МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТРАНСПОРТА»

Кафедра «Информационные технологии»

Д. В. БАЛАЩЕНКО, Д. В. ЗАХАРОВ

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С/С++ (ЧАСТЬ 1)

Пособие по дисциплине
«Информатика и информационные технологии»
для студентов электротехнических специальностей

Одобрено методической комиссией факультета

Гомель 2006

УДК 004.438 (075.8)
ББК 32.973-018.1
Б 20

Р е ц е н з е н т – доцент кафедры “Вычислительные машины и программирование” ГГУ им. Ф. Скорины к.ф-м.н. *Жадан М.И.*

Балащенко Д. В. Захаров Д. В.

Программирование на языке C/C++ (Часть 1). – Гомель: УО “БелГУТ”, 2006. – с.

ISBN.

В пособии рассмотрены основы программирования на языке C/C++. Предназначено для студентов электротехнических специальностей.

УДК 004.438 (075.8)

ББК 32.973-018.1
Б 20

ISBN

Содержание

Введение

Данное пособие предназначено для оказания помощи в изучении языков программирования С и С++ – мощных и профессиональных языков программирования, предназначенных как для системного программирования, так и для разработки всех других видов программного обеспечения.

Первая версия языка С была разработана в 1972 г. сотрудником фирмы Bell Laboratories Деннисом Ритчи (Dennis Ritchie), когда он и Кен Томпсон (Ken Thompson) занимались созданием операционной системы UNIX. Язык С произошел от разработанного Томпсоном языка В. Важным моментом является то, что язык С был создан в качестве инструмента для программистов-практиков, поэтому главная цель разработки этого языка программирования, в отличие от разработки языка Pascal, предназначенного для начального обучения программированию, заключалась в том, чтобы сделать его полезным при создании различных прикладных программ. Расцвет языка С пришел на 80-е годы XX века, но он остается одним из ведущих языков программирования, являясь бесспорным лидером в области системного программирования. Язык С является языком разработки операционных систем семейства UNIX(LINUX), которые являются наиболее технологически развитыми среди всех операционных систем и востребованными при использовании в областях, предъявляющих повышенные требования к безопасности и надежности.

Для языка С характерна достаточно высокая переносимость. Компиляторы с данного языка написаны для всех платформ, используемых в настоящее время. Программы на С получаются быстрыми и компактными, теряя всего лишь порядка 20% производительности по сравнению с программами, написанными на языке Ассемблера, выигрывая на порядок в простоте и скорости написания программ. Программы на языке С могут обладать скоростью выполнения в несколько раз большей (до сотен) по сравнению с другими языками программирования. Их можно настраивать либо на максимальное быстродействие, либо на экономное использование доступной памяти. И, наконец, язык программирования С является важным языком программирования как сам по себе, так и как промежуточная ступень при переходе на С++. Язык С++ – это язык С, к которому добавили средства объектно-ориентированного программирования. Язык С++ является, практически, расширением языка С. При изучении языка С происходит изучение языка С++. Язык С также имеет и другие достоинства и, конечно же, недостатки (первых намного больше, чем вторых). Компактность языка С в сочетании с большим количеством операторов дает возможность создавать программный код, понимание которого чрезвычайно затруднительно. В конце концов, какой язык программирования может похвастать ежегодным конкурсом на самую запутанную программу?

Отличительной чертой данного пособия является то, что оно предназначено для оказания помощи в программирования на языке С/С++ независимо

от аппаратной платформы и операционной системы, в которой планируется написание программы. Пособие ориентировано на создание программ на языке С и компиляцию их С-компилятором. Однако эти же программы можно без изменений компилировать С++ - компилятором.

Все примеры программ, представленных в пособии, являются полностью переносимыми, и откомпилированы с помощью компиляторов: gcc3.2 (Suse Linux 9.2, Slackware Linux 10.2, FreeBSD 5.4, Solaris 10) и Microsoft Visual C++ 7.0 (Windows XP SP2).

1 ЭТАПЫ СОЗДАНИЯ ПРОГРАММЫ НА ЯЗЫКЕ С

Язык программирования С является компилируемым языком. С целью представления того, как создается программа на языке С разобьем процесс написания программы на семь этапов.

***Этап 1*:** Определение целей создания программы

Естественно, что начинать программирование необходимо с ясного понимания того, что требуется от программы, т.е. определить, что программа должна делать в каждой конкретной ситуации. Необходимо обдумать какого рода информация требуется программе, какие вычисления и другие действия будет выполнять программа, а также какую информацию она должна отображать.

***Этап 2*:** Проектирование программы

После того как в общих чертах будет сформулировано, какие действия должны выполняться программой, следует решить, каким образом программа будет выполнять эти действия. Также необходимо принять решение относительно того, каким образом будут представлены данные в программе и, возможно, во вспомогательных файлах. Все это реализуется на уровне общих понятий, а не в виде исходного кода программы. На этом этапе разрабатывается блок-схема алгоритма программы.

***Этап 3*:** Создание программного кода

После создания блок-схемы программы можно приступать к ее реализации на практике, т. е. написанию программного кода на языке С. Можно набросать на бумаге свои идеи, но в конце концов, придется ввести код программы в компьютер. Особенности этого процесса зависят от среды программирования. В общем случае для создания того, что называется файлом исходного кода, применяется текстовый редактор. Файл исходного кода содержит код программы на языке С. На этом шаге следует также задокументировать результаты работы. Самый простой и эффективный способ – использование комментариев, позволяющий включить объяснения в исходный код программы.

***Этап 4*:** Компиляция

Следующий очень ответственный этап – это компиляция исходного кода. Детали этого процесса зависят от среды программирования. Мы же рассмотрим этот процесс в общих чертах.

Компилятор – это программа, в задачи которой входит преобразование исходного кода в исполняемый код программы. Исполняемый код – это код программы на машинном языке, т.е. родном языке компьютера. Компьютеры различных типов имеют разные машинные языки, и компилятор языка С преобразует код программы на языке С в код программы на конкретном машинном языке. Компилятор также проверяет отсутствие ошибок в программном коде на языке С. Если компилятор обнаруживает ошибки, он выдает соответствующее сообщение и не создает исполняемый файл.

Основная стратегия, используемая при создании программ на языке С, заключается в использовании программ, преобразующих файл исходного кода в исполняемый файл, который содержит готовый к выполнению код на машинном языке. Это преобразование осуществляется в два этапа: компиляция и компоновка. Компилятор преобразует исходный код в промежуточный, а компоновщик объединяет его с другим кодом и создает исполняемый файл. Этот двухступенчатый подход применяется с целью упрощения создания модульных программ (т.е. разбиения программ на модули). Модули можно компилировать по отдельности, а позднее с помощью компоновщика объединять компилированные модули. Таким образом, если потребуется изменить один модуль, не придется повторно компилировать остальные. Также компоновщик комбинирует программу с предварительно скомпилированным кодом библиотек.

В С системе перед началом фазы компиляции автоматически выполняется программа препроцессора. Препроцессор С подчиняется специальным командам – директивам препроцессора, которые определяют, какие действия должны быть выполнены перед компиляцией программы. Обычно это подключение заголовочных файлов и замена специальных символов в тексте программы. Препроцессор автоматически запускается компилятором перед тем, как начнется преобразование программы в машинный язык.

В процессе преобразования исходного кода в код на машинном языке создается файл объектного кода (объектный файл). Хотя объектный файл содержит код на машинном языке, его нельзя выполнить. Объекный файл содержит результат трансляции исходного кода, но он еще не является завершенной программой.

Первый элемент, который не включается в файл объектного кода, именуется кодом запуска. Этот код выполняет функции интерфейса между программой и операционной системой. Второй элемент, не включаемый в файл объектного кода, – это код стандартных библиотечных подпрограмм. Практически во всех программах на языке С используются стандартные подпрограммы (функции), которые входят в стандартную библиотеку языка С.

Компоновщик (linker) собирает все эти три элемента – объектный код программы, стандартный код запуска для конкретной операционной систе-

мы, библиотечный код и помещает их всех в один файл, который называется исполняемым файлом. Что касается библиотечного кода, то компоновщик извлекает из библиотеки только коды необходимых функций.

В итоге и объектный файл, и исполняемый файл состоят из инструкций машинного языка. Однако объекный файл содержит результат перевода на машинный язык только кода программы, созданной программистом, а исполняемый файл – также и машинный код для используемых стандартных библиотечных подпрограмм и для кода запуска. В одних системах компилирующие и компонующие программы необходимо выполнять по отдельности. В других системах компилятор автоматически запускает программу компоновщика, так что программист должен ввести только команду компиляции.

***Этап 5*:** Выполнение программы

Традиционно, исполняемый файл – это программа, которая выполняется компьютером. Во многих средах для выполнения программы достаточно ввести имя исполняемого файла. При этом загрузчик помещает в память исполняемый код программы с диска. И, наконец, компьютер, под контролем ЦПУ, выполняет программу, последовательно прочитывая ее инструкцию одну за другой.

***Этап 6*:** Тестирование и отладка программы

То, что программа работает – уже является хорошим признаком, но не исключено, что она работает неправильно. Поэтому необходимо проверить, делает ли она именно то, что от нее требуется. Вы можете обнаружить, что в некоторых ваших программах имеются ошибки. Отладка – это процесс обнаружения и устранения программных ошибок.

Существует множество возможностей для совершения ошибок. Серьезную ошибку легко допустить на этапе проектирования. Возможна некорректная реализация вполне здравых идей. Вы можете пропустить ввод непредвиденных данных, и ход программы может быть нарушен. Вероятно некорректное использование возможностей языка С.

К счастью, ситуация не так безнадежна. Многие виды ошибок перехватываются компилятором. Существуют также различные способы, позволяющие обнаруживать ошибки, которые не перехватываются компилятором.

***Этап 7*:** Сопровождение и модернизация программы

Когда создается программа для себя или для кого-то другого, возможно, что эта программа будет широко применяться в дальнейшем. Возможно, будет решено реализовать в программе какую-то новую умную идею или адаптировать программу для работы в другой компьютерной системе. Все эти задачи значительно упрощаются, если четко задокументировать программу и следовать корректным методам проектирования программ.

2 ЛИНЕЙНЫЕ ПРОГРАММЫ

2.1 Общие сведения

Простейшей алгоритмической структурой является линейная последовательность операций, которые выполняются по очереди и именно в том порядке, в котором записаны. Программную реализацию такой алгоритмической структуры называют линейной программой. Линейные алгоритмы и линейные программы обычно предназначены для решения относительно простых задач, в которых не предусмотрен выбор из нескольких возможных альтернатив или циклическое повторение каких-либо операций.

Исторически сложилось так, что первая программа, предлагаемая при изучении языков программирования, выводит фразу: "HELLO, WORLD!" (Приветствую тебя, МИР!). Мы не намерены отступать от этой доброй традиции.

Приведем листинг этой простой программы.

Листинг 2.1

```
#include <stdio.h> //Это директива препроцессора
int main(void){
  printf("HELLO");
  printf(", WORLD!\n");
  /* предыдущая строка выводит фразу WORLD! на экран */
  return 0;
}
```

Сначала наберите исходный текст этой программы в каком-либо редакторе, установленном в системе, присвоив файлу имя с расширением .c, что должно удовлетворить требованиям вашей системы, предъявляемым к именам файлов. Можно, например, использовать имя first.c. Теперь скомпилируйте и выполните программу. Если все прошло гладко, то на экране должно появиться следующее: HELLO, WORLD! Эти слова мы можем увидеть в третьей и четвертой строках программы, но что же случилось с остальными символами этих строк и другими строками?

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

Этот оператор является директивой препроцессора языка С. Компиляторы языка С выполняют предварительную обработку исходного кода программы перед ее компиляцией. Эта обработка называется препроцессорной, т.к. она выполняется препроцессором. Оператор `#include <stdio.h>` дает компилятору команду включить в программу информацию из файла `stdio.h`, который существует во всех программных пакетах компиляторов С. Файл `stdio.h` содержит информацию о функциях ввода-вывода, в частности, о функции `printf()`, которая и обеспечивает вывод на экран фразы "HELLO, WORLD!".

За символами // в первой строке находится комментарии - замечания, помогающие лучше понять программу. Они предназначены только для программиста и игнорируются компилятором. Данные комментарии действуют только до конца текущей строки. Между символами /* и */ также находятся комментарии. Все, что находится между ними - комментарий и его можно распространять на несколько строк.

```
#int main(void){
```

Все программы на языке С состоят из функций. Наша первая программа состоит из одной функции с именем main. Эта функция входит в состав всех программ, написанных на языке С. Круглые скобки являются признаком того, что она является функцией. Слово int указывает на то, что функция main возвращает целое число, а слово void указывает на то, что она не имеет аргументов. Открывающая фигурная скобка { в первой строке отмечает начало операторов, образующих функцию. Определение функции заканчивается закрывающей фигурной скобкой }. Эта пара скобок и часть программы между ними называется блоком. Блок – важная программная единица в языке С.

```
printf("HELLO");
```

При выполнении первого оператора printf на экране отображается фраза HELLO, причем курсор остается на той же строке.

```
printf(", WORLD!\n");
```

Следующий оператор добавляет запятую и слово WORLD с восклицательным знаком в конец предыдущей фразы, выведенной ранее на экран. Последовательность символов \n дает команду компьютеру начать новую строку, т.е. передвинуть курсор на начало следующей строки и является примером управляющей последовательности.

```
return 0;
```

Функция на языке С может передавать (или возвращать) в программу, в которой она вызывается, некоторое число. Пока считайте, что эта строка необходима в связи с требованием стандарта ISO/ANSI C, предъявляемыми к правильной записи функции main().

Особое внимание обращается на то, что при программировании на С необходимо обращать внимание на соблюдение регистра букв при вводе ключевых слов и имен переменных.

Давайте посмотрим на вторую программу на С, исходный код которой представлен в листинге 2.2.

Листинг 2.2

```
#include <stdio.h>
int main(void){
  int i;
  i=14;
```

```
    printf("Я учусь на 1-ом курсе БелГУТа\n");
    printf("Моим любимым числом является %d\n", i);
    return 0;
}
```

Данная программа является более объемной и сложной.

```
int i;
```

Этот оператор объявляет, что в программе используется переменная с именем `i` и что эта переменная имеет тип `int` (целочисленная переменная). При именовании переменных следует использовать осмысленные имена (например, `sheep_count` лучше имени `x3`), хотя, если программа маленькая, проще применять имена переменных типа `a`, `b` ,`c` , `a1` и т.д. В вашем распоряжении имеются буквы нижнего и верхнего регистра, цифры и знак подчеркивания, _. Первым символом должна быть буква или символ подчеркивания. В операционных системах и библиотеках С часто используются имена идентификаторов с одним или двумя первыми символами подчеркивания, поэтому желательно избегать использования таких имен. В именах переменных языка С учитывается регистр символов, поэтому имена `count`, `Count`, `couNT`, в отличие от некоторых других языков, обозначают разные переменные. Кроме того, имена, состоящие из одних прописных букв, имеют особый смысл, который будет пояснен позже, поэтому пока их лучше не использовать.

Оператор объявления является очень важным в языке С. В рассматриваемом примере объявляются два объекта. Во-первых, объявляется, что где-то в этой функции имеется переменная с именем `i`. Во-вторых, `int` означает, что переменная `i` объявляется как целое число, т.е. число без десятичной точки или дробной части (ключевое слово `int` представляет пример типа данных). Компилятор применяет эту информацию для выделения подходящего объема хранилища в памяти, предназначенного для использования переменной `i`. В языке С все переменные должны быть объявлены до их использования!!! Здесь имеется в виду, что должен быть представлен список всех переменных, которые используются в программе, причем требуется указать тип каждой переменной. Объявления переменных должны располагаться между левой фигурной скобки функции и перед первым исполняемым оператором.

```
i=1;
```

При выполнении оператора `i=1;` значение 1 присваивается переменной `i`. Рассмотренная ранее строка `int i;` применялась для выделения области памяти для переменной `i`, а строка `i=1;` применяется для присваивания переменной `i` значения 1. В операторе присваивания значение, стоящее справа от знака равенства, присваивается переменной, стоящей слева.

```
printf("Я учусь на 1-ом курсе БелГУТа\n");
```

Смотри пример выше.

```
printf("Моим любимым числом является %d\n", i);
```

Последний оператор printf выводит на экран фразу в кавычках со встроенным в нее значением переменной i (которое равно 1). Последовательность символов %d определяет, где и в какой форме будет выводится значение переменной i.

Если вы выполните данную программу, то увидите, что на экране будет напечатано следующее: Моим любимым числом является 14.

При печати строки символы %d заменены цифрой 14 и значение переменной также равно 14. Символ % предупреждает программу о том, что значение переменной должно быть выведено именно в этом месте, а символ d означает, что содержимое переменной должно быть напечатано как десятичное (основание 10) целое число.

Рассмотрим следующую программу:

Листинг 2.3

```
/*программа, считывающая и выводящая номер текущего
  месяца и года*/
#include <stdio.h>
int main(){
  int y, m;//здесь объявляются переменные
  printf("Введите номер месяца: ");
  scanf("%d", &m);
  printf("Введите номер года: ");
  scanf("%d", &y);
  printf("Сейчас идет %d месяц %d года\n ", m, y);
  return 0;
}
```

Программа начинается с комментария, который поясняет назначение программы. Такой вид документирования очень полезен, особенно по истечении нескольких месяцев после создания программы. Комментарий также встречается в строке, где объявляются переменные.

Для того, чтобы пользователь программы знал, в какой момент требуется ввести с клавиатуры данные, применяется так называемое приглашение к вводу: printf("Введите номер месяца: ");.

