

Раздел I. Основы алгоритмизации

1 - 2 **Типы алгоритмов.** Понятие алгоритма. Форма записи алгоритма. Типы структур в алгоритмах. Линейные алгоритмы. Разветвляющиеся алгоритмы. Циклические алгоритмы. Схемы алгоритмов.

Раздел II. Структурное программирование

2 – 10. **Введение в язык Си. Структура программы.** Алфавит языка. Идентификаторы. Ключевые слова. Знаки операций. Константы. Комментарии.

Типы данных. Директивы препроцессора. Функция `main()`.

3. – 22 **Базовые операции.** Базовые операции

4. – 30 **Функции ввода-вывода. Простые операторы. Математические функции.** Ввод-вывод символов. Форматированный ввод-вывод.

5. – 38 **Структурные операторы.** Классификация операторов, условные операторы.

6. – 48 **Структурные операторы.** Операторы цикла, операторы безусловного перехода.

7. – 60 **Структурированные типы данных. Массивы.** Понятие массива. Одномерные массивы. Строковый литерал. Чтение и запись строк. Двумерные массивы. Массивы строк.

8. – 69 **Указатели. Динамическое выделение памяти.** Понятие указателя. Указательные переменные.

9. – 78 **Функции пользователя.** Объявление и определение функций.

10. – 85 **Структуры, объединения, перечисления.** Понятие структуры. Доступ к полям структуры. Понятия объединения и перечисления. Битовые поля.

11. – 92 **Файлы.**

Раздел I. Основы алгоритмизации

1.1. Типы алгоритмов.

Алгоритм - конечная последовательность точно определенных действий, приводящих к однозначному решению поставленной задачи. Главная особенность любого алгоритма - *формальное исполнение*, позволяющее выполнять заданные действия (команды) не только человеку, но и различным техническим устройствам (исполнителям). Процесс составления алгоритма называется *алгоритмизацией*.

Свойства алгоритмов

Дискретность – значения новых величин (данных) вычисляются по определенным правилам из других величин с уже известными значениями.

Определенность (детерминированность) – каждое правило из набора однозначно, а сами данные однозначно связаны между собой, т.е. последовательность действий алгоритма строго и точно определена.

Результативность (конечность) – алгоритм решает поставленную задачу за конечное число шагов.

Массовость – алгоритм разрабатывается так, чтобы его можно было применить для целого класса задач, например, алгоритм вычисления определенных интегралов с заданной точностью.

Способы описания алгоритмов

Алгоритмы могут быть заданы: словесно, таблично, графически (с помощью блок-схем). *Словесное* задание описывает алгоритм с помощью слов и предложений. *Табличное* задание служит для представления алгоритма в форме таблиц и расчетных формул. *Графическое* задание, или *блок-схема*, - способ представления алгоритма с помощью геометрических фигур, называемых *блоками*. Последовательность блоков и соединительных линий образуют блок-схему. Описание алгоритмов с помощью блок-схем является наиболее наглядным и распространенным способом задания алгоритмов. Блок-схемы располагаются сверху вниз. Линии соединения отдельных блоков показывают направление процесса обработки в схеме. Каждое такое направление называется ветвью. Алгоритм независимо от его структуры всегда имеет по одному блоку «Начало» и «Конец». Его

ветви должны в конце сойтись, и по какой бы ветви не было бы начато движение, оно всегда должно привести к блоку «Конец».

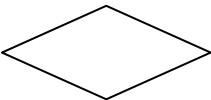
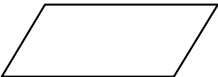
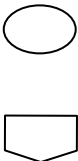
При задании алгоритма с помощью блок-схемы используются строго определенные блоки. Основные типы блоков приведены в таблице 1. Следует отметить, что все блоки нумеруются. В этом случае номера проставляются вверху слева от блока (блоки «Начало», «Конец» и соединительные блоки не нумеруются). Стрелки на соединяющих линиях обычно не ставят при направлении сверху вниз и слева направо; если направление противоположное, то его показывают стрелкой на линии. отметить, что все блоки нумеруются. В этом случае номера проставляются вверху слева от блока (блоки «Начало», «Конец» и соединительные блоки не нумеруются). Стрелки на соединяющих линиях обычно не ставят при направлении сверху вниз и слева направо; если направление противоположное, то его показывают стрелкой на линии.

Графическое описание алгоритма

Графическое изображение алгоритма – это представление его в виде схемы, состоящей из последовательности блоков (геометрических фигур), каждый из которых отображает содержание очередного шага алгоритма. А внутри фигур кратко записывают действие, выполняемое в этом блоке. Такую схему называют блок-схемой или структурной схемой алгоритма, или просто схемой алгоритма.

Правила изображения фигур сведены в единую систему программной документации (дата введения последнего стандарта ГОСТ 19.701.90 – 01.01.1992).

По данному ГОСТу графическое изображение алгоритма – это схема данных, которая отображает путь данных при решении задачи и определяет этапы их обработки.

Наименование	Обозначение	Пояснение
Пуск – останов		Начало, конец алгоритма, останов, вход, выход в подпрограмму
Процесс		Вычислительная операция или группа операций
Решение		Разветвление в алгоритме, проверка условий
Предопределенный процесс		Программа, стандартная подпрограмма
Ввод-вывод		Ввод-вывод в общем виде
Документ		Вывод результатов на бумагу
Дисплей		Ввод-вывод данных на дисплей
Линии потока		Соединительные линии между блоками алгоритмов
Соединители		Разрыв линий потока на странице, на разных страницах

Типы алгоритмов

Алгоритмы бывают *линейные, разветвляющиеся и циклические.*

Линейный алгоритм не содержит логических условий, имеет одну ветвь обработки и изображается линейной последовательностью связанных друг с другом блоков. Условное изображение линейного алгоритма может быть представлено на рис. 1.1

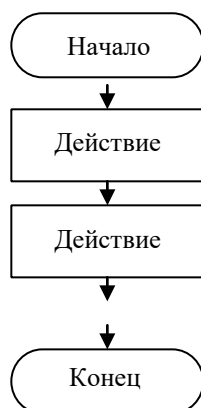


Рис. 1.1. Условное изображение линейного алгоритма.

Пример простейшего линейного процесса

Наиболее часто в практике программирования требуется организовать расчет некоторого арифметического выражения при различных исходных данных. Например, такого:

$$z = \frac{tg^2 x}{\sqrt{x^2 + m^2}} + x^{(m+1)} \sqrt{x^2 + m^2}$$

где $x > 0$ – вещественное, m – целое.

Разработка алгоритма обычно начинается с составления схемы. Продумывается оптимальная последовательность вычислений, при которой, например, отсутствуют повторения. При написании алгоритма рекомендуется переменным присваивать те же имена, которые фигурируют в заданном арифметическом выражении либо иллюстрируют их смысл.

Для того чтобы не было «длинных» операторов, исходное выражение полезно разбить на ряд более простых. В нашей задаче предлагается схема вычислений, представленная на рис. 1.2.

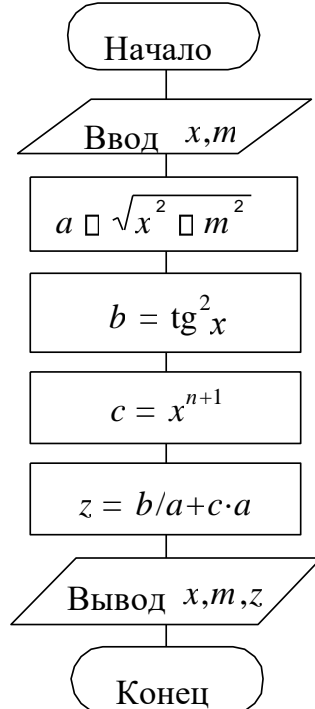
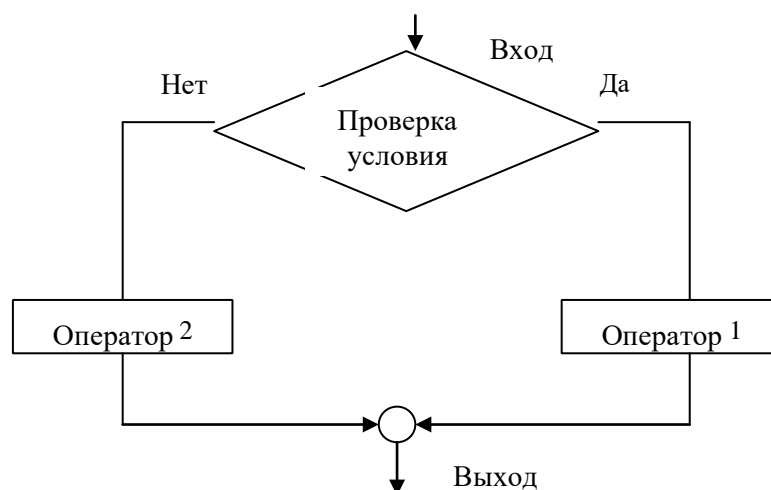


Рис. 1.2. Схема линейного процесса

Она содержит ввод и вывод исходных данных, линейный вычислительный процесс, вывод полученного результата. Заметим, что выражение $\sqrt{x^2 + m^2}$ вычисляется только один раз. Введя дополнительные переменные a , b , c , мы разбили сложное выражение на ряд более простых.

Разветвляющийся алгоритм содержит одно или несколько логических условий и имеет несколько ветвей обработки.

Условное изображение разветвления представлено на рис. 1.3. Структура РАЗВЕТВЛЕНИЕ предусматривает проверку условия, после которого вычислительный процесс развивается по одной из двух ветвей (в зависимости от ответа на поставленный в условии вопрос). Каждый из путей (ветвей) ведет к общему выходу.



Пример разветвляющегося процесса

Построить алгоритм вычисления значения функции y , заданной формулой

$$y = \begin{cases} x^2 + 2, & \text{если } x \leq 1 \\ x^2 - 2x + 4, & \text{если } x > 1 \end{cases}$$

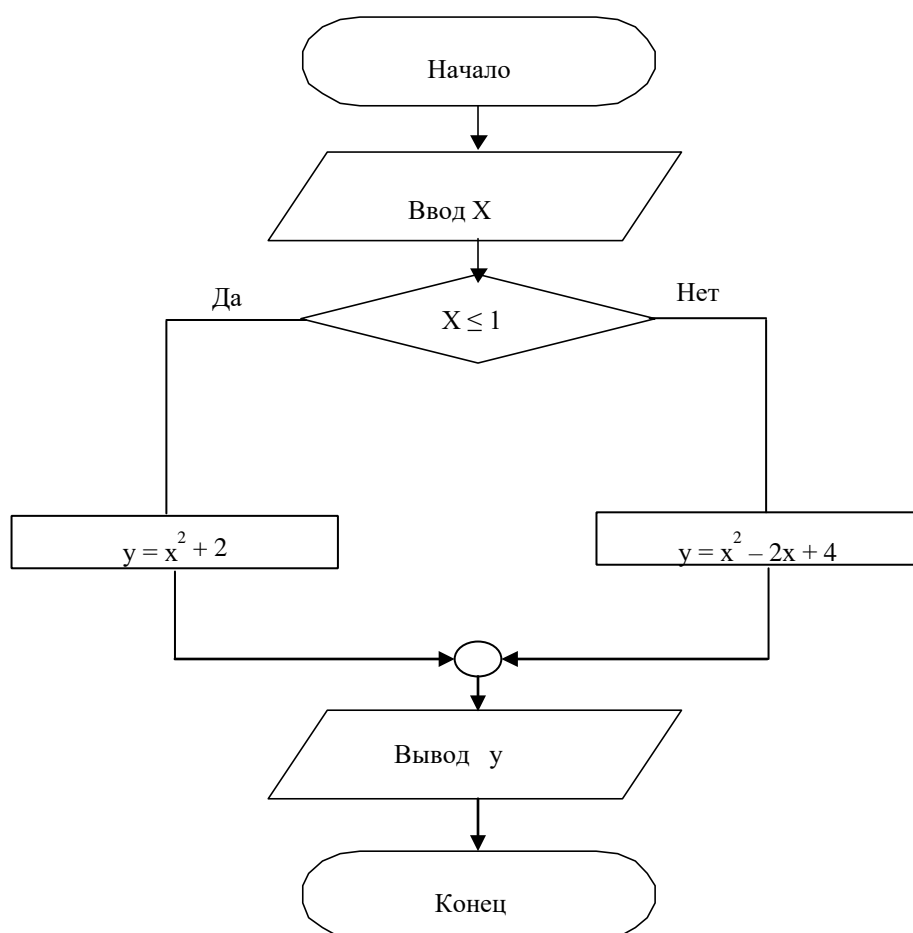


Рис. 1.4. Блок-схема примера

Если $x > 1$, то y вычисляется по формуле $y = x^2 - 2x + 4$. Блок-схема алгоритма решения задачи приведена на рис. 1.4.

Циклический алгоритм содержит один или несколько циклов – многократно повторяемых частей алгоритма. Цикл, не содержащий внутри себя других циклов, называют простым. Если он содержит внутри себя другие циклы или разветвления, то цикл называют сложным или вложенным. Любой цикл характеризуется одной или несколькими переменными, называемыми параметрами цикла, от анализа значений которых зависит выполнение цикла. *Параметр цикла* – переменная, принимающая при каждом вхождении в цикл

новое значение. Условное изображение циклического алгоритма представлено на рис. 1.5.

Базовую структуру Цикл с предусловием можно использовать для описания циклического процесса при решении любой задачи, т.к. она предусматривает возможность обхода этого цикла в случае невыполнения условия при первой проверке условия. Базовую структуру Цикл с постусловием можно использовать только в тех случаях, когда из условия задачи следует, что хотя бы один раз цикл обязательно должен выполняться.

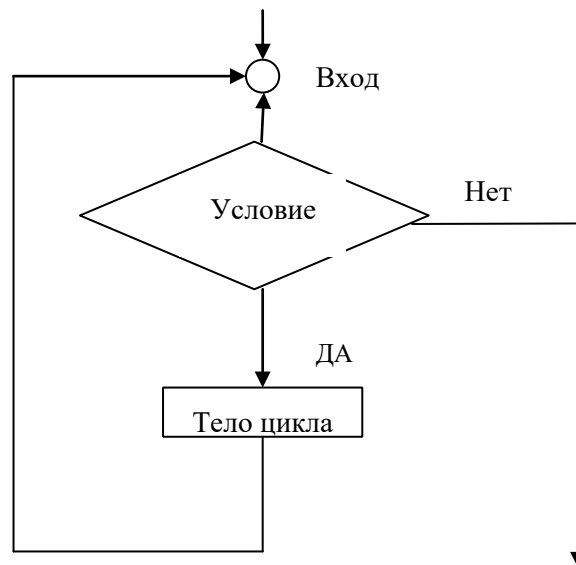


Рис. 1.5 Условное изображение циклического алгоритма

Пример циклического процесса

Вычислить значение функции $y = \sin x$, представленной в виде разложения в ряд, с заданной точностью, т.е. до тех пор, пока разность между соседними слагаемыми не станет меньше заданной точности:

$$y = \sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Схема алгоритма, приведенная на рис. 1.7, реализует циклический процесс, в состав которого (в блоке проверки $|E| < eps$) входит участок разветвления.

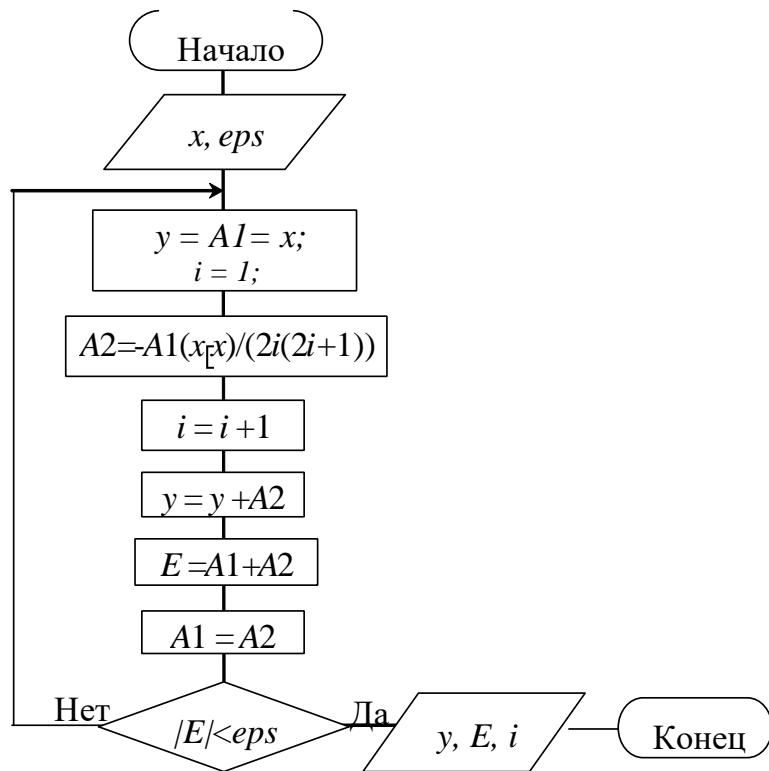


Рис. 1.6. Схема циклического алгоритма

Раздел II. Структурное программирование

2.1. Введение в язык Си. Структура программы.

Язык программирования Си Введение в Си

Язык Си был создан в 1972 г. сотрудником фирмы Bell Laboratories в США Денисом Ритчи.

