Если передать программе число 10, то получите результат 55. Пока в нашей программе всего один поток, но тем не менее программа демонстрирует запуск потока, а также использование общей переменной.

11.6.3. Завершение потока

Поток завершает выполнение задачи, когда:

- потоковая функция выполняет `return` и возвращает результат произведенных вычислений;
- в результате вызова завершения исполнения потока `pthread_exit();`
- в результате вызова отмены потока `pthread_cancel();`
- одна из нитей совершает вызов `exit()`
- основной поток в функции `main()` выполняет `return`, и в таком случае все потоки процесса резко сворачиваются.

Синтаксис проще, чем при создании потока.

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Если в последнем варианте старший поток из функции `main()` выполнит `pthread_exit()` вместо `exit()` или `return`, то тогда остальные потоки продолжат исполняться, как ни в чем не бывало.

11.6.4. Ожидание потока

Функция `pthread_join()` ожидает завершения потока обозначенного THREAD_ID. Если этот поток к тому времени был уже завершен, то функция немедленно возвращает значение. Суть функции в том, чтобы синхронизировать потоки. Синтаксис функции выглядит так:

```
#include <pthread.h>
int pthread_join (pthread_t THREAD_ID, void ** DATA);
```

При удачном завершении `pthread_join()` возвращает код 0, ненулевое значение сигнализирует об ошибке.

Если указатель DATA отличается от NULL, то туда помещаются данные, возвращаемые потоком через функцию `pthread_exit()` или через инструкцию `return` потоковой функции. Несколько потоков не могут ждать завершения одного. Если они пытаются выполнить это, один поток завершается успешно, а все остальные — с ошибкой ESRCH. После завершения `pthread_join()`, пространство стека связанное с потоком, может быть использовано приложением.

В каком-то смысле функция `pthread_joini()` похожа на вызов `waitpid()`, ожидающую завершения исполнения процесса, но с некоторыми отличиями. Во-первых, все потоки одноранговые, среди них отсутствует иерархический порядок, в то время как процессы образуют дерево и подчинены иерархии родитель — потомок. Поэтому возможно ситуация, когда поток А, породил поток Б, тот в свою очередь заделал В, но затем после вызова функции `pthread_join()` А будет ожидать завершения В или же наоборот. Во-вторых, нельзя дать указание одному ожидай завершение любого потока, как это возможно с вызовом waitpid(-1, &status, options). Также невозможно осуществить неблокирующий вызов `pthread_join()`.

11.6.5. Досрочное завершение потока

В некоторых случаях необходимо досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией `pthread_cancel`.

```
int pthread_cancel (pthread_t THREAD_ID);
```

При удачном завершении `pthread_cancel()` возвращает код 0, ненулевое значение сигнализирует об ошибке.

Важно понимать, что несмотря на то, что `pthread_cancel()` возвращается сразу и может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. Дело в том, что поток не только может самостоятельно выбрать момент завершения в ответ на вызов `pthread_cancel()`, но и вовсе его игнорировать. Вызов функции `pthread_cancel()` следует рассматривать как запрос на выполнение досрочного завершения потока. Поэтому, если для вас важно, чтобы поток был удален, нужно дождаться его завершения функцией `pthread_join()`.

11.6.6. Отсоединение потока

Любому потоку по умолчанию можно присоединиться вызовом `pthread_join()` и ожидать его завершения. Однако в некоторых случаях статус завершения потока и возврат значения нам не интересны. Все, что нам надо, это завершить поток и автоматически выгрузить ресурсы обратно в распоряжение ОС. В таких случаях мы обозначаем поток отсоединившимся и используем вызов `pthread_detach()`.

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

При удачном завершении `pthread_detach()` возвращает код 0, ненулевое значение сигнализирует об ошибке.

11.7. Потоки C11

Потоки C11 представляют собой, по сути, оболочку для POSIX-потоков[1]. Подключить необходимую библиотеку можно так:

```
#include <threads.h>
```

При компиляции программы нужно указать опцию `-lstdthreads`.

Базовые типы:

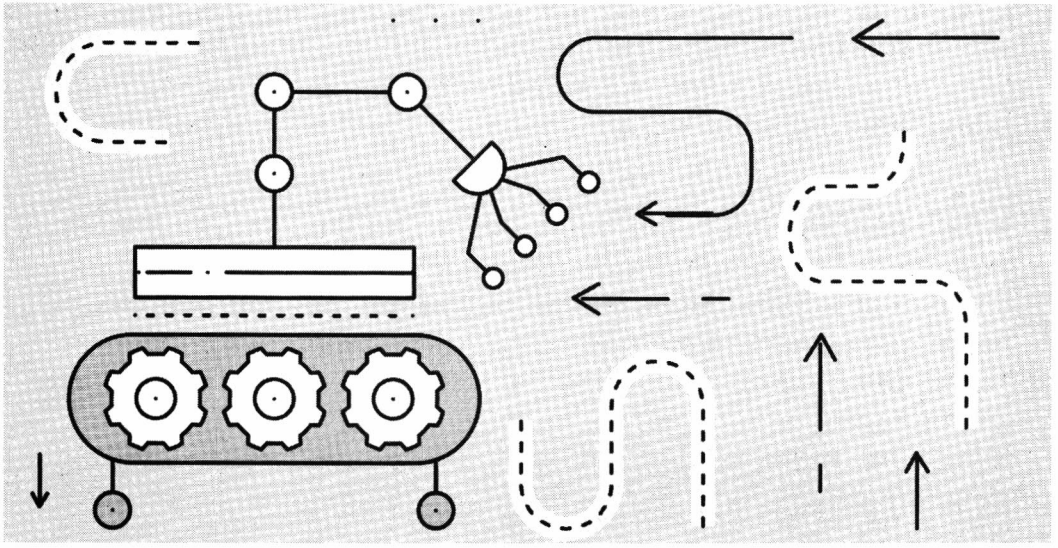
- `thrd_t` – тип для представления потока;
- `mtx_t` – тип для мьютекса;
- `cnd_t` – тип для условной переменной.