Рассмотрим следующую строку программы:

```
scanf("%d", &m);
```

Функция scanf() осуществляет считывание с клавиатуры. В данном случае считывается одно целое число и записывается в переменную m. Спецификатор %d в функции scanf() означает, что с клавиатуры считывается целое число, а &m – что вводимое число записывается в переменную m. Обратите внимание на наличие символа &. Амперсанд, когда он используется совместно с именем переменной, сообщает функции scanf() ячейку памя-

ти, в которой хранится переменная m. В последующем компьютер будет хранить величину для m в этой ячейке. Использование амперсанда очень часто смущает начинающих программистов, или людей программировавших до этого на других языках. На данный момент просто запомните, что в операторе scanf() необходимо имя переменной предварять знаком амперсанда. При считывании чисел с клавиатуры наличие данного символа является обязательным. В случае пропуска & программа может компилироваться нормально, но при выполнении будет возникать ошибка. На многих системах эта ошибка в процессе исполнения программы называется "ошибкой сегментации" или "нарушением доступа". Подобная ошибка происходит, когда программа пытается получить доступ к той части памяти компьютера, к которой пользовательские программы доступа не имеют.

```
printf("Сейчас идет %d месяц %d года\n", m, y);
```

На этот раз при использовании функции printf() были выполнены 2 подстановки.

Первая пара символов %d в кавычках была замещена значением первой переменной (m) в списке, следующим за текстом в кавычках; а вторая пара символов была замещена значением второй переменной (y). В списке переменных, которые выводятся на экран, их имена разделяются запятыми, а сам список следует за текстом, заключенном в кавычки.

Программа, представленная в листинге 2.4 является еще более сложной.

Листинг 2.4

```
/*программа для перевода суммы в долларах в рублевый
   эквивалент*/
#include <stdio.h>
int main(void){
  float dol;
  float rub;
  float kurs;
  printf("Введите курс доллара к рублю:");
  scanf("%f", &kurs);//считывается курс доллара
  printf("Введите количество долларов:");
  scanf("%f", &dol);
  rub=dol*kurs;//вычисляется количество рублей
  printf("$ %f - это %.2f рублей\n", dol, rub);
  return 0;
}
```

Здесь мы ввели 3 вещественных переменных типа float. Вещественные числа – это числа, у которых имеется дробная часть (5.6, -45.7382 и т.д.). Эти переменные можно было бы ввести в одной строке следующим образом:

```
float dol, rub, kurs;
```

В программе производится операция умножения. Как и в множестве других языков программирования звездочка * является знаком умножения. Таким образом, оператор

```
rub=dol*kurs;
```

означает следующее: «взять значение переменной dol, умножить его на значение переменной kurs и результат вычисления присвоить переменной rub».

Чтобы вывести на печать (на экран) значение переменной вещественного типа (т. е. число с плавающей точкой), в функции printf() применяется спецификатор %f. Кроме этого, при выводе количества рублей используется модификатор .2, который определяет, что данные, выводимые на экран, будут содержать два знака после десятичной точки.

Для ввода данных в программу с клавиатуры используется функция scanf(). В ней можно также увидеть спецификатор %f, который означает, что с клавиатуры будет считываться число с плавающей точкой, а &kurs и &dol – что вводимые числа присваиваются переменным kurs и dol соответственно. Вычисления могут выполняться и непосредственно внутри оператора printf(). Мы могли бы скомбинировать вычисление количества рублей (rub=dol*kurs) и печать результата в один оператор:

```
printf("$ %f – это %.2f рублей\n", dol, dol*kurs);
```

2.2 Арифметика в языке С

Большинство написанных на языке С программ выполняют какие-либо вычисления.

Арифметические операции представлены в таблице

Таблица 1.1 – **Арифметические операции**

Действие в С	Арифметическая операция	Алгебраическое выражение	Выражение на С
Сложение	+	x+2	x+2
Вычитание	-	p-t	p-t
Умножение	*	xy	x*y
Деление	/	x/y, x:y	x/y
Взятие по модулю	%	x mod y	x%y

Отметим использование специальных символов, не используемых в алгебре. Так звездочка (*) означает умножение, а знак процента (%) означает операцию взятия по модулю, которая будет объяснена ниже. В алгебре если мы хотим умножить x на y, мы можем просто написать переменные, обозначаемые одной буквой, рядом, т.е. xy. В языке С так делать нельзя! Необходимо явно обозначать умножение с использованием знака звездочки.

Результатом деления двух целых чисел также будет целое число. Например, значение выражения 17/4 будет 4, а 7/3 равно 2. В языке С существует операция взятия по модулю, %, которая дает значение остатка после деления нацело. Операция взятия по модулю может применяться только к целым операндам. Выражение `x%y` означает остаток от деления `x` на `y`. Таким образом, результатом `17%4` будет 1, а `8%3`, соответственно, 2.

Арифметические операции должны записываться в строчку, т.к. в таком виде легче ввести программу в компьютер. Круглые скобки используются в выражениях на языке С точно так же, как и в алгебраических выражениях. Например, для того, чтобы умножить x на y+z необходимо писать: `x*(y+z)`.

Язык С оценивает арифметические выражения (т.е. вычисляет их численное значение) в последовательности, определяемой правилами старшинства операций, которые в общем такие же, как в алгебре:

1 Выражения или части выражений, находящиеся внутри скобок, оцениваются в первую очередь. Таким образом, скобки обладают наивысшим приоритетом. В случае вложенных скобок выражение, находящееся во внутренней паре скобок оценивается первым.

2 Следующими выполняются операции умножения, деления и взятия по модулю. Если в выражении имеются несколько операций умножения, деления или взятия по модулю, вычисление производится слева направо. Операции умножения, деления и взятия по модулю являются операциями равного приоритета.

3 Последними выполняются операции сложения и вычитания. Если в выражении имеется несколько операций сложения или вычитания, вычисление производится слева направо. Операции сложения и вычитания также являются операциями равного приоритета.

При проведении математических вычислений может потребоваться вычисление значений математических функций, таких как косинус, натуральный логарифм и т.д. Для этого необходимо подключить заголовочный файл `math.h`, в котором дано описание этих функций, а также приведены значения некоторых констант таких как число "Пи". При компиляции программы в среде Unix с помощью компилятора gcc необходимо использовать опцию компилятора –lm, которая подключает математическую библиотеку:

```
$ gcc  -o  vis  vis.c  -lm
```

Описание математических функций языка С дано в приложении Б.

3 ТИПЫ ДАННЫХ

3.1 Общие сведения

3.1.1 Переменные и константы

Компьютеры могут складывать числа, сортировать информацию, рисовать картинки, обрабатывать мультимедийную информацию и др. Для решения всех этих и других задач программы должны работать с данными, которые представляются в виде чисел. Значения некоторых данных устанавливаются до начала выполнения программы и сохраняются неизменными в ходе выполнения программы. Такие данные называются константами. Более строго: константа – это ячейка или набор ячеек оперативной памяти, содержимое которой (которых) не может меняться в ходе выполнения программы. Значения других данных могут изменяться в ходе программы, например, путем присваивания новых значений. Эти данные называются переменными. Переменная – это ячейка или набор ячеек оперативной памяти, содержимое которой (которых) может изменяется в ходе выполнения программы.

3.1.2 Ключевые слова, обозначающие типы данных

Кроме различий между переменными и константами существуют также различия между данными разных типов. Одни данные являются числами, другие – буквами или другими символами. В языке С существует несколько основных типов данных, которые можно разделить на два семейства, в зависимости от того, в каком формате данные хранятся в памяти компьютера: *целочисленные типы данных* и *типы данных с плавающей точкой*.

3.2 Целочисленные типы данных

Целое число – это число без дробной части. Пример: 25, -84, 1. Такие числа, как 2.432, 25.67 и 5.00, не являются целыми.

3.2.1 Тип данных int

В языке С существует несколько разновидностей целочисленных данных. Основным типом целочисленных данных является `int`. К данным типа `int` относятся целые числа со знаком. Диапазон возможных значений зависит от компьютерной системы и операционной системы, установленной на компьютере. Так, например, в ОС MS DOS для хранения данных типа `int` отводится 16 битов, поэтому диапазон возможных значений простирается от –32768 до +32767. На современных 32 разряд-

ных компьютерах, работающих под управлением ОС Windows, Linux, Unix, Mac OS для хранения данных этого типа отводится 32 бита. Диапазон возможных значений от –2147483648 до +2147483647. Для того чтобы не запоминать эти многозначные числа, достаточно знать, что вместо них можно использовать символические константы, соответственно, INT_MIN и INT_MAX, определенные в заголовочном файле `limits.h`. Обычно для хранения данных типа `int` отводится одно машинное слово, чем обеспечивается наибольшая скорость работы программы, в которой используется данный тип данных.

При объявлении переменных типа `int` сначала идет ключевое слово `int`, затем имя переменной и в заключение – точка с запятой. Несколько переменных можно объявлять как по отдельности, так и перечислив имена переменных через запятую после ключевого слова `int`. Кстати, такой подход применим и к другим типам данных. При объявлении переменных создаются переменные, но им не присваиваются значения. Инициализировать переменную в языке С можно прямо в операторе объявления:

```
int i=32;
int count=3, pass=-34;
```

Для вывода на печать данных типа `int` можно использовать функцию `printf()`. При этом используются символы `%d` для того, чтобы указать место, где должно печататься целое число. Необходимо тщательно следить за тем, равняется ли количество спецификаторов формата в функции `printf()` количеству отображаемых на экране значений.

Обычно в языке С предполагается, что целочисленная константа является десятичным числом. Но иногда удобнее использовать восьмеричную и шестнадцатеричную системы счисления. Чтобы указать компьютеру, что число представлено в шестнадцатеричной системе счисления, необходимо добавить префикс `0x` или `0X` (нуль-икс) перед числом, записанным в шестнадцатеричной системе. `0x10` - это 16 в десятичной системе. Подобно этому префикс `0` (нуль) означает, что число является восьмеричным. Восьмеричные и шестнадцатеричные числа трактуются как числа без знака.

При печати на экране числа в восьмеричном виде вместо спецификатора формата `%d` необходимо использовать `%o`. Для того чтобы отобразить целое число в шестнадцатеричном виде, используется спецификатор `%x`.

3.2.2 Другие типы целочисленных данных

При изучении языка С большинство потребностей, связанных с целыми числами, удовлетворят данные типа `int`. Однако по ряду причин вам могут потребоваться и другие типы целочисленных данных.

В языке С используются три ключевых слова, обозначающие модификации основного типа целочисленных данных: `short`, `long` и `unsigned`.

1 Данные типа `short int` (коротко `short`) могут занимать меньший объем памяти, чем данные типа `int` и поэтому экономить память, если используются небольшие по величине числа. Так же, как и данные типа `int`, являются данными со знаком.
2 Данные типа `long int` (коротко `long`) могут занимать больший объем памяти, чем данные типа `int` и поэтому используются для представления больших чисел. Так же, как и данные типа `int`, являются данными со знаком.
3 Данные типа `long long int` (коротко `long long`) могут занимать еще больший объем памяти, чем данные типа `long int` и поэтому используются для представления очень больших чисел. Так же как и данные типа `int` являются данными со знаком.
4 Тип данных `unsigned int` (`unsigned`) используется для представления только положительных чисел. Диапазон возможных значений данных этого типа смещен по отношению к диапазону данных типа `int`.
5 В стандарте С90 распознаются как допустимые следующие типы данных: `unsigned long int` (`unsigned int`) и `unsigned short int` (`unsigned short`). Стандарт С99 добавляет типы данных `unsigned long long int` (`unsigned long long`).

Для вывода данных типа `unsigned int` используется спецификатор `%u`. Чтобы вывести данные типа `long`, используется спецификатор формата `%ld`. Префикс `l` можно также использовать вместе с префиксами `x` и `o` (для вывода на экран чисел в шестнадцатеричном и восьмеричном виде соответсвенно). Для обозначения типа данных `short` можно использовать префикс `h`.

3.2.3 Применение символов: тип данных char

Данные типа `char` представляют собой символы, такие как буквы и знаки препинания, но на самом деле они относятся к целочисленному типу данных. Это происходит потому, что данные типа `char` представляют собой фактически целые числа, а не символы. Данные типа char определяются как 8-битовые элементы памяти. Объявляются данные типа `char` также, как и переменные других типов:

```
char bukva;
char simvol1, simvol2;
```

Для инициализации переменной bukva применяется следующий код:

```
bukva='H';
```

Некоторые символы кода ASCII являются непечатаемыми. Например, они могут представлять такие действия, как переход на следующую строку или выдачу звукового сигнала. Наилучшим способом представления таких

символов в С - это использование специальных управляющих последовательностей.

Таблица 3.1 – **Управляющие последовательности**

Последовательность	Значение
\a	Звуковой сигнал
\n	Новая строка
\\	Обратная наклонная черта (\)
\'	Одиночная кавычка
\"	Двойная кавычка
\0oo	Восьмеричное значение
\xhh	Шестнадцатеричное значение

Когда символьным переменным присваиваются управляющие последовательности, они должны быть заключены в одиночные кавычки.

Можно создать оператор

```
alpi='\n';
```

и затем вывести на экран значение переменной alpi, чтобы перейти на следующую строку.

3.2.4 Печать символов

Для вывода на печать символов в функции printf() используется спецификатор %c. Так как символьная переменная хранится в памяти как 1-байтовое целочисленное значение, то при выводе переменной типа char с помощью обычного спецификатора %d вы получите целое число. Спецификатор %c означает, что функция printf() должна преобразовать целое число в соответствующий символ.

В следующем листинге приведена программа, выводящая переменную типа char обоими способами.

Листинг 2.1

```
#include <stdio.h>
int main(void){
  char ch;
  printf("Введите символ: ");
  scanf("%c", &ch);
  printf("Код для %c будет %d.\n", ch, ch);
  return 0;
}
```

В результате выполнения программы получилось следующее:

```
Введите символ: B
Код для B будет 66.
```

Обратите внимание на то, что спецификаторы в функции `printf()` определяют, каким образом данные отображаются на экране, а не то, как они хранятся в памяти. В одних реализациях языка С данные типа `char` могут принимать значения в диапазоне от -128 до +127. В других реализациях данные типа `char` являются данными без знака, и принимают значения в диапазоне от 0 до 255. В описании компилятора должно быть указано, что представляют собой данные типа `char`; это же можно узнать, просмотрев заголовочный файл `limits.h`.

3.3 Данные типа float, double и long double

В программах с математическими вычислениями часто используются *числа с плавающей точкой*. В языке С такие числа называются данными типа `float`, `double` и `long double`. Применение чисел с плавающей точкой дает возможность представлять гораздо больший диапазон чисел, включая десятичные дроби.

В стандарте С установлено, что данные типа `float` должны иметь, как минимум, шесть значащих цифр, а диапазон их возможных значений должен лежать в пределах от 10^{-37} до 10^{+37}.

Для представления чисел с плавающей точкой в языке С имеется также тип данных `double` (для чисел двойной точности). Минимальный диапазон возможных значений для данных типа `double` установлен такой же, как и для данных типа `float`, а минимальное число значащих цифр увеличено до 10. Для представления данных типа `double` обычно используется не 32, а 64 бита.

Стандарт С допускает еще один тип данных с плавающей точкой: `long double`. Этот тип данных обеспечивает еще большую точность.

Переменные с плавающей точкой объявляются и инициализируются, как и целочисленные переменные:

```
float nout;
double d1, year;
long double sw=12.38483;
```

Для вывода на печать чисел типа `float` и `double` в десятичной форме в функции `printf()` применяется спецификатор формата `%f`, а в экспоненциальной форме – спецификатор `%e`.

В языке С имеется встроенный оператор `sizeof`, который определяет размеры данных различных типов (в байтах).

Для вывода данных типа `unsigned int` используется спецификатор `%u`. Чтобы вывести данные типа `long`, используется спецификатор формата `%ld`. Префикс `l` можно также использовать вместе с префиксами `x` и `o` (для вывода на экран чисел в шестнадцатеричном и восьмеричном виде соответ-

свенно). Для обозначения типа данных `short` можно использовать префикс `h`.

При работе с функцией `printf()` нужно проявлять особую осторожность, так как на самом деле здесь не приведены все случаи использования спецификаторов формата. В случае неправильной спецификации можно получить совсем не те результаты, что ожидалось.

4 ВВОД-ВЫВОД

4.1 Ввод-вывод в C. Использование функций scanf() и printf().

Важная составляющая часть решения любой задачи – представление результатов. В этой главе мы детально рассмотрим форматирующие возможности функций `scanf()` и `printf()`. Эти функции соответственно вводят данные из стандартного потока ввода и выводят данные в стандартный поток вывода. Четыре других функции, которые действуют на стандартных потоках ввода/вывода – `gets()`, `puts()`, `getchar()` и `putchar()` будут рассмотрены позже.

Весь ввод и вывод выполняется посредством потоков – последовательностей символов с построчной организацией. Каждая строка содержит нулевое или большее число символов и заканчивается символом новой строки.

При запуске программы к ней автоматически присоединяются три потока. Стандартный поток ввода `stdin` обычно присоединяется к клавиатуре, а стандартный поток вывода `stdout` – к устройству вывода информации на экран монитора. Операционные системы нередко позволяют переадресовать эти потоки на другие устройства. Третий поток – стандартный поток ошибок `stderr` – также присоединяется к экрану. В него выводятся сообщения об ошибках.

Функции `printf()` и `scanf()` позволяют пользователю общаться с программой. Они называются функциями ввода/вывода (input/output). Исторически данные функции не были частью определения языка C. Первоначально язык C оставлял реализацию процедуры ввода/вывода авторам компиляторов. В стандартах C90 и C99 описываются стандартные версии этих функций, и мы будем следовать стандарту.

Несмотря на то, что `printf()` – это функция вывода, а `scanf()` – функция ввода, в их работе много общих черт. Рассмотрим сначала работу функции `printf()`, а затем перейдем к описанию функции `scanf()`.

4.2 Функция printf()

4.2.1 Форматированный вывод с использованием функции printf()

С помощью функции `printf()` можно точнейшим образом форматировать вывод программы. Инструкции, которые мы даем функции `printf()`, зависят от типа выводимой переменной. Так, например, мы используем спецификатор `%d` при печати целого числа и спецификатор `%c` при печати символа. Эти указания называются спецификаторами преобразования, поскольку они определяют, каким образом преобразуются данные в форму, пригодную для вывода. В таблице № 4.1 приведены спецификаторы преобразования.