По замыслу автора, язык Си должен был обладать противоречивыми свойствами. С одной стороны, это язык программирования высокого уровня, поддерживающий методику структурного программирования (подобно Паскалю). С другой стороны, этот язык должен обеспечивать возможность создавать такие системные программы, как компиляторы и операционные системы. До появления Си подобные программы писались исключительно на языках низкого уровня – Ассемблерах, Автокодах. Первым системным программным продуктом, разработанным с помощью Си, стала операционная система UNIX. Из-за упомянутой выше двойственности свойств нередко в литературе язык Си называют языком среднего уровня. Стандарт Си был утвержден в 1983 г. Американским национальным институтом стандартов (ANSI) и получил название ANSI C.

Этапы работы с программой на Си в системе программирования

На рисунке 1 прямоугольниками отображены системные программы, а блоки с овальной формой обозначают файлы на входе и на выходе этих программ.

1. С помощью *текстового редактора* формируется текст программы и сохраняется в файле с расширением `crr`. Пусть, например, это будет файл с именем `example. crr`.

2. Осуществляется этап *препроцессорной обработки*, содержание которого определяется директивами препроцессора, расположенными перед заголовком программы (функции). В частности, по директиве `#include` препроцессор подключает к тексту программы заголовочные файлы (`*.h`) стандартных библиотек.

3. Происходит *компиляция текста* программы на Си. В ходе компиляции могут быть обнаружены синтаксические ошибки, которые должен исправить программист. В результате успешной компиляции получается объектный код программы в файле с расширением `obj`. Например, `example.obj`.

4. Выполняется этап *компоновки* с помощью системной программы Компоновщик (Linker). Этот этап еще называют *редактированием связей*. На данном этапе к программе подключаются библиотечные функции. В результате компоновки создается исполняемая программа в файле с расширением `exe`. Например, `example.exe`.

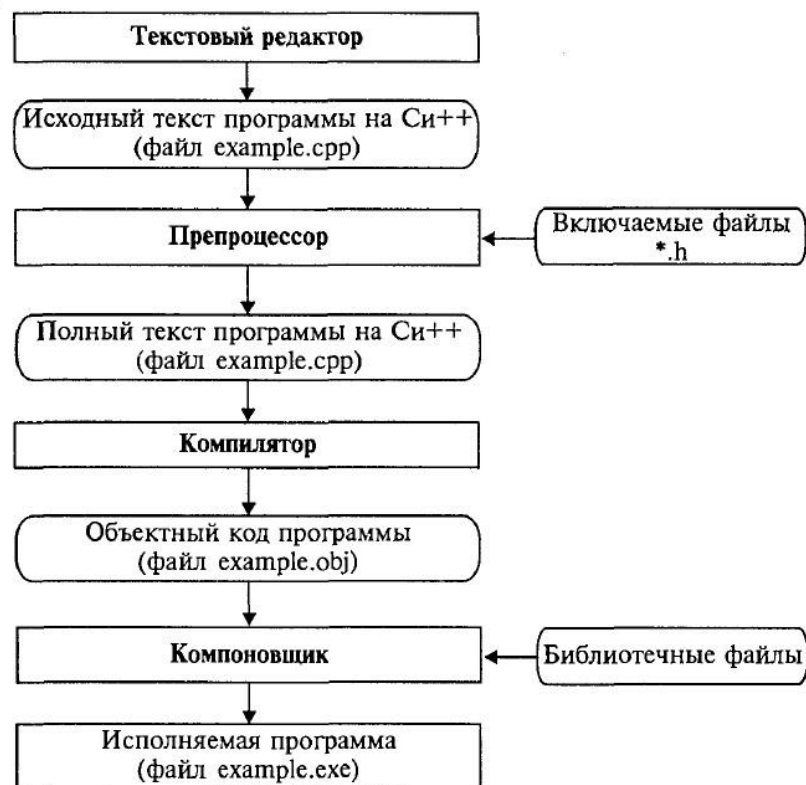


Рисунок 1 – Этапы работы с программой на Си

Базовые элементы языка Си

Алфавит

В алфавит языка Си входят:

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

В комментариях, строках и символьных константах могут использоваться и другие знаки (например, русские буквы).

Комбинации некоторых символов, не разделенных пробелами, интерпретируются как один значимый символ. К ним относятся:

++ -- == && || << >> >= <= += -= *= /= ?: /* */ //

В Си в качестве ограничителей комментариев могут использоваться как пары символов /* и */, так и символы //. Признаком конца такого комментария является невидимый символ перехода на новую строку.

Примеры:

/* Это комментарий, допустимый в Си */

//Это строчный комментарий, используемый в Си

Из символов алфавита формируются лексемы – единицы текста программы, которые при компиляции воспринимаются как единое целое и не могут быть разделены на более мелкие элементы. К лексемам относятся идентификаторы, служебные слова, константы, знаки операций, разделители.

Идентификаторы

Последовательность латинских букв, цифр, символов подчеркивания (), начинающаяся с буквы или символа подчеркивания, является идентификатором. Например:

B12 rus hard_RAM_disk MAX ris_32

В отличие от Паскаля в Си различаются прописные и строчные буквы. Это значит, что, например, flag, FLAG, Flag, FlAg – разные идентификаторы.

Ограничения на длину идентификатора могут различаться в разных реализациях языка. Компиляторы фирмы Borland позволяют использовать до 32 первых символов имени. В некоторых других реализациях допускаются идентификаторы длиной не более 8 символов.

Служебные (ключевые) слова

Как и в Паскале, служебные слова в Си – это идентификаторы, назначение которых однозначно определено в языке. Они не могут быть использованы как свободно выбираемые имена. Полный список

служебных слов зависит от реализации языка, т. е. различается для разных компиляторов. Однако существует неизменное ядро, которое определено стандартом Си. Вот его список:

asm	friend	static
auto	goto	struct
break	if	switch
case	inline	template
catch	int	this
char	long	throw
class	new	try
const	operator	typedef
continue	private	typeid
default	protected	union
delete	public	unsigned
do	register	virtual
double	return	void
else	short	volatile
enum	signed	while
extern	sizeof	static
float		

Типы данных

Концепция типов данных является важнейшей стороной любого языка программирования. Схема типов данных для языка Си представлена на рисунке 2.



Рисунок 2 – Типы данных

В Си имеется четыре базовых арифметических (числовых) типа данных. Из них два целочисленных – *char*, *int* – и два плавающих (вещественных) – *float* и *double*. Кроме того, в программах можно использовать некоторые модификации этих типов, описываемых с помощью служебных слов – модификаторов. Существуют два модификатора размера – *short* (короткий) и *long* (длинный) – и два модификатора знаков – *signed* (знаковый) и *unsigned* (беззнаковый). Знаковые модификаторы применяются только к целым типам.

Тип величины связан с ее формой внутреннего представления, множеством принимаемых значений и множеством операций, применимых к этой величине. В таблице 1 перечислены арифметические типы данных Си, указан объем занимаемой памяти и диапазон допустимых значений. Размер типа *int* и *unsigned int* зависит от размера слова операционной системы, в которой работает компилятор Си. В 16-разрядных ОС этим типам соответствуют 2 байта, в 32-разрядных – 4 байта.

Таблица 1 – Арифметические типы данных языка Си

Тип данных	Размер (байт)	Диапазон значений	Эквивалентные названия типа
<i>char</i>	1	−128...+127	<i>signed char</i>
<i>int</i>	2/4	зависит от системы	<i>signed</i> , <i>signed int</i>
<i>unsigned char</i>	1	0...255	нет
<i>unsigned int</i>	2/4	зависит от системы	<i>unsigned</i>
<i>short int</i>	2	−32768...32767	<i>short</i> , <i>signed short int</i>
<i>unsigned short</i>	2	0...65535	<i>unsigned short int</i>
<i>long int</i>	4	−2147483648...2147483647	<i>long</i> , <i>signed long int</i>
<i>unsigned long int</i>	4	0...4294967295	<i>unsigned long</i>
<i>float</i>	4	±(3.4E−38...3.4E+38)	нет
<i>double</i>	8	±(1.7E−308...1.7E+308)	нет
<i>long double</i>	10	±(3.4E−4932...1.1E+4932)	нет

- если не указан базовый тип, то по умолчанию подразумевается *int*;
- если не указан модификатор знаков, то по умолчанию подразумевается *signed*;
- с базовым типом *float* модификаторы не употребляются;

- модификатор `short` применим только к базовому типу `int`.

В Си величины типа `char` могут рассматриваться в программе и как символы, и как целые числа. Все зависит от способа использования этой величины.

Среди базовых типов нет логического типа данных. Между тем в Си используются логические операции и логические выражения. В качестве логических величин в Си выступают целые числа. Интерпретация их значений в логические величины происходит по правилу: *равно нулю – ложь, не равно нулю – истина*.

Описание переменных

Описание переменных в программах на Си имеет вид:

имя_типа список_переменных;

Примеры описаний:

char symbol,ee;

unsigned char code;

int number, row;

unsigned long long_number;

float x, X, cc3;

double e, b4;

long double max_num;

Одновременно с описанием можно задать начальные значения переменных. Такое действие называется *инициализацией переменных*. Описание с инициализацией производится по следующей схеме:

тип имя_переменной = начальное_значение;

Например:

float pi=3.14159,c=1.23;

unsigned int year=2000;

Константы

Целые константы

Целые десятичные числа, начинающиеся не с нуля, например: 4, 356, -128.

Целые восьмеричные числа, запись которых начинается с нуля, например: 016, 077.

Целые шестнадцатеричные числа, запись которых начинается с символов 0x, например: 0x1A, 0x253, 0xFFFF.

Тип константы компилятор определяет по следующим правилам: если значение константы лежит в диапазоне типа *int*, то она получает тип *int*; в противном случае проверяется, лежит ли константа в диапазоне типа *unsigned int*, в случае положительного ответа она получает этот тип; если не подходит и он, то пробуются тип *long* и, наконец, *unsigned long*. Если значение числа не укладывается в диапазон типа *unsigned long*, то возникает ошибка компиляции.

Вещественные константы

Если в записи числовой константы присутствует десятичная точка (2.5) или экспоненциальное расширение (1E-8), то компилятор рассматривает ее как вещественное число и ставит ей в соответствие тип *double*. Примеры вещественных констант: 4 4 . 3.14159 44E0 1.5E-4.

Символьные и строковые константы

Символьные константы заключаются в апострофы. Например: 'A', 'a', '5', '+'. Строковые константы, представляющие собой символьные последовательности, заключаются в двойные кавычки. Например: "rezult", "введите исходные данные".

Особую разновидность символьных констант представляют так называемые *управляющие символы*. Их назначение – управление выводом на экран. Как известно, такие символы расположены в начальной части кодовой таблицы ASCII (коды от 0 до 31) и не имеют графического представления. В программе на Си они изображаются парой символов, первый из которых '\'.
Вот некоторые из управляющих символов:

'\n' – переход на новую строку;

'\t' – горизонтальная табуляция;

'\a' – подача звукового сигнала.

Именованные константы

В программе на Си могут использоваться именованные константы. Употребляемое для их определения служебное слово `const` принято называть *квалификатором доступа*. Квалификатор `const` указывает на то, что данная величина не может изменяться в течение всего времени работы программы. . Примеры описания константных переменных:

```
const float pi=3.14159;
```

```
const int iMIN=1, iMAX=1000;
```

Еще одной возможностью ввести именованную константу является использование препроцессорной директивы `#def ine` в следующем формате:

```
#define <имя константы> <значение константы>
```

Например:

```
#define iMIN 1
```

```
#define iMAX 1000
```

Тип констант явно не указывается и определяется по форме записи. В конце директивы не ставится точка с запятой.

На стадии препроцессорной обработки указанные имена заменяются на соответствующие значения.

Использование комментариев в тексте программы

Комментарий - это набор символов, которые игнорируются компилятором, на этот набор символов, однако, накладываются следующие ограничения. Внутри набора символов, который представляет комментарий не может быть специальных символов, определяющих начало и конец комментариев, соответственно (`/*` и `*/`). Отметим, что комментарии могут заменить как одну строку, так и несколько. Например:

```
/* комментарии к программе */
```

```
/* начало алгоритма */
```

или

```
/* комментарии можно записать в следующем виде, однако надо  
быть осторожным, чтобы внутри последовательности, которая  
игнорируется компилятором, не попались операторы программы,  
которые также будут игнорироваться */
```

Неправильное определение комментариев.

/* комментарии к алгоритму */ решение краевой задачи */ */

ИЛИ

/* комментарии к алгоритму решения */ краевой задачи */

Структура программы

Программа, написанная на языке С, состоит из директив препроцессора, объявлений глобальных переменных, одной или нескольких функций, среди которых одна главная (main) функция управляет работой всей программы.

Общая структура программы на языке С имеет вид:

<директивы препроцессора>

<определение типов пользователя – typedef>

<прототипы функций>

<определение глобальных объектов>

<функции>

Функции, в свою очередь, имеют структуру:

<класс_памяти> <тип> <имя функции> (<объявление
параметров>)

{ - начало функции

<определение локальных объектов>

<операции и операторы>

} - конец функции

Среди функций обязательно присутствует **главная функция** с именем `main`. Простейшая программа содержит только главную функцию и имеет следующую структуру:

директивы препроцессора

```
void main()
```

{ определения объектов;

исполняемые операторы;

}

Перед компиляцией программы на языке С автоматически выполняется предварительная (препроцессорная) обработка текста программы. С помощью директив препроцессора задаются необходимые действия по преобразованию текста программы перед компиляцией.

Директивы записываются по следующим правилам:

- все препроцессорные директивы должны начинаться с символа #;
- все директивы начинаются с первой позиции;
- сразу за символом # должно следовать наименование директивы, указывающее текущую операцию препроцессора.

Наиболее распространены директивы **#include** и **#define**.

Директива **#include** используется для подключения к программе заголовочных файлов (обычных текстов) с декларацией стандартных библиотечных функций. При заключении имени файла в угловые скобки < > поиск данного файла производится в стандартной директории с этими файлами. Если же имя файла заключено в двойные кавычки " ", то поиск данного файла осуществляется в текущем директории.

Например:

#include <stdio.h> - подключение файла с объявлением стандартных функций файлового ввода-вывода;

#include <conio.h> - функции работы с консолью;

#include <graphics.h> - графические функции;

#include <math.h> - математические функции.

Директива **#define** (определить) создает макроконстанту и ее действие распространяется на весь файл.

Например: #define PI 3.1415927

В ходе препроцессорной обработки программы идентификатор PI заменяется значением 3,1415927.

Библиотечные заголовочные файлы ANSI Си[править | править исходный текст]<assert.h> Содержит макрос утверждений, используемый для обнаружения логических и некоторых других типов ошибок в отлаживаемой версии программы.

<locale.h> Для setlocale() и связанных констант.
Используется для выбора соответствующего языка.
<math.h> Для вычисления основных математических функций.
<stdio.h> Реализует основные возможности ввода и вывода в языке Си. Этот файл содержит весьма важную функцию printf.

<stdlib.h> Для выполнения множества операций, включая конвертацию, генерацию псевдослучайных чисел, выделение памяти, контроль процессов, окружения, сигналов, поиска и сортировки.

<string.h> Для работы с различными видами строк.

<math.h> Для типовых математических функций.
(Появилось в C99)

<time.h> Для конвертации между различными форматами времени и даты.

Пример программы:

```
#include <stdio.h>
#include <conio.h>          /* Директивы препроцессора
*/
#define PI 3.1415927
void main()                // Заголовок главной
функции
{                          // Начало функции
int num;                  // Декларирование
                          // переменной num
num=13 ;                 // Операция присваивания
clrscr();                // Очистка экрана
printf(" \n  Число pi=%7.5f\n %d - это опасное число
\n", PI, num);
}                          // Конец функции
```

В первых двух строках программы указаны директивы препроцессора #include, по которым происходит подключение заголовочных файлов, содержащих декларации функций ввода-вывода (stdio.h) для функции printf() и работы с консолью (conio.h) для функции clrscr(). Следующая директива создает макроконстанту PI и подставляет вместо ее имени значение 3,1415927 по всему тексту программы. В главной функции main декларируется переменная целого типа num. Далее этой переменной присваивается значение 13. Функция printf() выводит на экран строки:

Число pi =3.1415927

13 – это опасное число

Как видно, функция представляет собой набор операций и операторов, каждый из которых оканчивается символом ; (точка с запятой).

В тексте программы использованы комментарии между парой
СИМВОЛОВ
/* */ и после пары символов //.

2.2. Базовые операции.

Выражения и операции

Во всех языках программирования под выражением подразумевается конструкция, составленная из констант, переменных, знаков операций, функций, скобок. Выражение определяет порядок вычисления некоторого значения. Если это числовое значение, то такое выражение называют арифметическим. Вот несколько примеров арифметических выражений, записанных по правилам языка Си:

$a+b$ $12.5-z$ $2*(X+Y)$
 $x++$ $++b$ $--n*2$ $n*=1$

Три первых выражения имеют традиционную форму для языков программирования высокого уровня. Следующие четыре выражения специфичны для языка Си.

Операция, применяемая к одному операнду, называется *унарной*, а операция с двумя операндами – *бинарной*.

Арифметические операции

К арифметическим операциям относятся:

- вычитание или унарный минус;
- + сложение или унарный плюс;
- * умножение;
- / деление;
- % деление по модулю;
- ++ унарная операция увеличения на единицу (инкремент);
- унарная операция уменьшения на единицу (декремент).

Все операции, кроме деления по модулю, применимы к любым числовым типам данных. Операция % применима только к целым числам.

Рассмотрим особенности выполнения операции деления. Если делимое и делитель – целые числа, то и результат – целое число. Например, значение выражения $5/3$ будет равно 2, а при вычислении $1/5$ получится 0.