1 См. https://en.cppreference.com/w/c/thread

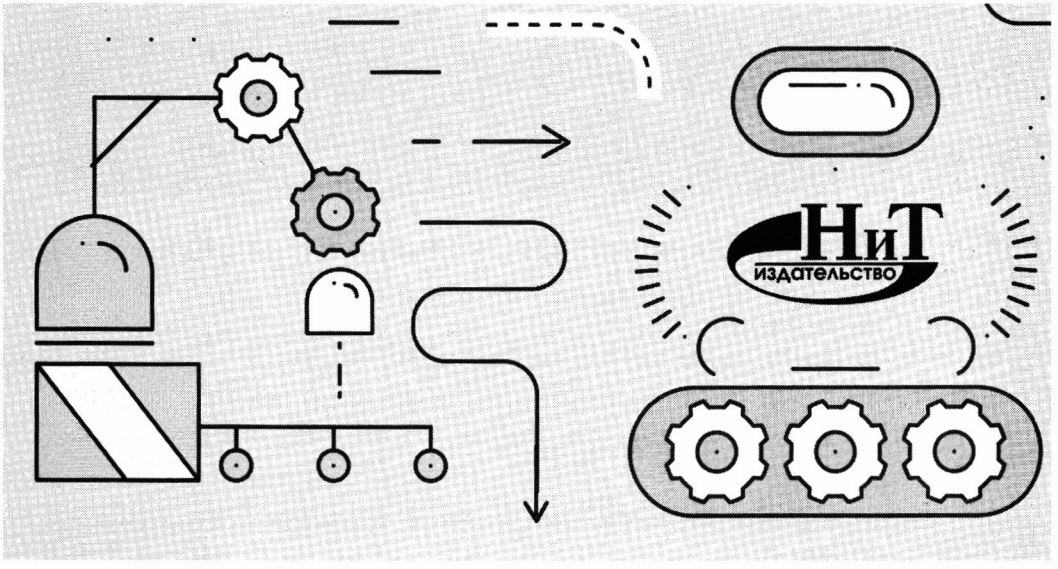
Создание потока осуществляется функцией `thrd_create()`, для подсоединения к потоку используется функция `thrd_join()`. Поскольку потоки в C11 – это оболочка для POSIX-потоков, то работа с функциями осуществляется аналогично, отличаются только имена функций:

- `pthread_*() -> thrd_*()`
- `pthread_mutex_*() -> mxt_*()`
- `pthread_cond_*() -> cnd_*()`

Также нужно помнить, что нет эквивалента для функции `ptread_self()`, а также что потоки, мьютексы и условные переменные создаются и инициализируются без указания определенных атрибутов.



Часть III. Практическое программирование в Си



В этой части книги мы поговорим о двух, на наш взгляд, самых интересных темах – о сетевом программировании и о разработке игр. Поскольку у нашей книги более практическое направление, то мы напишем приложение клиент-сервер и разработает игру с использованием библиотеки glut (OpenGL).

Глава 12. Работа с сетью в Си

12.1. Разработка программы-сервера

В этом примере мы напишем две программы – сервер и клиент. Программа-сервер после запуска сразу же перейдет в режим ожидания ("прослушивания") новых клиентов. Максимальное количество клиентов – 3.

Но сначала нам нужно изучить некоторые структуры и функции, необходимые нашим программам.

Первым делом нам нужно подключить следующие заголовочные файлы:

```
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Они содержат все, что нам нужно для работы с сетью – типы данных, а также сетевые функции.

Для организации связи нам необходимо создать структуру SOCKADDR_IN. Она необходима для выполнения команды `bind` и в ней содержатся параметры связи, такие как порт и атрибуты. Вот ее описание.

```
struct sockaddr_in{
   short sin_family;
   unsigned short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];
};
```

В данной структуре есть вложенная структура `sin_addr` и она описана вот так:

```
struct in_addr {
   union {
      struct{
      unsigned char
         s_b1,
         s_b2,
         s_b3,
         s_b4;
      } S_un_b;
      struct{
      unsigned short
         s_w1,
         s_w2;
      } S_un_w;
      unsigned long S_addr;
   } S_un;
};
```

Первой функцией, которую мы вызовем, будет `socket()`:

```
int sock1, sock2;
sock1 = socket (AF_INET, SOCK_STREAM, 0);
```

С помощью вызова функции `socket()` в области ядра создается неименованный сокет, и возвращается его socket descriptor – целое число, которое мы будем хранить в переменной sock1.

Первым аргументом этой функции передается тип домена. Т.к. мы будем использовать сеть – то используем тип сокета AF_INET (IPv4). Вторым аргументом – SOCK_STREAM, который указывает на тип протокола. Для TCP – это будет SOCK_STREAM, для UDP – SOCK_DGRAM.

Третий аргумент функции `socket()` оставляем по умолчанию – тут ядро само решит какой тип протокола использовать (т.к. мы указали SOCK_STREAM – то будет выбран TCP).

Далее нам нужно вызвать функцию `bind()`. Ее прототип выглядит так:

```
int bind (
   SOCKET s,
   const struct sockaddr FAR*  name,
   int namelen
);
```

Первый параметр – это сокет, и он уже у нас есть (переменная `sock1`). Второй параметр – наша структура `sockaddr`, которую нам еще предстоит заполнить, третий – длина структуры `sockaddr`.

Структуру адреса мы заполняем так:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = SERVER_PORT;
```

Итак, мы указываем семейство протоколов – IPv4, говорим, что будем работать с любым адресом (INADDR_ANY) и указываем номер порта (содержится в константе SERVER_PORT).

Когда у нас есть сокет и структура адреса, мы можем связать открытый ранее сокет с определенным портом/адресом:

```
bind(sock1, (struct sockaddr *)&sin, sizeof(sin));
```

Если все нормально, то функция вернет 0 или SOCKET_ERROR в случае ошибки. Правильнее вызывать ее так:

```
if (bind(sock1, (struct sockaddr *)&sin, sizeof(sin)) ==
SOCKET_ERROR)
{
  printf("Error bind socket");
  exit(1);
}
else printf("Bind OK\n");
```

Далее в цикле `while()` мы ждем подключения клиента. Сокет клиента будет называться `sock2`.

Как только подключиться клиент, мы отправим ему сообщение "Version", в ответ на которое клиент передаст свое имя – "Client v.0.1". Для отправки сообщения мы будем использовать обычную функцию write:

```
write (sock2, MSG_TO_SEND, sizeof(MSG_TO_SEND));
```

Обратите внимание на ее параметры. Мы указываем, куда мы хотим записать данные (sock2), сообщение для отправки (MSG_TO_SEND) и размер этого сообщения.

Сервер прочитает переданную от клиентов информацию и выведет ее на консоль. Чтение информации мы осуществляем путем обычной функции `read()`:

```
ans_len = read (sock2, buffer, BUF_SIZE);
```

Мы читаем данные из сокета `sock2`, помещаем их в буфер `buffer`, размер буфера задается последним параметром.

Клиент, в свою очередь, выведет на консоль запрос сервера.

После обработки ответа клиента мы закрываем сокет функцией `shutdown()` и ждем нового клиента.