Ниже приводится формат, определяющий использование функции `printf()`:

```
printf(управляющая_строка, элемент1, элемент2,…);
```

Т а б л и ц а № 4.1 – **Спецификаторы преобразования**

Спецификатор преобразования	Результат
`%a`	Число с плавающей точкой, шестнадцатеричные цифры и p-запись (C99).
`%A`	Число с плавающей точкой, шестнадцатеричные цифры и p-запись (C99).
`%c`	Одиночный символ.
`%d`	Десятичное целое число со знаком.
`%e`	Число с плавающей точкой, экспоненциальное представление (e-представление).
`%E`	Число с плавающей точкой, E-представление.
`%f`	Число с плавающей точкой, десятичное представление.
`%g`	Используется %f или %e, в зависимости от значения. Стиль %e используется, если степень меньше чем -4, больше либо равна заданной точности.
`%G`	Используется %f или %E, в зависимости от значения. Стиль %e используется, если степень меньше чем -4, больше либо равна заданной точности.
`%i`	Десятичное целое число со знаком.
`%o`	Восьмеричное целое число без знака.
`%p`	Указатель.
`%s`	Строка символов.
`%u`	Десятичное целое число без знака.
`%x`	Шестнадцатеричное целое число без знака, использование шестнадцатеричных цифр 0f.
`%X`	Шестнадцатеричное целое число без знака, использование шестнадцатеричных цифр 0F.
`%%`	Печать знака процента.

Параметры элемент1, элемент2 и т.д. определяют выводимые элементы, которые нужно напечатать. В их качестве могут выступать переменные, константы и даже целые выражения, значение которых вычисляется при выводе на печать. Параметр управляющая_строка – это строка символов, описывающая, каким образом необходимо выполнить вывод на печать элементов. Управляющая строка должна содержать спецификатор преобразования для каждого выводимого элемента.

Рассмотрим небольшой пример:

```
printf("Мне %d лет и моя зарплата составляет "
    "%f рублей.\n", year, bonus);
```

Управляющая строка – это фраза в двойных кавычках. Она содержит два спецификатора преобразования соответственно для year и bonus. Особо обращаем внимание на то, что необходимо использовать один спецификатор преобразования для каждого элемента в списке, который следует за управляющей строкой.

Не делайте так:

```
printf("Мне %d лет и моя зарплата составляет "
"%f рублей.\n", year);
```

Здесь не указано значение для второго спецификатора `%f`. То, к чему приведет вас такая небрежность, зависит от типа вашей системы, но в лучшем случае вы получите какую-либо абракадабру.

Если вам необходимо вывести только фразу, то не нужны никакие спецификаторы преобразования.

```
printf("Мне 18 лет и я учусь на 1-ом курсе БелГУТа\n");
```

Поскольку функция `printf()` использует символ % для идентификации спецификации преобразования, то возникает проблема при распечатке символа %. Выходом из этой ситуации является использование двух символов %%:

```
printf("Прирост ВВП составил %d%%\n", 10);
```

4.2.2 Печать с заданием ширины поля и точности представления

Точный размер поля, в котором печатаются данные задается шириной поля. Если ширина поля больше, чем необходимо для печати данных, то данные обычно выравниваются внутри поля по его правому краю. Целое число, задающее ширину поля, вставляется в спецификацию преобразования между знаком процента (%) и спецификатором преобразования. Ширина поля может быть использована со всеми спецификаторами преобразования. Для вывода значений, превышающих текущее значение ширины поля, она автоматически увеличивается. При выводе отрицательных значений требуется учесть, что знак "минус" отрицательного значения занимает одну символьную позицию ширины поля. Одной из распространенных ошибок явля-

ется задание недостаточной ширины поля для вывода текущего значения. Это может привести к смешиванию данных и спутыванию результатов.

Функция `printf()` дает также возможность задавать точность представления, с которой будут напечатаны данные. Точность имеет различный смысл для различных типов данных. Если она используется при выводе целых чисел, то она показывает минимальное количество цифр, которые должны быть выведены на печать. Если выводимое значение содержит меньше цифр, чем задано точностью, то будут дополнительно напечатаны нули. Для целых чисел точность по умолчанию равна 1. Если точность используется со спецификаторами преобразования значений с плавающей точкой `e`, `E`, `f`, то точность – это количество цифр, которые будут напечатаны после десятичной точки. Если при печати значений с плавающей точкой задана меньшая точность, чем число десятичных разрядов дробной части, то это значение округляется. Для спецификаторов преобразования `g` и `G` точность – это максимальное количество значащих цифр, которые будут выведены на экран. Для того, чтобы использовать точность представления, необходимо поместить между знаком процента и спецификатором преобразования десятичную точку с последующим целым числом, задающим точность представления. Ширина поля и точность могут быть объединены, для чего между знаком процента и спецификатором преобразования нужно вставить значение ширины поля, десятичную точку и последующее значение точности, например:

```
printf("%7.3f\n", 12.53323);
```

Данный оператор выводит на экран число 12.533 в поле шириной 7 символов с выравниванием по правому краю.

Ширину поля и точность представления можно задать, используя целочисленные выражения в списке аргументов после строки управления форматом. Для этого вставьте * (звездочку) вместо ширины поля или точности (или вместо того и другого). Звездочки при печати будут заменены соответствующими значениями из списка аргументов. Значение аргумента для ширины поля может быть отрицательным, а для точности представления – только положительным. Отрицательное значение ширины поля приводит к выравниванию вывода по левому краю поля. Оператор

```
printf("%*.*f\n", 9, 2, 12.53323);
```

использует значение 9 для ширины поля, 2 для точности представления и выводит значение 12.53 с выравниванием по правому краю.

4.3 Форматированный ввод с использованием функции scanf()

Функция `scanf()` содержит строку управления форматом, которая описывает формат входных данных.

Функция `scanf()` имеет следующий формат:

`scanf(строка_управления_форматом, другие_аргументы);`

В строке управления форматом содержится описание входного формата, а другие аргументы – это указатели на переменные, в которых сохраняются вводимые данные.

В таблице 4.2 приведены спецификаторы преобразования, используемые при вводе различных типов данных.

Таблица 4.2 – **Спецификаторы преобразования для функции scanf(), определенные в стандарте C99**

Спецификатор преобразования	Значение
`%c`	Интерпретирует результат ввода в качестве символа.
`%d`	Интерпретирует результат ввода как десятичное целое число со знаком.
`%e, %f, %g, %a`	Интерпретирует результат ввода как число с плавающей точкой (%a определен в стандарте C99).
`%E, %F, %G, %A`	Интерпретирует результат ввода как число с плавающей точкой (%A определен в стандарте C99).
`%i`	Интерпретирует результат ввода как десятичное целое число со знаком.
`%o`	Интерпретирует результат ввода как восьмеричное целое число со знаком.
`%p`	Интерпретирует результат ввода как указатель (адрес).
`%s`	Интерпретирует результат ввода как строку; ввод начинается с первого символа, не являющегося служебным, и включает все символы до следующего служебного символа.
`%u`	Интерпретирует результат ввода как десятичное целое число без знака.
`%x, %X`	Интерпретирует результат ввода как шестнадцатеричное целое число со знаком.

Спецификатор `%i` дает возможность вводить десятичные, восьмеричные и шестнадцатеричные числа. При вводе чисел с плавающей точкой может быть использован любой из спецификаторов `e`, `E`, `g`, `G`, `f`, `F`, `a`, `A`.

Листинг 4.1 – Ввод-вывод целых чисел

```
#include <stdio.h>
int main(){
  int a1, a2, a3, a4, a5, a6;
  printf("Введите 6 целых чисел: ");
  scanf("%d%i%i%o%u%x", &a1, &a2, &a3, &a4, &a5, &a6);
  printf("Были введены числа: %d %d %d %d %d %d\n",
     a1, a2, a3, a4, a5, a6);
  return 0;
}
```

При выполнении программы получим следующее:

```
Введите 6 целых чисел: 123 0x14 54 13 756 a2
Были введены числа: 123 20 54 11 756 162
```

Символы и строки вводятся с помощью спецификаторов преобразования `c` и `s` соответственно. Для чтения определенного числа символов из входного потока со спецификацией преобразования `scanf()` может использоваться значение ширины поля.

Л и с т и н г 4.2 – Ввод данных с заданием ширины поля

```
#include <stdio.h
int main(){
  int x, y;
  printf("Введите 7 цифр целого числа: ");
  scanf("%2d%d", &x, &y);
  printf("Были введены числа %d и %d\n", x, y);
  return 0;
}
```

При выполнении программы получим следующее:

```
Введите 7 цифр целого числа: 1234567
Были введены числа 12 и 34567
```

Часто бывает необходимо пропустить некоторые символы входного потока. Например, дата может быть введена как 18-11-2005.

Каждое число даты нужно сохранить, но символы тире, которыми они разделены, могут быть отброшены. Чтобы устранить ненужные символы их необходимо включить в строку управления форматом `scanf()` (пробельные символы – пробел, новая строка и табуляция – пропускают все пробельные символы, предшествующие значимым данным). Так, чтобы пропустить знаки тире при вводе, можно использовать оператор

```
scanf("%d-%d-%d", &d, &m, &y);
```

Но дата может быть введена и в другом формате: 18/11/2005.

По этой причине функция `scanf()` имеет возможность установить символ подавления присваивания `*` (звездочка). Символ подавления присваивания дает возможность функции `scanf()` читать любой тип данных и отбрасывать их, не присваивая переменной.

5 ОПЕРАТОРЫ ВЕТВЛЕНИЯ

5.1 Общие сведения

Рассмотрим небольшую программу, представленную в листинге 5.1:

Л и с т и н г 5.1

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

```
int main(void){
  int x;
  printf("Введите число: ");
  scanf("%d", &x);
  if (x<100)
  printf("Число %d меньше, чем 100\n", x);
  return 0;
}
```

В данной программе вводится целое число с клавиатуры и, в том случае если введенное число меньше, чем 100, выводится сообщение об этом. Оператор `if` является наиболее простым из операторов ветвлений. За ключевым словом `if` следует условие ветвления, *заключенное в круглые скобки*. Если будет выполнено условие, стоящее в скобках, то будет выполнен оператор следующий за `if`.

Рассмотрим другой пример:

Листинг 5.2

```
#include <stdio.h>
#define STO 100
int main(void){
  int x;
  printf("Введите число: ");
  scanf("%d", &x);
  if (x>STO){
     printf("%d больше, чем %d\n", x, STO);
     printf("Я знал это!!!\n");
  }
  return 0;
}
```

Эта программа интересна тем, что в ней используется еще одна директива препроцессора `#define`. Препроцессор C позволяет определять константы.

```
#define STO 100
```

После компиляции программы величина 100 будет подставлена повсюду, где была использована `STO`. Подобная подстановка именуется подстановкой во время компиляции. Константы, определенные таким образом, часто называют именованными. Сначала следует служебное слово `#define`. Затем указывается символическое имя (`STO`) для константы и далее – значение 100. В общем случае все выглядит следующим образом:

```
#define NAME value
```

Вы можете заменить выбранное символическое имя для `NAME` и соответствующее значение для `value`. Точка с запятой здесь не используется!!! По правилам хорошего тона в программировании имя символической констан-

ты записывается в верхнем регистре. Тогда при разборе программы вы сразу определите, что это символическая константа, а не переменная. Имена, используемые для символических констант, должны удовлетворять тем же правилам, что и имена переменных.

Следующим, на что следует обратить свое внимание, является то, что тело оператора `if` может состоять только из одного оператора, что было продемонстрировано в предыдущей программе, так и из нескольких операторов, *заключенных в фигурные скобки*.

Оператор `if` позволяет совершать действие в том случае, если выполняется некоторое условие. Если же условие не выполняется, никакого действия не выполняется. Однако очень часто встречаются ситуации, когда нам необходимо совершить одно действие в случае выполнения условия и другое действие в случае невыполнения этого условия. Здесь нас может выручить оператор ветвления (`if...else...`). Он состоит из оператора `if`, за которым следует блок операторов, и ключевого слова `else`, за которым следует еще один блок операторов.

В листинге 5.3 представлена модифицированная программа из предыдущего примера.

Листинг 5.3

```
#include <stdio.h>
#define STO 100
int main(void){
  int x;
  printf("Введите число: ");
  scanf("%d", &x);
  if (x>STO){
     printf("%d больше чем %d\n", x, STO);
     printf("Я знал это!!!\n");
  }
  else
     printf("%d не больше чем %d\n", x, STO);
  return 0;
}
```

В зависимости от истинности или ложности условия ветвления, программа выводит на экран соответствующее сообщение.

В следующем примере рассматривается программа, в которой вычисляется сумма всех цифр 3-значного числа, причем осуществляется проверка того, является ли введенное число 3-значным.

Листинг 5.4

```
#include <stdio.h>
int main(){
  int x, sum;
  printf("Введите 3-значное число: ");
  scanf("%d", &x);
```

```
    if ((x / 1000 == 0) && (x / 100 != 0)){/*проверка
       того, является ли число 3-значным*/
       sum=x/100 + x%100/10 + x%10;//вычисление суммы
       printf("Сумма цифр числа %d = %d\n", x, sum);
    }
    else
       printf("Вы ввели не 3-значное число!!!\n");
    return 0;
}
```

Обратите внимание на строку `if`...

```
if ((x / 1000 == 0) && (x / 100 != 0)){
```

Здесь осуществляется проверка того, является ли число 3-значным.

Результат вычисления `x/1000` сравнивается с 0 (== это логическое равенство).

Если результат целочисленного деления `x` на `1000` будет равным `0`, то это означает, что `x<1000`. Не правда ли? В этой строке есть еще одна проверка: `x/100!=0`. Знак `!=` в языке С означает НЕ РАВНО. Таким образом, если при целочисленном делении числа `x` на `100` получается число отличное от `0`, то это означает, что `x>=100`. Объединив эти два условия (первое – число `x` меньше `1000`, второе – число `x` больше либо равно `100`) при помощи знака `&&` (логическое умножение), мы получаем проверку того, что введенное число является 3-значным. Логическое умножение объединяет два условия, которые должны выполняться одновременно. Также широко распространено логическое сложение (знак `||` - должно выполняться хотя бы одно условие) и логическое отрицание (знак `!` – не должно выполняться условие).

Логические операции `||` и `&&` выполняются слева направо; как только какое-то условие в `||` оказывается истинным (а в `&&` ложным) – дальнейшие условия просто не вычисляются!!!

Небезынтересной является также следующая строка:

```
sum=x/100 + x%100/10 + x%10;
```

Здесь мы вначале (`x/100`) выбираем первую цифру из трехзначного числа `x` (сотни). Далее (`x%100`) получаем остаток от деления на `100`, т.е. двузначное число (десятки и единицы), а после этого, применяя деление на `10`, получаем вторую цифру (десятки). И, наконец, (`x%10`) – сразу получаем третью цифру (единицы). Осталось их только сложить...

5.2 Оператор выбора

В том случае, если необходимо сделать выбор из нескольких (более 2-х) альтернативных путей дальнейшей работы программы целесообразно использовать оператор выбора `switch`.

Рассмотрим пример:

Листинг 5.5

```
#include <stdio.h>
int main(){
  int x;
  printf("Введите скорость вращения пластинки: ");
  scanf("%d", &x);
  switch (x){
     case 33: //если пользователь ввел число 33
        printf("Долгоиграющий формат\n");
        break;
     case 47: //если пользователь ввел число 47
        printf("Формат сингла\n");
        break;
     case 78: //если пользователь ввел число 78
        printf("Устаревший формат\n");
        break;
     default: //если введено какое-то другое число
        printf("Такого формата нет.\n");
        break;
  }
  return 0;
}
```

Эта программа печатает одно из трех сообщений в зависимости от того, какое из чисел 33, 47 или 78 – введет пользователь. За ключевым словом switch следуют круглые скобки, содержащие имя переменной, от значения которой зависит дальнейшее выполнение программы: `switch(x)`. Скобки ограничивают набор операторов `case`. Каждый раз за ключевым словом case следует константа, после значения которой стоит двоеточие:

```
case 33:
```

Тип констант, употребляемых в операторах case, должен совпадать с типом переменной, стоящей в скобках оператора `switch`, но они должны быть целочисленными.

Перед входом в тело `switch` программа должна инициализировать каким-либо значением переменную, стоящую внутри оператора `switch`, поскольку это значение будет сравниваться с константами, стоящими в теле ветвления `switch`. Если какое-либо из сравнений даст истинный результат, то операторы, стоящие за соответствующим сравнением, будут исполняться до тех пор, пока не встретится слово `break`.

В конце последовательности операторов каждой из `case`-секций помещен оператор `break`. Этот оператор завершает выполнение ветвления `switch`. Управление в этом случае передается первому оператору, следующему за конструкцией `switch`. Не забывайте ставить оператор `break`, поскольку при его отсутствии управление будет передано операторам, относящимся к другим веткам `switch`, что, скорее всего, создаст нежелательный

эффект для работы программы (хотя в отдельных случаях это может быть полезным).

Если значение переменной в операторе switch не совпадает ни с одним из значений констант, указанных внутри ветвления, то управление будет передано в конец switch без выполнения каких-либо действий.

Последняя ветвь switch программы в листинге 5.5 начинается не со слова case, а с нового ключевого слова default. Оно предназначено для того, чтобы программа могла выполнить некоторую последовательность действий в том случае, если ни одно из значений констант не совпало со значением switch-переменной. В примере мы выводим сообщение о том, что такой формат не существует и предложение попробовать еще.

Попробуем закрепить наши знания, рассмотрев следующую программу. В ней создается простейший калькулятор, позволяющий осуществлять операции умножения, деления, сложения и вычитания двух чисел.