Если хотя бы один из операндов имеет вещественный тип, то и результат будет вещественным. Например, операции $5./3$, $5./3.$, $5/3$. дадут вещественный результат 1.6666.

Операции инкремента и декремента могут применяться только к переменным и не могут – к константам и выражениям. Операция ++ увеличивает значение переменной на единицу, операция -- уменьшает значение переменной на единицу. Оба знака операции могут записываться как перед операндом (префиксная форма), так и после операнда (постфиксная форма), например: ++x или x++, --a или a--. Три следующих оператора дают один и тот же результат:

$x=x+1$; ++x; x++ .

Различие проявляется при использовании префиксной и постфиксной форм в выражениях. Проиллюстрируем это на примерах. Первый пример:

$a=3$; $b=2$;

$c=a++*b++$;

В результате выполнения переменные получают следующие значения: $a=4$, $b=3$, $c=6$.

Второй пример: $a=3$; $b=2$; $c=++a*++b$;

Результаты будут такими: $a=4$, $b=3$, $c=12$.

Объяснение следующее: при использовании постфиксной формы операции ++ и -- выполняются после того, как значение переменной было использовано в выражении, а префиксные операции – до использования. Поэтому в первом примере значение переменной c вычислялось как произведение 3 на 2, а во втором – как произведение 4 на 3.

По убыванию старшинства арифметические операции расположены в следующем порядке:

++, --, - (унарный минус), *, /, %, +, -

Одинаковые по старшинству операции выполняются в порядке слева направо. Для изменения порядка выполнения операций в выражениях могут применяться круглые скобки.

Операции отношения

В Си используется следующий набор операций отношения:

< меньше,
 <= меньше или равно,
 > больше,
 >= больше или равно, равно,
 != не равно.

В стандарте Си нет логического типа данных. Поэтому результатом операции отношения является целое число: если отношение истинно – то 1, если ложно – то 0.

Примеры отношений:

$a < 0$, $101 \geq 105$, $'a' == 'A'$, $'a' != 'A'$

Результатом второго и третьего отношений будет 0 – ложь; результат четвертого отношения равен 1 – истина; результат первого отношения зависит от значения переменной a .

Логические операции

Три основные логические операции в языке Си:

! операция отрицания (**НЕ**),

&& конъюнкция, логическое умножение (**И**),

|| дизъюнкция, логическое сложение (**ИЛИ**).

Правила их выполнения определяются таблицей 2, где A, B – операнды логического выражения, принимающие значения Т – истина, F – ложь.

Таблица 2 - Таблица истинности

A	B	!A	A && B	A B
T	T	F	T	T
T	F	F	F	T
F	F	T	F	F
F	T	T	F	T

Например, логическое выражение, соответствующее двойному неравенству $0 < x < 1$ в программе на Си запишется в виде следующего логического выражения:

$$x > 0 \ \&\& \ x < 1$$

Здесь не понадобились круглые скобки для выделения операций отношения, так как в языке Си операции отношения старше конъюнкции и дизъюнкции. По убыванию приоритета логические операции и операции отношения расположены в следующем порядке:

!
> < >= <=
== !=
&&

II

Помимо рассмотренных в Си имеются *поразрядные* логические операции. Эти операции выполняются над каждой парой соответствующих двоичных разрядов внутреннего представления операндов. Их еще называют *битовыми* логическими операциями. Знаки битовых логических операций:

& поразрядная конъюнкция (**И**),
| поразрядная дизъюнкция (**ИЛИ**),
^ поразрядное исключающее **ИЛИ**,
~ поразрядное отрицание (**НЕ**).

Битовые логические операции вместе с операциями поразрядного сдвига влево (<<) и вправо (>>) позволяют добраться до каждого бита внутреннего кода. Чаще всего такие действия приходится выполнять в системных программах.

Операция присваивания

Присваивание в Си является операцией, а не оператором. Знак операции присваивания =. Присваивание, как любой другой знак операции, может несколько раз входить в выражение. Например:

a=b=c=x+y;

Присваивание имеет самый низкий приоритет (ниже только у операции «запятая»). Кроме того, операция присваивания — правоассоциативная. Это значит, что несколько подряд расположенных присваиваний выполняются справа налево. Поэтому в приведенном выше выражении первой выполнится операция

сложения, затем переменной с присвоится значение суммы, затем это значение присвоится переменной b и в конце – переменной a.

В языке Си имеются дополнительные операции присваивания, совмещающие присваивание с выполнением других операций. Среди них: +=, -=, /=, *=, %=. Приоритет у них такой же, как и у простого присваивания.

Примеры использования этих операций:

a+=2 эквивалентно a=a+2,

x-=a+b эквивалентно x=x-(a+b),

p/=10 эквивалентно p=p/10,

m*=n эквивалентно m=m*n,

г%=5 эквивалентно г=г%5.

Вместо выражения a=a+2 предпочтительнее писать в программе a+=2, поскольку второе выражение будет вычисляться быстрее.

Операция преобразования типа

Применение этой операции имеет следующий формат:

(имя_типа) операнд

Операндом могут быть константа, переменная, выражение. В результате значение операнда преобразуется к указанному типу.

Примеры использования преобразования типа:

(long)8, (float)1, (int)x%2

По поводу последнего выражения заметим, что приоритет операции «тип» выше деления (и других бинарных арифметических операций), поэтому сначала значение переменной x приведется к целому типу (отбросится дробная часть), а затем выполнится деление по модулю.

Следующий фрагмент программы иллюстрирует одну из практических ситуаций, в которой потребовалось использовать преобразование типа:

float c; int a=1, b=2; c=(float)a/b;

В результате переменная c получит значение 0,5. Без преобразования типа ее значение стало бы равно 0.

Операция sizeof

Эта операция имеет две формы записи:

sizeof(тип) и sizeof(выражение)

Результатом операции является целое число, равное количеству байтов, которое занимает в памяти величина явно указанного типа или величина, полученная в результате вычисления выражения. Ее приоритет выше, чем у бинарных арифметических операций, логических операций и отношений.

Примеры использования операции:

sizeof(int)	результат – 2
sizeof (1)	результат –2
sizeof (0.1)	результат –8
sizeof(char)	результат – 1
sizeof('a')	результат –2

Операция «запятая»

Эта необычная операция используется для связывания нескольких выражений в одно. Несколько выражений, разделенных запятыми, вычисляются последовательно слева направо. В качестве результата такого совмещенного выражения принимается значение самого правого выражения. Например, если переменная *x* имеет тип *int* , то значение выражения (*x*=3, 5**x*) будет равно 15.

Операция «условие ?:»

Это единственная операция, которая имеет три операнда. Формат операции:

выражение1 ? выражение2 : выражение3;

Данная операция реализует алгоритмическую структуру ветвления. Алгоритм ее выполнения следующий: первым вычисляется значение выражения 1, которое обычно представляет собой некоторое условие. Если оно истинно, т. е. не равно 0, то вычисляется выражение 2 и полученный результат становится результатом операции. В противном случае в качестве результата берется значение выражения 3.

Пример. Вычисление абсолютной величины переменной *X* можно организовать с помощью одной операции:

X<0 ? -X : X;

Пример. Выбор большего значения из двух переменных а и b:
 $\text{max}=(a \leq b)?b:a;$

Пример. Заменить большее значение из двух переменных а и b на единицу: $(a > b)?a:b=1;$

Правила языка в данном случае позволяют ставить условную операцию слева от знака присваивания.

В языке Си круглые () и квадратные скобки [] рассматриваются как операции, причем эти операции имеют наивысший приоритет.

Сведём в общую таблицу 3 операции языка Си, расположив их по рангам. Ранг операции – это порядковый номер в ряду приоритетов. Чем больше ранг, тем ниже приоритет.

Таблица 3 - Приоритеты (ранги) операций

Ранг	Операции
1	() [] -> •
2	!-+--+ – & * (тип) sizeof (унарные)
3	% (мультипликативные бинарные)
4	+ - (аддитивные бинарные)
5	(поразрядного сдвига)
6	< <= >= > (отношения)
7	= = ! = (отношения)
8	поразрядная конъюнкция «И»)
9	зрядная конъюнкция «И»)
10	зрядная дизъюнкция «ИЛИ»)
11	&& (конъюнкция «И»)
12	(дизъюнкция «ИЛИ»)
13	?: (условная)
14	= *= /= %= += -= &= ^= = << = >>=
15	, («запятая»)

2.3. Функции ввода-вывода. Простые операторы.

Математические функции.

Форматированный вывод

Оператор вызова функции `printf()` имеет следующую структуру:
`printf(форматная_строка, список_аргументов);`

Форматная строка ограничена двойными кавычками (т.е. является текстовой константой) и может включать в себя произвольный текст, управляющие символы и спецификаторы формата. Список аргументов может отсутствовать или же состоять из выражений, значения которых выводятся на экран (в частном случае из констант и переменных).

Оператор `printf("\n a=")` ; содержит текст ("a=") и управляющие символы ("\n"). Текст выводится на экран в том виде, в котором он записан. Управляющие символы влияют на расположение на экране выводимых знаков. В результате выполнения этого оператора на экран с новой строки выведутся символы a=.

Признаком управляющего символа является значок \. Ниже приводится их список:

- \n – перевод строки;
- \t – горизонтальная табуляция;
- \r – возврат курсора к началу новой строки;
- \a – сигнал-звонок;
- \b – возврат на один символ (одну позицию);
- \f – перевод (прогон) страницы;
- \v – вертикальная табуляция.

Оператор `printf("\n Площадь треугольника=%f", s);` содержит все виды параметров функции `printf()`. Список аргументов состоит из одной переменной `s`. Ее значение выводится на экран. Пара символов `% f` является спецификацией формата выводимого значения переменной `s`. Значок `%` – признак формата, а буква `f` указывает на то, что выводимое число имеет вещественный (плавающий) тип и выводится на экран в форме с фиксированной точкой. Например, если в результате вычислений переменная `s` получит значение 32,435621, то на экран выведется:

Площадь треугольника=32.435621

Спецификатор формата определяет форму внешнего представления выводимой величины. Вот некоторые спецификаторы формата: %с – символ;

%s – строка;

%d – целое десятичное число (тип int);

%u – целое десятичное число без знака (тип unsigned);

%f – вещественные числа в форме с фиксированной точкой;

%e – вещественные числа в форме с плавающей точкой (с мантиссой и порядком). Например, после выполнения следующих операторов

```
float m,p;
```

```
int k;
```

```
m=84.3; k=-12; p=32.15; printf("\n m=%f\t k=%d\t p=%e",m, k,p) ;
```

на экран выведется строка:

```
m=84.299999 k=-12 p=3.21500e+01
```

Здесь дважды используемый управляющий символ табуляции \t отделил друг от друга выводимые значения. Из этого примера видно, что соответствие между спецификаторами формата и элементами списка аргументов устанавливается в порядке их записи слева направо.

К спецификатору формата могут быть добавлены числовые параметры: ширина поля и точность. Ширина – это число позиций, отводимых на экране под величину, а точность – число позиций под дробную часть (после точки). Параметры записываются между значком % и символом формата и отделяются друг от друга точкой. Внесем изменения в оператор вывода для рассмотренного выше примера.

printf("\n m=%f.2f\tk=%5d\t p=%8.2e\t p=%11.4e",m,k,p,p); В результате на экране получим:

```
m=84.30 k= -12 p= 32.15 p= 3.2150e+01
```

Если в пределы указанной ширины поля выводимое значение не помещается, то этот параметр игнорируется и величина будет выводиться полностью.

К спецификаторам формата могут быть добавлены модификаторы в следующих вариантах:

%ld – вывод long int;

%hu – вывод short unsigned;

%Lf – вывод long double.

Форматированный ввод с клавиатуры

Оператор вызова функции scanf() имеет следующую структуру:

scanf(форматная_строка, список_аргументов);

Данная функция осуществляет чтение символов, вводимых с клавиатуры, и преобразование их во внутреннее представление в соответствии с типом величин. В функции scanf() форматная строка и список аргументов присутствуют обязательно. Например оператор:

scanf ("%f",&a);

Здесь "%f" - форматная строка; &a – список аргументов, состоящий из одного элемента. Этот оператор производит ввод числового значения в переменную a.

Символьную последовательность, вводимую с клавиатуры и воспринимаемую функцией scanf(), принято называть входным потоком. Функция scanf() разделяет этот поток на отдельные вводимые величины, интерпретирует их в соответствии с указанным типом и форматом и присваивает переменным, содержащимся в списке аргументов.

Список аргументов – это перечень вводимых переменных, причем перед именем каждой переменной ставится значок &. Это знак операции «взятие адреса переменной».

Форматная строка заключается в кавычки (как и для printf()) и состоит из списка спецификаций. Каждая спецификация имеет следующую структуру:

%[*] [ширина поля ввода] [модификатор] спецификатор

Из них обязательным элементом является % и спецификатор. Для ввода числовых данных используются следующие спецификаторы:

d – для целых десятичных чисел (тип int);

u – для целых десятичных чисел без знака (тип unsigned int);

f – для вещественных чисел (тип float) в форме с фиксированной точкой;

e – для вещественных чисел (тип float) в форме с плавающей точкой.

Звездочка в спецификации позволяет пропустить во входном потоке определенное количество символов. *Ширина поля* – целое

положительное число, позволяющее определить число символов из входного потока, принадлежащих значению соответствующей вводимой переменной. Как и в спецификациях вывода для функции `printf()`, в спецификациях ввода функции `scanf()` допустимо использование модификаторов `h`, `l`, `L`. Они применяются при вводе значений модифицированных типов:

`hd` – для ввода значений типа `short int`;

`ld` – для ввода значений типа `long int`;

`lf`, `le` – для ввода значений типа `double` в форме с фиксированной и плавающей точкой;

`Lf`, `Le` – для ввода значений типа `long double` в форме с фиксированной и плавающей точкой.

Все три величины `a`, `b`, `c` можно ввести одним оператором:

```
scanf("%f%f%f",&a,&b,&c);
```

Если последовательность ввода будет такой:

```
5 3.2 2.4 <Enter>
```

то переменные получают следующие значения: `a = 5,0`, `b = 3,2`, `c = 2,4`. Разделителем в потоке ввода между различными значениями может быть любое количество пробелов, а также другие пробельные символы: знак табуляции, конец строки. Только после нажатия на клавишу `Enter` вводимые значения присвоятся соответствующим переменным. До этого входной поток помещается в буфер клавиатуры и может редактироваться.

Функции `getchar ()`, `putchar ()`, `gets ()`, `puts()`

Функции `getchar()` и `putchar()` выполняют ввод и вывод символа соответственно.

Самым простым вводом данных является чтение по одному символу из стандартного файла ввода `stdin` (находится в заголовочном файле `stdio.h`) с помощью функции **`getchar ()`**.

Обращение: `c=getchar ()`;

присваивает переменной `c` очередной вводимый с клавиатуры символ.

Функция **`putchar(c)`** выдает значение «`c`» в стандартный файл вывода `stdout` (находится в заголовочном файле `stdio.h`).

Таким образом, чтобы ввести символ используется функция **getchar()**, а чтобы вывести его на экран – функция **putchar()**.

Пример. Программа выводит на экран все прописные латинские буквы. Последовательное прибавление к очередному значению **c** обеспечивает выбор очередной буквы ввиду их лексической упорядоченности.

```
#include <stdio.h>
#include <conio.h>
main()
{
    char c;
    c='A' ;
    while(c<='Z')
    {putchar(c);
    c=c+1;    }
    getch();}
```

Для работы не с одиночными символами, а массивами из символов - *строками* в С используется большое количество различных функций, наиболее распространенными из которых являются **gets()** и **puts()**.

Функция **gets (имя-строки)** читает символы из строки ввода до тех пор, пока не встретит символ новой строки „\n“, который создается нажатием клавиши <ввод>. К прочитанным символам (без“\n“) присоединяется нульсимвол „\0“ и полученное значение присваивается соответствующей переменной-строке. При обнаружении EOF функция **gets()** возвращает значение NULL.

Функция **puts (строка)** выводит символы до тех пор, пока не встретит символ „\0“, который заменяет символ новой строки „\n“.

Функция **cprintf()**, как и функция **printf()**, используется для вывода на экран сообщений и значений переменных, но при этом имеется возможность задать цвет выводимых символов и цвет фона. Заголовочный файл **conio.h**.

Функция **void textcolor(int цвет);** (заголовочный файл **conio.h**) задает цвет для выводимого функциями **cputs()** и **cprintf()** текста. В качестве параметра *цвет* обычно используют одну из перечисленных ниже именованных констант.

Таблица 5

Цвет	Константа	Значение константы
1	2	3
Черный	BLACK	0
Синий	BLUE	1
Зеленый	GREEN	2
Бирюзовый	CYAN	3
Красный	RED	4
Сиреневый	MAGENTA	5
Коричневый	BROWN	6
Светло-серый	LIGHTGRAY	7
Серый	DARKGRAY	8
Голубой	LIGHTBLUE	9
Светло-зеленый	LIGHTGREEN	10
Светлобирюзовый	LIGHTCYAN	11
Алый	LIGHTRED	12
Светлосиреневый	LIGHTMAGENTA	13
Желтый	YELLOW	14
Белый (яркий)	WHITE	15

Функция ***void textbackground(int цвет);*** (заголовочный файл `conio.h`) задает цвет фона, на котором появляется текст, выводимый функциями `cputs()` и `cprintf()`. В качестве параметра *цвет* обычно используют одну из перечисленных ниже именованных констант.