С целью упрощения исходного кода как сервера, так и клиента, обработку ошибок производить не будем, поэтому будьте готовы к тому, что ваш клиент выдаст сообщение Segmentation fault и завершит работу, если вы, например, укажите неправильное имя сервера или порт.

Теперь рассмотрим полный исходный код нашей программы.

Листинг 12.1. Программа-сервер (server.c)

```
#include <sys/types.h>
#include <netdb.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

#define SERVER_PORT 1234
#define BUF_SIZE 64
#define MSG_TO_SEND "Version\n"

int main () {

int sock1, sock2;

int ans_len, total=0;

char buffer[BUF_SIZE];
struct sockaddr_in sin, client;

sock1 = socket (AF_INET, SOCK_STREAM, 0);
memset ((char *)&sin, '\0', sizeof(sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = SERVER_PORT;

bind(sock1, (struct sockaddr *)&sin, sizeof(sin));

printf("Server running...\n");
```

```
listen (sock1, 3);

while (1) {
ans_len = sizeof(client);
sock2 = accept (sock1, &client, &ans_len);
write (sock2, MSG_TO_SEND, sizeof(MSG_TO_SEND));
total+=1;
ans_len = read (sock2, buffer, BUF_SIZE);
write (1, buffer, ans_len);
printf("Client no %d\n",total);
shutdown (sock2, 0);
close (sock2);
   };
return 0;
}
```

Теперь разберемся, что есть что. Сначала мы определяем некоторые макросы:

```
#define SERVER_PORT 1234
#define BUF_SIZE 64
#define MSG_TO_SEND "Version\n"
```

Первый – это номер порта сервера, именно этот порт будет прослушивать наша программа. **Второй** макрос – это размер буфера передаваемых данных. **Третий** – это наш запрос клиенту.

Нам понадобятся сразу два сокета:

```
int sock1, sock2;
```

Первый сокет – это сокет сервера, а через второй сокет мы будем производить обмен данными с клиентом.

Следующие две переменные

```
int ans_len, total=0;
```

Переменная `ans_len` используется для хранения размера передаваемой клиентом информации – фактически размер структуры struct sockaddr_in. Вторая переменная (total) – это общий счетчик числа клиентов. Данная переменная используется для вывода порядкового номера клиента.

Переменная buffer размера BUF_SIZE – это наш буфер для обмена информацией. Нам нужны две структуры типа sockaddr_in – одна для сервера (sin) и одна для клиента (client).

В строке

```
sock1 = socket (AF_INET, SOCK_STREAM, 0);
```

мы создаем наш, "серверный", сокет. Набор протоколов – TCP/IP, режим с установлением соединения.

Затем мы инициализируем структуру sin:

```
memset ((char *)&sin, '\0', sizeof(sin));

sin.sin_family = AF_INET;          // TCP/IP
sin.sin_addr.s_addr = INADDR_ANY; // можем работать на любом
                                   //адресе
sin.sin_port = SERVER_PORT;        // указываем порт (1234)
```

После создания сокета и инициализации структуры sin, нужно связать наш сокет с адресом и портом сервера:

```
bind (sock1, (struct sockaddr *)&sin, sizeof(sin));
```

Оператор

```
listen (sock1, 3);
```

означает, что мы будем прослушивать сокет sock1 (порт 1234) и максимальное число клиентов не должно превышать 3.

Как и любой нормальный сервер, мы должны работать в бесконечном цикле, постоянно обрабатывая запросы клиентов:

```
while (1) {
ans_len = sizeof(client);
sock2 = accept (sock1, &client, &ans_len);
write (sock2, MSG_TO_SEND, sizeof(MSG_TO_SEND));
total+=1;
ans_len = read (sock2, buffer, BUF_SIZE);
write (1, buffer, ans_len);
printf("Client no %d\n",total);
```

```
shutdown (sock2, 0);
close (sock2);
 };
```

Конечно, любой нормальный сервер при поступлении определенных сигналов, например, SIG_HUP должен корректно перезапуститься или вообще завершить работу. Наш сервер этого не делает – обработку сигналов, я надеюсь, вы можете добавить сами. Установить обработчик сигнала можно так:

```
#include "sock.h"
#include <signal.h>

/* обработчик сигнала SIGPIPE        */
sigpipe_handler()
{
  err_quit("Получен SIGPIPE \n");
}

main()
{

/* установка обработчика сигнала SIGPIPE      */
  signal(SIGPIPE, sigpipe_handler);
```

В бесконечном цикле мы:

- получаем размер структуры `client`

```
ans_len = sizeof(client);
```

- создаем сокет `sock2`, через который будем обмениваться данными с клиентом. Если в очереди `listen` нет клиентов, мы переходим в состояние ожидания

```
sock2 = accept (sock1, &client, &ans_len);
```

- как только подключится клиент, мы отправим ему сообщение MSG_TO_SEND

```
write (sock2, MSG_TO_SEND, sizeof(MSG_TO_SEND));
```

- увеличиваем счетчик клиентов

```
total+=1;
```

- получаем размер прочитанных данных, сами данные записываются в буфер `buffer`

```
ans_len = read (sock2, buffer, BUF_SIZE);
```

- выводим прочитанные данные на стандартный вывод

```
write (1, buffer, ans_len);
```

- завершаем сеанс связи

```
shutdown (sock2, 0);
```

- закрываем сокет

```
close (sock2);
```

Теперь мы можем откомпилировать нашу программу:

```
gcc -o server server.c
```

Запускаем:

```
./server
```

Программа перешла в состояние ожидания новых клиентов. Далее будет приведена иллюстрация, демонстрирующая в работе обе программы.

12.2. Программа-клиент

Программа-клиент несколько проще, чем сервер.