Листинг 5.6

```
/*простейший калькулятор*/
#include <stdio.h>
int main(){
  double a, b;
  int x;
  //-------------------------------------------------
  //выбор пункта меню
  printf("Выберите операцию\n");
  printf("1. Умножение (a*b) 2. Деление (a/b)\n");
  printf("3. Сложение (a+b) 4. Вычитание (a-b)\n");
  //в следующей строке проверяется правильности выбора
  //пункта меню
  if ((scanf("%d", &x)!=1) || (x>4) || (x<1)) {
     printf("Неправильно выбран пункт!!!\n");
     return 0;
  }
  else {
     printf("Введите a и b: ");
     scanf("%lf %lf", &a, &b);
     switch (x){
        case 1: // "*"
           printf("%.3f*%.3f=%.3f\n", a, b, a*b);
           break;
        case 2: // "/"
           printf(" %.3f / %.3f = %.3f\n", a, b, a/b);
           break;
        case 3: // "+"
           printf(" %.3f+%.3f = %.3f\n", a, b, a+b);
           break;
        case 4: // "-"
```

```
            printf(" %.3f - %.3f = %.3f\n", a, b, a-b);
            break;
         default: break;
      }
   }
   return 0;
}
```

Интересным здесь является следующий фрагмент кода:

```
if ((scanf("%d", &x)!=1) || (x>4) || (x<1)) {
  printf("Неправильно выбран пункт!!!\n");
  return 0;
}
```

Здесь имеется условный оператор `if`. Аргументом является условие, состоящее из трех условий, объединенных операций логического сложения. Т.е. истинное значение (`true`) будет получено в том случае, если хотя бы один из приведенных операторов будет истинен. Так как имеется всего 4 пункта меню, то выбор пользователем пунктов с номером больше, чем 4 и меньше, чем 1 будет запрещен, и программа будет завершаться. При этом напечатается соответствующее сообщение. Завершает программу оператор `return 0;`.

Во-первых посмотрите на функцию `scanf()`:

```
((scanf("%d", &x)!=1)
```

Мы реализовали проверку на правильность ввода пунктов меню, если пользователь будет вводить числа. Однако пользователь может по ошибке ввести букву. Программа при этом будет работать некорректно. Для того, чтобы обеспечить защиту от ввода некорректных данных, необходимо предусмотреть и этот вариант. Функция `scanf()` возвращает количество успешно считанных аргументов, а так как мы пытаемся считать целое число (спецификатор `%d`), то при попытке ввести что-то другое функция `scanf()` возвратит 0, а не 1. (Здесь надо заметить, что при попытке ввода вещественного числа будет считана целая часть и сообщение об ошибке не последует). Далее мы проверяем значение, возвращенное данной функцией, и если оно отлично от 1, то завершаем работу программы, сообщив пользователю.

6 БИНАРНЫЕ ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Язык С включает поразрядные логические операции и операции сдвига.

6.1 Поразрядные логические операции языка С

Поразрядные операции используются для операций над битами целочисленных операндов (`char`, `short`, `int`, `long`, `long long`; как `signed` так

и unsigned). В поразрядных операциях обычно используют беззнаковые целые. Они называются поразрядными, потому что работают с каждым битом. Поразрядные операции не следует путать с обычными логическими операциями (&&, || и !), которые оперируют со значениями в целом.

1 Дополнение или поразрядное отрицание: ~

При выполнении унарной (действующей на один операнд) операции поразрядного дополнения все биты инвертируются, т.е. все биты, равные 1, устанавливаются равными 0, а все биты, равные 0, устанавливаются равными 1:

~ (01110101) //исходное выражение

(10001010) //результат применения операции дополнения

2 Операция поразрядное И (побитовое умножение): &

В ходе выполнения операции & производится побитовое сравнение двух операндов. Для каждой позиции бита результирующий бит будет равен 1, если оба бита в соответствующих операндах равны 1.

Следовательно:

(10110111) & (01101101) //выражение

(00100101) //результат применения операции поразрядного И.

Язык С также располагает комбинированной операцией И поразрядного присваивания: &=.

3 Операция поразрядного ИЛИ (побитовое сложение): |

В ходе выполнения операции | производится побитовое сравнение двух операндов. Для каждой позиции бита результирующий бит будет равен 1, если хотя бы один из соответствующих битовых операндов равен 1.

Следовательно:

(10100110) | (10001000) // выражение

(10101110) //результат применения операции поразрядного ИЛИ.

Язык С также располагает комбинированной операцией ИЛИ поразрядного присваивания: |=.

4 Операция поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ: ^

В ходе выполнения операции ^ производится побитовое сравнение двух операндов. Для каждой позиции бита результирующий бит будет равен 1, если один или второй (но не оба одновременно) соответствующий бит в операндах равен 1.

Следовательно:

(10101101) ^ (11001011) //выражение

(01100110) //результат применения операции исключающее ИЛИ.

Язык С также располагает комбинированной операцией ИСКЛЮЧАЮЩЕЕ ИЛИ поразрядного присваивания: ^=.

6.2 Операции смещения

Рассмотрим операции сдвига языка С. В результате применения этих операций биты смещаются влево или вправо.

1 Операция сдвига влево: <<

Операция сдвига влево, <<, смещает биты значения левого операнда влево на расстояние, значение которого определено в правом операнде. Битам, освобожденным в результате смещения, присваиваются нули, а биты, перемещаемые за крайнюю левую позицию операнда, теряются.

В примере, представленном ниже, каждый бит перемещается влево на две позиции:

(10010011) <<2 //выражение

(01001100) //результат

Эта операция приводит к появлению нового битового значения, но не изменяет операнды. Здесь можно использовать операцию присваивающего левого смещения (<<=), что позволит изменить фактическое значение переменной.

2 Операция сдвига вправо: >>

Операция сдвига вправо, >>, смещает биты значения левого операнда вправо на расстояние, значение которого определено в правом операнде. Битам, освобожденным в результате смещения, присваиваются нули, а биты, перемещаемые за крайнюю правую позицию операнда, теряются.

В примере, представленном ниже каждый бит перемещается влево на две позиции:

(10010011) >>4 //выражение

(00001001) //результат

Эта операция приводит к появлению нового битового значения, но не изменяет операнды. Здесь можно использовать операцию присваивающего правого смещения (>>=), что позволит изменить фактическое значение переменной.

Операции поразрядного сдвига могут обеспечивать быстрое и эффективное умножение и деление чисел со степенью 2.

число <<n //умножает число на 2^n.

число >>n //делит число на 2^n.

Рассматриваемые операции смещения аналогичны процедуре смещения точки в десятичной системе счисления при умножении или делении числа на 10.

6.3 Применение масок

Часто операция поразрядного И используется совместно с масками. Маска – это битовый шаблон, определенные биты которого установлены равными соответственно вкл (1) и выкл (0). Предположим, что мы опреде-

лили символическую константу MASKA равную 2, что в двоичной системе соответствует 00000010, где только один бит имеет значение 1, отличное от нуля. Применение оператора

```
niss = niss & MASKA;
```

приводит к тому, что все биты niss (за исключением бита 1) устанавливаются равными 0, поскольку любой бит, который комбинируется с 0 с помощью операции &, выдает в результате 0 (логическое умножение). Первый бит остается неизменным (нумерация битов в байте идет с нуля). (Если бит 1, то `1 & 1 = 1;` если бит равен 0, то `0 & 1 = 0`). Этот процесс называется наложением маски, т.к. нули в маске скрывают соответствующие биты в переменной niss.

Можно записать вышеприведенный оператор следующим образом:

```
niss &= MASKA;
```

6.4 Включение битов

В случае необходимости можно включить специфические биты, не изменяя остальные. Компьютеры управляют аппаратными средствами с помощью значений, пересланных портам. Чтобы включить какое-нибудь устройство, например, дисковод, необходимо установить 1 бит, не изменяя остальные. Сделать это можно с помощью операции побитовое ИЛИ.

Например, пусть опять же значение символической константы MASKA будет равно 2. Оператор

```
niss = niss | MASKA;
```

устанавливает первый бит niss в значение 1 и оставляет все другие биты неизменными. Можно также применить операцию ИЛИ поразрядного присваивания:

```
niss |= MASKA;
```

При этом в значение 1 устанавливаются те биты в niss, которые также включены в MASKA, а другие биты при этом остаются неизменными.

По аналогии с тем, как можно включать одни биты, не затрагивая других, их можно и отключать. Пусть символическая константа MASKA равна 253, что в двоичной системе будет равно 11111101. Тогда применение оператора `niss = niss & MASKA;`

установит первый бит переменной niss в 0, не изменяя остальных.

6.5 Переключение битов

Иногда бывает необходимо осуществить переключение битов – установка его значения в 0, если его первоначальное значение было равно 1, и

наоборот. Для переключения битов используют операцию поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ. Если значения битов объединяются с маской с помощью операции ^, значения, соответствующие 1 в маске переключаются, а значения, соответствующие 0 в маске, остаются неизменными. Пусть MASKA равна 2 (00000010 в двоичной системе счисления). Тогда для переключения первого бита в переменной niss необходимо выполнить следующее:

```
niss = niss ^ MASKA; (или niss ^= MASKA;)
```

7 ЦИКЛЫ

Действие циклов заключается в повторении определенной части вашей программы некоторое количество раз. Это повторение будет продолжаться до тех пор, пока выполняется соответствующее условие. Когда значение выражения, задающего условие, становится ложным, выполнение цикла прекращается, а управление передается оператору, следующему непосредственно за циклом.

В С существует 3 типа циклов: for (цикл со счетчиком), while (цикл с предусловием), и do ... while (цикл с постусловием).

7.1 Цикл со счетчиком for

Цикл for организует выполнение фрагмента программы фиксированное число раз. Как правило (хотя и не всегда), этот тип цикла используется тогда, когда заранее известно число повторений тела цикла.

В примере, представленном в листинге 7.1 выводится на экран таблица целых чисел от 0 до 50 и квадратные корни этих чисел. Кроме того, после каждого числа, кратного 10 выводится строка с дефисами, отделяя группы чисел.

Листинг 7.1

```
#include <math.h>
#include <stdio.h>
#define LIMIT 50
int main(){
  int x;
  double z;
  printf("Таблица значений функции z=sqrt(x)\n");
  printf("================\n");//рисуем шапку таблицы
  printf("|  x  |   z   |\n");
  printf("================\n");
  for (x=1; x<=LIMIT; x++){//заголовок цикла for
     z=sqrt(x); printf("|%5d | %7.2f|\n", x, z);
```

```
    if (x%10==0)/*если остаток от деления x на 10 равен
      0, то печатается линия с дефисами*/
      printf("---------------\n");
  }
  return 0;
}
```

В начале программы, предел целых чисел в таблице задается с помощью директивы препроцессора `define`. Это очень удобно, особенно в больших программах, когда при необходимости изменить границы таблицы не нужно отслеживать верхний предел по всей программе, а достаточно только осуществить замену в одном месте. Четыре верхних оператора `printf()` рисуют шапку таблицы. Оператор for управляет циклом. Он состоит из ключевого слова `for`, за которым следую круглые скобки, содержащие три выражения, разделенные точкой с запятой:

```
for (x=1; x<=LIMIT; x++)
```

Первое из этих трех выражений называют инициализирующим, второе – условием проверки, а третье – инкрементирующим. Эти три выражения, как правило (но не всегда), содержат одну переменную, которую обычно называют *счетчиком цикла*. В данном примере счетчиком цикла является переменная `x`. Под телом цикла понимается та часть кода, которая периодически исполняется в цикле:

```
z=sqrt(x);
printf("|%5d | %7.2f|\n", x, z);
/*если остаток от деления x на 10 равен 0, то печатается
  линия с дефисами*/
if (x%10==0)
  printf("-----------------\n");
```

В нашем случае тело цикла состоит из нескольких операторов, поэтому они заключены в фигурные скобки, и образуют блок, так как, согласно правилам языка С, в теле цикла может быть только один оператор. Если же в тело цикла необходимо поместить несколько операторов, то их нужно заключать в фигурные скобки.

Инициализирующее выражение вычисляется только один раз – в начале выполнения цикла. В нашей программе значение счетчика цикла устанавливается равным 1.

Как правило, условие выполнения цикла содержит в себе операцию отношения. Условие проверяется каждый раз перед исполнением тела цикла и определяет, нужно ли выполнять цикл еще раз или нет. Если условие выполняется, то цикл исполняется еще раз. В противном случае управление передается оператору, следующему за циклом. В нашем примере управление передается оператору `return 0;`

Инкрементирующее выражение предназначено для изменения значения счетчика цикла. Часто такое изменение сводится к инкрементированию счетчика. Модификация счетчика происходит после того, как тело цикла полностью выполнится.

В нашем примере цикл выполняется 50 раз. В первый раз он выполняется при значении `x=1`. После того, как он выполнится первый раз, значение `x` станет равным 2, и цикл будет исполняться еще раз и т.д. Последнее исполнение цикла произойдет при `x=50`, поскольку условием выполнения цикла служит `x<=50`.

Обратите внимание на оператор `x++` в заголовке цикла. Это – операция инкремента. Операция инкремента выполняет простое действие: в результате ее выполнения происходит инкремент (приращение) значения операнда на 1. Существуют две разновидности этой операции. В первом случае символы `++` предшествуют переменной (`++x`) –это префиксная форма. Во втором случае символы `++` следуют сразу за переменной (`x++`) – постфиксная форма. Пробелы между именем переменной и операцией недопустимы!!! Чтобы понять отличие этих двух форм, рассмотрим маленький пример:

```
apps = a++;//a увеличивается после того, как ее значение было использовано
bips = ++b;//b увеличивается до того, как ее значение было использовано
```

В постфиксной форме инкремента значение переменной `apps` будет присвоено до увеличения значения переменной `a` на 1. В префиксной форме инкремента значение переменной `bips` будет присвоено после увеличения значения переменной `b` на 1.

В тех случаях, когда одна из операций инкремента используется сама по себе, как в нашем примере (`x++`), не имеет значения, какая форма используется.

Имеется операция декремента `--` (уменьшение) переменной на 1, которая имеет также имеет префиксную и постфиксную формы. Особенности применения операции декремента аналогичны операции инкремента.

Возможно, нам понравится созданная в программе таблица, и мы захотим сохранить ее для дальнейшего использования или печати. Для этого таблицу необходимо сохранить в файле. Существуют два способа получения программы для работы с файлами. Первый путь заключается в применении специальных функций, используемых для открытия и закрытия файлов, чтения и записи файлов, а также осуществления других операций. Второй путь связан с использованием перенаправления потоков ввода/вывода. Этот подход является более ограниченным, но его значительно проще реализовать. Перенаправление ввода дает возможность программе использовать для ввода вместо клавиатуры файл, а также перенаправлять вывод в файл вместо экрана. Более подробно перенаправление рассмотрено в 7.4.2.

7.2 Цикл с предусловием while

Цикл for выполняет последовательность действий определенное количество раз. В том случае, когда неизвестно, сколько раз понадобится выполнить эту последовательность действий используется цикл while (цикл с предусловием).

Ниже представлен листинг программы, производящей перевод километров в мили и выводящей результат в виде таблицы.

Листинг 6.2

```
#include <stdio.h>
#define SCALE 0.6214
int main(void){
  double kilometres, miles;
  printf("Таблица перевода километров в мили\n");
  printf("Километры Мили\n");
  printf("----------------------\n");
  kilometres=1.0;
  while (kilometres<=20){
     miles=SCALE*kilometres;
     printf("%7.1f %7.2f\n", kilometres, miles);
     kilometres+=1.0;
  }
  return 0;
}
```

С помощью директивы препроцессора define вводится множитель для перевода SCALE. Сначала печатается шапка таблицы, а затем, в цикле while – и сама таблица. Цикл с предусловием напоминает цикл for. Он содержит условие продолжения цикла, но не содержит ни инициализирующих, ни инкрементирующих выражений. До тех пор, пока выполняется условие выполнения цикла (выражение в скобках справа от слова while), исполнение цикла продолжается. В данной программе цикл будет выполняться до тех пор, пока переменная kilometres будет меньше или равна 20.

Тело цикла должно содержать оператор, изменяющий значение переменной цикла, иначе цикл будет бесконечным!!! Таким оператором является kilometres+=1.0; В языке С имеются дополнительные операции присваивания: *=, /=, +=, -=, %=. В записи каждой такой операции имя переменной стоит слева от знака, а выражение – справа. Переменной присваивается новое значение, равное ее старому значению, скорректированному на величину выражения, стоящего справа. Результат зависит от используемой операции, например:

Таблица 7.1 – **Дополнительные операции присваивания**

Операция	Пояснение
alpha+=20	То же самое, что и alpha=alpha+20

`betta-=2`	То же самое, что и `betta=betta-2`
`gamma*=3`	То же самое, что и `gamma=gamma*3`
`Theta/=3.67`	То же самое, что и `thetta=thetta/3.67`
`Unique%=4`	То же самое, что и `unique=unique%4`

В примерах таблицы использовались простые числа, но можно использовать и более сложные выражения:

```
y*=4*s+12*x
```

то же самое, что и

```
y=y*(4*s+12*x)
```

Дополнительные операции присваивания обладают таким же низким приоритетом, что и операция =, т. е. меньшим, чем * или –.

Этими формами пользоваться необязательно. Однако они компактнее и более наглядны.

7.3 Цикл с постусловием do …while

Циклы `while` и `for` являются циклами с предусловием. Проверка истинности условия осуществляется перед каждой итерацией цикла, поэтому существует вероятность того, что операторы, предусматриваемые циклом, никогда не будут выполнены. В языке С имеется цикл, в котором проверка условия выполняется на выходе из каждой итерации цикла (цикл с постусловием), благодаря чему гарантируется выполнение операторов хотя бы один раз. Это цикл – `do while`.

В листинге 7.3 представлен пример решения предыдущей задачи с использованием цикла с постусловием.

Листинг 7.3

```
#include <stdio.h>
#define SCALE 0.6214
int main(void){
  double kilometres, miles;
  printf("Таблица перевода километров в мили\n");
  printf("Километры Мили\n");
  printf("----------------------\n");
  kilometres=1.0;
  do {
     miles=SCALE*kilometres;
     printf("%7.1f %7.2f\n", kilometres, miles);
     kilometres+=1.0;
  } while (kilometres<=20);
  return 0;
}
```

После служебного слова do (до слова while) в фигурных скобках заключено тело цикла. После выполнения тела цикла осуществляется проверка условия, стоящего справа от слова while в круглых скобках. При выполнении этого условия тело цикла выполняется еще раз. После чего следует новая проверка условия продолжения цикла и т.д. до тех пор, пока условие будет оставаться истинным.