Таблица 6

Цвет	Константа	Значение константы
Черный	BLACK	0
Синий	BLUE	1
Зеленый	GREEN	2
Бирюзовый	CYAN	3
Красный	RED	4
Сиреневый	MAGENTA	5

Коричневый	BROWN	6
Светлосерый	LIGHTGRAY	7

Функция ***void gotoxy(int x, int y);*** (заголовочный файл `conio.h`) переводит курсор в позицию с указанными координатами. Координата *x* задает номер колонки, координата *y* – номер строки, на пересечении которых находится знакоместо, куда переводится курсор.

Функция ***void clrscr(void);*** (заголовочный файл `conio.h`) очищает экран и закрашивает его цветом, заданным функцией `textbackground()`.

Функция ***void window(int x1, int y1, int x2, int y2);*** (заголовочный файл `conio.h`) определяет окно – область экрана. параметры *x1*, *y1* задают координаты левого верхнего угла окна относительно экрана, параметры *x2*, *y2* – правого нижнего.

Математические функции

В языке C математические функции находятся в подключаемом заголовочном файле `math.h` и представлены в таблице 4.

Таблица 4— Математические функции (заголовочный файл `math.h`)

Обращение	Тип аргумента	Тип результата	Функция
<code>abs(x)</code>	<code>int</code>	<code>int</code>	абсолютное значение целого числа
<code>acos(x)</code>	<code>double</code>	<code>double</code>	арккосинус (радианы)
<code>asin(x)</code>	<code>double</code>	<code>double</code>	арксинус (радианы)
<code>atan(x)</code>	<code>double</code>	<code>double</code>	арктангенс (радианы)
<code>ceil(x)</code>	<code>double</code>	<code>double</code>	ближайшее целое, не меньшее <i>x</i>
<code>cos(x)</code>	<code>double</code>	<code>double</code>	косинус (<i>x</i> в радианах)
<code>exp(x)</code>	<code>double</code>	<code>double</code>	e^x — экспонента от <i>x</i>
<code>fabs(x)</code>	<code>double</code>	<code>double</code>	абсолютное значение вещественного <i>x</i>
<code>floor(x)</code>	<code>double</code>	<code>double</code>	наибольшее целое, не превышающее <i>x</i>
<code>fmod(x, y)</code>	<code>double</code> <code>double</code>	<code>double</code>	остаток от деления нацело <i>x</i> на <i>y</i>
<code>log(x)</code>	<code>double</code>	<code>double</code>	логарифм натуральный — $\ln x$
<code>log10(x)</code>	<code>double</code>	<code>double</code>	логарифм десятичный — $\lg x$
<code>pow(x, y)</code>	<code>double</code> <code>double</code>	<code>double</code>	<i>x</i> в степени <i>y</i> — x^y
<code>sin(x)</code>	<code>double</code>	<code>double</code>	синус (<i>x</i> в радианах)
<code>sinh(x)</code>	<code>double</code>	<code>double</code>	гиперболический синус
<code>sqrt(x)</code>	<code>double</code>	<code>double</code>	корень квадратный (положительное значение)
<code>tan(x)</code>	<code>double</code>	<code>double</code>	тангенс (<i>x</i> в радианах)
<code>tanh(x)</code>	<code>double</code>	<code>double</code>	гиперболический тангенс

2.4. Структурные операторы.

Операторы

Все операторы языка СИ могут быть условно разделены на следующие категории:

- условные операторы, к которым относятся оператор условия if и оператор выбора switch;
- операторы цикла (for, while, do while);
- операторы перехода (break, continue, return, goto); - другие операторы (оператор "выражение", пустой оператор).

Операторы в программе могут объединяться в составные операторы с помощью фигурных скобок. Любой оператор в программе может быть помечен меткой, состоящей из имени и следующего за ним двоеточия.

Все операторы языка СИ, кроме составных операторов, заканчиваются точкой с запятой ";".

Оператор выражение

Любое выражение, которое заканчивается точкой с запятой, является оператором.

Выполнение оператора выражение заключается в вычислении выражения. Полученное значение выражения никак не используется, поэтому, как правило, такие выражения вызывают побочные эффекты. Заметим, что вызвать функцию, не возвращающую значения можно только при помощи оператора выражения. Правила вычисления выражений были сформулированы выше.

Примеры:

```
++ i;
```

Этот оператор представляет выражение, которое увеличивает значение переменной i на единицу.

```
a=cos(b * 5);
```

Этот оператор представляет выражение, включающее в себя операции присваивания и вызова функции.

```
a(x,y);
```

Этот оператор представляет выражение состоящее из вызова функции.

Пустой оператор

Пустой оператор состоит только из точки с запятой. При выполнении этого оператора ничего не происходит. Он обычно используется в следующих случаях:

- в операторах `do`, `for`, `while`, `if` в строках, когда место оператора не требуется, но по синтаксису требуется хотя бы один оператор;
- при необходимости пометить фигурную скобку.

Синтаксис языка СИ требует, чтобы после метки обязательно следовал оператор. Фигурная же скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор.

Пример:

```
int main ( )
{
    :
    { if (...) goto a; /* переход на скобку */
      { ...
        }
    a;; }
    return 0;
}
```

Составной оператор

Составной оператор представляет собой несколько операторов и объявлений, заключенных в фигурные скобки:

```
{ [объявление]
  :
  оператор; [оператор];
  :
}
```

Заметим, что в конце составного оператора точка с запятой не ставится.

Выполнение составного оператора заключается в последовательном выполнении составляющих его операторов.

Пример:

```

int main ()
{
int   q,b;
double t,d;
    :
    if (...)
    {
        int   e,g;
        double f,q;
        :
    }
    :
    return (0);
}

```

Переменные e,g,f,q будут уничтожены после выполнения составного оператора. Отметим, что переменная q является локальной в составном операторе, т.е. она никоим образом не связана с переменной q объявленной в начале функции main с типом int. Отметим также, что выражение стоящее после return может быть заключено в круглые скобки, хотя наличие последних необязательно.

Оператор if

Формат оператора:

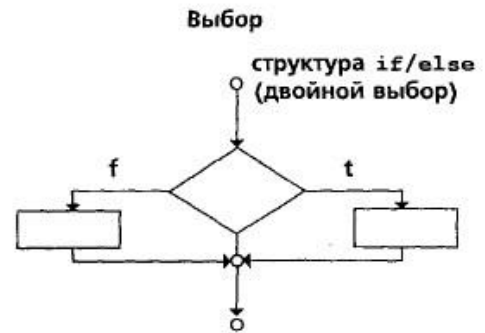
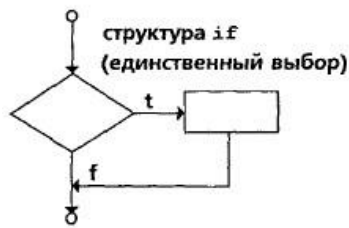
if (выражение) оператор-1; [else оператор-2;]

Выполнение оператора if начинается с вычисления выражения.

Далее выполнение осуществляется по следующей схеме:

- если выражение истинно (т.е. отлично от 0), то выполняется оператор-1.
- если выражение ложно (т.е. равно 0),то выполняется оператор-2.
- если выражение ложно и отсутствует оператор-2 (в квадратные скобки заключена необязательная конструкция), то выполняется следующий за if оператор.

После выполнения оператора if значение передается на следующий оператор программы, если последовательность выполнения операторов программы не будет принудительно нарушена использованием операторов перехода. if (выражение) оператор; if (выражение) оператор_1; else оператор_2;



Пример:

```
if (i < j)  i++;
else { j = i-3;  i++; }
```

Этот пример иллюстрирует также и тот факт, что на месте оператор-1, так же как и на месте оператор-2 могут находиться сложные конструкции.

Допускается использование вложенных операторов if. Оператор if может быть включен в конструкцию if или в конструкцию else другого оператора if. Чтобы сделать программу более читабельной, рекомендуется группировать операторы и конструкции во вложенных операторах if, используя фигурные скобки. Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово else с наиболее близким if, для которого нет else.

Примеры:

```
int main ( )
{
    int t=2, b=7, r=3;
    if (t>b)
    {
        if (b < r) r=b;
    }
    else r=t;
    return (0);
}
```

В результате выполнения этой программы r станет равным 2.

Если же в программе опустить фигурные скобки, стоящие после оператора if, то программа будет иметь следующий вид:

```

int main ( )
{
    int t=2,b=7,r=3;
    if ( a>b )
        if ( b < c ) t=b;
        else      r=t;
    return (0);
}

```

В этом случае r получит значение равное 3, так как ключевое слово else относится ко второму оператору if, который не выполняется, поскольку не выполняется условие, проверяемое в первом операторе if.

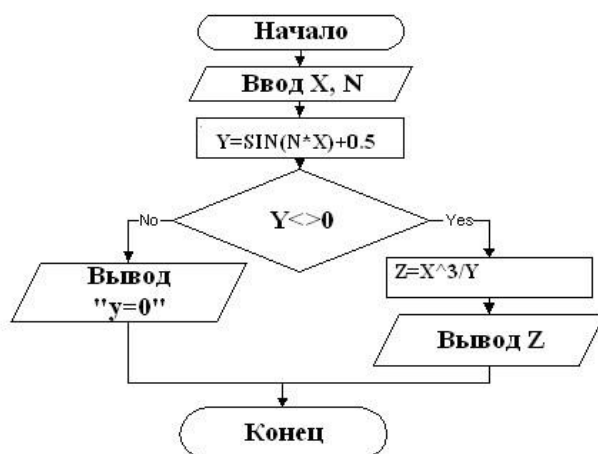
Следующий фрагмент иллюстрирует вложенные операторы if:

```

char ZNAC;
int x,y,z;
:
if (ZNAC == '-') x = y - z;
else if (ZNAC == '+') x = y + z;
    else if (ZNAC == '*') x = y * z;
        else if (ZNAC == '/') x = y / z;
            else ...

```

Пример. Составить программу для вычисления значения функции $z = \frac{x^3}{y}$, где $y = \sin(nx) + 0.5$.



Решение:

Рисунок 5 – Блок-схема алгоритма решения задачи из примера 1

Программа:

```

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

```

```

#include <math.h>
void main()
{
int n; float x,y,z;
scanf("%d",n);
scanf("%f",x);
y=sin(n*x)+0.5;
if (y!=0) {
z=pow(x,3)/y;
printf("z=%f",z);
}
else printf("y=0");
getch(); }

```

Из рассмотрения этого примера можно сделать вывод, что конструкции использующие вложенные операторы if, являются довольно громоздкими и не всегда достаточно надежными. Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора switch.

Оператор switch

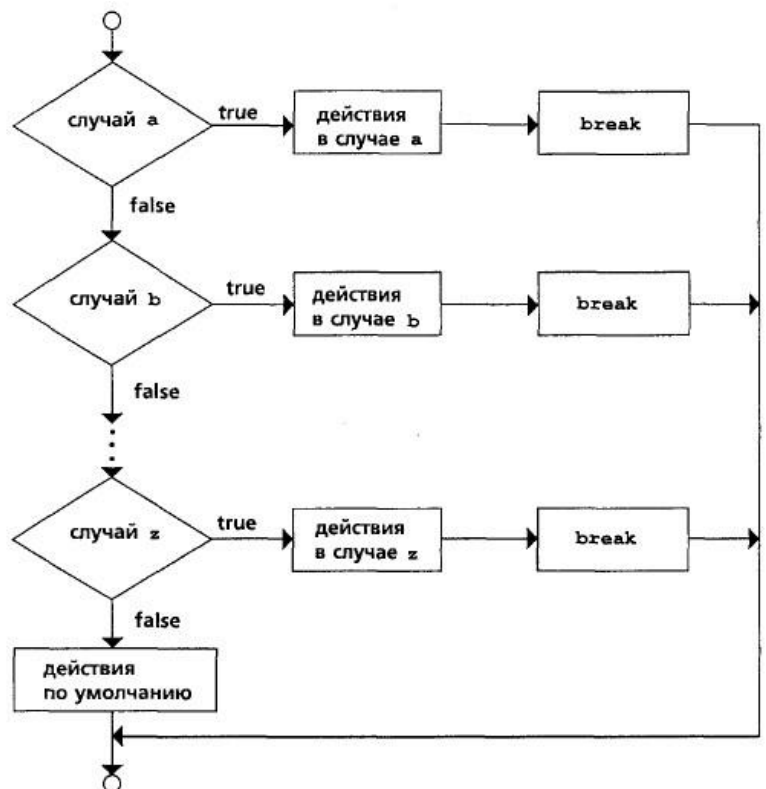


Рис.3. Множественный выбор

Оператор switch предназначен для организации выбора из множества различных вариантов. Формат оператора следующий:

```
switch ( выражение )  
{ [объявление]  
:  
[ case константное-выражение1]: [ список-операторов1]  
[ case константное-выражение2]: [ список-операторов2]  
:  
:  
[ default: [ список операторов ]]  
}
```

Схема выполнения оператора switch следующая:

- вычисляется выражение в круглых скобках;
- вычисленные значения последовательно сравниваются с константными выражениями, следующими за ключевыми словами case;
- если одно из константных выражений совпадает со значением выражения, то управление передается на оператор, помеченный соответствующим ключевым словом case;
- если ни одно из константных выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом default, а в случае его отсутствия управление передается на следующий после switch оператор.

Выражение, следующее за ключевым словом switch в круглых скобках, может быть любым выражением, допустимыми в языке СИ, значение которого должно быть целым. Отметим, что можно использовать явное приведение к целому типу, однако необходимо помнить о тех ограничениях и рекомендациях, о которых говорилось выше.

Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора switch состоит из нескольких операторов, помеченных ключевым словом case с последующим константным-выражением. Следует отметить, что использование целого константного выражения является существенным недостатком, присущим рассмотренному оператору.

Так как константное выражение вычисляется во время трансляции, оно не может содержать переменные или вызовы функций. Обычно в качестве константного выражения используются целые или символьные константы.

Все константные выражения в операторе switch должны быть уникальны. Кроме операторов, помеченных ключевым словом case, может быть фрагмент, помеченный ключевым словом default.

Список операторов может быть пустым, либо содержать один или более операторов. Причем в операторе switch не требуется заключать последовательность операторов в фигурные скобки.

Отметим также, что в операторе switch можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом case, однако в объявлениях не должна использоваться инициализация.

Отметим интересную особенность использования оператора switch: конструкция со словом default может быть не последней в теле оператора switch. Ключевые слова case и default в теле оператора switch существенны только при начальной проверке, когда определяется начальная точка выполнения тела оператора switch. Все операторы, между начальным оператором и концом тела, выполняются вне зависимости от ключевых слов, если только какой-то из операторов не передаст управления из тела оператора switch. Таким образом, программист должен сам позаботиться о выходе из case, если это необходимо. Чаще всего для этого используется оператор break.

Для того, чтобы выполнить одни и те же действия для различных значений выражения, можно пометить один и тот же оператор несколькими ключевыми словами case.

Пример:

```
int i=2;
switch (i)
{
    case 1: i += 2;
    case 2: i *= 3;
    case 0: i /= 2;
    case 4: i -= 5;
    default:    ;
}
```

```
}
```

Выполнение оператора switch начинается с оператора, помеченного case 2. Таким образом, переменная i получает значение, равное 6, далее выполняется оператор, помеченный ключевым словом case 0, а затем case 4, переменная i примет значение 3, а затем значение -2. Оператор, помеченный ключевым словом default, не изменяет значения переменной.

Рассмотрим ранее приведенный пример, в котором иллюстрировалось использование вложенных операторов if, переписанной теперь с использованием оператора switch.

```
char ZNAC;  
int x,y,z;  
switch (ZNAC)  
{  
    case '+': x = y + z; break;  
    case '-': x = y - z; break;  
    case '*': x = y * z; break;  
    case '/': x = u / z; break;  
    default : ;  
}
```

Использование оператора break позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора switch, путем передачи управления оператору, следующему за switch.

Отметим, что в теле оператора switch можно использовать вложенные операторы switch, при этом в ключевых словах case можно использовать одинаковые константные выражения.

Пример:

```
:  
switch (a)  
{  
case 1: b=c; break;  
case 2:  
    switch (d)  
    { case 0: f=s; break;  
      case 1: f=9; break;
```

```
        case 2: f-=9; break;
    }
    case 3: b-=c; break;
    :
}
```

Оператор break

Оператор `break` обеспечивает прекращение выполнения самого внутреннего из объединяющих его операторов `switch`, `do`, `for`, `while`. После выполнения оператора `break` управление передается оператору, следующему за прерванным.

2.5 Структурные операторы.

В Си существуют все три типа операторов цикла: цикл с предусловием, цикл с постусловием и цикл с параметром.

Цикл с предусловием

Формат оператора цикла с предусловием:

while (выражение) оператор;

Цикл повторяет свое выполнение, пока значение выражения отлично от нуля, т. е. заключенное в нем условие цикла истинно.

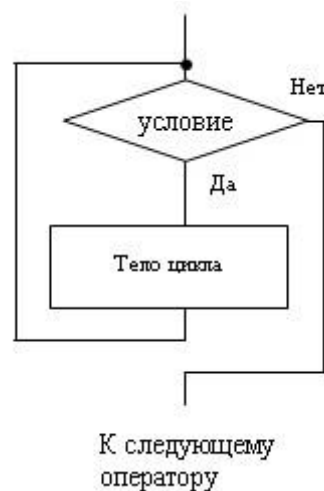


Рисунок 1– Алгоритм цикла с предусловием

В качестве примера использования оператора цикла рассмотрим программу вычисления факториала целого положительного числа N!.

Пример 1.