Листинг 12.2. Программа-клиент (client.c)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <memory.h>
#include <stdio.h>
```

```
#define SERVER_HOST "localhost"
#define SERVER_PORT 1234
#define CLIENT_PORT 1235
#define MSG "Client v.0.1\n"

main () {
int sock;
int ans_len;int BUF_SIZE = 64;

char buffer[BUF_SIZE];
struct hostent *h;
struct sockaddr_in client, server;

sock = socket (AF_INET, SOCK_STREAM, 0);
memset ((char *)&client, '\0', sizeof(client));

client.sin_family = AF_INET;
client.sin_addr.s_addr = INADDR_ANY;
client.sin_port = CLIENT_PORT;

bind (sock, (struct sockaddr *)&client, sizeof(client));
memset ((char *)&client, '\0', sizeof(server));

h = gethostbyname (SERVER_HOST);

server.sin_family = AF_INET;

memcpy ((char *)&server.sin_addr,h->h_addr,h->h_length);

server.sin_port = SERVER_PORT;

connect (sock, &server, sizeof(server));

ans_len = recv (sock, buffer, BUF_SIZE, 0);
write (1, buffer, ans_len);
send (sock, MSG, sizeof(MSG), 0);

close (sock);
exit (0);
}
```

Для подключения к серверу нам понадобиться следующая информация: имя узла сервера, номер порта сервера, номер порта клиента:

```
#define SERVER_HOST "localhost"
#define SERVER_PORT 1234
#define CLIENT_PORT 1235
#define MSG "Client v.0.1\n"
```

Константа MSG – это сообщение, которое будет передано серверу. Как и в случае с сервером, нам понадобятся две структуры типа `sockaddr_in`:

```
struct hostent *h;
struct sockaddr_in client, server;
```

Структура типа `hostent` нам нужна для получения адреса сервера.

Создаем сокет, заполняем информацию о клиенте и связываем сокет:

```
sock = socket (AF_INET, SOCK_STREAM, 0);
memset ((char *)&client, '\0', sizeof(client));

client.sin_family = AF_INET;
client.sin_addr.s_addr = INADDR_ANY;
client.sin_port = CLIENT_PORT;

bind (sock, (struct sockaddr *)&client, sizeof(client));
```

Перед подключением к серверу нужно определить его IP-адрес:

```
h = gethostbyname (SERVER_HOST);
```

Подключаемся к серверу:

```
// набор протоколов
server.sin_family = AF_INET;
// задаем адрес сервера
memcpy ((char *)&server.sin_addr,h->h_addr,h->h_length);
// указываем порт сервера
server.sin_port = SERVER_PORT;

connect (sock, &server, sizeof(server));
```

После подключения к серверу принимаем его запрос, выводим его на стандартный вывод, отправляем ему свое сообщение и закрываем сокет:

```
ans_len = recv (sock, buffer, BUF_SIZE, 0);
write (1, buffer, ans_len);
send (sock, MSG, sizeof(MSG), 0);
close (sock);
```

Вот в принципе, и все. Результат работы сервера и клиента показан на рис. 12.1.

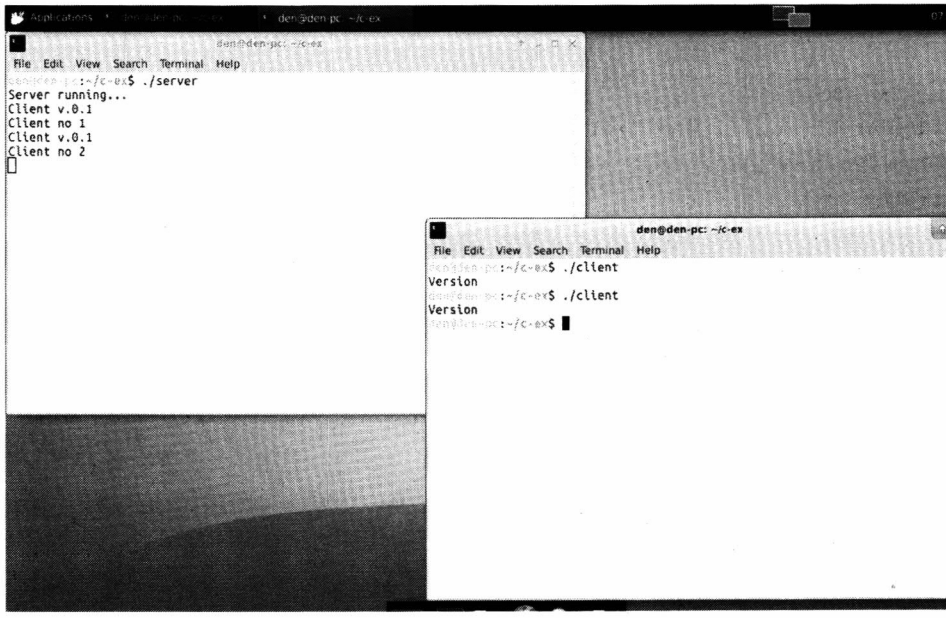


Рис. 12.1. Клиент и сервер

12.3. Многопоточный сервер

Приведенные ранее примеры сервера и клиента хороши, но на практике редко, когда можно встретить однопоточный сервер. Как правило, серверы запускают несколько потоков, чтобы обслужить сразу нескольких клиентов. Попробуем написать многопоточный сервер.

Как обычно, сначала объявим необходимые заголовочные файлы

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<arpa/inet.h>          // inet_addr
#include<unistd.h>                  // write
#include<pthread.h>                 // для потоков, сборка с
lpthread
```

При компиляции нашей программы не забудьте указать параметр `-lpthread`:

```
gcc mserver.c -o mserver -lpthread
```

В функции `main()` мы создаем сокет. В процессе работы сервера будем выводить отладочные сообщения, чтобы понимать, на какой стадии находится программа:

```
    int socket_desc , client_sock , c;
    struct sockaddr_in server , client;

    // Создаем сокет
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1)
    {
        printf("Ошибка при создании сокета");
    }
    puts("Сокет создан");
```

Подготавливаем адресную структуру и выполняем вызов `bind()`. На этот раз мы производим обработку ошибок `bind()`:

```
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 1234 );

    //Bind
    if( bind(socket_desc,(struct sockaddr *)&server ,
sizeof(server)) < 0)
    {
        //print the error message
```

```
        perror("bind failed. Error");
        return 1;
    }
    puts("bind выполнен");

    // Слушаем
    listen(socket_desc , 3);
```

Теперь начинается самое интересное. Мы случаем соединения и для обработки каждого нового соединения мы создаем новый поток. В качестве обработчика, то есть функции потока, мы используем функцию `connection_handler()`. Если поток создать не удалось, мы завершаем работу программы.

```
    puts("Ждем новые соединения...");
    c = sizeof(struct sockaddr_in);
    pthread_t thread_id;

    while( (client_sock = accept(socket_desc, (struct sockaddr
*)&client, (socklen_t*)&c)) )
    {
        puts("Соединение создано");

        if( pthread_create( &thread_id , NULL ,
           connection_handler , (void*) &client_sock) < 0)
        {
            perror("Ошибка при создании потока\n");
            exit(1);
        }

        puts("Обработчик назначен");
    }

   if (client_sock < 0)
    {
        perror("accept failed");
        return 1;
    }
```