Ранее мы рассмотрели операции поразрядного смещения. Применим теперь оператор цикла for для перевода целого положительного числа из десятичной в двоичную систему счисления.

Листинг 7.5

```
#include <stdio.h>
int main(){
  unsigned short int x;
  int i;
  printf("Enter an unsigned integer (0 - 65535): ");
  scanf("%u", &x);
  unsigned short int MASKA=1<<15;
  printf("%7u = ", x);
  for (i=1; i<=16; i++){
     putchar(x & MASKA ? '1' : '0');
     x<<=1;
     if (i%8 == 0)
        putchar(' ');
  }
  putchar('\n');
  return 0;
}
```

Данная программа выводит на печать целое без знака число в двоичном представлении группами по 8 бит. В качестве маски MASKA используется число 2^{16}=32768 (1000000000000000 в двоичном представлении), полученное сдвигом 1 влево на 15 разрядов. Можно было бы задать маску и так: `unsigned short int MASKA=32768;`

Оператор

```
putchar(x & MASKA ? '1' : '0');
```

определяет, что надо напечатать для текущего крайнего слева бита переменной x. Функция putchar() выводит символ в стандартный поток вывода. А вот какой символ выводить, определяется тернарной операцией ?:. Все, что стоит до знака ? – логическое выражение; после ? и до : – оператор, который выполняется в случае истинности логического выражения; а что после : – оператор, выполняющийся в случае ложности логического выражения.

7.4 Дополнительные сведения

7.4.1 Вложенные циклы

Вложенные циклы используются в том случае, когда один цикл выполняется в теле другого цикла. Очень часто вложенные циклы используются для упорядоченного отображения данных в виде строк и столбцов. Один цикл может взять на себя обработку всех строк, в то время как второй выполняет обработку всех столбцов в строке. Пример такой программы представлен в листинге 6.4. Здесь имеется 2 цикла со счетчиком. Первый производит обработку строк, в которых печатается знак '*', а во втором цикле производится вывод этих знаков в столбцах соответствующей строки.

Листинг 6.4

```
#include <stdio.h>
int main(){
  int x, i, j;
  printf("Введите сторону квадрата: ");
  while (1){
     if((scanf("%d", &x)==1) && x>=1 && x<=20)
     break;
     else {
        printf("Введите сторону квадрата от 1 до 20: ");
        continue;
     }
  }
  for (i=0; i<x; i++){
     for (j=0; j<x; j++)
        putchar('*');
     putchar('\n');
  }
  return 0;
}
```

Функция putchar() выводит символ * в стандартный поток вывода.

7.4.2 Перенаправление ввода/вывода

Допустим, у нас имеется программа program, которая считывает день и номер месяца вашего рождения. Далее она печатает на экране сообщение:

Меня зовут Иванов Алексей. Я родился … … 1986 года.

День и месяц рождения можно вводить с клавиатуры, а можно поместить в файл (например, birth.dat) и запуская программу следующим образом:

```
program <birth.dat
```

получить тот же самый результат. Аналогично и с выводом в файл:

```
program >resultat.dat
```

При выполнении данной команды просьба ввести день и месяц рождения и результаты будут перенаправлены в файл resultat.dat.

Более того, можно перенаправлять ввод из файла и одновременно осуществлять вывод в файл:

```
program <birth.dat >resultat.dat
```

Поэтому для вывода нашей таблицы в файл необходимо набрать в командной строке:

```
program >tablitsa.dat
```

и вместо вывода на экран мы получим в текущем каталоге новый файл, в котором и будет наша таблица.

Очень удобно использовать перенаправление ввода, когда используется или отлаживается программа, в которой приходиться вводить большой объем данных (матрицы, элементы таблиц и т.д.), т.к. именно многократный процесс ввода одних и тех же данных может вызвать легкую нервозность (как минимум).

8 ОДНОМЕРНЫЕ МАССИВЫ

8.1 Общие сведения

Допустим, необходимо сохранить в памяти температуру каждого дня месяца. Можно для каждого дня использовать свою переменную. Однако, в данном случае, необходимо объявлять 31 переменную. При этом обработка данных (поиск наибольшего, наименьшего и т.д.) будет чрезвычайно затруднена. В подобных случаях обычно используются массивы.

Массив – это набор переменных одного типа, имеющих одно и то же имя.

Общая форма объявления одномерного массива:

тип имя_переменной [размер];

где тип – базовый тип массива,
имя_переменной – имя массива,
размер – количество элементов массива.
Например:

```
double mass[31];
```

Здесь происходит объявление массива `mass`, состоящего из 31 элемента, каждый из которых имеет тип `double`.

Примеры правильного объявления массива указаны в таблице 8.1.

Таблица 8.1 – **Правильное объявление массива**

Объявление	Пояснения

`double mass[10];`	объявление массива с именем `mass`, состоящего из 10 элементов, каждый из которых имеет тип `double`;
`int a[30];`	объявление массива с именем a, состоящего из 30 элементов, каждый из которых имеет тип `int`;
`float b, c[5];`	объявление переменной `b` типа `float` и массива c, состоящего из 30 элементов, каждый из которых имеет тип `float`;
`short int i,j[8],k;`	объявление массива `j`, состоящего из 8 элементов, каждый из которых имеет тип `short int` и переменных `i` и `k` типа `short int`;

Варианты неправильного объявления массива указаны в таблице 8.2

Т а б л и ц а 8.2 – **Неправильное объявление массива**

Ошибка в объявлении	Пояснения
`double mass[0];`	массив `mass` не может иметь нулевое количество элементов;
`int a[-8];`	количество элементов массива не может иметь отрицательное значение;
`int c[5.0];`	количество элементов массива не может иметь вещественное значение;
`int k=7,i[k];`	количество элементов массива не может быть переменной (в C99 данная конструкция ошибкой уже не является);

В C99 возможно объявление массива, размер которого указан с помощью переменной.

Каждый элемент массива имеет свой индекс. Индекс первого элемента любого массива равен 0.

На рисунке 8.1 представлен массив, который объявляется как

```
int b[12];
```

Рисунок 8.1 – Схематичное изображение массива

Доступ к элементу массива осуществляется с помощью имени массива и индекса. Индекс элемента массива помещается в квадратных скобках после имени.

Например, следующий оператор присваивает элементу массива `mass` с индексом 5 значение 37.56 (рисунок 8.2):

```
mass[5]=37.56;
```

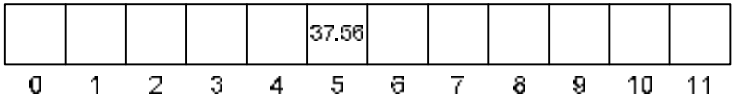


Рисунок 8.2 – Присваивание элементу `mass`[5] значения 37.56

Необходимо обратить внимание, что элемент `mass[5]` по счету является шестым.

Объем памяти, необходимый для хранения одномерного массива вычисляется по следующей формуле:

кол_байт=`sizeof`(базовый_тип) × размер_массива

Например, для массива описанного как

```
double mass[10];
```

количество занимаемых байт составит

кол_байт=`sizeof`(double) × размер_массива=8 × 10 = 80 байт.

В таблице 8.3 приведены подсчеты.

Т а б л и ц а 8.3 – **Расчет количества занимаемых байт массивом**

Объявление	Подсчет
`int bb[34];`	кол_байт=sizeof(int) × размер_массива=4 × 34 = 136 байт.
`float w[20];`	кол_байт=sizeof(float) × размер_массива=4 × 20 = 80 байт.
`double mass[10];`	кол_байт=sizeof(double) × размер_массива=8 × 10 = 80 байт.
`char c[67];`	кол_байт=sizeof(char) × размер_массива=1 × 67 = 67 байт.

Важно отметить, что во время выполнения программы не проверяется соблюдение границ массива.

Например, при компиляции следующей программного кода ошибки не произойдет.

```
int a[10],i;
i=12;
a[i]=5; //здесь происходит за границы массива
```

Однако при выполнении происходит обращение к несуществующему элементу `a[12]`, что приводит к разрушению соседних участков памяти.

Проверка индексов должна вестись самим программистом. Если программист не уверен, что индекс не выйдет за допустимую границу (не станет большим и равным количеству элементов массива), он может использовать оператор `if`.

```
int a[10],i;
i=12;
if(i<=9) a[i]=5;
  else printf("Попытка выхода за пределы массива\n");
```

В данном случае обращения к элементу a[12] не произойдет, т.к. индекс 12 не удовлетворяет условию оператора if. Вместо этого на экран выведется сообщение о попытке выхода за пределы массива.

Та же самая ошибка произойдет при выполнении, если попытаться обратиться к элементу с отрицательным индексом.

Например:

```
int a[10],i;
i=-56;
a[i]=5;
```

При выполнении компиляции ошибка опять не будет обнаружена.

Для того чтобы ограничить индекс массива как сверху, так и снизу необходимо написать двойное условие в операторе if.

```
int a[10],i;
i=-56;
if ((i>=0)&&(i<=9))
    a[i]=5;
else printf("Попытка выхода за пределы массива\n");
```

В данном случае выход за пределы массива невозможен.

Однако проверку необходимо осуществлять только в том случае, если нет уверенности в том, что выход за пределы массива не произойдет.

Если обращение идет к элементам массива, например, в цикле for и переменная-счетчик, используемая как индекс массива, имеет допустимые пределы, дополнительную проверку вводить не следует.

```
int a[10],i;
for (i=0; i<=9; i++)
  a[i]=i;
```

Например, в данном случае выход за пределы массива невозможен, т.к. проверка осуществляется с помощью цикла for.

8.2 Ввод и вывод массива.

Рассмотрим пример ввода массива, который объявлен, как int b[3]

```
for (i=0;i<3;i++)
  scanf("%d",&b[i]);
```

Ниже рассмотрен пошагово ввод массива.

Таблица 8.4 – **Ввод массива**

Выполняемый оператор	Выполняемое действие	Пояснение
for(i=0;i<3;i++)	i=0	Присваивание начального значения для i
for(i=0;i<3;i++)	i<3 (0<3)	Проверка условия
scanf("%d",&b[i]);	scanf("%d",&b[0])	Ввод элемента

`for(i=0;i<3;i++)`	`i++`	Увеличение `i` на 1
`for(i=0;i<3;i++)`	`i<3 (1<3)`	Проверка условия
`scanf("%d",&b[i]);`	`scanf("%d",&b[1])`	Ввод элемента `b[1]`
`for(i=0;i<3;i++)`	`i++`	Увеличение `i` на 1
`for(i=0;i<3;i++)`	`i<3 (2<3)`	Проверка условия
`scanf("%d",&b[i]);`	`scanf("%d",&b[2])`	Ввод элемента `b[2]`
`for(i=0;i<3;i++)`	`i++`	Увеличение `i` на 1
`for(i=0;i<3;i++)`	`i<3 (3<3)`	Проверка условия и выход из цикла

Элементы массива можно вводить как через пробел, так и через другие символы-разделители (конец строки, символ табуляции).

Вышеописанный метод является кратким в записи, однако при вводе больших массивов, пользователь может запутаться с номером вводимого элемента массива.

Поэтому можно перед функцией scanf() добавить функцию `printf()`, которая бы выводила номер вводимого элемента массива.

```
for (i=0;i<n;i++){
  printf("a[%d]=",i);
  scanf("%f",&a[i]);
}
```

В данном случае элементы массива лучше вводить через символ конца строк (нажатие клавиши "Enter"). После ввода каждого элемента будет отображаться приглашение ввести следующий элемент массива.

Выводить на экран элементы массива можно тоже разными способами. Наиболее распространенными являются:

– в одну строку через пробел;

– каждый элемент на новой строке, с обозначением номера элемента.

Если необходимо вывести весь массив в одну строку через пробел, необходимо вставить следующий программный код:

```
int i;
float a[5];
  ...
for(i=0; i<5; i++)
  printf("%f ",a[i]);
```

В данном случае на экран выводится массив, значения которого имеют тип `float`, о чем говорит спецификатор `%f`. После спецификатора `%f` ставиться пробел.

Вывод в данном случае будет выглядеть следующим образом:

```
0.125126 56.358532 19.330423 80.874046 58.500931
```

Для вывода элементов в каждой строке необходимо в функцию `printf()` добавить символ конца строки `'\n'`:

Если еще необходимо, чтобы выводился индекс элемента массива, код необходимо изменить следующим образом

```
int i;
float a[5];
  ...
for(i=0; i<5; i++)
  printf("a[%d]=%f\n",i,a[i]);
```

Результат вывода показан ниже:

```
a[0]=0.125126
a[1]=56.358532
a[2]=19.330423
a[3]=80.874046
a[4]=58.500931
```

Если необходимо вывести другой тип элементов массива, то вместо `%f` необходимо подставить другой спецификатор.

8.3 Генерация псевдослучайных чисел.

Часто в учебных программах, использующих массивы, заполнение элементов осуществляют не с помощью функции ввода, а псевдослучайными числами.

Прежде всего необходимо отметить, что истинно случайные числа можно получить только в результате случайных экспериментов (подбрасывание монеты, процесс радиоактивного распада и т. д.). Такие числа сгенерировать на компьютере практически невозможно, т.к. любой элемент компьютера имеет четко предсказуемую логику, и состояние его строго определено.

Однако существуют алгоритмы, которые вырабатывают определенную последовательность чисел, похожую на случайную. Такие числа называют псевдослучайными.

Принцип генерации псевдослучайных чисел заключается в том, что задается исходное число, а на его основе по определенному алгоритму вычисляются другие числа.

Исходное число задается с помощью функции `srand()`, находящейся в библиотеке `stdlib.h`.

Общий вид функции

`srand (unsigned` нач_число);

где нач_число – начальное число, для генерации псевдослучайной последовательности.

Пример:

```
srand(53);
```

– задает начальное число 53 для генерации псевдослучайной последовательности.

Генерация случайных чисел осуществляется с помощью функции `rand()`, находящейся в той же библиотеке `stdlib.h`. При каждом обращении к функции возвращается целое число в интервале между нулем и значением `RAND_MAX` (Обычно равно `UINT_MAX`).

Общий вид функции:

```
int rand();
```

Пример

```
a=rand();
```

– генерирует очередное псевдослучайное число и записывает его в целочисленную переменную `a`.

Простейший пример программы, заполняющей псевдослучайными числами массив и выводящей его значения на экран, показан в листинге 8.1.

Последовательность псевдослучайных чисел определяется вводимым в начале программы числом `n`.

Листинг 8.1 – Программа генерации псевдослучайных чисел на основе введенного числа

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
  int a[20],i,n;
  printf("Введите исходное число n:");
  scanf("%d",&n);
  srand(n);
  for (i=0; i<20; i++)
     a[i]=rand();
  for (i=0; i<20; i++)
     printf("a[%d]=%d\n",i,a[i]);
  return 0;
}
```

При разных вводимых `n` будут генерироваться разные последовательности.

Однако, псевдослучайная последовательность будет целиком зависеть от вводимого пользователем числа `n`. При одинаковых `n` последовательность псевдослучайных чисел будет одинакова.

Для того, чтобы этого не происходило, необходимо в качестве аргумента `srand()` использовать значение, не повторяющееся при каждом запуске программы.

Для этого можно использовать функцию `time()`, которая возвращает текущее значение времени в секундах, начиная от 1 января 1970 года.

Можно записать следующим образом:

```
srand(time(NULL));
```

В данном случае в функцию `srand()` будет подставляться текущее время, которое для каждого запуска программы будет уникальным. В результате каждый раз будет получена новая псевдослучайная последовательность, не похожая на предыдущие и не зависящая от пользователя.

Часто необходимо генерировать не весь диапазон целых чисел от 0 до `RAND_MAX`, а только в определенных пределах от `n_min` до `n_max`.

Рассмотрим сначала случай, когда нужно сгенерировать целые числа от 0 до `n_max`.

Для того, чтобы генерировались числа именно в этом диапазоне можно, как один из вариантов, взять остаток от деления числа сгенерированного командой `rand()` на `n_max+1`.

```
a[i]=rand()%(n_max+1);
```

Теперь рассмотрим случай когда необходимо получить числа в диапазоне от `n_min` до `n_max`.

```
a[i]=rand()%(n_max-n_min+1)+n_min;
```

Ниже приведен листинг программы, заполняющей целочисленный массив размером `MASS=20` псевдослучайной последовательностью чисел в диапазоне от `N_MIN=-5` до `N_MAX=47` и выводящий его на экран.

Л и с т и н г 8.2 – Программа генерации случайных чисел на основе текущего времени

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N_MIN -5
#define N_MAX 47
#define MASS 20
int main(void){
  int a[MASS],i;
  srand(time(NULL));
  for (i=0; i<MASS; i++)
     a[i] = rand()%(N_MAX - N_MIN +1)+ N_MIN;
  for (i=0; i<MASS; i++)
     printf("a[%d]=%d\n",i,a[i]);
  return 0;
}
```

8.4 Примеры задач

8.4.1 Нахождение максимального значения в массиве.

Допустим, у нас есть массив с введенными значениями элементов. Необходимо найти значение максимального элемента.

Для этого необходимо объявить дополнительную переменную (напр. max), в которую будем записывать максимальное значение массива. Тип переменной должен совпадать с типом массива.

Алгоритм нахождения максимального элемента следующий.

Мысленно ограничим массив одним элементом с нулевым индексом. Если представить, что это весь массив, то в данной ячейке находится единственное, в то же время являющеюся максимальным, значение. Поэтому присваиваем переменной max значение элемента с индексом 0.

Добавим к рассмотрению элемент с индексом 1. Проверим, будет ли значение данного элемента больше значения переменной max. Если «да», то значит значение элемента с индексом 1 больше значения элемента с индексом 0, следовательно, записываем значение элемента с индексом 1 в переменную max. Если «нет», то значение элемента с индексом 1 не больше значения элемента с индексом 0, следовательно, сразу переходим к следующему шагу. В любом случае в переменной max будет находиться максимальное из первых двух элементов массива.