//программа вычисления факториала

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

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

```
void main()
```

```
{ long int F;
```

```
int i,N;
```

```
printf("N=");
```

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

```
F=i=1;
```

```
While(i<=N) { F=F*i; i++; }
```

```
printf("\n %d \n%d",N,F); getch(); }
```


Пример 2. Составить программу приближенного вычисления суммы бесконечного ряда $S = 1 + x/1! + x^2/2! + \dots$

Решение:

По условию задачи считается, что нужное приближение получено, если вычислена сумма нескольких первых слагаемых, и очередное слагаемое оказалось по модулю меньше, чем данное малое положительное число \square .

Введем обозначения: EPS - малое положительное число, например 0.003; S - сумма; i - переменная, последовательно принимающая значения 0, 1, 2,...; Y - значение $x^i/i!$, которое вычисляется путем умножения предыдущего результата на x/i .

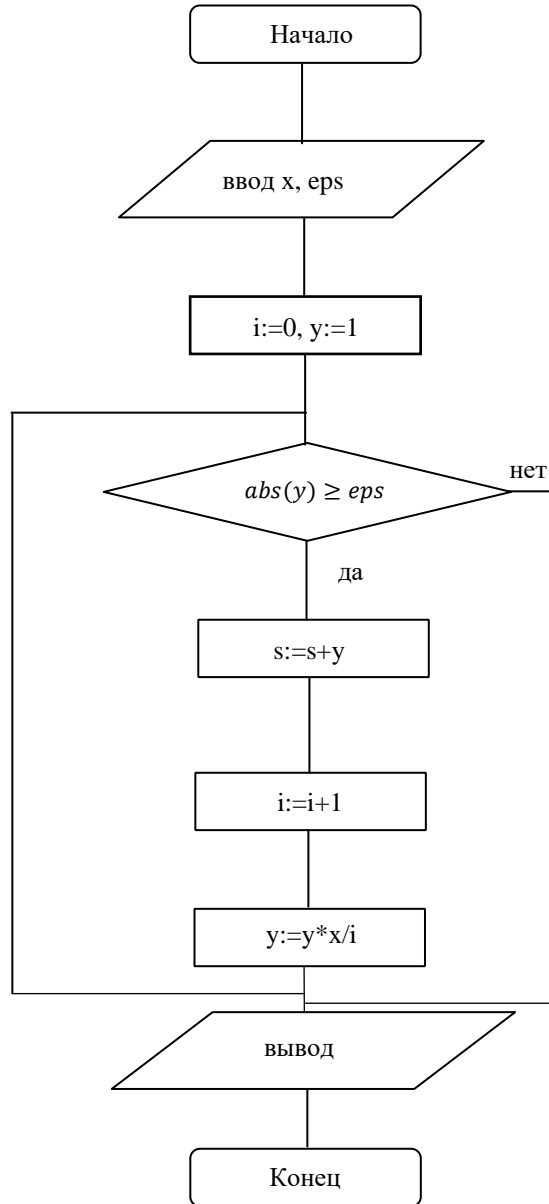


Рисунок 2 – Блок схема алгоритма решения задачи из примера 2

Программа:

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

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

```
void main()
```

```
{ int i=0; float x,y=1,eps,s=0;
```

```
printf ("Введите x, eps");
```

```
scanf("%f%f",&x,&eps);
```

```
while (abs(y)>=eps)
```

```
// начало цикла
```

$$\{ \text{ s} := \text{ s} + \text{ y};$$
$$\mathbf{i} := \mathbf{i} + 1;$$

```

y:=y*x/i;
}                                     // конец цикла
printf("\n сумма s=%f",s);
getch();
}

```

Результат выполнения программы:

Введите x, eps

2 0.003

Сумма S= 7.39

Пример 3. Найти натуральное число **n**, удовлетворяющее условию $n^3 - n^2 = 100$, или вывести сообщение об отсутствии такого числа.

Решение:

При решении такого класса задач поиск начинается с наименьшего натурального числа, т.е. 1, и продолжается (с увеличением значения **n** на шаг 1) до тех пор, пока выражение в левой части меньше значения в правой части заданного условия. Выход из цикла может осуществиться в двух случаях: при равенстве левой и правой частей (что означает, что искомое число найдено) или, когда значение в левой части больше значения в правой части (что означает, что искомого числа не существует).

Программа:

```

#include <stdio.h>
#include <conio.h>
void main()
{ int n=1;
while (pow(n,3)-pow(n,2) < 100)
n=n+1;
if (pow(n,3)-pow(n,2) == 100) printf("\n n=%d",n);
else printf("\n такого числа нет");
getch();}

```

Интересно свойство следующего оператора:

```
while (1);
```

Это бесконечный пустой цикл. Использование в качестве выражения константы 1 приводит к тому, что условие повторения

цикла все время остается истинным и работа цикла никогда не заканчивается. Тело в этом цикле представляет собой пустой оператор.

Цикл с постусловием

Формат оператора цикла с постусловием:

do оператор while (выражение);

Цикл выполняется до тех пор, пока выражение отлично от нуля, т.е. заключенное в нем условие цикла истинно. Выход из цикла происходит после того, как значение выражения станет ложным, иными словами равным нулю.

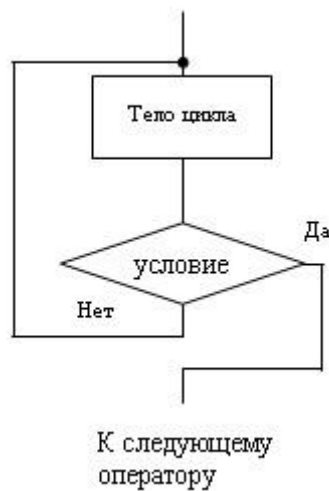


Рисунок 3– Алгоритм цикла с постусловием

В качестве примера рассмотрим программу вычисления $N!$, в которой используется цикл с постусловием.

Пример 1.

// Программа вычисления факториала

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

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

```
void main()
```

```
{long int F;
```

```
int i,N; printf("N=");
```

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

```
F=i=1; do F=F*I; i++;
```

```
while(i<=N);
```

```
printf("\n %d \n%d",N,F);  
getch(); }
```

Пример 2. Вычислить и напечатать множество значений функции $y=x^2+b$ для x , изменяющегося от -10 до 10 с шагом 2.

Решение:

Пусть $b=5$. Введем обозначения: x - переменная, отображающая значения аргумента x ; y - переменная, отображающая значения функции y ; b - константа b ; dx - шаг изменения x . Переменная y вычисляется по известной формуле на интервале изменения переменной x с помощью оператора цикла `do while`.

Блок-схема алгоритма приведена на рисунке 10.

Программа:

```
#include <stdio.h>  
#include <conio.h>  
const int b=5;  
void main()  
{ float x,y,dx;  
printf ("Введите x,dx");  
scanf ("%f%f",&x,&dx);  
do                                     // начало цикла y:=x*x+b;  
printf("\n x=%f y=%f",x,y);  
x=x+dx;  
while (X<=10) ;                       // конец цикла  
getch();}
```

Результат выполнения программы:

Вводите $x=-10.0$, $dx=20.0$

-10.0 20.0

$x= -10$ $y=105$

$x=10$ $y=105$

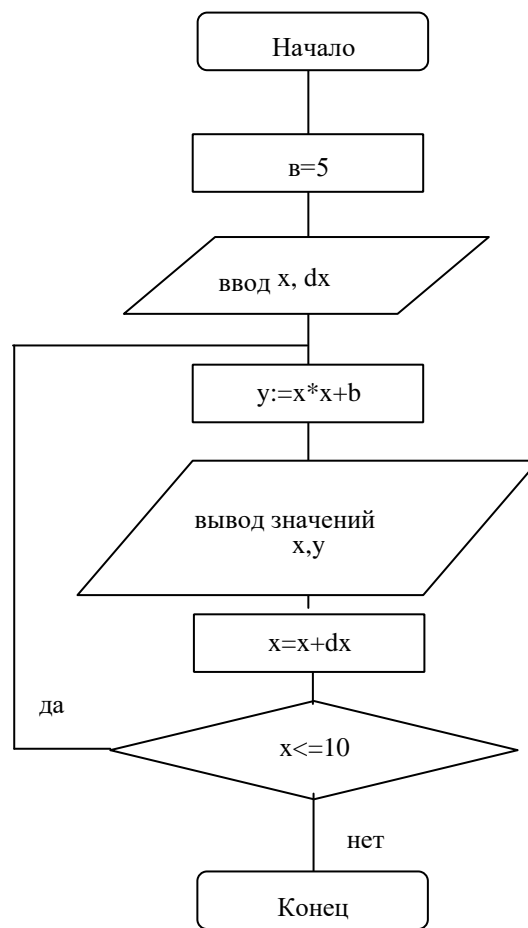


Рисунок 4 – Блок схема алгоритма решения задачи из примера 2

Цикл с параметром

Формат оператора цикла с параметром:

**for (выражение_1; выражение_2; выражение_3)
оператор;**

Выражение 1 выполняется только один раз в начале цикла. Обычно оно определяет начальное значение параметра цикла (инициализирует параметр цикла). Выражение 2 – это условие выполнения цикла. Выражение 3 обычно определяет изменение параметра цикла, оператор – тело цикла, которое может быть простым или составным. В последнем случае используются фигурные скобки.

Алгоритм выполнения цикла for представлен на блок-схеме на рисунке 5.

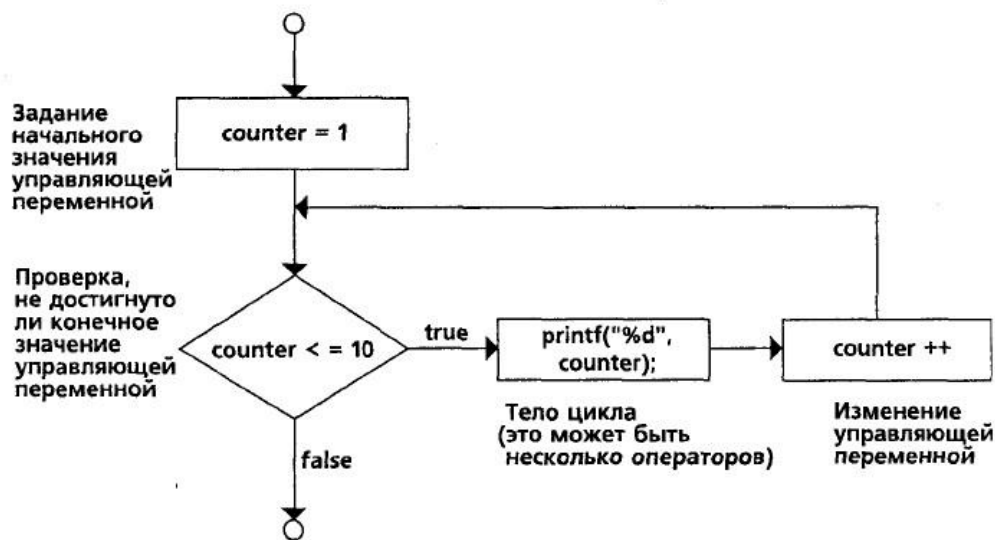


Рисунок 5 – Алгоритм цикла с параметром

С помощью цикла for нахождение $N!$ можно организовать следующим образом:

```
F=1;
```

```
for(i=1;i<=N;i++) F=F*i;
```

Используя операцию «запятая», можно в выражение 1 внести инициализацию значений сразу нескольких переменных:

```
for(F=1,i=1;i<=N;i++) F=F*i;
```

Некоторых элементов в операторе for может не быть, однако разделяющие их точки с запятой обязательно должны присутствовать. В следующем примере инициализирующая часть вынесена из оператора for:

```
F=1;
```

```
i=1;
```

```
for(i=1;i<=N;i++) F=F*i;
```

В языке Си оператор for является достаточно универсальным средством для организации циклов. С его помощью можно программировать даже итерационные циклы, что невозможно в Паскале. Вот пример вычисления суммы элементов гармонического ряда, превышающих заданную величину ϵ :

```
for(n=1,S=0;1.0/n>eps && n<INT_MAX;n++) S+=1.0/n;
```

//И наконец, эта же самая задача с пустым телом цикла:

```
for(n=1,S=0;1.0/n>eps && n<INT_MAX; S+=1.0/n++);
```

Следующий фрагмент программы на Си++ содержит два вложенных цикла for. В нем запрограммировано получение на экране таблицы умножения.

```
for(x=2;x<=9;x++)  
for(y=2;y<=9;y++)  
printf("\n %d*%d=", x*y);
```

На экране будет получен следующий результат:

2*2=4

2*3=6

9*8=72

9*9=81

Пример 1. Составить программу вычисления К первых членов арифметической прогрессии, заданных рекуррентной формулой $A_{n+1} = A_n + 2$, где $A_1=5$, $K=6$.

Решение:

Введем обозначения: x - член арифметической прогрессии a_n ,
 k - количество членов прогрессии,
 n - параметр цикла.

Блок-схема алгоритма приведена на рисунке 12

Программа:

```
#include <conio.h>  
#include <stdio.h>  
void main () {  
    //раздел описания переменных  
    int n,k; float x;  
    k=6;  
    printf("/n x="); scanf("%f",&x);  
    //начало цикла  
    for (n=2;n<=k; n++)  
    {  
        //вычисление членов прогрессии  
        x=x+2;  
        printf("x=%f", x); };  
    //конец цикла  
    getch(); }
```

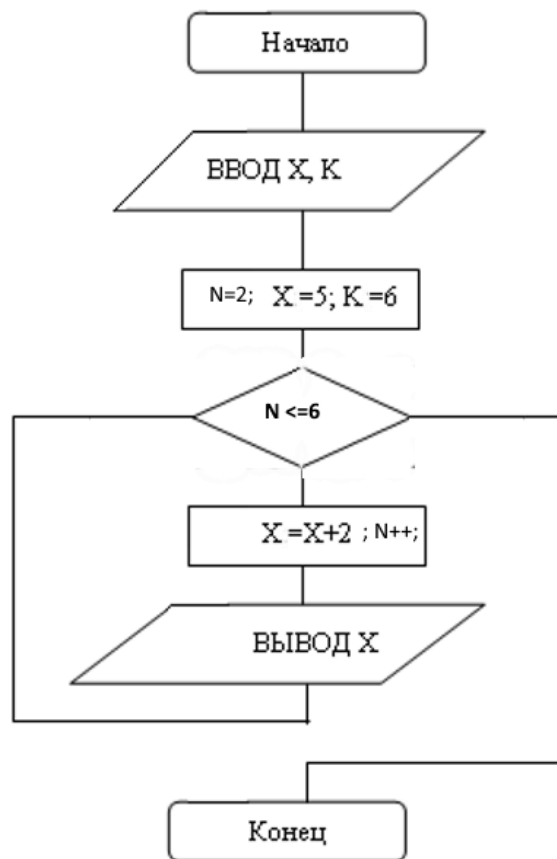



Рисунок 6– Блок-схема алгоритма решения задачи из примера 1

Результат выполнения программы:

$x = 5.00$
 $x = 7.00$
 $x = 9.00$
 $x = 11.00$
 $x = 13.00$
 $x = 15.00$

Пример 2. Среди первых 1000 членов последовательности A_n найти номер и значение первого положительного члена, если $A_n = -100 - n + n^3$. Предусмотреть случай, когда последовательность не содержит искомого элемента.

Программа:

```

#include <stdio.h>
#include <conio.h>
void main () {
int n,pr; float a;
pr=0;
for (n=1; n<=1000; n++) {

```

```

a=-100-n+pow(n,3);
if a>0 {
printf(“a=%f n=%d”,a,n);
pr=1;
break;
}
}
if (pr=0) printf(“положительных значений нет”); getch(); }

```

Оператор continue

Если выполнение очередного шага цикла требуется завершить до того, как будет достигнут конец тела цикла, используется оператор `continue`. Следующий фрагмент программы обеспечивает вывод на экран всех четных чисел в диапазоне от 1 до 100.

```

for(i=1;i<=100;i++)
{if(i%2) continue; printf("\t%d",i); }

```

Для нечетных значений переменной / остаток от деления на 2 будет равен единице, этот результат воспринимается как значение «истина» в условии ветвления, и выполняется оператор `continue`. Он завершит очередной шаг цикла, выполнение цикла перейдет к следующему шагу.

Оператор goto

Оператор безусловного перехода `goto` существует в языке Си, как и во всех других языках программирования высокого уровня. Однако с точки зрения структурного подхода к программированию его использование рекомендуется ограничить. Формат оператора:

goto метка;

Метка представляет собой идентификатор с последующим двоеточием, ставится перед помечаемым оператором.

Одна из ситуаций, в которых использование `goto` является оправданным — это необходимость «досрочного» выхода из вложенного цикла. Вот пример такой ситуации:

```

for(...)
{ while (...)

```

```
{ for(...)  
{ ... goto exit ...}  
}  
}
```

exit: printf("выход из цикла");

При использовании оператора безусловного перехода необходимо учитывать следующие ограничения:

- нельзя входить внутрь блока извне;
- нельзя входить внутрь условного оператора (if ...else...);
- нельзя входить внутрь переключателя;
- нельзя входить внутрь цикла.

2.6. Структурированные типы данных. Массивы.

Массив – это структура однотипных элементов, занимающих непрерывную область памяти. С массивом связаны следующие его свойства: имя, тип, размерность, размер.

Формат описания массива следующий:

тип элементов имя [константное выражение]

Константное выражение определяет размер массива, т. е. число элементов этого массива.

Одномерные массивы

Согласно описанию

int A[10];

объявлен одномерный массив с именем А, содержащий 10 элементов целого типа. Элементы массива обозначаются индексированными именами.