Основная магия происходит в функции `connection_handler()`:

```
void *connection_handler(void *socket_desc)
{
    // Получаем дескриптор сокета
    int sock = *(int*)socket_desc;
    int read_size;
    char *message , client_message[2000];

    // Приветствует клиента
    message = "Hi! I am your connection handler\n";
    write(sock , message , strlen(message));

    message = "Please send me a text: \n";
    write(sock , message , strlen(message));

    // Читаем сообщение от клиента
    while( (read_size = recv(sock , client_message , 2000 , 0)) >
0 )
    {
                        // Добавляем символ конца строки
     client_message[read_size] = '\0';

     // Отправляем клиенту его же сообщение
         write(sock , client_message , strlen(client_message));

     // Очищаем буфер
     memset(client_message, 0, 2000);
    }

    if(read_size == 0)
    {
        puts("Клиент отключен");
        fflush(stdout);
    }
    else if(read_size == -1)
    {
        perror("Сбой при recv ");
    }

    return 0;
}
```

Немного придется изменить код самого клиента. Ничего связанного с потоками, просто мы немного изменили "протокол" общения клиента и сервера и теперь нужно переписать приложение-клиент:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int main()
{
int socket_desc, val;
struct sockaddr_in client_addr;
char buffer[256];
socket_desc = socket(AF_INET, SOCK_STREAM, 0);

printf("Введите номер порта\n");
int port;
scanf("%d", &port);
client_addr.sin_family = AF_INET;
client_addr.sin_addr.s_addr =  INADDR_ANY;
client_addr.sin_port = htons(port);

if(connect(socket_desc, (struct sockaddr*)&client_addr,
sizeof(client_addr)) == 0)
   printf("Подключены к серверу, порт %d\n", port); else {
   printf("Сбой!\n");
       exit(1);
}

while(1)
{

   printf("Сообщение для передачи сервера: ");
   bzero(buffer,256);
   scanf("%s", buffer);
```

```
    write(socket_desc,buffer,strlen(buffer));
    bzero(buffer,256);

    read(socket_desc,buffer,255);
    printf("Сообщение от сервера: %s\n",buffer);

}
    close(socket_desc);
    return 0;
}
```

Чтобы тестировать было проще, в клиенте мы предусмотрели ввод порта сервера, к которому будет подключаться клиент – так будет проще программисту, ведь при смене порта придется перекомпилировать только сервер.

Глава 13.
Практический пример компьютерная игра

Разработка игры на Си – процесс довольно кропотливый и заслуживает написания отдельной книги. Впрочем, если поискать, такие книги есть. Сложность разработки заключается в том, что по стандарту Си не содержит никаких функций, даже самых примитивных, для работы с графикой. В том же Pascal есть функции для построения графических примитивов и можно запрограммировать пусть и примитивную, но все же графику. Следовательно, программисту нужно выбрать и изучить сторонние библиотеки, которые он будет использовать для работы с графикой.

С другой стороны, такой подход на практике себя оправдывает. Зачем изучать примитивные графические функции, если все равно они не будут использоваться в реальных приложениях? На практике вы будете использовать ту же OpenGL для работы с графикой, а там совсем другие функции. Поэтому если уже и тратить время на изучения функций для работы с графикой, то лучше изучать то, с чем будете работать непосредственно.

Что же касается выбора языка программирования, то современные игры чаще написаны на C++/C#, нежели на Си. Причина тому – отсутствие поддержки принципов объектно-ориентированного программирования. Но тем не менее, на Си можно написать игру. На нем были написаны следующие игры:

- Quake 1/2/3/TA
- Doom 1/2
- OpenArena
- Tremulous
- Alien Arena
- Medal of Honor
- Wolfenstein ET
- Jedi Academy
- Jedi Outcast
- Call of Duty 2
- Soldier of Fortune

Игры известные, но как вы видите, все старые. А все потому что современные игровые движки и библиотеки для работы с графикой поддерживают C++/C#. В этой главе мы конечно напишем простенький шутер, но если хотите заниматься разработкой игр серьезно, смотрите в сторону C++.

13.1. Идея игры

Создадим простенький космический шутер. Главным персонажем игры будет пришелец, сидящий в космическом корабле. Его задача – остаться в живых, а для этого ему нужно истребить все камни, летящие в его сторону.

Пусть идея не отличается особой оригинальностью, но все же это лучше, чем тетрис или змейка.

13.2. Выбор библиотеки

Как мы знаем, для работы с графикой (в том числе для разработки игры) используют чаще всего или DirectX или OpenGL. Мы остановимся на послед-

ней, поскольку OpenGL работает, как в Windows, так и в Linux, а вот при использовании DirectX мы привяжемся только к Windows. Впрочем, не все разработчики игр заботятся о кроссплатформенности – наоборот, выбирают то, что им лучше/проще. Нам же нужно охватить большую аудиторию, мы не знаем, какая у читателя операционная система, какой компилятор, поэтому мы будем использовать OpenGL.

Чтобы начать программировать с использованием этой библиотеки, нужно установить пакеты freeglut3 и freeglut3-dev. Первый – это библиотека freeglut. Второй – это include-файлы, необходимые для разработки приложения. freeglut – открытая альтернатива OpenGL Utility Toolkit. GLUT позволяет пользователю создавать окна, предоставляющие контекст OpenGL на широком спектре платформ, и управлять ими, а также взаимодействовать с мышью, клавиатурой и джойстиком.

Конечно, freeglut – это не DirectX, но для нашей примитивной игрушки, вполне достаточно.

Для компиляции нашей программы нужно ввести команду:

```
c99 -I/usr/include/GL gl.c -lglut -lGL -lCLU -lX11 -lm
```

Разберемся, что есть что. Команда c99 – это компилятор, поддерживающий стандарт C99. По сути, это тот же gcc, но уже установлен стандарт языка. Нам эта команда "экономит" количество символов в самой команде компиляции, поскольку нам не нужно указывать параметр, сообщающий компилятору стандарт языка Си.

Параметр –I указывает, где "лежат" необходимые нам файлы библиотеки freeglut. Далее следует имя нашей программы (gl.c), а после – список подключаемых библиотек. Мы подключаем glut, GL, GLU, X11 и библиотеку math для математических вычислений.

Мы не указали имя выходного файла, поэтому после компиляции можно запустить программу так:

```
./a.out
```

Конечно, можно и указать опцию –o при желании:

```
c99 -I/usr/include/GL gl.c -lglut -lGL -lCLU -lX11 -lm -o game
```

Тогда запустить игру можно будет так:

```
./game
```

13.3. Основы glut

Данный практический пример будет для вас и домашним заданием. Вам предстоит разобраться самостоятельно с библиотекой `glut`. Конечно, мы рассмотрим основные функции, а дальше – сами. Как уже отмечалось, на тему разработки игр можно написать отдельную книгу и даже не одну. Рассказать о разработке игры и рассмотреть все функции библиотеки – физически невозможно.

Функция `main()` нашей игры будет выглядеть так:

```
#include <glut.h> //подключаем заголовочный файл glut.h
…
void main(int argc, char** argv) {

   FILE *fp = fopen("HighScoreFile.txt" ,"r") ;

   glutInit(&argc, argv);
   glutInitWindowPosition(90 ,0);
   glutInitWindowSize(1200,700);
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
   glutTimerFunc(50,UpdateColorIndexForSpaceshipLights,0);
   glutCreateWindow("SPACESHOOTER");
   glutDisplayFunc(display);
   glutKeyboardFunc(keys);
   glutPassiveMotionFunc(passiveMotionFunc);
   glBlendFunc(GL_SRC_ALPHA ,GL_ONE_MINUS_SRC_ALPHA);
   glutIdleFunc(idleCallBack);
   glutMouseFunc(mouseClick);
   glGetIntegerv(GL_VIEWPORT ,m_viewport);
   myinit();
   SetDisplayMode(GAME_SCREEN);
   initializeStoneArray();
   glutMainLoop();
}
```

Первая строчка открывает файл, который нам понадобится для записи счета игры, а дальше следуют функции инициализации игры. Разберемся, для чего они нужны.

Все функции инициализации в glut начинаются с glutInit. Первое, что вы должны сделать, это вызвать функцию glutInit.

```
void glutInit(int *argc, char **argv);
```

параметры:

- int argc - количество аргументов
- char** argv - их описание в виде указателя на строку

После инициализации glut нам нужно инициализировать окно приложения. Сначала установим положение окна, вернее его верхний левый угол. Для этого мы используем функцию glutInitWindowPosition.

```
void glutInitWindowPosition(int x, int y);
```

Параметры:

- x - число пикселей от левой части экрана.
- y - количество пикселей от верхней части экрана.

Далее мы выбираем размер окна. Для этого мы используем функцию glutInitWindowSize.

```
void glutInitWindowSize(int width, int height);
```

Параметры:

- width - ширина окна;
- height - высота окна;

Затем вы должны определить режим отображения с помощью функции glutInitDisplayMode.

```
void glutInitDisplayMode(unsigned int mode);
```

Параметры:

- mode - определяет режим отображения.

Вы можете использовать режим, чтобы определить цвет, а также количество и тип буферов.

Константами для определения цвета модели являются:

- GLUT_RGBA или GLUT_RGB - выбирает окно RGBA.
- GLUT_INDEX - выбирает режим индексированного цвета.

Режим отображения также позволяет выбрать одно-или двухместных буфера окна. Для этого используются константы:

- GLUT_SINGLE - режим одинарной буферизации.
- GLUT_DOUBLE - режим двойной буферизации, подходит для анимации.

Также существуют специализированные режимы буфера:

- GLUT_ACCUM - буфер накопления.
- GLUT_STENCIL - Буфер трафарета.
- GLUT_DEPTH -буфер глубины.

Итак, предположим, вы хотите создать окно в цветовом пространстве RGB, с двойной буферизацией. Все, что вам нужно сделать, это прописать соответствующие константы для того, чтобы создать необходимый режим.

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
```

После этих шагов, окно может быть создано с помощью glutCreateWindow.

```
int glutCreateWindow(char *title);
```

Параметры:

- `title` - имя создаваемого окна.

Мы перед созданием окна создаем таймер, который каждые 50 мс будет запускать функцию обновления цветовых индексов корабля (`UpdateColorIndexForSpaceshipLights`):

```
glutTimerFunc(50,UpdateColorIndexForSpaceshipLights,0);
```

Далее мы указываем, какие функции будут отвечать за рендеринг экрана и обработку нажатий клавиш:

```
glutDisplayFunc(display);
glutKeyboardFunc(keys);
```

Позже мы их рассмотрим. Далее у нас функция `glutPassiveMotionFunc()`. Она относится к обработчикам событий. Она задает обработчик, который активизируется, когда изменено положение курсора, но клавиша мыши не нажата. А если нужно установить обработчик при изменении положения курсора или нажатии клавиши мышки, то используются следующие функции:

- `glutMouseFunc()` – обработчик нажатий клавиши мыши.
- `glutMotionFunc()` – обработчик мыши при изменении положения курсора.

Функция `glutIdleFunc()` определяет обработчик простоя, то есть функцию, которая будет запущена при простое. При простое у нас будет вызываться функция `idleCallBack()`, которая будет вызывать, в свою очередь, функцию `display()` – основную нашу функцию рендеринга. Можно было бы вызвать напрямую `display()`, но дополнительный обработчик понадобится, если мы захотим выполнить еще какие-либо действия.

Функция `glutMouseFunc()` устанавливает обработчик нажатия кнопки мыши. О ней мы уже говорили.

Функция `glGetIntegerv` возвращает запрошенное значение. В данном случае мы запрашиваем значение GL_VIEWPORT, которое записывается в переменную `m_viewport`. Значение GL_VIEWPORT состоит из четырех значений – координаты x, y окна, а также ширина и высота окна – именно в такой последовательности. Именно поэтому наша переменная должна быть объявлена как массив из четырех элементов:

```
GLint m_viewport[4];
```

Далее вызывается функция `myinit()`. Она не является библиотечной, как вы уже догадались. Она выполняет некоторые инициализационные действия. Основное, что она делает – это выполняет визуализацию сцены. Если вы хотите увеличить или уменьшить игровое пространство, вам нужно изменить вызов функции `gluOrtho2D()`, которую как раз и вызывает функция `myinit()`:

```
void myinit()
{
  glClearColor(0.5,0.5,0.5,0);
```

```
    glColor3f(1.0,0.0,0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

        gluOrtho2D(-1200,1200,-700,700);
    glMatrixMode(GL_MODELVIEW);
}
```

Для создания сцены необходимо задать область вывода объектов и способ проецирования.

