**40.Числа с плавающей точкой. Описание переменных с плавающей точкой. Перечисляемый тип данных (enum).**

В языке Си такие данные описываются типом **float** - с плавающей точкой одинарной точности; **double** − с плавающей точкой двойной точности;**long** **double** - тип с плавающей точкой повышенной точности;

**Описание переменных с плавающей точкой.**

Переменные с плавающей точкой описываются и инициализируются точно таким же образом, что и переменные целого типа.

**Перечисляемый тип данных (enum)**

При объявлении переменной перечисляемого типа с ней связывается набор именованных целочисленных констант, называемых перечисляемыми константами. В каждый момент времени переменная может иметь значение одной из констант, и к ней можно обращаться по имени.

Все константы и переменные имеют тип int, каждая константа автоматически получает по умолчанию некоторое значение, если не указывается какое-либо другое

**41.Модификаторы доступа. Модификатор const. Определение констант через #define. Модификатор volatile. Cовместное использование const и volatile.**

**Модификаторы доступа.**

Модификаторы const и volatile являются новыми в С и C++. Они были добавлены в стандарте ANSI С для того, чтобы различать переменные, которые никогда не изменяются **(const),** и переменные, значение которых может измениться непредсказуемо **(volatile).**

**Модификатор const.**

Иногда необходимо использовать некоторою величину, которая не изменяется на протяжении всей программы. Такая величина называется константой. В С и C++ для объявления константы перед ключевым словом (int, float или double) пишется const

**Определение констант через #define**

В С и C++ существует другой способ описания констант: директива препроцессора **#define**. Общий формат данной строки следующий: директива #define, литерал sales\_team(имя константы), литерал 10 (значение константы). Встретив такую команду, препроцессор осуществит глобальную замену в программном коде имени *sales\_team* числом 10. Причем никакие другие значения идентификатору *sales\_team* не могут быть присвоены, поскольку переменная с таким именем в программе формально не описана.

**Модификатор volatile**

Ключевое слово **volatile** указывает на то, что данная переменная в любой момент может быть изменена в результате выполнения внешних действий, не контролируемых программой. **volatile int event\_time;**

Подобное описание переменной может понадобиться в том случае, когда переменная обновляется системными устройствами, например таймером. При получении сигнала от таймера выполнение программы прерывается и значение переменной изменяется.

**Одновременное использование const и volatile**

Допускается одновременное употребление ключевых слов **const** и **volatile** при объявлении переменных. Так, в следующей строке создается переменная, обновляемая извне, но значение которой не может быть изменено самой программой:

**const volatile** *соnstant\_event\_time***;**

**42. Модификаторы pascal, cdecl, near, far и huge. Модификатор pascal. Модификатор cdecl. Модификаторы near, far и huge.**

Так как встроенный компоновщик Microsoft Visual C/C++ различает (по умолчанию) заглавные и строчные буквы, предполагается, что все внешние идентификаторы, объявленные в программе, сохраняют символ подчеркивания и тот вид, который они имели при объявлении (название и регистр букв).

**Pascal**

Применение модификатора **pascal** к идентификатору приводит к тому, что идентификатор преобразуется к верхнему регистру и к нему не добавляется символ подчеркивания. Этот идентификатор может использоваться для именования в программе на языке Си глобального объекта, который используется также в программе на языке Паскаль. В объектном коде, сгенерированном компилятором языка Си, и в объектном коде, сгенерированном компилятором языка Паскаль, идентификатор будет представлен идентично.Функции типа **pascal** не могут иметь переменное число аргументов, как, например, функция **printf**. Поэтому нельзя использовать завершающее многоточие в списке параметров функции типа **pascal**.

**Cdecl**

Существует опция компиляции, которая присваивает всем функциям и указателям на функции тип **pascal**. Это значит, что они будут использовать вызывающую последовательность, принятую в языке Паскаль, а их идентификаторы будут приемлемы для вызова из программы на Паскале. При этом можно указать, что некоторые функции и указатели на функции используют вызывающую последовательность, принятую в языке Си, а их идентификаторы имеют традиционный вид для идентификаторов языка Си. Для этого их объявления должны содержать модификатор **cdecl**.

**Near, Far, Huge**

Эти модификаторы оказывают воздействие на работу с адресами объектов.

Указатель типа **near** — 16-битовый

Указатель типа **far** — 32-битовый

Указатель типа **huge** — 32-битовый

—операции отношения ==, !=, <, >, <=, >= выполняются корректно и предсказуемо над указателями типа **huge**, но не над указателями типа **far**;

—при использовании указателей типа **huge** требуется дополнительное время, т. к. программы нормализации должны вызываться при выполнении любой арифметической операции над этими указателями. Объем кода программы также возрастает.