Нижнее значение индекса равно 0:

A[0], A[1], A[2], A[3], A[4], A[5], A[6], A[7], A[8], A[9]

В Си нельзя определять произвольные диапазоны для индексов. Размер массива, указанный в описании, всегда на единицу больше максимального значения индекса.

Размер массива может явно не указываться, если при его объявлении производится инициализация значений элементов. Например:

int p[]={2, 4, 6, 10, 1};

В этом случае создается массив из пяти элементов со следующими значениями:

p[0]=2, p[1]=4, p[2]=6, p[3]=10, p[4]=1

В результате следующего объявления массива

int M[6]={5, 3, 2};

будет создан массив из шести элементов. Первые три элемента получат инициализированные значения. Значения остальных будут либо неопределенными, либо равны нулю, если массив внешний или статический.

Рассмотрим несколько примеров программ обработки одномерных массивов.

Пример 1. Ввод с клавиатуры и вывод на экран одномерного массива.

Программа:

```
//Ввод и вывод массива
#include <stdio.h>
#include <conio.h>
void main()
{ int i, A [ 5 ] ;
  clrscr();
  for(i=0; i<5; i++)
  { printf("\nA[%d]=",i);
    scanf("%d", &A[i]); }
  for(i=0; i<5; i++)
  printf("\n A[%d]=%d",i,A[i] );
}
```

Двумерные массивы

Двумерный массив трактуется как одномерный массив, элементами которого является массив с указанным в описании типом элементов. Например, оператор

float R[5][10];

объявляет массив из пяти элементов, каждый из которых есть массив из десяти вещественных чисел. Отдельные величины этого массива обозначаются именами с двумя индексами: R[0][0], R [0][1] , ..., R [4][9]. Объединять индексы в одну пару скобок нельзя, т.е. запись R[2, 3] ошибочна.

Порядок расположения элементов многомерного массива в памяти такой, что прежде всего меняется последний индекс, затем предпоследний и т.д., и лишь один раз пробегает свои значения первый индекс.

При описании многомерных массивов их также можно инициализировать. Делать это удобно так:

```
int M[3][3]={ 11,12,13,
```

21,22,23,

31,32,33 };

Рассмотрим примеры программ обработки матриц – числовых двумерных массивов.

Пример 1. Ввести матрицу $a[2][3]$ и найти сумму ее элементов.

Решение:

Блок-схема алгоритма приведена на рисунке 16. Программа:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a[2][3],i,j,s=0;
    clrscr();
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            scanf("%d",&a[i][j]);
    for(i=0;i<3;i++)
        {
            for(j=0;j<2;j++)
                s=s+a[i][j];
        }
    printf("\ns=%d",s);
    getch();
}
```

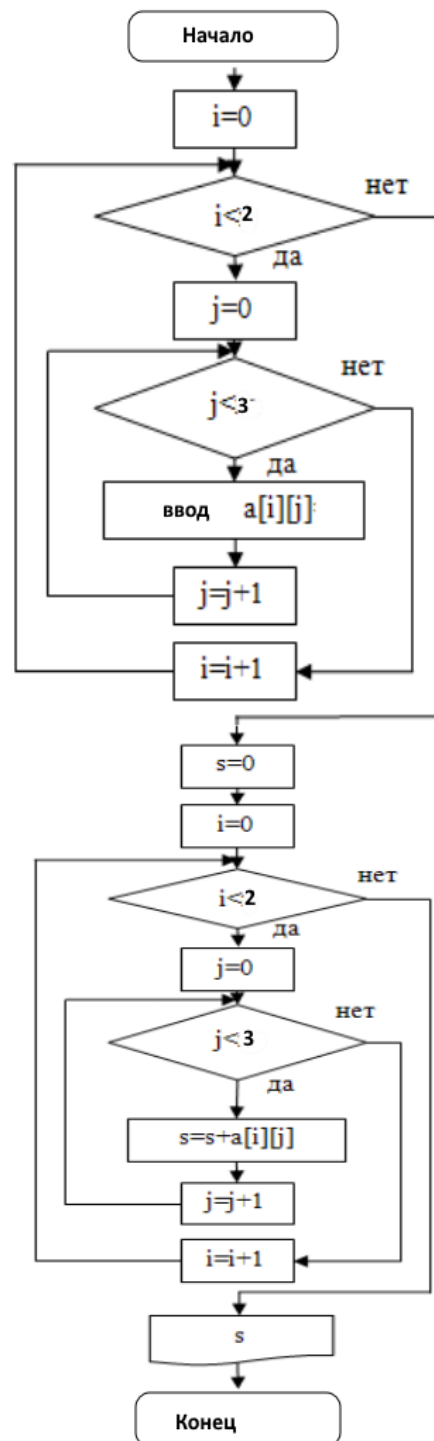


Рисунок 1— Блок-схема алгоритма решения задачи из примера 1

Строковый литерал

Строкой называется массив символов, заканчивающийся символом `'\0'`.

На каждый символ отводится 1 байт. Каждый символ имеет свой код (от 0 до 255), эти коды определяются по специальной таблице. Количество символов последовательности называется длиной строки.

Массив символов: `char s[10]`.

В массиве, в котором храниться строковая переменная, сразу после последнего символа строки ставиться специальный символ «`\0`». Поскольку символ «`\0`» всегда занимает один элемент массива, максимальная длина строки, которая может храниться в массиве, на единицу меньше его размера.

Строка описывается как символьный массив: `имя_типа`

`имя_массива[константное выражение];`

Например:

```
int a[10];          /* массив с именем a, содержащий 10 элементов  
целого    типа */
```

```
char b[50]; /* массив b, содержащий 50 элементов типа char */
```

Начальное значение строки можно задать при объявлении в двойных кавычках после знака равенства:

```
char s[80] = "Привет, Вася!";
```

Символы в кавычках будут записаны в начало массива `s`, а затем — признак окончания строки `'\0'`. Оставшиеся символы не меняются, и в локальных строках там будет «мусор». Можно написать и так

```
char s[] = "Привет, Вася!";
```

В этом случае компилятор подсчитает символы в кавычках, выделит памяти на 1 байт больше и занесет в эту область саму строку и завершающий ноль.

Если строка не будет изменяться во время работы программы, то можно объявить константу (постоянную строку) так:

```
const char PRIVET[] = "Привет, Вася!";
```

Стандартный ввод и вывод

Для ввода и вывода строк с помощью функций `scanf()` и `printf()` используется специальный формат `"%s"`:

```
#include<stdio.h>
void main()
{
    char Name[50];
    printf("Как тебя зовут? ");
    scanf("%s", Name);
    printf("Привет, %s!", Name);
}
```

Заметьте, что в функцию `scanf()` надо передать просто имя строки (без знака `&`), ведь имя массива является одновременно адресом его начального элемента.

Однако у функции `scanf()` есть одна особенность: она заканчивает ввод, встретив первый пробел. Если надо ввести всю строку целиком, включая пробелы (то есть до нажатия на клавишу `Enter`), то применяется функция `gets(s)`;

Для вывода строки на экран можно (кроме `printf()`) использовать и функцию `puts()`, которая после вывода строки еще и дает команду перехода на новую строку.

Пример. Ввести символьную строку и заменить в ней все буквы 'А' на буквы 'Б'.

Будем рассматривать строку как массив символов. Надо перебрать все элементы массива, пока мы не встретим символ `'\0'` (признак окончания строки) и, если очередной символ – это буква 'А', заменить его на 'Б'. Для этого используем цикл `while`, потому что мы заранее не знаем длину строки. Условие цикла можно сформулировать так: «пока не конец строки».

```
#include<stdio.h>
void main()
{
    char s[80];
    int i;
    puts( "Введите строку" );
    gets(s);
```



```

        i=0; // начать с первого символа, s[0]
        while (s[i] != „\0“ ) // пока не достигли конца строки
        {
                if ( s[i] == „А“ ) s[i]=“Б”;
                i++;
        }
        puts ( s );
}

```

Заметьте, что одиночный символ записывается в апострофах, а символьная строка – в кавычках.

Функции для работы со строками

Для использования этих функций надо включить в программу заголовочный файл

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

Длина строки – strlen()

Эта функция определяет текущую длину строки (не считая '\0').

```

int len;
char s[] = "Hello, world! ";
len = strlen(s);
printf ( "Длина строки %s равна %d", s, len );

```

Сравнение строк – strcmp()

Для сравнения двух строк используют функцию **strcmp()**. Функция возвращает ноль, если строки равны (то есть «разность» между ними равна нулю) и ненулевое значение, если строки различны. Сравнение происходит по кодам символов, поэтому функция различает строчные и заглавные буквы – они имеют разные коды.

```

char s1[] = "Вася", s2[] = "Петя";
if ( 0 == strcmp(s1,s2) )
        printf("Строки %s и %s одинаковы", s1, s2);
else printf("Строки %s и %s разные", s1, s2);

```

Если строки не равны, функция возвращает «разность» между первой и второй строкой, то есть *разность кодов первых различных символов*. Эти числа можно использовать для сортировки строк – если

«разность» отрицательна, значит первая строка «меньше» второй, то есть стоит за ней в алфавитном порядке.

Задача. Ввести две строки и вывести их в алфавитном порядке.

```
#include <stdio.h>
#include <string.h>
void main()
{ char s1[80], s2[80];
  printf ("Введите первую строку");
  gets(s1);
  printf ("Введите вторую строку");
  gets(s2);
  if ( strcmp(s1,s2) <= 0 )
    printf("%s\n%s", s1, s2);
  else printf("%s\n%s", s2, s1);
}
```

Иногда надо сравнить не всю строку, а только первые несколько символов. Для этого служит функция **strncmp()** (с буквой **n** в середине). Третий параметр этой функции – количество сравниваемых символов. Принцип работы такой же – она возвращает нуль, если заданное количество первых символов обеих строк одинаково.

```
char s1[80], s2[80];
printf ("Введите первую строку");
gets(s1);
printf ("Введите вторую строку");
gets(s2);
if ( 0 == strncmp(s1, s2, 2) )
  printf("Первые два символа %s и %s одинаковы", s1, s2);
else
  printf("Первые два символа %s и %s разные", s1, s2);
```

Один из примеров использования функции **strcmp()** – проверка пароля.

Пример. Составить программу, которая определяет, сколько цифр в символьной строке. Программа должна работать только при вводе пароля «куку».

```
#include<stdio.h>
#include<string.h>
main()
{
  char pass[] = "куку", // правильный пароль
```

```

s[80]; // вспомогательная строка int i, count = 0;
printf ("Введите пароль "); gets(s);
if ( strcmp ( pass, s ) != 0 )
{
    printf ( "Неверный пароль" );
    return 1; // выход по ошибке, код ошибки 1
}
printf ("Введите строку");
gets(s);
i = 0;
while ( s[i] != '\0' ) {
    if ( s[i] >= '0' && s[i] <= '9' )
        count ++; }
printf("\nНашли %d цифр", count);
}

```

В этой программе использован тот факт, что коды цифр расположены в таблице символов последовательно от '0' до '9'.

Копирование строк

В копировании участвуют две строки, они называются «*источник*» (строка, откуда копируется информация) и «*приемник*» (куда она записывается или добавляется).

При копировании строк надо проверить, чтобы для строки-приемника было выделено достаточно места в памяти.

Простое копирование выполняет функция `strcpy()`. Она принимает два аргумента: сначала строка-приемник, потом – источник (порядок важен!).

```

char s1[50], s2[10];
gets(s1);
strcpy ( s2, s1); // s2 (приемник) <- s1 (источник)
puts ( s2 );

```

Следующая строка скопирует строку `s2` в область памяти строки `s1`, которая начинается с ее 6-ого символа, оставив без изменения первые пять:

```

strcpy ( s1+5, s2 );

```

Функция позволяет скопировать только заданное количество символов, она называется `strncpy` и принимает в третьем параметре количество символов, которые надо скопировать. Важно помнить, что эта функция *НЕ записывает завершающий нуль*, а только копирует символы (в отличие от нее `strcpy()` всегда копирует завершающий нуль). Функция `strncpy()` особенно полезна тогда, когда надо по частям собрать строку из кусочков.

```
char s1[] = "Ку-ку", s2[10];  
strncpy ( s2, s1, 2 ); // скопировать 2 символа из s1 в s2  
puts ( s2 );          // ошибка! нет последнего '\0'  
s2[2] = '\0';         // добавляем символ окончания строки  
puts (s2);            // вывод
```

Объединение строк

Функция – `strcat()` позволяет добавить строку-источник в конец строкиприемника (завершающий нуль записывается автоматически).

```
char s1[80] = "Могу, ",  
s2[] = "хочу, ", s3[] = "надо!";  
strcat ( s1, s2 ); // дописать s2 вконец s1  
puts ( s1 );       // "Могу, хочу, "  
strcat ( s1, s3 ); // дописать s3 вконец s1  
puts ( s1 );       // "Могу, хочу, надо!"
```

2.7. Указатели. Динамическое выделение памяти.

Указатель – это переменная, которая может содержать адрес некоторого объекта в памяти компьютера, например адрес другой переменной. И через указатель, установленный на переменную можно обращаться к участку оперативной памяти, отведенной компилятором под ее значения.

Указатель объявляется следующим образом:

тип *идентификатор;

Примеры описаний указателей:

int *pti; char *ptc; float *ptf;

После такого описания переменная pti может принимать значение указателя на величину целого типа; переменная ptc предназначена для хранения указателя на величину типа char; переменная ptf – на величину типа float.

С указателями связаны две унарные операции: & и *. Операция & означает «взять адрес» операнда (т.е. установить указатель на операнд). Данная операция допустима только над переменными. Операция * имеет смысл: «значение, расположенное по указанному адресу» и работает следующим образом:

- определяется местоположение в оперативной памяти переменной типа указатель.
- извлекается информация из этого участка памяти и трактуется как адрес переменной с типом, указанным в объявлении указателя.
- производится обращение к участку памяти по выделенному адресу для проведения некоторых действий.

Пример 1:

```
int x,      /* переменная типа int */
    *y;      /* указатель на элемент данных типа int */
y=&x;       /* y - адрес переменной x */
*y=1;      /* косвенная адресация указателем поля x
            “по указанному адресу записать 1”, т.е. x=1; */
```

Пример 2:

```
int i, j=8, k=5, *y;
```

```

y=&i;
*y=2;    /* i=2 */
y=&j;    /* переустановили указатель на переменную j */
*y+=i;    /* j+=i , т.е. j=j+1 -> j=j+2=10 */
y=&k;    /* переустановили указатель на переменную k */
k+=*y;    /* k+=k, k=k+k = 10 */
(*y)++;    /* k++, k=k+1 = 10+1 = 11 */

```

Указателю-переменной можно присвоить значение другого указателя либо выражения типа указатель с использованием, при необходимости операции приведения типа (приведение необязательно, если один из указателей имеет тип "void *").

```

int i, *x;
char *y;
x=&i;    /* x -> поле объекта int */
y=(char *)x;    /* y -> поле объекта char */
y=(char *)&i;    /* y -> поле объекта char */

```

Рассмотрим фрагмент программы:

```

int a=5, *p, *p1, *p2;
p=&a; p2=p1=p;
++p1; p2+=2;
printf("a=%d, p=%d, p=%p, p1=%p, p2=%p.\n",a,p,p,p1,p2);

```

Результат выполнения:

a=5, *p=5, p=FFC8, p1=FFCC, p2=FFD0.

Конкретные значения адресов зависят от ряда причин: архитектура компьютера, тип и размер оперативной памяти и т.д.

Операции над указателями (адресная арифметика)

Записывая выражения и операторы, изменяющие значения указателей, необходимо помнить главное правило: *единицей изменения указателя является размер соответствующего ему типа.*

Например, int *pti; char *ptc; float *ptf;

Выполнение операторов pti=pti+1; или pti++; изменит значение указателя pti на 2, в результате чего он примет значение FFC2. В результате выполнения оператора pti--; значение указателя уменьшится на 2 и станет равным FFBE.

Аналогично для указателей других типов:

ptc++; увеличит значение указателя на 1;

ptf++; увеличит значение указателя на 4.

Разрешается сравнивать указатели и вычислять разность двух указателей. При сравнении могут проверяться отношения любого вида (" $>$ ", " $>=$ ", " $<$ ", " $<=$ ", " $==$ ", " $!=$ "). Наиболее важными видами проверок являются отношения равенства или неравенства.

В заголовочном файле `stdio.h` определена константа – нулевой указатель с именем `NULL`. Ее значение можно присваивать указателю. Например:

```
ptf=NULL;
```

Нулевой указатель обозначает отсутствие конкретного адреса ссылки.

Пример. Ввести два целых числа. Вычислить сумму, разность и деление этих чисел.