Каждый раз, рассматривая новый элемент массива, сравниваем его со значением переменной max. Если данный элемент больше, то значит, найдено новое максимальное значение, следовательно, записываем это значение в переменную max. Если данный элемент не больше, значит, максимальное значение из рассмотренных ячеек было найдено ранее.

Простейший пример программы рассмотрен в листинге 8.3

Л и с т и н г 8.3 – Программа нахождения максимального значения в одномерном массиве

```
#include <stdio.h>
#define N 10
int main(void){
  double m[N],max;
  int i;
  printf("Введите элементы массива:\n");
  for (i=0; i<N; i++) {
     printf("Введите m[%d]: ",i);
     scanf("%lf",&m[i]);
  }
  max=m[0];
  for (i=1; i<N; i++)
     if (m[i]>max) max=m[i];
```

```
  printf("Введенный массив:\n");
  for (i=0; i<N; i++)
     printf("%g ",m[i]);
  printf("\nМаксимальное значение max=%g\n",max);
  return 0;
}
```

8.4.2 Нахождение минимального значения в массиве.

Алгоритм нахождения минимального значения в массиве аналогичен алгоритму нахождения максимального значения. Вместо переменной max используется переменная min.

Разница проявляется на 2-ом и 3-ем шагах. Проверяется, будет ли значение текущего проверяемого элемента **меньше**, чем значение переменной min.

Если «да», то переменной min присваивается значение данного элемента.

Код программы приведен в листинге 8.4. В данной программе заполнения массива производится с помощью генератора псевдослучайных чисел в диапазоне от 50 до 300.

Л и с т и н г 8.4 – Программа нахождения минимального значения в одномерном массиве

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 10
#define N_MAX 300
#define N_MIN 50
int main(void){
  double m[N],min;
  int i;
  srand(time(NULL));
  for (i=0; i<N; i++)
     m[i] = rand()%(N_MAX - N_MIN +1)+ N_MIN;
  min=m[0];
  for (i=1; i<N; i++)
     if (m[i]<min) min=m[i];
  printf("Введенный массив:\n");
  for (i=0; i<N; i++)
     printf("%g ",m[i]);
  printf("\nМинимальное значение min=%g\n",min);
  return 0;
}
```

8.4.3 Нахождение индекса максимального элемента массива.

Данный алгоритм схож с нахождением максимального значения.

Дополнительно вводится переменная i_max целочисленного типа. В самом начале переменной max присваивается значение элемента с индексом 0, а переменной i_max индекс этого элемента (равен нулю).

На каждом шаге сравнивается значение переменной max и текущего элемента массива. Если значение элемента массива больше, переменной max присваивается это значение, а переменной i_max присваивается индекс данного элемента.

По окончании проверки массива в переменной max будет находиться максимальное значение массива, а в переменной i_max – индекс максимального элемента.

Листинг 8.5 программы приведен ниже.

Л и с т и н г 8.5 – Программа нахождения индекса максимального элемента массива

```
#include <stdio.h>
#define N 10
int main(void){
  double m[N],max;
  int i,i_max;
  printf("Введите элементы массива:\n");
  for (i=0; i<N; i++) {
     printf("Введите m[%d]: ",i);
     scanf("%lf",&m[i]);
  }
  max=m[0];
  i_max=0;
  for (i=1; i<N; i++)
     if (m[i]>max) {
        max=m[i];
        i_max=i;
     }
  printf("Введенный массив:\n");
  for (i=0; i<N; i++)
     printf("%g ",m[i]);
  printf("\nМаксимальное значение max=%g\n",max);
  printf("находится в %d элементе\n",i_max+1);
  return 0;
}
```

8.4.4 Подсчет количества положительных элементов.

В данном случае объявляется дополнительная целочисленная переменная (напр. n_p), в которой будет подсчитываться количество положительных элементов.

Вначале присваиваем этой переменной значение 0. (т.к. положительных элементов еще не найдено).

На каждом проходе цикла проверяем, является ли текущий элемент массива положительным.

Если «да», значит, найден положительный элемент и значение переменной n_p увеличиваем на 1.

По окончании цикла в переменной n_p будет находится количество положительных переменных массива.

Ниже приведен листинг 8.6 данной программы.

Л и с т и н г 8.6 – Программа нахождения количества положительных элементов массива

```
#include <stdio.h>
#define N 10
int main(void){
  double m[N];
  int i,k;
  printf("Введите элементы массива:\n");
  for (i=0; i<N; i++) {
     printf("Введите m[%d]: ",i);
     scanf("%lf",&m[i]);
  }
  k=0;
  for (i=0; i<N; i++)
     if (m[i]>0)
        k++;
  printf("Введенный массив:\n");
  for (i=0; i<N; i++)
     printf("%g ",m[i]);
  printf("\nКоличество положительных элементов=%d\n",k);
  return 0;
}
```

8.4.5 Подсчет количества элементов четных, нечетных и кратных некоторому числу.

Определение на четность, нечетность или кратность какому-либо числу определяется с помощью операции остатка от деления.

Если остаток от деления элемента массива на какое-то число равен нулю, то значение элемента массива кратно этому числу.

Если число при делении на 2 не имеет остатка, то оно является четным, иначе – нечетным.

Ниже приведен листинг программы нахождения количества нечетных чисел:

Л и с т и н г 8.7 – Программа нахождения количества нечетных элементов массива, сгенерированных с использованием генератора псевдослучайных чисел.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 10
int main(void){
  int m[N],i,k;
  srand(time(NULL));
  for (i=0; i<N; i++)
     m[i]=rand()%100;
  k=0;
  for (i=0; i<N; i++)
     if (m[i]%2!=0)
        k++;
  printf("Введенный массив:\n");
  for (i=0; i<N; i++)
     printf("%d ",m[i]);
  printf("\nКоличество нечетных элементов=%d\n",k);
  return 0;
}
```

9 ДВУМЕРНЫЕ МАССИВЫ.

9.1 Общие сведения

Двумерный массив – это массив одномерных массивов.

Объявление двумерного массива выглядит следующим образом:

```
int b[3][5];
```

Данный двумерный массив можно представить, как 3-х элементный массив 5-ти элементных массивов типа `int` или как квадратную матрицу, содержащую 3 строки и 5 столбцов. Графическое представление данного массива представлено на рисунке 9.1.

Нумерация строк производится от 0 до 2 и столбцов – от 0 до 4.

Обращение к элементу двумерного массива выглядит следующим образом:

```
b[1][3]
```

Здесь происходит обращение к ячейке с индексом 3 массива `b[1]`

Часто двумерный массив условно представляют в виде матрицы, состоящий из строк и столбцов. Первый индекс указывает номер строки (номер_строки=индекс+1), а второй – номер столбца (номер_столбца=индекс+1). Тогда

```
b[1][3]
```

можно представить, как обращение к ячейке массива b, расположенной на пересечении строки 2 и столбца 4.

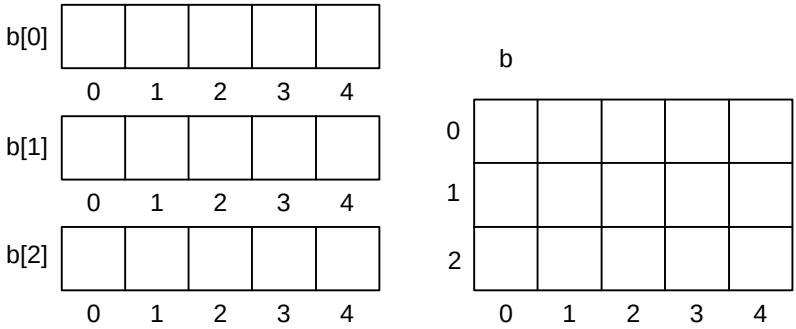


Рисунок 9.1 – Представление 2-х мерного массива

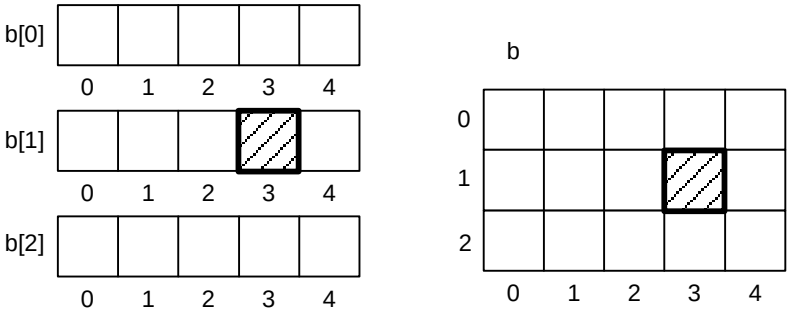


Рисунок 9.2 – Обращение к элементу b[1][3]

9.2 Ввод-вывод 2-х мерного массива.

Ввод-вывод 2-х мерного массива может производиться с помощью 2-х циклов `for`, вложенных друг в друга.

Один из них имеет переменную-счетчик `i`, которая обозначает номер текущей строки, а второй переменную-счетчик `j`, обозначающую столбец.

Допустим, у нас есть 2-х мерный массив вещественных значений размером M×N (M строк и N столбцов).

Ввод можно осуществить следующим способом:

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
     scanf("%lf",&m[i][j]);
```

При этом не отображаются индексы введенного элемента, а значения можно вводить, как в строку через пробел, так и нажимая после каждого ввода клавишу Enter.

Если необходимо, чтобы перед вводом каждой строки выводился ее номер, код будет выглядеть так:

```
for (i=0; i<M; i++){
  printf("Введите %d строку:\n",i+1);
  for (j=0; j<N; j++)
     scanf("%lf",&m[i][j]);
}
```

Если же нужно, чтобы перед вводом каждого элемента отображались его индексы, код необходимо переделать следующим образом:

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++){
     printf("Введите m[%d][%d]: ",i,j);
     scanf("%lf",&m[i][j]);
  }
```

Вывод на экран матрицы можно также осуществлять несколькими способами:

– построчно;

– каждый элемент в новой строке.

Для того, чтобы вывести массив построчно необходимо в программу добавить следующий фрагмент:

```
printf("Введенный массив:\n");
for (i=0; i<M; i++){
  for (j=0; j<N; j++)
     printf("%g ",m[i][j]);
  printf("\n");
 }
```

В данном коде находятся 2 цикла, вложенных друг в друга. Внутренний цикл с функцией `printf("%g ",m[i][j])` отвечает за вывод значений строки массива построчно через пробел. Во внешнем цикле находится внутренний цикл и команда `printf("\n")`.

Возможный результат выполнения данного кода показан ниже:

```
Введенный массив:
4.3 6.4 3.6
2.3 4.7 3.5
```

```
3.4 7.6 4.3
```

Если необходимо вывести каждый элемент в новой строке, можно воспользоваться данным кодом:

```
printf("Введенный массив:\n");
for (i=0; i<M; i++){
  for (j=0; j<N; j++)
     printf("m[%d][%d]= %g\n",i,j,m[i][j]);
  printf("\n");
}
```

При этом каждое значение элемента массива выводится с новой строки. Перед значением выводятся номера строки и столбца текущего элемента. Ниже показан результат использования данного кода:

```
Введенный массив:
m[0][0]= 4.3
m[0][1]= 6.4
m[0][2]= 3.6
m[1][0]= 2.3
m[1][1]= 4.7
m[1][2]= 3.5
m[2][0]= 3.4
m[2][1]= 7.6
m[2][2]= 4.3
```

10 УКАЗАТЕЛИ.

10.1 Общие сведения

Каждый байт в ОЗУ имеет свой адрес, значение которого обычно обозначается шестнадцатеричным числом. Максимальное значение величины адреса зависит от разрядности системы. Для 32-х битных систем для хранения адреса используются 32 бита, и максимальная значение составляет 2^{32}-1. На рисунке 10.1 схематично показано фрагмент оперативной памяти компьютера. Каждая ячейка обозначает один байт

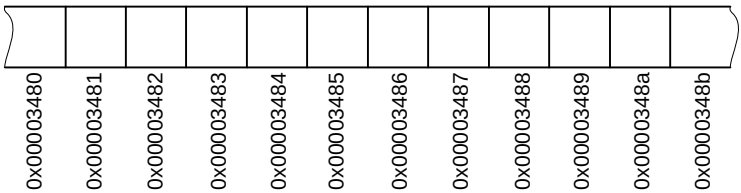


Рисунок 10.1 – Фрагмент ОЗУ компьютера

Для хранения переменных в зависимости от типа отводится определенное количество байт. При этом адресом переменной является адрес первого

байта участка памяти, в которой хранится переменная. Например, после выполнения следующего программного кода

```
short a;
float b;
```

переменные a и b могут расположиться в памяти следующим образом (рисунок 10.2).

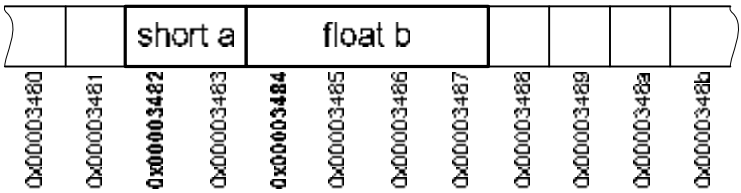


Рисунок 10.2 – Возможное расположение переменных short a и float b.

Причем, адрес переменной `a` будет `0x00003482`, а переменной `b` – `0x00003484`.

Указатель – это переменная, значением которой является адрес некоторого объекта (обычно другой переменной) в памяти компьютера. Например, если одна переменная содержит адрес другой переменной, то говорят, что первая переменная указывает (ссылается) на вторую.

Например на рисунках 10.3 и 10.4 в ячейки по адресам от `0x00003488` до `0x0000348b` (т.к. любой указатель для 32-битных систем занимает 4 байта) записан, в первом случае (рисунок 10.3), адрес переменной `short a`, а во втором – адрес переменной `float b` (рисунок 10.4).

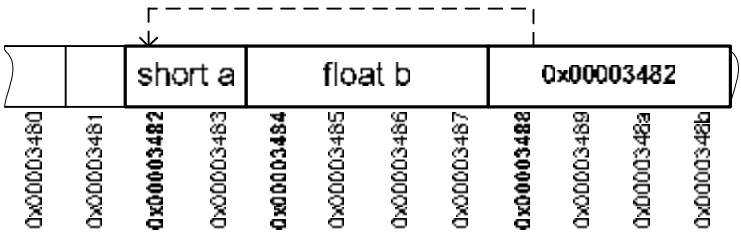


Рисунок 10.3 – Указатель на пременную short a.

В первом случае говорят, что по адресу `0x00003488` записан указатель на переменную типа `short`, а во втором – указатель на переменную типа `float`.

Объявление указателя состоит из имени базового типа, символа * и имени переменной. Общая форма объявления указателя следующая:

тип *имя_переменной_1;

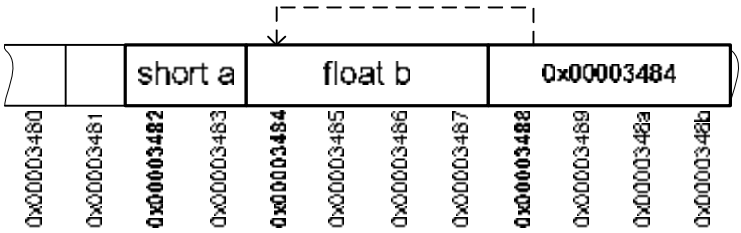


Рисунок 10.4 – Указатель на переменную `float b`

Здесь тип — это базовый тип указателя, т.е. тип переменной, на которую данный указатель ссылается; имя_переменной_1 определяет имя переменной-указателя.

Например, для объявления указателя на переменную типа `short` необходимо записать

```
short *c;
```

А для объявления указателя на переменную типа `float`:

```
float *d;
```

В качестве имени указателя может использоваться любое неиспользуемое имя.

Базовый тип указателя определяет тип объекта, на который указатель будет ссылаться. Фактически указатель любого типа может ссылаться на любое место в памяти. Однако выполняемые с указателем операции существенно зависят от его типа. Например, если объявлен указатель типа `int *`, компилятор предполагает, что любой адрес, на который он ссылается, содержит переменную типа `int`, хоть это может быть и не так. Следовательно, объявляя указатель, необходимо убедиться, что его тип совместим с типом объекта, на который он будет ссылаться.

10.2 Операции над указателями

В языке С определены две операции для работы с указателями: `*` и `&`. Оператор `&` - это унарный оператор, возвращающий адрес своего операнда.

Например, оператор

```
c = &a;
```

присваивает переменной `c` адрес переменной `a`. Адрес и значение переменной — это совершенно разные понятия. Оператор `&` можно представить себе как оператор, возвращающий адрес объекта.

Например, при выполнении следующего кода программы будут объявлены и расположены в памяти переменные a и b типа short и указатель c на данный тип.

```
short a, b, *c;
```

При этом в памяти данные переменные могут располагаться следующим образом, как показано на рисунке 10.5.

Тогда после выполнения следующего кода в переменную c запишется адрес переменной a (рисунок 10.6).

```
c = &a;
```

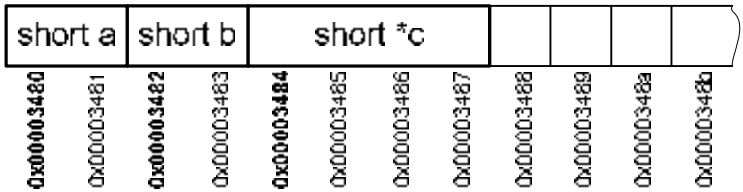


Рисунок 10.5 – Возможное расположение в памяти переменных short a, short b и переменной-указателя short *c

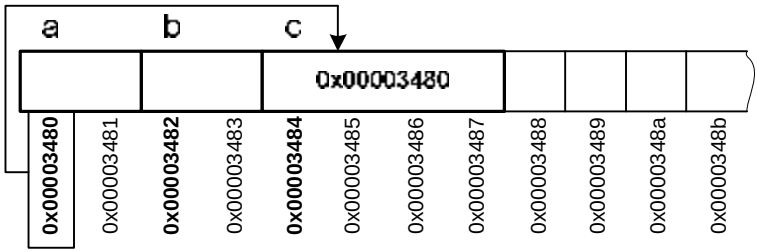


Рисунок 10.6 – При выполнении операции c = &a; в переменную-указатель c копируется адрес переменной a.