Если область вывода не задана программистом, то в OpenGL используется установленная по умолчанию зона в виде куба видимости 2x2x2 с началом

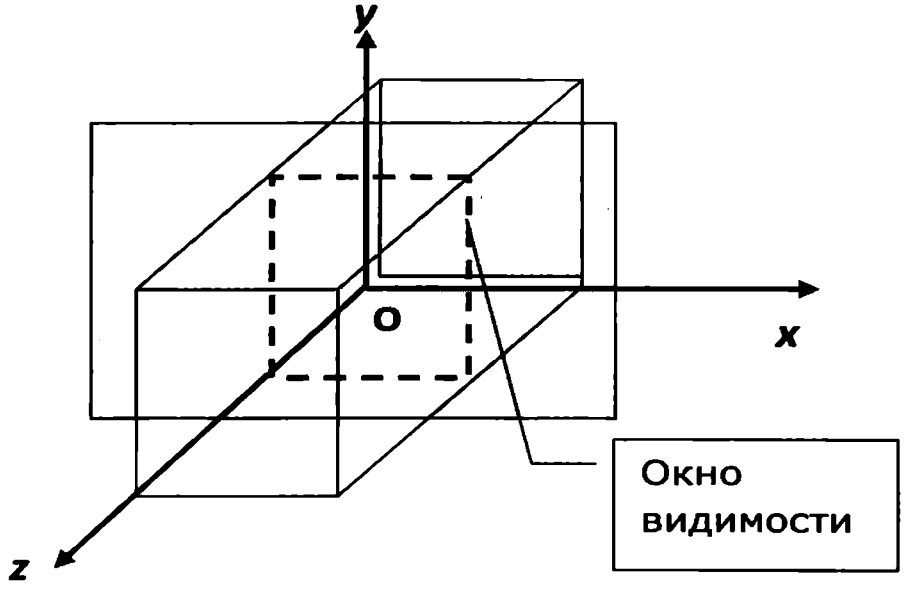


Рис. 13.1. Область вывода и система координат OpenGL Oxyz

Система координат в OpenGL Oxyz расположена таким образом, что ось Oz направлена в сторону противоположную направления зрения.

Окно видимости (Windows) масштабируется в пределах [-1;1] по осям Ox, Oy. Изображение по умолчанию воспроизводится на плоскости z=0.

Существует два типа проецирования: параллельная и перспективная. *Ортогональная проекция* – это частный случай параллельной проекции, при которой проецирующие лучи ортогональны картинной плоскости.

При ортогональном проецировании точка (x,y,z) на объекте проецируется в точку (x,y,0) на плоскости проекции. В OpenGL ортогональная проекция, характеризуемая параллелепипедом видимости, задаётся функцией `glOrtho()`, объявленной следующим образом:

```
glOrtho(left,right,bottom,top,near,far)
```

Таким образом, видны все объекты, которые попали внутрь параллелепипеда видимости. Если необходимо работать только с плоскостью Oxy, можно воспользоваться специальной функцией библиотеки GLU. `gluOrtho2D()`. Определение функции имеет вид

```
gluOrtho2D(left,right,bottom,top);
```

Результат выполнения этой функции будет таким же, как и после вызова функции `glOrtho()`, где для аргументов near и far установлены значения -1.0 и 1.0 соответственно.

Функция `SetDisplayMode()` задает режим отображения экрана. У нас будет два режима – режим игры и режим меню:

```
void SetDisplayMode(int modeToDisplay) {
     switch(modeToDisplay){
     case GAME_SCREEN: glClearColor(0, 0, 0, 1);break;
     case MENU_SCREEN : glClearColor(1, 0 , 0, 1);break;
   }
}
```

В зависимости от выбранного режима функция вызывает функцию `glClearColor()`, которая задает значения очистки цветом буфера цвета. При вызове франции используется 4 параметра цвета:

```
glClearColor(GLfloat red, GLfloat green, GLfloat blue, GLfloat
alpha);
```

Функция `initializeStoneArray()` случайным образом заполняет массив камней, летящих в сторону корабля. Вы можете изменить ее по своему усмотрению, код ее приводить не станем, поскольку он очень прост и не имеет к графике, по сути, никакого отношения.

Наконец, мы вызываем функцию `glutMainLoop()`. В каждом приложении OpenGL функция `main()` должна заканчиваться вызовом `glutMainLoop()`.

13.4. Основная функция display()

Функция `display()` управляет отображением экранов игры. Экранов у нас будет четыре – игровой экран, он выводится функцией `GameScreenDisplay()`, экран меню (`startScreen`), экран инструкций (вызывается функцией `InstructionsScreenDisplay()`) и экран Game over.

```
void display() {

   glClear(GL_COLOR_BUFFER_BIT);
   glViewport(0,0,1200,700);
      // если игра началась и нет gameOver, отображаем экран игры
   if(startGame && !gameOver)
      GameScreenDisplay();
       // если пользователь запросил инструкции,
   else if(instructionsGame)
      InstructionsScreenDisplay();
      // если пользователь проиграл
   else if(gameOver)
      GameOverScreen();
   // Экран меню
   else if(startScreen){
         startScreenDisplay();
         if(gameQuit || startGame || optionsGame ||
instructionsGame){
            //startScreen = false;

            if(startGame){
               SetDisplayMode(GAME_SCREEN);
               startScreen = false;

            } else if(gameQuit)
               exit(0);

            } else if(instructionsGame) {
               SetDisplayMode(GAME_SCREEN);
               InstructionsScreenDisplay();
```

```
        }
    }

  // сбрасываем значения масштаба
  glScalef(1/2 ,1/2 ,0);
  glFlush();  // освобождаем буферы GL
  glLoadIdentity(); // сбрасываем матрицу в исходное
состояние
  glutSwapBuffers(); // обмен буферов
}
```

13.5. Обработка нажатий клавиш клавиатуры и мыши

Ранее в качестве обработчика нажатий клавиатуры мы установили функцию `keys()`.Функция, используемая в качестве аргумента `glutKeyboardFunc` должна иметь три аргумента. В первом содержится ASCII код нажатой клавиши, оставшиеся два аргумента обеспечивают положение курсора мыши при нажатии клавиши, относительно верхнего левого угла клиентской области окна.

```
void keys(unsigned char key, int x, int y)
{
      if(key == 'd') xOne+=SPACESHIP_SPEED;
      if(key == 'a') xOne-=SPACESHIP_SPEED;
      if(key == 'w') {yOne+=SPACESHIP_SPEED;}
      if(key == 's') {yOne-=SPACESHIP_SPEED;}
      if(key == 'd' || key == 'a' || key == 'w' || key == 's')
            somethingMovedRecalculateLaserAngle();

      display();

}
```

При нажатий клавиш A и D мы изменяем координаты корабля по горизонтали, а при нажатии W и S – по вертикали. Значение координаты изменяется на скорость корабля – значение SPACESHIP_SPEED. Если были

нажаты эти клавиши, то нам нужно пересчитать угол лазера, поэтому мы вызываем функцию somethingMovedRecalculateLaserAngle().