```
#include<stdio.h>
#include<stdio.h>
void main()
{ int a, b, *p1, *p2, *p3;
  float *p4;
  clrscr();
  puts("Введите 2 целых числа");
  scanf ("%d%d", a, b);
  *p1=a+b;
  *p2=a*b;
  *p3=a-b;
  *p4=(float)a/b;
  printf ("%d + %d=%d\n", a, b, *p1);
  printf ("%d * %d=%d\n", a, b, *p2);
  printf ("%d - %d=%d\n", a, b, *p3);
  printf ("%d / %d=%f\n", a, b, *p4);
  getch();
}
```


ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ ПАМЯТИ

Указатели чаще всего используют при работе с динамической памятью, которую иногда называют «*куча*» (перевод английского слова *heap*). Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти производится только через указатели. Время жизни динамических объектов – от точки создания до конца программы или до явного освобождения памяти.

Динамическая переменная хранится в некоторой области ОП, не обозначенной именем, и обращение к ней производится через переменную указатель.

Библиотечные функции

Функции для манипулирования динамической памятью в стандарте Си следующие:

*void *calloc(unsigned n, unsigned size);* – выделение памяти для размещения *n* объектов размером *size* байт и заполнение полученной области нулями; возвращает указатель на выделенную область памяти;

*void *malloc (unsigned n)* – выделение области памяти для размещения блока размером *n* байт; возвращает указатель на выделенную область памяти;

*void *realloc (void *b, unsigned n)* – изменение размера размещенного по адресу *b* блока на новое значение *n* и копирование (при необходимости) содержимого блока; возвращает указатель на перераспределенную область памяти; при возникновении ошибки, например, нехватке памяти, эти функции возвращают значение *NULL*, что означает отсутствие адреса (нулевой адрес);

*void free (void *b)* – освобождение блока памяти, адресуемого указателем *b*.

Для использования этих функций требуется подключить к программе в зависимости от среды программирования заголовочный файл *alloc.h* или *malloc.h*.

Создание одномерного динамического массива

В языке Си размерность массива при объявлении должна задаваться константным выражением.

Если до выполнения программы неизвестно, сколько понадобится элементов массива, нужно использовать динамические массивы, т.е. при необходимости работы с массивами переменной размерности вместо массива достаточно объявить указатель требуемого типа и присвоить ему адрес свободной области памяти (захватить память).

Память под такие массивы выделяется с помощью функций *malloc* и *calloc* во время выполнения программы. Адрес начала массива хранится в переменной-указателе. Например:

```
int n = 10;  
double *b = (double *) malloc(n * sizeof (double));
```

В примере значение переменной *n* задано, но может быть получено и программным путем.

Обнуления памяти при ее выделении не происходит. Инициализировать динамический массив при декларации нельзя.

Обращение к элементу динамического массива осуществляется так же, как и к элементу обычного – например *a[3]*. Можно обратиться к элементу массива и через косвенную адресацию – **(a + 3)*. В любом случае происходят те же действия, которые выполняются при обращении к элементу массива, декларированного обычным образом.

После работы захваченную под динамический массив память необходимо освободить, для нашего примера *free(b)*;

Таким образом, время жизни динамического массива, как и любой динамической переменной – с момента выделения памяти до момента ее освобождения.

Пример работы с динамическим массивом:

```
#include <alloc.h>  
#include <stdio.h>  
void main()  
{ double *x; int n;  
printf("\nВведите размер массива – ");  
scanf("%d", &n);  
// Захват памяти  
if ((x = (double*)calloc(n, sizeof(*x)))==NULL)
```

```

    { puts("Ошибка ");          return;    }
    ...
    // Работа с элементами массива
    ...
    free(x);    }    // Освобождение памяти

```

Создание двумерного динамического массива

Напомним, что имя двумерного массива – указатель на указатель. В данном случае сначала выделяется память на указатели, расположенные последовательно друг за другом, а затем каждому из них выделяется соответствующий участок памяти под элементы.

```

...
int **m, n1, n2, i, j;
puts(" Введите размеры массива (строк, столбцов): ");
scanf("%d%d", &n1, &n2);
// Захват памяти для указателей – A (n1=3)
m = (int**)calloc(n1, sizeof(int*));
for (i=0; i<n1; i++)
    // Захват памяти для элементов – B (n2=4)
    *(m+i) = (int*)calloc(n2, sizeof(int));
for ( i=0; i<n1; i++)
for ( j=0; j<n2; j++)
m[i] [j] = i+j;      // (*(m+i)+j) = i+j;
...
for(i=0; i<n; i++)
    free(m[i]);      // Освобождение памяти
free(m);
...

```

Указатели и символьные массивы (строки). Для решения задач, сводящихся к задаче о выделении подстроки, начало которой не совпадает с началом содержащей ее строки, нужно воспользоваться указателями. Пример: выделить из текста N символов, начиная с позиции m -го символа.

Для решения задачи определим указатель `pAux` на тип данных `char` (т.е.

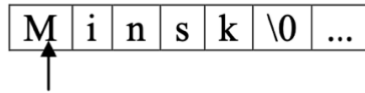
на любые переменные типа `char`), которому с помощью операции взятия адреса присвоим адрес первого элемента строки `str2`.

```
char *pAux;
```

```
pAux=&(str2[0]);
```

Эта ситуация иллюстрируется на рисунке:

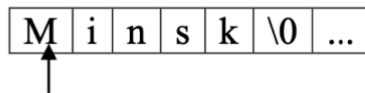
```
char str2[]="Minsk";
```



```
pAux=&(str2[0]);
```

После того как мы увеличим исходное значение указателя `pAux` на единицу (`pAux= pAux+1`), он будет показывать на следующий байт памяти, т.е. на следующий элемент массива `str2` с элементами типа `char`, каждый из которых занимает в памяти компьютера всего один байт:

```
char str2[]="Minsk";
```



```
pAux= pAux+1;
```

Ситуация, показанная на этом рисунке, является исходной для выделения из строки `str2` подстроки, начинающейся со второй позиции. Например, следующий вызов функции `strncpy()`:

```
strncpy(str1, pAux, 3);
```

приведет к тому, что в буфер `str1` будет скопирована подстрока "ins", т.е. три символа из строки `str2`, начиная с позиции символа, на который показывает указатель `pAux`. Убедиться в этом на практике можно с помощью следующей программы.

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

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

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

```
void main() { char str1[128];
```

```
char str2[]="Minsk";
```

```
char *pAux;
```

```
pAux=&(str2[0]);
```

```
pAux=pAux+1;
```

```
memset(str1, 0, sizeof(str1));
```

```
strncpy(str1, pAux, 3);
```

```
printf("str1 □ %s\n", str1);  
getch();  
}
```

2.8. Функции пользователя.

Если в программе возникает необходимость частого обращения к некоторой группе операторов, то рационально сгруппировать такую группу операторов в самостоятельный блок, к которому можно обращаться, указывая его имя. Такие самостоятельные программные блоки называются **подпрограммами пользователя**.

Функции – подпрограммы пользователя, которые возвращают какие-либо значения.

Функция – это вспомогательная программа (подпрограмма), предназначенная для получения некоторого объекта-результата (например, числа).

Функция имеет следующий вид:

```
Тип_результата  Имя_функции ([Список_параметров])  
{  
    Тело_функции  
}
```

Тип_результата – это тип возвращаемого из функции результата. Типом возвращаемого значения может быть любой тип данных, кроме массива или функции, но может быть указатель на массив или указатель на функцию. Если тип возвращаемого значения **void**, то это означает, что функция не возвращает никакого значения.

Имя_функции – это любой правильно написанный идентификатор.

Список_параметров – это список разделенных запятыми объявлений тех параметров, которые получает функция при вызове. Для каждого параметра, передаваемого в функцию, указывается его тип и имя. Если функция *не получает никаких значений*, список параметров задается как **void**, т.е. список параметров пустой.

Тело_функции – это блок или составной оператор. В теле функции может быть оператор **return**, который возвращает полученное значение функции в точку вызова. Он имеет следующую форму:

return выражение;

Любая функция должна быть *объявлена (прототип функции), вызвана и определена (описание функции)*.

Объявление (прототип) функции задает имя функции, тип возвращаемого значения и список передаваемых параметров. Прототип функции указывает компилятору тип данных, возвращаемый функцией, количество и тип параметров, которые ожидает функция, а также порядок их следования. Также прототип функции сообщает компилятору о том, что далее в тексте программы будет приведено ее полное определение.

Прототип функции имеет следующий вид:

Тип_результата Имя_функции ([Список_параметров]);

Пример прототипа функции fun, которая получает три целых числа и одно вещественное, а возвращает вещественное значение:

`double fun(int, int, int, double);`

При **вызове** (активизации) **функции** указываются: *имя функции* и *фактические параметры* (т.е. значения, которые передаются при вызове функции).

Существуют два способа передачи параметров в функцию: **по адресу** (с помощью указателя) и **по значению**.

Определение функции содержит, кроме объявления, тело функции, которое представляет собой последовательность описаний и операторов.

Пример. Написать программу, которая вводит целое число и определяет сумму его цифр. Использовать функцию, вычисляющую сумму цифр числа.

Для того чтобы найти последнюю цифру числа, надо взять остаток от его деления на 10. Затем делим число на 10, отбрасывая его последнюю цифру, и т.д. Сложив все эти остатки-цифры, мы получим сумму цифр числа.

```
#include <stdio.h>
#include <conio.h>
int SumDigits (int N)    // заголовок функции
{                        // начало функции
    int d, sum = 0;      // тело функции
    while (N != 0 )
    {
        d = N % 10;
        sum = sum + d;
```

```

        N = N / 10;
    }
    return sum ;    // возврат суммы
}
void main()
{
    int N, s ;
    puts( "Введите целое число ");
    scanf ("%d", &N);
    s = SumDigits (N)           // Вызов функции
    printf (" Сумма цифр числа %d = %d\n", N, s );
    getch();
}

```

Очень часто надо составить функцию, которая просто решает какой-то вопрос и отвечает на вопрос «Да» или «Нет». Такие функции называются *логическими*. В языке Си ноль означает ложное условие, а единица – истинное.

Логическая функция – это функция, возвращающая **1** (если ответ «Да») или **0** (если ответ «Нет»). Логические функции используются, главным образом, в двух случаях:

- если надо проанализировать ситуацию и ответить на вопрос, от которого зависят дальнейшие действия программы;
- если надо выполнить какие-то сложные операции и определить, была ли при этом какая-то ошибка.

Пример. Ввести число *N* и определить, простое оно или нет. Использовать функцию, которая отвечает на этот вопрос.

Теперь расположим тело функции ниже основной программы. Чтобы транслятор знал об этой функции во время обработки основной программы, надо объявить её заранее.

```

#include <stdio.h>
#include <conio.h>
int Prime (int N);    // объявление функции
void main()
{
    int N;
    puts( "Введите целое число ");

```



```

scanf ("%d", &N);
if ( Prime(N) )           // Вызов функции
printf (" Число %d – простое\n", N );
else printf (" Число %d – составное\n", N );
getch();
}
int Prime (int N)  // описание функции
{
    for ( int i=2; i*i <=N; i++ )
        if ( N % i == 0) return 0 ; // нашли делитель – составное
    return 1 ; // не нашли ни одного делителя – простое
}

```

По определению функция может вернуть только одно значение-результат. Если надо вернуть два и больше результатов, приходится использовать специальный прием – **передачу параметров по ссылке**.

Пример. Написать функцию, которая определяет максимальное и минимальное из двух целых чисел.

В следующей программе используется достаточно хитрый прием: мы сделаем так, чтобы функция изменяла значение переменной, которая принадлежит основной программе. Один результат (минимальное из двух чисел) функция вернет как обычно, а второй – за счет изменения переменной, которая передана из основной программы.

```

#include <stdio.h>
#include <conio.h>
int MinMax (int a, int b, int &Max)
{
    if (a>b) { Max = a; return b; }
    else { Max = b; return a; }
}
void main()
{
    int N, M, min, max ;
    puts( "Введите 2 целых числа ");
    scanf ("%d%d", &N, &M );
}

```

```

min = MinMax (N, M, max);           // Вызов функции
printf ("min= %d, max = %d\n", min, max );
getch();
}

```

Глобальные и локальные переменные

Глобальные переменные – это переменные, объявленные вне основной программы и подпрограмм.

Глобальные переменные доступны из любой функции. Поэтому их надо объявлять вне всех подпрограмм. Остальные переменные, объявленные в функциях, называются *локальными* (местными), поскольку они известны только той подпрограмме, где они объявлены.

- Если в подпрограмме объявлена локальная переменная с таким же именем, как и глобальная переменная, то используется **локальная** переменная.

- Если имена глобальной и локальной переменных совпадают, то для обращения к глобальной переменной в подпрограмме перед ее именем ставится два двоеточия:

```
:: var = ::var * 2 + var;
```

Везде, где можно, надо передавать данные в функции через их параметры. Если же надо, чтобы подпрограмма меняла значения переменных, надо передавать параметр по ссылке.

Рекурсивной (самовызываемой или самовызывающей) называют функцию, которая прямо или косвенно вызывает сама себя.

При каждом обращении к рекурсивной функции создается новый набор объектов автоматической памяти, локализованных в коде функции.

Возможность прямого или косвенного вызова позволяет различать прямую или косвенную рекурсии. Функция называется косвенно рекурсивной в том случае, если она содержит обращение к другой функции, содержащей прямой или косвенный вызов первой функции. В этом случае по тексту определения функции ее рекурсивность (косвенная) может быть не видна. Если в функции используется вызов этой же функции, то имеет место прямая рекурсия, т.е. функция по определению рекурсивная.

Рекурсивные алгоритмы эффективны в задачах, где рекурсия использована в самом определении обрабатываемых данных. Поэтому изучение рекурсивных методов нужно проводить, вводя динамические структуры данных с рекурсивной структурой. Рассмотрим вначале только принципиальные возможности, которые предоставляет язык Си для организации рекурсивных алгоритмов.

В рекурсивных функциях необходимо выполнять следующие правила.

1. При каждом вызове в функцию передавать модифицированные данные.

2. На каком-то шаге должен быть прекращен дальнейший вызов этой функции, это значит, что рекурсивный процесс должен шаг за шагом упрощать задачу так, чтобы для нее появилось нерекурсивное решение, иначе функция будет вызывать себя бесконечно.

3. После завершения очередного обращения к рекурсивной функции в вызывающую функцию должен возвращаться некоторый результат для дальнейшего его использования.

Пример 1. Заданы два числа a и b , большее из них разделить на меньшее, используя рекурсию.

Текст программы может быть следующим:

...

```
double proc(double, double);
```

```
void main (void)
```

```
{
```

```
double a,b;
```

```
puts(" Введи значения a, b : ");
```

```
scanf("%lf %lf", &a, &b);
```

```
printf("\n Результат деления : %lf", proc(a,b));
```

```
}
```

```
//----- Функция -----
```

```
double proc( double a, double b) {
```

```
if ( a< b ) return proc ( b, a );
```

```
else return a/b;
```

```
}
```

Если a больше b , условие, поставленное в функции, не выполняется и функция *proc* возвращает нерекурсивный результат.

Пусть теперь условие выполнилось, тогда функция *proc* обращается сама к себе, аргументы в вызове меняются местами и последующее обращение приводит к тому, что условие вновь не выполняется и функция возвращает нерекурсивный результат.

2.9. Структуры, объединения, перечисления.

Структура – это структурированный тип данных, представляющий собой поименованную совокупность разнотипных элементов. Тип *структура* обычно используется при разработке информационных систем, баз данных.

Формат описания структурного типа следующий:

```
struct имя_типа  
    {определения_элементов};
```

В конце обязательно ставится точка с запятой (это оператор). Элементы структуры называются *полями*. Каждому полю должно быть поставлено в соответствие имя и тип.

Например, сведения о выплате студентам стипендии требуется организовать в виде:

```
    |  
struct student {char fam[30];  
    int  kurs;  
    char grup[5];  
    float stip;};
```

После этого `student` становится именем структурного типа, который может быть назначен некоторым переменным. В соответствии со стандартом СИ это нужно делать так:

```
struct student stud1, stud2;
```

Здесь `stud1` и `stud2` – переменные структурного типа.

Допускается и другой вариант описания структурных переменных, когда можно вообще не задавать имя типа, а описывать сразу переменные:

```
struct {char fam[30];  
    int  kurs;  
    char grup[5];  
    float stip;  
} stud1, stud2, *pst;
```

В этом примере кроме двух переменных структурного типа объявлен указатель `pst` на такую структуру.

Обращение к полям структурной величины производится с помощью *уточненного* имени следующего формата:

имя_структуры.имя_элемента

Примеры уточненных имен для описанных выше переменных:

stud1.fam; stud1.stip;

Значения элементов структуры могут определяться вводом, присваиванием, инициализацией. Пример инициализации в описании:

student stud1={"Кротов", 3, "ПО313", 350};

Пусть в программе определен указатель на структуру

student *pst, stud1;

Тогда после выполнения оператора присваивания

pst=&stud1;

к каждому элементу структурной переменной stud1 можно обращаться тремя способами. Например, для поля fam

stud1.fam или (*pst).fam или pst->fam

В последнем варианте используется *знак операции доступа к элементу структуры*: ->. Аналогично можно обращаться и к другим элементам этой переменной.

Допускается использование массивов структур. Например, сведения о 100 студентах могут храниться в массиве, описанном следующим образом:

student stud[100];

Тогда сведения об отдельных студентах будут обозначаться, например, так: stud[1].fam, stud[5].kurs и т.п. Если нужно взять первую букву фамилии 25-го студента, то следует писать: stud[25].fam[0].

Пример 1. Ввести сведения об N студентах. Определить фамилии студентов, получающих самую высокую стипендию.

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

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

```
void main()
```

```
{ const N=30; int i; long maxs;
```

```
struct student {char fam[15];
```

```
                int kurs;
```

```
                char grup[3];
```

```
                long stip;  };
```

```
student stud[N];
```

```
clrscr();
```

```
for (i=0; i<N; i++)
```

```
{ printf("%d-й студент",i);
```

```

        printf("\n"Фамилия:"); scanf("%s",&stud[i].fam);
        printf("Курс:"); scanf("%d",&stud[i].kurs);
        printf("Группа:"); scanf("%s",&stud[i].grup);
        printf("Стипендия:"); scanf("%ld",&stud[i].stip); }
maxs=0;
for (i=0; i<N; i++)
    if (stud[i].stip>maxs) maxs=stud[i].stip;
printf("\n Студенты, получающие макс. стипендию %ld  руб.",
maxs);
for (i=0; i<N; i++)
    if (stud[i].stip==maxs) printf("\n%s", stud[i].fam);}

```

Объединение – поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента.

Объединенный тип данных декларируется подобно структурному типу:

```

union ID_объединения {
    описание полей    };

```

Пример описания объединенного типа:

```

union word {
    int nom;
    char str[20]; };

```

Пример объявления объектов объединенного типа:

```

union word *p_w, mas_w[100];

```

Объединения применяют для экономии памяти в случае, когда объединяемые элементы логически существуют в разные моменты времени либо требуется разнотипная интерпретация поля данных.

Практически все вышесказанное для структур имеет место и для объединений. Декларация данных типа *union*, создание переменных этого типа и обращение к полям объединений производится аналогично структурам.

Пример использования переменных типа *union*:

```

...
typedef union q {
    int a;

```

```

double b;
char s[5];
} W;
void main(void)
{
W s, *p = &s;
s.a = 4;
printf("\n Integer a = %d, Sizeof(s.a) = %d", s.a, sizeof(s.a));
p -> b = 1.5;
printf("\n Double b = %lf, Sizeof(s.b) = %d", s.b, sizeof(s.b));
strcpy(p->s, "Minsk");
printf("\n String a = %s, Sizeof(s.s) = %d", s.s, sizeof(s.s));
printf("\n Sizeof(s) = %d", sizeof(s));
}

```

Результат работы программы:

Integer a = 4, Sizeof(s.a) = 2

Double b = 1.500000, Sizeof(s.b) = 4

String a = Minsk, Sizeof(s.s) = 5

Sizeof(s) = 5

Перечисления – средство создания типа данных посредством задания ограниченного множества значений.

Определение перечисляемого типа данных имеет вид

```

enum ID_перечисляемого_типа {
    список_значений };

```

Значения данных перечисляемого типа указываются идентификаторами, например:

```
enum marks { zero, one, two, three, four, five };
```

Компилятор последовательно присваивает идентификаторам списка значений целочисленные величины 0, 1, 2,... . При необходимости можно явно задать значение идентификатора, тогда очередные элементы списка будут получать последующие возрастающие значения. Например:

```
enum level {
    low=100, medium=500, high=1000, limit };

```


Константа *limit* по умолчанию получит значение, равное 1001.

Примеры объявления переменных перечисляемого типа:

```
enum marks Est;
```

```
enum level state;
```

Переменная типа *marks* может принимать только значения из множества {zero, one, two, three, four, five}.

Основные операции с данными перечисляемого типа:

- присваивание переменных и констант одного типа;
- сравнение для выявления равенства либо неравенства.

Практическое назначение перечисления – определение множества различающихся символических констант целого типа.

Пример использования переменных перечисляемого типа:

...

```
typedef enum {  
mo=1, tu, we, th, fr, sa, su  
} days;
```

```
void main(void)
```

```
{
```

```
days w_day; // Переменная перечисляемого типа
```

```
int t_day, end, start;
```

```
// Текущий день недели, начало и конец недели соответственно
```

```
puts(" Введите день недели (от 1 до 7) : ");
```

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

```
w_day = su;
```

```
start = mo;
```

```
end = w_day - t_day;
```

```
printf("\n Понедельник – %d день недели, \
```

```
сейчас %d – й день и \n\
```

```
до конца недели %d дн. ", start, t_day, end );
```

```
}
```

Результат работы программы:

Введите день недели (от 1 до 7) : 5

*Понедельник – 1 день недели, сейчас 5-й день и
до конца недели 2 дн.*

Битовые поля – это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа

«да/нет». Минимальная адресуемая ячейка памяти – 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа), не превышающая разрядности поля типа *int*:

```
struct fields {  
    unsigned int flag: 1;  
    unsigned int mask: 10;  
    unsigned int code: 5;    };
```

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом – по имени. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры. Следует учитывать, что операции с отдельными битами реализуются гораздо менее эффективно, чем с байтами и словами, так как компилятор должен генерировать специальные коды и экономия памяти под переменные оборачивается увеличением объема кода программы. Размещение битовых полей в памяти зависит от компилятора и аппаратуры. В основном битовые поля размещаются последовательно в поле типа *int*, а при нехватке места для очередного битового поля происходит переход на следующее поле типа *int*. Возможно объявление безымянных битовых полей, а длина поля 0 означает необходимость перехода на очередное поле *int*:

```
struct areas {  
    unsigned f1: 1;  
        : 2; – безымянное поле длиной 2 бита;  
    unsigned f2: 5;  
        : 0 – признак перехода на следующее поле int;  
    unsigned f3:5;  
        double data; char buffs[100]; } ; // структура может  
содержать элементы
```

Битовые поля могут использоваться в выражениях как целые числа соответствующей длины поля разрядности в двоичной системе исчисления. Единственное отличие этих полей от обычных объектов – запрет операции определения адреса (&). Следует учитывать, что использование битовых полей снижает быстродействие программы по сравнению с представлением данных в полных полях из-за необходимости выделения битового поля.

2.10. Файлы.

Аналогом понятия внутреннего файла в языке СИ является понятие потока. Поток в СИ не ставится в соответствие тип. *Поток – это байтовая последовательность, передаваемая в процессе ввода-вывода.*

Поток должен быть связан с каким-либо внешним устройством или файлом на диске.

Основные отличия файлов в СИ состоят в следующем: здесь отсутствует понятие типа файла и, следовательно, фиксированной структуры записи файла. Любой файл рассматривается как байтовая последовательность:

байт 0	байт 1	байт 2	...	EOF
--------	--------	--------	-----	-----

EOF – стандартная константа, обозначающая признак конца файла.

Существуют стандартные потоки и потоки, объявляемые в программе. Последние обычно связываются с файлами на диске, создаваемыми программистами. Стандартные потоки назначаются и открываются системой автоматически. С началом работы любой программы открываются 5 стандартных потоков, из которых основными являются следующие:

- `stdin` – поток стандартного ввода (обычно связан с клавиатурой);
- `stdout` – поток стандартного вывода (обычно связан с дисплеем);
- `stderr` – вывод сообщений об ошибках (связан с дисплеем).

Поток для работы с дисковым файлом должен быть открыт в программе.

Работа с дисковым файлом начинается с объявления указателя на поток. Формат такого объявления:

`FILE *имя_указателя;`

Например, `FILE *fp;`

Слово `FILE` является стандартным именем структурного типа, объявленного в заголовочном файле `stdio.h`. В структуре `FILE` содержится информация, с помощью которой ведется работа с

потоком, в частности: указатель на буфер, указатель (индикатор) текущей позиции в потоке и т.д.

Следующий шаг – открытие потока, которое производится с помощью стандартной функции **fopen()**. Эта функция возвращает конкретное значение для указателя на поток и поэтому ее значение присваивается объявленному ранее указателю. Соответствующий оператор имеет формат:

```
имя_указателя=fopen(имя_файла, режим_открытия);
```

Параметры функции **fopen()** являются строками, которые могут быть как константами, так и указателями на символьные массивы. Например,

```
fp=fopen("test.dat","r");
```

Здесь **test.dat** – это имя физического файла в текущем каталоге диска, с которым теперь будет связан поток с указателем **fp**. Параметр режима **r** означает, что файл открыт для чтения.

Существуют следующие режимы открытия файла и соответствующие им параметры:

<i>Параметр</i>	<i>Режим</i>
r	открыть для чтения
w	создать для записи
a	открыть для добавления
r+	открыть для чтения и записи
w+	создать для чтения и записи
a+	открыть для добавления или создать для чтения и записи

Открытие уже существующего файла для записи ведет к потере прежней информации в нем. Если такой файл еще не существовал, то он создастся. Открывать для чтения можно только существующий файл.

Поток может быть открыт либо для текстового, либо для двоичного режима обмена.

Смысл понятия текстового файла: это последовательность символов, которая делится на строки специальными кодами – возврат каретки (код 13) и перевод строки (код 10). Если файл открыт в текстовом режиме, то при чтении из такого файла комбинация символов «возврат каретки – перевод строки» преобразуется в один символ **\n** – переход к новой строке.

При записи в файл осуществляется обратное преобразование. При работе с двоичным файлом никаких преобразований символов не происходит, т.е. информация переносится без всяких изменений.

Указанные выше параметры режимов открывают текстовые файлы. Если требуется указать на двоичный файл, то к параметру добавляется буква b. Например, rb, wb или r+b. В некоторых компиляторах текстовый режим обмена обозначается буквой t, т.е. записывается a+t или rt.

Если при открытии потока по какой-либо причине возникла ошибка, то функция fopen() возвращает значение константы NULL. Эта константа также определена в файле stdio.h. ошибка может возникнуть из-за отсутствия открываемого файла на диске, нехватки места в динамической памяти и т.п. Поэтому желательно контролировать правильность прохождения процедуры открытия файла:

```
FILE *fp;  
If (fp=fopen("test.dat","r")==NULL  
{ puts("Не могу открыть файл\n");  
return; }
```

В случае ошибки программа завершит выполнение с закрытием всех ранее открытых файлов.

Закрытие файла осуществляет функция fclose(), прототип которой имеет вид:

int fclose(FILE *fptr); Здесь fptr обозначает формальное имя указателя на закрываемый поток. Функция возвращает ноль, если операция закрытия прошла успешно. Другая величина означает ошибку.

Запись и чтение символов

Запись символов в поток производится функцией putc() с прототипом

int putc(int ch, FILE *fptr); Если операция прошла успешно, то возвращается записанный символ. В случае ошибки возвращается константа EOF.

Считывание символа из потока, открытого для чтения, производится функцией gets() с прототипом

```
int gets(FILE *fptr);
```

Функция возвращает значение считываемого из файла символа. Если достигнут конец файла, то возвращается значение EOF. Это происходит лишь в результате чтения кода EOF.

Функция gets() возвращает значение типа int. То же самое можно сказать и про аргумент ch в описании функции puts(). Используется же в обоих случаях только младший байт. Поэтому обмен при обращении может происходить и с переменными типа char.

Пример 1. Составить программу записи в файл символьной последовательности, вводимой с клавиатуры. Пусть признаком завершения ввода будет символ *.

```
//Запись символов в файл
#include <stdio.h>
void main()
{ FILE *fp; char c;
  if (( fp=fopen("test.dat","w"))==NULL)
  {puts("Не могу открыть файл! \n"); return;}
  puts("Вводите символы. Признак конца – *");
  while ((c=getchar())!="*")  putc(c,fp); fclose(fp); }
```

В результате на диске (в каталоге, определяемом системой) будет создан файл с именем test.dat, который заполнится вводимыми символами. Символ * в файл не запишется.

Пример 2. Файл требуется последовательно прочитать и содержимое вывести на экран.

```
//Чтение символов из файла
#include <stdio.h>
#include <conio.h>
void main()
{ FILE *fp; char c;
  clrscr();
  if (( fp=fopen("test.dat","r"))==NULL)
  {puts("Не могу открыть файл! \n"); return;}
  while ((c=getc(fp))!=EOF) putchar(c);
  fclose(fp);
}
```

Запись и чтение целых чисел

Запись целых чисел в поток без преобразования их в символьную форму производится функцией `putw()` с прототипом

```
int putw(int, FILE *fptr);
```

Если операция прошла успешно, то возвращается записанное число. В случае ошибки возвращается константа `EOF`.

Считывание целого числа из потока, открытого для чтения, производится функцией `getw()` с прототипом `int getw(FILE *fptr);`

Функция возвращает значение считываемого из файла числа. Если прочитан конец файла, то возвращается значение `EOF`.

Специальные функции обмена с файлами имеются только для символьного и целого типов данных. В общем случае используются функции чтения и записи блоков данных. С их помощью можно записывать в файл и читать из файла вещественные числа, массивы, строки, структуры. При этом сохраняется форма внутреннего представления данных.

Функция записи блока данных имеет прототип

```
int fread(void*buf, int bytes, int n, FILE*fptr);
```

Здесь

`buf` – указатель на адрес данных, записываемых в файл; `bytes` – длина в байтах одной единицы записи (блока данных);

`n` – число блоков, передаваемых в файл;

`fptr` – указатель на поток.

Если запись выполнялась благополучно, то функция возвращает число записанных блоков (значение `n`).

Функция чтения блока данных из файла имеет прототип

```
int fwrite(void*buf, int bytes, int n, FILE*fptr);
```

Пример 3. следующая программа организует запись блоков в файл строки (символьного массива), а также чтение и вывод на экран записанной информации.

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

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

```
void main()
```

```
{ FILE *stream;
```

```
char msg[ ]="this is a test";
```

```
char buf[20];
```



```

if (( stream=fopen("DUMMY.FILL","w+"))==NULL)
{puts("Не могу открыть файл \n"); return;}
// Запись строки в файл fwrite(msg, strlen(msg)+1, 1, stream);
// Установка указателя на начало файла
fseek(stream, 0, SEEK_SET);
// Чтение строки из файла
fread(buf, strlen(msg)+1, 1, stream);
printf("%s \n",buf);
fclose(stream);
}

```

В этой программе поток открывается в режиме w+ (создание для записи с последующим чтением). Поэтому закрывать файл после записи не потребовалось. Новым элементом данной программы по сравнению с предыдущими является использование функции установки указателя потока в заданную позицию. Ее формат

```
int fseek(указатель_на поток, смещение, начало_отсчета);
```

Начало отсчета задается одной из констант, определенных в файле stdio.h:

SEEK_SET (имеет значение 0) – начало файла;

SEEK_CUR (имеет значение 1) – текущая позиция;

SEEK_END (имеет значение 2) – конец файла.

Смещение определяет число байт, на которое надо сместить указатель относительно заданного начала отсчета. Смещение может быть как положительным, так и отрицательным числом. Оба параметра имеют тип long.

Форматный обмен с файлами

С помощью функции форматного вывода можно формировать на диске текстовый файл с результатами вычислений, представленными в символьном виде. В дальнейшем этот файл может быть просмотрен на экране, распечатан на принтере, отредактирован с помощью текстового редактора. Общий вид функции форматного вывода:

```
int fprintf (указатель_на_поток, форматная_строка,          спи-
сок_переменных);
```

Используемая нами ранее функция printf () для организации вывода на экран является частным вариантом функции fprintf (). Функция printf () работает лишь со стандартным потоком stdin,

который всегда связывается системой с дисплеем. Не будет ошибкой, если в программе вместо `printf()` написать `fprintf(stdin, ...)`.

Правила использования спецификаторов форматов при записи в файлы на диске точно такие же, как и при выводе на экран.

Пример 4. Составим программу, по которой будет рассчитана и записана в файл таблица квадратных корней для целых чисел от 1 до 10. Для контроля эта же таблица выводится на экран.

```
//Таблица квадратных корней
#include <stdio.h>
#include <iostream.h>
#include <math.h>
void main()
{ FILE *fp;
  int x;
  fp = fopen("test.dat", "w");
  //Вывод на экран и в файл шапки таблицы
  printf("\t Таблица квадратных корней \n");
  fprintf(fp, "\t Таблица квадратных корней \n ");
  printf("\t x\t\tsqrt(x) \n");
  fprintf(fp, "\t x\t\tsqrt(x) \n ");
  \\Вычисление и вывод таблицы квадратных корней
  \\на экран и в файл
  for(x = 1; x<=10; x++)
  { printf("\t%f\t%f\n", float(x), sqrt(float(x)));
    fprintf(fp, "\t%f\t%f\n", float(x), sqrt(float(x)));
  }
  fclose(fp); }
```

Форматный ввод из текстового файла осуществляется с помощью функции `fscanf()`, общий формат которой выглядит следующим образом:

```
int fscanf(указатель_на_поток, форматная_строка, список_адресов_переменных);
```

Данной функцией удобно пользоваться в тех случаях, когда исходные данные заранее подготавливаются в текстовом файле.

Создать программу, в которой реализованы создание, добавление и просмотр файла, содержащего информацию о фамилии и среднем балле студентов. Процесс добавления информации заканчивается при нажатии точки.

```
#include <stdio.h>
#include <stdlib.h>
struct Sved {
    char Fam[30];
    double S_Bal;
} zap,zapt;
char Spis[]="c:\\work\\Sp.dat";
FILE *F_zap;
FILE* Open_file(char*, char*);
void main (void)
{
    int i, j, kodR, size = sizeof(Sved), kod_read;
    while(1) {
        puts("Создать – 1\n Добавить – 3\n Просмотреть – 2\n Выход – 0");
        scanf("%d",&kodR);
        switch(kodR) {
            case 1:
            case 3:
                if(kodR==1) F_zap = Open_file (Spis,"w+");
                else F_zap = Open_file (Spis,"a+");
                while(2) {
                    puts("\n Fam (. – end) ");
                    scanf("%s",zap.Fam);
                    if((zap.Fam[0])=='.') break;
                    puts("\n Ball: ");
                    scanf("%lf",&zap.S_Bal);
                    fwrite(&zap,size,1,F_zap);
                }
                fclose(F_zap);
                break;
            case 2: F_zap = Open_file (Spis,"r+"); int nom=1;
                while(2) {
                    if(!fread(&zap,size, 1, F_zap)) break;
```

```

        printf("%2d: %20s %5.2lf\n",
               nom++, zap.Fam, zap.S_Bal);
    }
    fclose(F_zap);
    break;
case 0: return;          // exit(0);
}                        // Закрывает switch()
}                        // Закрывает while()
}

// Функция обработки ошибочной ситуации при открытии файла
FILE* Open_file(char *file, char *kod)
{
FILE *f;
    if(!(f = fopen(file, kod))) {
        puts("Open File Error!");
        exit(1);
    }
    return f;
}

```