Вторая операция для работы с указателями (ее знак, т.е. оператор, *) выполняет действие, обратное по отношению к &. Оператор * — это унарный оператор, возвращающий значение переменной, расположенной по указанному адресу. Например, если c содержит адрес переменной a, то оператор

```
b = *c;
```

присваивает переменной b значение переменной a. Действие оператора * можно выразить словами "значение по адресу", тогда предыдущий оператор может быть прочитан так: "b получает значение переменной, расположенной по адресу c".

Например, рассмотрим фрагмент кода:

```
short a, b, *c;
a=64;
c = &a;
```

После выполнения возможное расположение переменных в памяти и их значения будут такими, как показано на рисунке 10.7.
После выполнения оператора

```
b = *c;
```

в переменную b запишется значение переменной, адрес которой записан в переменной c, т.е. переменной a, как показано на рисунке 10.8.

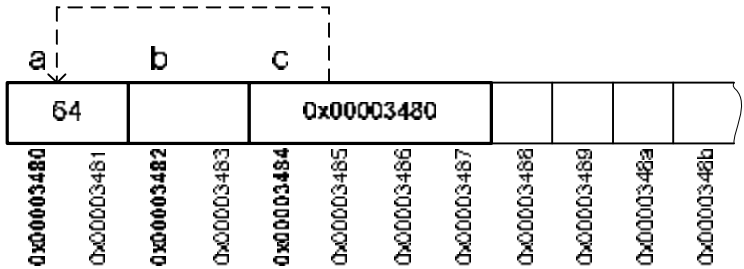


Рисунок 10.7 – Указатель-переменная c указывает на переменную a

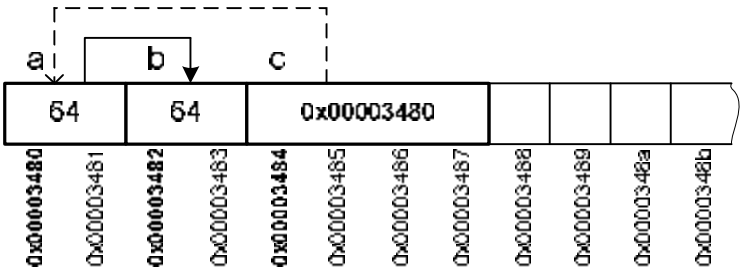


Рисунок 10.8 – Выполнение операции b = *c. Переменной b присваивается значение, находящейся по адресу, записанному в переменной-указателе c.

В языке C допустимы только 2 арифметические операции над указателями: суммирование и вычитание.

Предположим, что текущее значение указателя c типа short равно 0x00003480.

Это значит, что первый байт переменной типа short расположен по адресу 0x00003480, а сама переменная занимает 2 байта (с 0x00003480 по 0x00003481).

Тогда после операции увеличения

```
c++;
```

указатель примет значение 0x00003482, а не 0x00003481. Т. е. при увеличении указателя, он будет ссылаться на следующее число типа `short`. (рисунок 10.9)

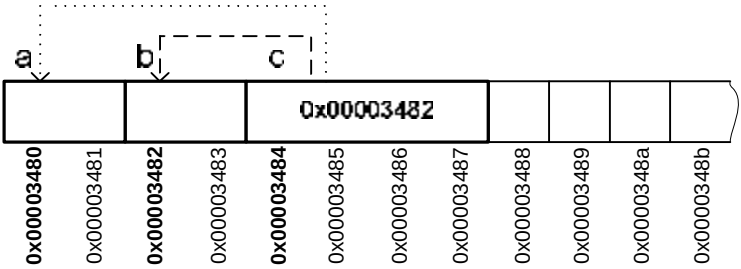


Рисунок 10.9 – После выполнении операции `c++`, указатель-переменная с указывает на следующий элемент

То же самое справедливо и для операции уменьшения. Например, если `p1` равно 0x00003482, то после выполнения оператора:

```
c--;
```

значение `c` будет 0x00003480.

В общем случае операции адресной арифметики подчиняются следующим правилам. При выполнении операции инкремента над переменной-указателем, данный указатель будет ссылаться на следующий объект своего базового типа. После выполнения операции декремента – на предыдущий объект. Т. е. для всех указателей адрес увеличивается или уменьшается на величину, равную размеру объекта того типа, на который они указывают.

К указателям также можно добавлять или вычитать целые числа. Например, выполнение оператора

```
c=c+10;
```

«передвигает» указатель с на 10 объектов в сторону увеличения адреса.

10.3 Массивы и указатели.

Имя массива является одновременно указателем на первый элемент массива, т.е. обозначения `a` и `&a[0]` – эквивалентны.

Каждый элемент массива занимает определенное количество байт, зависящее от его типа. Кроме того, все элементы в памяти располагаются последовательно.

Обратиться к элементу массива можно 2-мя способами:

1. С помощью индекса массива:

Например:

```
int a[10];
a[4]=10;
```

2. С помощью адресной арифметики:

```
int a[10];
*(a+4)=10;
```

10.4 Указатели NULL.

В языке С используют понятие пустого указателя. Значения данного указателя равно нулю и обычно обозначается при помощи макроса `NULL`.

Допустим

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

Пустой указатель никуда не указывает, и при попытке обратиться к памяти с его помощью возникает ошибка.

Однако он может быть полезен, если необходимо показать, что указатель временно не используется.

Кроме того, некоторые функции могут возвращать значение `NULL`.

10.5 Вывод значения указателя на экран.

Для вывода на экран значения указателя следует использовать функцию `printf()` со спецификатором `%p`.

Например:

```
#include <stdio.h>
int main(void){
  int a=3, *b;
  b=&a;
  printf("Переменная a находится по адресу %p\n",b);
  return 0;
}
```

Данная программа определяет адрес переменной `a`, присваивает его переменной-указателю `b` и выводит это значение на экран.

11 СИМВОЛЫ И СТРОКИ.

11.1 Символы.

В языке С можно оперировать символами. Однако для этого не существует отдельного типа данных. Для хранения символов можно использовать любой целочисленный тип, но чаще всего используется тип `char`.

Символьные значения заключаются в одинарные кавычки. Например:

```
'a'
```

Каждому символу соответствует свое числовой код. Таблица кодов символов ASCII приведена в приложении Д. Например, символу `'a'` (a латинское) соответствует код 97, а символу `'b'`– код 98. При этом важно понимать, что в памяти хранится именно код, т. е. число, соответствующее данному символу.

Любым целочисленным переменным можно присваивать, как сам символ, так и его код.

Допустим, в программе объявлена переменная `ch` типа `char`. Тогда следующие записи будут эквивалентны:

```
ch='a';   ch=97;
```

Т. е. присваивая переменной `c` символ `'a'`, мы, по сути, записываем в переменную `ch` его код (в данном случае 97).

Для вывода символа на экран необходимо использовать спецификатор `%c`, если же нужно вывести его код – спецификатор `%d`.

В таблице представлены результаты вывода на экран:

Таблица 11.1 – **Результаты использования функции printf() со спецификаторами %c и %d**

Оператор	Результаты вывода на экран
`printf("%c", 'a');`	*a*
`printf("%c", 97);`	*a*
`printf("%d", 'a');`	*97*
`printf("%d", 97);`	*97*

В следующей программе на экран выводится строчные символы латинского алфавита и их коды.

Листинг 11.1 – Программа вывода кодов символов на экран

```
#include <stdio.h>
int main(void){
  char ch;
  printf("Таблица кодов\n");
  for (ch='a'; ch<='z'; ch++)
     printf("%c - %d\n", ch, ch);
}
```

В программе используется переменная ch типа char. В цикле переменной ch присваивается значение кода символа 'a'(т.е. 97). Цикл будет выполняться до тех пор, пока значение кодов символов не достигнет 'z'. На каждой итерации цикла происходит вывод на экран символа и его кода, а также увеличение кода текущего символа на 1 (переход к следующему символу).

Фрагмент вывода данной программы на экран приведен ниже:

```
Таблица кодов
a - 97
b - 98
c – 99
……..
y - 121
z - 122
```

В листинге 11.2 показана программа, выводящая все символы с кодами от 0 до 255. Необходимо обратить внимание, что в данном случае вместо типа char используется тип int.

Л и с т и н г 11.2 – Программа вывода кодов символов на экран

```
#include <stdio.h>
int main(void){
  int ch;
  printf("Таблица кодов\n");
  for (ch=0; ch<=255; ch++)
     printf("%c - %d\n", ch, ch);
}
```

Что же произойдет, если в данной программе изменить тип переменной ch с int на unsigned char? Т. к. значения переменной типа unsigned char не могут быть более 255, и при прибавлении 1 к 255 произойдет переполнение (значение переменной станет равным 0), то условие выполнения цикла ch<=255 всегда будет равно "истина". Следовательно, цикл станет бесконечным и программа "зависнет".

К символам допустимы все операции, которые применимы к целым числам. По сути, операции производятся над их кодами. Например, вполне допустима следующая запись:

```
ch='a'+2;
```

В данном случае, результатом станет символ, код которого на 2 больше, чем код символа 'a', т.е. символ 'c'.

Хотя для вывода символа на экран можно использовать функцию printf() со спецификатором %c, лучше это делать с помощью специальной функции putchar(). Она преобразует код символа, хранящейся в переменной ch, в символ, который отображается на экране.

Функция вывода на экран символа имеет следующий формат:

putchar(int ch).

Она записывает символ, содержащийся в младшем байте параметра ch, в стандартный поток вывода stdout (обычно вывод на экран).

Пример записи функции putchar() показан ниже:

Таблица 11.2 – **Результаты вывода на экран функции putchar()**

Выполняемые операторы	Вывод на экран	Пояснения
`putchar(99);`	*c*	Выводит на экран символ с кодом 99.
`putchar('s');`	*s*	Выводит на экран символ 's'
`char ch='r';` `putchar(ch);`	*r*	Выводит на экран значение переменной ch в символьном виде
`char ch=100;` `putchar(ch);`	*d*	Выводит на экран значение переменной ch в символьном виде

Ввести символ с клавиатуры можно с помощью функции scanf() со спецификатором %c либо специальной функцией getchar().

Функция getchar() предназначена для чтения одного символа и имеет следующий вид

```
int getchar()
```

При чтении символа предполагается, что символ имеет тип unsigned char, который потом преобразуется в целый.

Примеры ввода символов записаны ниже:

Таблица 11.3 – **Ввод символа ch с клавиатуры**

Выполняемые операторы	Пояснения
`scanf("%c",&ch);`	Чтение символа с клавиатуры и запись его в переменную ch
`ch=getchar();`	

Ниже представлена программа чтения символа с клавиатуры и вывода его на экран.

Листинг 11.3 – Программа вывода введенного символа на экран

```
#include <stdio.h>
int main(void){
  char ch;
  printf("Введите символ ");
  ch = getchar();
  printf("\nВведенный символ ");
  putchar(ch);
}
```

Рассмотрим программу, в которой вводится символ и переводится в верхний регистр. В языке С в библиотеке ctype.h имеется функция toupper(), которая переводит букву в верхний регистр.

Листинг 11.4 – Программа вывода введенного символа на экран

```
#include <stdio.h>
#include <ctype.h>
int main(void){
  char ch1,ch2;
  printf("Введите символ ");
  ch1 = getchar();
  ch2=toupper(ch1);
  printf("Введенный символ ");
  putchar(ch1);
  printf("Символ в верхнем регистре ");
  putchar(ch2);
}
```

Другие функции для обработки символов, находящиеся в библиотеке `ctype.h`, приведены в приложении В.

11.2 Строки

В языке С нет специальных типов переменных, предназначенных для хранения строк. Строки в С реализуются, как массивы символов. На каждую букву отводится один элемент массива. Объявление может выглядеть следующим образом:

```
char a[12];
```

В данном массиве можно хранить строки разной длины. Признаком окончания строки служит нулевой символ. Он обозначается как '\0'. Пример расположения строки показан на рисунке 11.1.

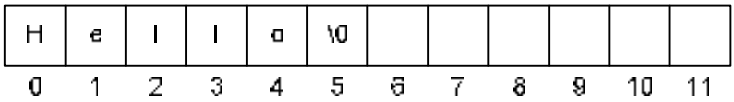


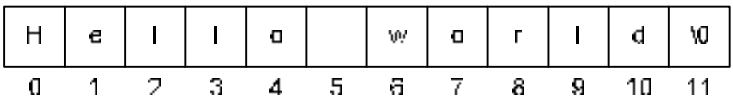
Рисунок 11.1 – Расположение в массиве строки `"Hello"`

В данном случае в символьном массиве записана строка `"Hello"`. Каждая буква записана в свой элемент массива. Последним стоит символ окончания строки `'\0'`. Все элементы, которые находятся после этого символа (в нашем случае с элемента с индексом 6), обычно игнорируются.

Для обозначения строк в программе служат двойные кавычки. Например:

```
"Hello world"
```

В памяти данная строка будет храниться, как показано на рисунке 11.2



Необходимо обратить внимание, что в строке символов последним всегда является символ '\0'. Поэтому для строки нужно резервировать на один элемент больше. Т. е., для строки "Hello world" (содержит 11 символов) массив должен быть объявлен с количеством элементов не менее 12.

Т. к. строка символов является массивом, возможен произвольный доступ к любому элементу массива.

В С при объявлении массива можно использовать знак присваивания для записи в массив какой-либо строки.

Например:

```
char s[12]="Hello";//допустимая запись
```

После этого в массив записываются все данные символы, последним автоматически записывается символ '\0'.

Однако для символьных массивов недопустимо присваивание строки с помощью знака = в другом месте программы.

Например, следующий код недопустим:

```
char s[12];
s="Hello";// недопустимая запись
```

Для того чтобы записать в строковый массив какое-либо значение можно воспользоваться функцией strcpy(), прототип которой находится в библиотеке string.h.

Данная функция имеет следующий формат:

```
strcpy(s1,s2);
```

Эта функция копирует строку s2 в строку s1.

Т.е. для того чтобы поместить в массив s строку "Hello" необходимо записать данную функцию следующим образом:

```
strcpy(s,"Hello");
```

Для того, чтобы определить длину строки, т.е. количество символов записанных до символа '\0', необходимо воспользоваться функцией strlen() (библиотека <string.h>). Данная функция имеет следующий формат:

```
strlen(s1)
```

Она возвращает длину строки, записанной в массиве s1. Символ '\0' при этом не учитывается.

Например:

```
int n;
char s[10]="Hello";
n=strlen(s);
```

После выполнения данного кода переменной n будет присвоено значение 5.

Для вывода на экран строки можно использовать функцию `printf()` со спецификатором `%s`:

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

Кроме этого, существует стандартная функция для вывода строк `puts()`. Она имеет следующий формат:

```
puts(char *s1);
```

где `s1` – символьный массив, выводимый на экран.

Особенностью данной функции является то, что последний символ `'\0'` преобразуется в символ `'\n'`, т.е. после вывода строки происходит переход на новую строку.

Пример вывода строки `s` показан ниже:

```
puts(s);
```

11.3 Буферизация символов.

11.3.1 Общие сведения

Важно отметить, что в стандарте С присутствуют только буферизированные функции ввода-вывода. Это значит, что все данные, которые вводятся или выводятся, сначала записываются в буфер, а потом только считываются функцией или устройством.

Существует 2 вида буферизации:

1 Построчная

2 Полная

При построчной буферизации буфер заполняется до появления символа `'\n'`.

При полной – заполнение происходит до тех пор, пока буфер не заполнится полностью.

11.3.2 Буфер ввода.

Стандартные функции в С работают не непосредственно с устройством, а через буфер.

Как только в программе встречается функция ввода, происходит обращение к буферу ввода.

Если буфер пуст, обращение передается устройству ввода (напр. клавиатура). При вводе с клавиатуры данные сначала записываются в буфер.

Далее происходит ожидание либо до полного заполнения буфера (при полной буферизации), либо до появления символа конца строки `'\n'` (при построчной буферизации).

После этого передается управление функции ввода, которая считывает данные из буфера.

Если после считывания какие-то данные в буфере остались, они будут считаны следующей функцией ввода.

Функции ввода с клавиатуры обычно работают с построчной буферизацией.

Л и с т и н г 11.4 – Программа, демонстрирующая принцип буферизации ввода

```
#include <stdio.h>
int main(void){
  int ch;
  while((ch=getchar())!='*')
     putchar(ch);
  return 0;
}
```

В данной программе функция `getchar()` пытается считать данные из буфера стандартного потока `stdin`.

Если буфер пуст, а так происходит в начале работы программы, функция `getchar()` ожидает заполнение буфера. Т.к. буферизация в данном случае является построчной, то происходит ожидание ввода символа `'\n'` (нажатие клавиши Enter).

После этого функция `getchar()` начинает считывать из буфера символы. При этом в условии цикла переменная `ch` проверяется на равенство символу `'*'`. Если равенства нет, то символ, хранящийся в переменной `ch`, выводится на экран. Последним обязательно является символ `'\n'` (вводится при нажатии клавиши Enter), который также выводится в виде перехода на новую строку.

Т. к. после этого буфер становится пустым (все символы считаны), то функция `getchar()` снова ожидает заполнения буфера до символа `'\n'`.

Если при чтении из буфера встретится символ `'*'` цикл прекращает выполнение и завершается выполнение программы.

В языке С определен макрос `EOF`. Обычно его значение принимается равным -1, но в некоторых компиляторах может быть и другое значение. Значение `EOF` возвращается функцией ввода при чтении из файла, когда достигнут конец файла или произошла ошибка чтения.