При нажатии кнопки мыши вызывается следующий обработчик:

```
void mouseClick(int buttonPressed ,int state ,int x, int y) {
if(buttonPressed == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
            mButtonPressed = true;
      else
            mButtonPressed = false;
      display();
}
```

Он говорит программе, что нажата левая клавиша мыши – устанавливает в true переменную mButtonPressed, далее передает управление функции display(). В игровом экране нажатие клавиши мыши означает стрельбу из лазерной пушки, поэтому программа вычислит угол лазерного луча (функция somethingMovedRecalculateLaserAngle()) и произведет выстрел. Если при этом луч попадет в камень, тогда будет увеличен счет игрока.

13.6. Отображение меню

Задачи функции отображения меню, следующие:

1. Вывести само меню. Надписи выводим на английском, чтобы не заморачиваться с кодировками и шрифтами в glut.
2. Определить, какой пункт выбрал пользователь
3. Выполнить выбранное пользователем действие – запуск игры, отображение инструкций или выход.

```
void startScreenDisplay()
{
   glLineWidth(50);
   SetDisplayMode(MENU_SCREEN);

            // Рисуем рамку меню
   glColor3f(0,0,0);
   glBegin(GL_LINE_LOOP);                  // Рамка
      glVertex3f(-750 ,-500 ,0.5);
```

```
    glVertex3f(-750 ,550 ,0.5);
    glVertex3f(750 ,550 ,0.5);
    glVertex3f(750 ,-500, 0.5);
  glEnd();

  glLineWidth(1);

          // Рисуем прямоугольник для Start Game
  glColor3f(1, 1, 0);
  glBegin(GL_POLYGON);            // прямоугольник Start Game
    glVertex3f(-200 ,300 ,0.5);
    glVertex3f(-200 ,400 ,0.5);
    glVertex3f(200 ,400 ,0.5);
    glVertex3f(200 ,300, 0.5);
  glEnd();

  glBegin(GL_POLYGON);            // прямоугольник Instructions
    glVertex3f(-200, 50 ,0.5);
    glVertex3f(-200 ,150 ,0.5);
    glVertex3f(200 ,150 ,0.5);
    glVertex3f(200 ,50, 0.5);
  glEnd();

  glBegin(GL_POLYGON);            // Прямоугольник Quit
    glVertex3f(-200 ,-200 ,0.5);
    glVertex3f(-200 ,-100 ,0.5);
    glVertex3f(200, -100 ,0.5);
    glVertex3f(200, -200 ,0.5);
  glEnd();

           // Проверяем координаты мыши
  if(mouseX>=-100 && mouseX<=100 && mouseY>=150 && mouseY<=200){
    glColor3f(0 ,0 ,1) ;
    if(mButtonPressed){
       // Если нажата левая кнопка мыши при этих координатах, то
      // мы запускаем игру - переменная startGame = true
       startGame = true ;
       gameOver = false;
       mButtonPressed = false;
    }
  } else
```

```
    glColor3f(0 , 0, 0);

        // Выводим надписи элементов меню
  displayRasterText(-100 ,340 ,0.4 ,"Start Game");

  if(mouseX>=-100 && mouseX<=100 && mouseY>=30 && mouseY<=80) {
    glColor3f(0 ,0 ,1);
    if(mButtonPressed){
      instructionsGame = true ;
      mButtonPressed = false;
    }
  } else
    glColor3f(0 , 0, 0);
  displayRasterText(-120 ,80 ,0.4 ,"Instructions");

  if(mouseX>=-100 && mouseX<=100 && mouseY>=-90 && mouseY<=-40){
    glColor3f(0 ,0 ,1);
    if(mButtonPressed){
      gameQuit = true ;
      mButtonPressed = false;
    }
  }
  else
    glColor3f(0 , 0, 0);
  displayRasterText(-100 ,-170 ,0.4 ,"    Quit");

}
```

Код этой функции очень прост и понятен. Если вам не понятно назначение (или параметры) тех или иных функций glut, найти описание функций можно по адресу https://www.opengl.org/resources/libraries/glut/.

13.7. Игровой экран

Функция `GameScreenDisplay()` отвечает за отображение основного игрового экрана:

```
void GameScreenDisplay()
{
  SetDisplayMode(GAME_SCREEN);
  DisplayHealthBar();
```

```
glScalef(2, 2 ,0);
if(alienLife){
   SpaceshipCreate();
}
else {
   gameOver=true;
   instructionsGame = false;
   startScreen = false;
}

StoneGenerate();
```

Рис. 13.2. Экран меню

Она также довольно проста и понятна. Сначала она регистрирует игровой экран, далее – отображает полоску с индикатором состояния игры, затем проверяет, жив ли наш пришелец. Если да, то прорисовывает космический корабль с помощью функции `SpaceshipCreate`, а если нет, то выводит экран Game over, путем установки переменной `gameOver` в true. Наконец, функция запускает генератор камней для прорисовки камней, летящих на космический корабль.

Рис. 13.3. Игровой процесс

Оставшуюся часть программы мы предлагаем вам исследовать самостоятельно. Игра состоит всего лишь из чуть более 800 строк. Информацию о любой glut-функции вы можете получить по адресу https://www.opengl.org/resources/libraries/glut/. Исходный код игры будет доступен на сайте издательства.

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

12+

ООО «Наука и Техника»

Лицензия №000350 от 23 декабря 1999 года.

192029, г. Санкт-Петербург, пр.Обуховской обороны, д. 107.

Подписано в печать 14.01.2019. Формат 70x100 1/16.

Бумага газетная. Печать офсетная. Объем 18 п. л.

Тираж 1200. Заказ №680.

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, г. Ульяновск, ул. Гончарова, 14.

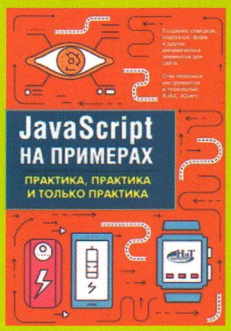
Кольцов Д.М.

Си НА ПРИМЕРАХ

ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА

Эта книга является превосходным учебным пособием для изучения языка программирования Си на примерах.
В книге рассмотрена базовая теоретическая часть языка Си, позволяющая ориентироваться в языке и создавать свои программы: операторы, логические конструкции, массивы, связанные списки и деревья, очереди и стеки, работа с файлами. Отдельное внимание уделено программированию различных алгоритмов, а также рассмотрению нововведений языка Си на момент 2019 года (стандарты C99, C11, современные практики использования, многопоточность).
В книге используется большое количество примеров с подробным анализом кода.
Будет полезна как начинающим программистам, студентам, так и всем, кто хочет быстро начать программировать на Си.

Издательство "Наука и Техника" рекомендует:



ISBN 978-5-94387-776-6

Издательство "Наука и Техника"
г. Санкт-Петербург

Для заказа книг:
(812) 412-70-26
e-mail: nitmail@nit.com.ru
www.nit.com.ru