Рассмотрим следующий код:

```
while((ch=getchar())!=EOF)
   putchar(ch);
```

При перенаправлении ввода функция `getchar()` считывает из файла по одному символу. И так будет до тех пор, пока функция не возвратит значение равное `EOF`. Значение `EOF` будет означать, что конец файла достигнут, и цикл прекратится.

Измененная программа будет выглядеть так, как показано в листинге 11.5

Л и с т и н г 11.5 – Программа, демонстрирующая использование макроса EOF

```
#include <stdio.h>
int main(void){
  int ch;
  while((ch=getchar())!=EOF)
     putchar(ch);
  return 0;
}
```

При вводе с клавиатуры значение макроса `EOF` может быть сгенерировано нажатием комбинации клавиш Ctrl+d для ОС Unix/Linux или Ctrl+z для DOS/Windows.

Рассмотрим программу, в которой подсчитывается количество цифр, прописных букв и строк в файле. Для подсчета количества цифр используется функция `isdigit()`, которая возвращает значение "истина", если обрабатываемый символ является цифрой. Для подсчета количества прописных букв используется функция `isupper()`, которая возвращает значение "истина", если обрабатываемый символ является прописной буквой.

Для подсчета количества строк необходимо подсчитать количество символов `'\n'`. Количество строк будет на 1 больше, чем символов `'\n'`. В переменной `n1` подсчитывается количество цифр, в `n2` – количество заглавных букв, а в `n3` – количество строк.

Л и с т и н г 11.6 – Программа, подсчитывающая количество цифр, прописных букв и количество строк

```
#include <stdio.h>
#include <ctype.h>
int main(void){
  int ch, n1, n2, n3;
  n1=n2=n3=0;
  while((ch=getchar())!=EOF){
     putchar(ch);
     if (isdigit(ch)) n1++;
     if (isupper(ch)) n2++;
     if (ch=='\n') n3++;
  }
  printf("Количество цифр равно %d\n",n1);
  printf("Количество строчных букв равно %d\n",n2);
  printf("Количество строк в файле %d\n",n3+1);
  return 0;
}
```

Например, в файле *os.txt* содержится следующий текст:

```
FreeBSD Unix 6
```

```
Red Hat Linux 9
DOS 6.22
Microsoft Windows XP
```

Необходимо выполнить перенаправление ввода. Если программа называется *program*, тогда в командной строке необходимо записать:

```
program < os.txt
```

После этого программа выдаст на экран результаты подсчета:

```
Количество цифр равно 5
Количество строчных букв равно 15
Количество строк в файле 4
```

11.3.3 Ввод строки

Ввод строки или символьного массива можно производить несколькими способами:

1 С помощью функции `scanf()`.

2 С помощью функции `gets()`.

3 С помощью функции `fgets()`.

4 С помощью цикла и функции `getchar()`.

1. Спецификатор `%s` используется для ввода и вывода строк. Код, который выводит строку с помощью функции `scanf()` можно записать следующим образом:

```
char a[30];
scanf("%s",a);
```

Т. к. имя массива будет являться указателем на первый элемент, то знак `'&'` перед a не ставится. Чтение символов происходит до первого символа-разделителя, т. е. символа новой строки (`'\n'`), пробела или знака табуляции (`'\t'`). Последним записывается нулевой символ (`'\0'`), как признак конца строки.

Если необходимо ограничить количество вводимых символов в спецификацию преобразования нужно добавить число считываемых символов, как показано ниже:

```
char a[30];
scanf("%29s",a);
```

В данном случае в символьный массив a записывается 29 символов, если раньше не встретится символ-разделитель. Тридцатым (в элемент `a[29]`) записывается нулевой символ `'\0'`. Если символ-разделитель встретится раньше, то будут записаны в массив только символы до первого символа-разделителя. Последним запишется нулевой символ `'\0'`.

Недостатком ввода строки символов с помощью строки `scanf()` является невозможность записать в строку пробелы.

Например, если попытаться ввести строку "Привет всем" в массив запишется только "Привет".

2. Функция gets() служит для ввода из стандартного потока stdin символов и помещает их в массив символов, адресуемый указателем str.

```
char *gets (char *str)
```

Последний символ '\n' при этом заменяется на символ '\0'.

Фрагмент ввода символьного массива с помощью данной функции будет выглядеть следующим образом:

```
char a[30];
gets(a);
```

В символьный массив a записываются все символы до символа '\n', при этом символ '\n' преобразуется в символ '\0'.

Преимуществом данного способа является возможность ввода пробелов и знаков табуляции. Т. е. возможно записать целое предложение, в то время, как с помощью команды scanf() – только слово.

Недостатком является невозможность ограничить число считываемых символов, т.е. возможно переполнение массива.

3. Функция fgets() имеет следующий формат.

```
char *fgets(char *str, int num, FILE *stream);
```

Функция fgets() читает из входного потока stream не более num-1 символов и помещает их в массив символов, адресуемый указателем str. Символы читаются до тех пор, пока не будет прочитан символ новой строки или значение EOF, либо пока не будет достигнут заданный предел.

Функция fgets() обычно служит для ввода строки из файла. Однако ее можно применять для ввода строки с клавиатуры, если задать в качестве имени файла стандартный поток ввода stdin.

Особенностью данной функции является то, что можно задавать максимальное количество символов, вводимых в строку. Поэтому переполнение массива невозможно. Однако данная функция не заменяет последний вводимый символ '\n' на '\0', что делают другие функции. Поэтому замену производят дополнительно.

Пример кода ввода показан ниже.

```
char a[30];
int i;
fgets(a,30,stdin);
i=0;
while(a[i]!='\n') i++;
a[i]='\0';
```

В данном коде вводится строка с помощью fgets(). Далее с помощью цикла находится символ '\n', после чего он заменяется на '\0'.

4. Можно вводить посимвольно массив в цикле с помощью функции `getchar()`.

Этот способ более гибкий, чем предыдущие два, хотя и более сложный.

Допустим, необходимо ограничить количество введенных символов. Тогда программный код будет выглядеть следующим образом:

```
int i;
char a[30];
for(i=0; i<=28; i++)
  a[i]=getchar();
a[29]='\0';
```

В цикле с помощью функции `getchar()` считываются 29 символов и присваиваются элементам массива с индексами от 0 до 28. Переменная `i` играет роль счетчика. В последний элемент `a[29]` записывается нулевой символ. В данном способе невозможен выход за границы массива, однако необходимо ввести не менее 28 символов, причем вводиться будут и символы новой строки `'\n'`.

Для того чтобы ввод строки прекращался по нажатию клавиши Enter (т.е. ввода символа `'\n'`), необходимо изменить наш код:

```
int i;
char a[30];
for(i=0; (a[i]=getchar())!='\n'; i++)
  continue;
a[i]='\0';
```

Символ, считываемый с помощью функции `getchar()`, присваивается элементу `a[i]` и одновременно проверяется на неравенство `'\n'`. Если встретится символ новой строки `'\n'`, цикл прекратится и вместо символа `'\n'` в элемент `a[i]` запишется нулевой символ `'\0'`. По сути, данный способ работает так же, как функция `gets()`, и имеет тот же недостаток – возможность выхода за пределы массива.

Если скомбинировать вышеописанные 2 способа, то данная часть программы будет выглядеть следующим образом:

```
int i;
char a[30];
for(i=0; (i<=28)&&((a[i]=getchar())!='\n'); i++)
  continue;
a[i]='\0';
```

Цикл `for` будет работать только в том случае, если одновременно будут выполняться 2 условия: индекс вводимого элемента массива меньше или равен 28 и не был введен символ новой строки `'\n'`. В данном случае невозможен выход за пределы символьного массива и ввод будет прекращаться нажатием клавиши Enter.

11.3.4 Очистка буфера.

При заполнении буфера стандартного потока `stdin` могут оказаться лишние символы, которые не все будут считаны функциями ввода.

Л и с т и н г 11.7 – Программа демонстрирующая дейстие буфера ввода

```
#include <stdio.h>
int main(void){
  int a,b;
  printf("Введите a: ");
  scanf("%d",&a);
  printf("a=%d\n",a);
  printf("Введите b: ");
  scanf("%d",&b);
  printf("b=%d",b);
  return 0;
}
```

В следующем программном коде сначала должно вводиться одно целое число. Оно записывается в переменную `a` и выводится на экран функцией `printf()`. После этого вводится второе число и записывается в переменную `b`, а затем тоже выводится на экран.

Рассмотрим случай, когда пользователь введет через пробел сразу два числа. При этом два числа записываются в буфер. В переменную `a` считывается первое число, а в буфере остается второе. После вывода значения переменной `a` из буфера сразу считывается второе число, записывается в переменную `b` и выводится на экран. Т. е. не происходит ожидание ввода второго числа.

Для того, чтобы этого не происходило, перед вводом переменной `b` необходимо очистить буфер. При этом используется то свойство буфера, что последним всегда будет символ `'\n'`.

Поэтому для очистки буфера необходимо просто считать все символы из буфера до `'\n'` включительно.

Это можно сделать с помощью следующего цикла:

```
while (getchar()!='\n')
  continue;
```

В условии цикла считывается символ из буфера. Он никуда не записывается, а просто проверяется на равенство `'\n'`. Так происходит до тех пор, пока не будет достигнут последний символ `'\n'`, который считывается и цикл прекращается. После этого буфер должен быть пустым.

Л и с т и н г 11.8 – Измененная программа.

```
#include <stdio.h>
int main(void){
  int a,b;
  printf("Введите a: ");
```

```
  scanf("%d",&a);
  printf("a=%d\n",a);
  while (getchar()!='\n')
     continue;
  printf("Введите b: ");
  scanf("%d",&b);
  printf("b=%d",b);
  return 0;
}
```

В данной программе из буфера считывается первое число и записывается в переменную a. После этого буфер очищается и программа готова к вводу нового числа, которое запишется в переменную b.

11.4 Поиск подстроки в строке.

Для поиска подстроки в строке служит функция strstr(). Она возвращает указатель на первое вхождение подстроки, адресуемой параметром str2, в строку, адресуемую параметром str1. Если совпадения не обнаружено, то возвращается нулевой указатель.

Ниже приведен листинг данной программы.

Л и с т и н г 11.9 – программа поиска подстроки в строке

```
#include <stdio.h>
#include <string.h>
int main(void){
  char s1[40],s2[40];
  puts("Введите строку:");
  gets(s1);
  puts("Введите подстроку:");
  gets(s2);
  if (strstr(s1,s2)!=NULL)
     printf("В строке \"%s\" присутСтвует подстрока\
       \"%s\"\n",s1,s2);
  else
     printf("В строке \"%s\" подстрока \"%s\" не\
       найдена\n",s1,s2);
  return 0;
}
```

В программе с помощью функции gets(), вводятся строка s1 и подстрока s2. Необходимо проверить, присутствует ли подстрока s2 в строке s1.

Возвращаемое значение функции strstr(s1,s2) проверяется на неравенство нулевому указателю. Значение данной функции не равно нулевому указателю, когда подстрока s2 содержится в строке s1. Поэтому в при-

веденном примере на экран выводится сообщение о том, что в строке `s1` присутствует подстрока `s2`. Иначе (в случае равенства значения функции нулевому указателю), выводится сообщение о том, что подстрока `s2` не содержится в строке `s1`.

В приложении Г указаны другие функции для работы со строками.

ПРИЛОЖЕНИЕ А.

Основные операции языка C/C++

В таблице приведен список основных операций в соответствии с их приоритетами (по убыванию приоритетов, операции с разными приоритетами разделены чертой).

Операция	Краткое описание
Унарные операции (действуют на один операнд)	
`++`	Увеличение на 1 (пробелы между символами операции не допускаются)
`--`	Уменьшение на 1 (пробелы между символами операции не допускаются)
`sizeof`	Размер
`~`	Поразрядное отрицание
`!`	Логическое отрицание
`-`	Унарный минус
`+`	Унарный плюс
`&`	Взятие адреса
`*`	Разадресация (разыменование)
`(type)`	Преобразование типа
Бинарные (действуют на два операнда) и тернарная (действует на три операнда) операции	
`*`	Умножение
`/`	Деление
`%`	Остаток от деления
`+`	Сложение
`-`	Вычитание
`<<`	Сдвиг влево
`>>`	Сдвиг вправо
`<`	Меньше
`<=`	Меньше или равно
`>`	Больше
`>=`	Больше или равно
`==`	Равно
`!=`	Не равно
`&`	Поразрядная конъюнкция ("И")
`^`	Поразрядное исключающее ИЛИ
`|`	Поразрядная дизъюнкция ("ИЛИ")
`&&`	Логическое "И"
`||`	Логическое "ИЛИ"

?:	Условная операция (тернарная)
= *= /= %= += –= <<= >>= &= |= ^=	Присваивание Умножение с присваиванием Деление с присваиванием Остаток от деления с присваиванием Сложение с присваиванием Вычитание с присваиванием Сдвиг влево с присваиванием Сдвиг вправо с присваиванием Поразрядное "И" с присваиванием Поразрядное "ИЛИ" с присваиванием Поразрядное исключающее "ИЛИ" с приваиванием
,	Последовательное вычисление

ПРИЛОЖЕНИЕ Б.

Математическая библиотека: math.h

Основные математические функции языка С:

Прототип функции	Описание функции
`double acos(double x);`	Возвращает угол (от 0 до Pi радиан), косинус которого равен x.
`double asin(double x);`	Возвращает угол (от –Pi/2 до Pi/2 радиан), синус которого равен x.
`double atan(double x);`	Возвращает угол (от –Pi/2 до Pi/2 радиан), тангенс которого равен x.
`double cos(double x);`	Возвращает косинус x (x выражен в радианах).
`double sin(double x);`	Возвращает синус x (x выражен в радианах).
`double tan(double x);`	Возвращает тангенс x (x выражен в радианах).
`double exp(double x);`	Возвращает значение экспоненты с аргументом x (e^x).
`double log(double x);`	Возвращает натуральный логарифм числа x (ln(x)).
`double log10(double x);`	Возвращает десятичный логарифм числа x (lg(x)).
`double pow(double x, double y);`	Возводит x в степень y.

`double fabs(double x);`	Возвращает модуль числа x.
`double sqrt(double x);`	Возвращает квадратный корень из числа x.

Некоторые константы, определяемые в заголовочном файле math.h:
M_PI – Число "Пи" (3.1415926…)
M_E – Экспонента (2.7182818…)

ПРИЛОЖЕНИЕ В

Обработка символов: ctype.h

Функции этой библиотеки берут аргументы типа int, представимые в форме unsigned char или в виде EOF; результат подстановки величин другого типа не определен. В таблице “true” используется в качестве обозначения термина “ненулевое значение”.

Прототип	Описание
`int isalnum(int c);`	Возвращает true, если с является буквой или числом.
`int isalpha(int c);`	Возвращает true, если с является буквой.
`int isblank(int c);`	Возвращает true, если с является пробелом или знаком горизонтальной табуляции (C99).
`int iscntrl(int c);`	Возвращает true, если с является управляющим символом, например, Ctrl+B.
`int isdigit(int c);`	Возвращает true, если с является цифрой.
`int isgraph(int c);`	Возвращает true, если с является печатным символом, но не пробелом.
`int islower(int c);`	Возвращает true, если с является символом нижнего регистра.
`int isprint(int c);`	Возвращает true, если с является печатным символом.
`int ispunct(int c);`	Возвращает true, если с является знаком пунктуации (любым печатным символом, но не пробелом, буквой или цифрой).
`int isspace(int c);`	Возвращает true, если с является “невидимым “ символом: пробел, новая строка, возврат каретки, горизонт. отступ и т.д.
`int isupper(int c);`	Возвращает true, если с является символом верхнего регистра.
`int isxdigit(int c);`	Возвращает true, если с является символом, представленным в шестнадцатеричной форме.
`int tolower(int c);`	Если аргумент является символом верхнего регистра, возвращает соответсвующий символ нижнего регистра; иначе возвращает исходный аргумент.
`int toupper(int c);`	Если аргумент является символом нижнего регистра, возвращает соответсвующий символ верхнего регистра; иначе возвращает исходный аргумент.

ПРИЛОЖЕНИЕ Г

Библиотека строковых функций: string.h

Прототип	Описание
`int strlen (const char *s);`	Возвращает число символов (исключая завершающие пробелы) в строке s
`char *strcpy (char *restrict s1, const char *restrict s2);`	Копирует строку, указанную указателем s2 (включая пустой символ), на место, указанное указателем s1; возвращает s1.
`char *strstr(const char *s1, const char *s2);`	Возвращает указатель на положение первого появления последовательности символов из s2 в строке s1 (исключая завершающие пробелы); возвращает NULL, если совпадений не найдено.
`int strcmp(const char *s1, const char *s2);`	Сравнивает строки, указанные указателями s1 и s2; две строки идентичны, если совпадают все пары; иначе строки сравнивают по первой несовпадающей паре; символы сравниваются с помощью значений кодов символов; функция возвращает нуль, если строки одинаковы; значение, которое меньше нуля, если первая строка меньше второй; и значение, превышающее нуль, если первая строка больше второй.
`char *strchr(const char *s, int c);`	Ищет первое появление c (преобразованного в char) в строке, указанной указателем s; пустой символ является частью строки; возвращает указатель на первое появление c или NULL, если ничего не найдено.
`char *strcpy(char *restrict s1, const char *restrict s2);`	Копирует строку, указанную указателем s2 (включая пустой символ), на место, указанное указателем s1; возвращает s1.
`char *strcat(char *restrict s1, const char *restrict s2);`	Дополняет копию строки, указанную указателем s2 (включая пробелы), в размещение, указанное указателем s1; первый символ строки s2 переписывает пустой символ строки s1; возвращает s1.

ПРИЛОЖЕНИЕ Д

Кодировка символов ASCII (от 33 до 126)

Для получения кода символа необходимо сложить индексы, находящиеся в заголовках столбца и строки.

Код	0	1	2	3	4	5	6	7	8	9
30				!	"	#	$	%	&	'
40	(	)	*	+		-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Например, необходимо найти код символа `'a'`.

Данный символ находится на пересечении столбца с индексом 7 и строки с индексом 90. Значит код символа `'a'` будет равен 90+7=97.