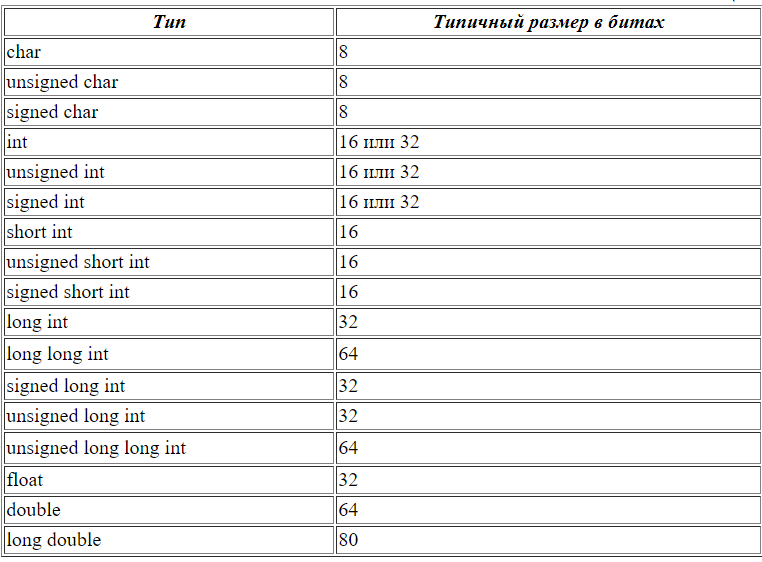
В СП MSC модификатор **huge** применяется только к массивам, размер которых превышает 64 К. В СП ТС недопустимы массивы больше 64 К, а модификатор **huge** применяется к функциям и указателям для спецификации того, что адрес функции или указуемого объекта имеет тип **huge**.

Для вызова функции типа **near** используются машинные инструкции ближнего вызова, для типов **far** и **huge** — дальнего.

**43. Тип данных char Другие типы и размеры данных. Преобразование типов данных. Явные преобразования типов при помощи операции приведения типа.**

**Тип данных char**

Этот тип определяет целые числа без знака в диапазоне от 0 до 255. Обычно такое целое размещается в одном байте памяти.



**Преобразование типов данных.**

Предположим, что выполняется следующая операция, в которой переменные **fresult** и **fvalue** имеют тип **float**, а переменная **ivalue** — тип **int**:

**fresult = fvalue \* ivalue;**

 Следовательно, это операция смешанного типа. При выполнении оператора значение ivalue будет преобразовано в число с плавающей точкой, а затем выполнится умножение. Компилятор распознает смешанные операции и генерирует код для выполнения следующих операций.

**Явные преобразования типов при помощи операции приведения типа.**

в некоторых случаях, несмотря на то, что автоматическое преобразование не выполняется, преобразование типов было бы желательным. В таких случаях необходимо особо указать, что требуется изменение типа. Кроме того, подобные явные указания облегчают другим программистам понимание используемых операторов. В языке С имеется несколько процедур, позволяющих указывать необходимость преобразования типа.

Одна из таких процедур называется приведением типа. Если нужно временно изменить формат некоторой переменной, достаточно перед ее именем указать в скобках имя типа, который необходимо получить. Например: если переменные **ivalue1** и **ivalue2** объявлены как int, а переменные **fvalue** и **fresult** как float, то три следующих оператора будут эквивалентны:

 Все три оператора выполняют преобразование в тип с плавающей точкой и деление переменных **ivalue1** и **ivalue2**. Третий оператор явно указывает, какие операции должны выполняться.

**44. Символьные строки. Массивы символов в C++ Функции работы со строками символов.**

**Символьные строки.**

«Символьная строка» — это последовательность, состоящая из одного или более символов. Кавычки не являются частью строки. Они вводятся только для того, чтобы отметить ее начало и конец, т. е. играют ту же роль, что и апострофы в случае одиночного символа.

В языке Си нет специального типа, который можно было бы использовать для описания строк. Вместо этого строки представляются в виде «массива» элементов типа char

Необходимо отметить, что на рисунке последним элементом массива является символ \0. Это «нуль-символ», и в языке Си он используется для того, чтобы отмечать конец строки.

Библиотека С++ содержит функции копирования строк (strcpy, strncpy), сравнения (strcmp, strncmp), объединения строк (strcat, strncat), поиска подстроки (strstr), поиска вхождения символа (strchr, strnchr, strbrk) определения длины строки strlen и другие.

**45. Символьные строки. Определение длины строк Копирование и конкатенация строк Сравнение строк Преобразование строк**

| **Функция** | **Пояснение** |
| --- | --- |
| **strlen(**имя\_строки**)** | определяет длину указанной строки, без учёта нуль-символа |
| **Копирование строк** | |
| **strcpy(**s1,s2**)** | выполняет побайтное копирование символов из строки  s2 в строку s1 |
| **strncpy(**s1,s2, n**)** | выполняет побайтное копирование n символов из строки  s2 в строку s1. возвращает значения s1 |
| **strcat(**s1,s2**)** | объединяет строку s2 со строкой s1. Результат сохраняется в s1 |
| **strncat(**s1,s2,n**)** | объединяет n символов строки s2 со строкой s1. Результат сохраняется в s1 |
| **Сравнение строк** | |
| **strcmp(**s1,s2**)** | сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 С учётом регистра |
| **strncmp(**s1,s2**)** | сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 С учётом регистра |
| **stricmp(**s1,s2**)** | сравнивает строку s1 со строкой s2 и возвращает результат типа **int**: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 Без учёта регистра |
| **strnicmp(**s1,s2**)** | сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2,  <0 — если s1>s2 Без учёта регистра |
| **Функции преобразования** | |
| **atof(**s1**)** | преобразует строку s1 в тип double |
| **atoi(**s1**)** | преобразует строку s1 в тип int |
| **atol(**s1**)** | преобразует строку s1 в тип long int |

**46.Символьные строки. Обращение строк Поиск символов Поиск подстрок**

Функция обращения строки **strrev()** меняет порядок следования символов на обратный (реверс строки). Данная функция имеет прототип:

**char\* strrev(char\* str)**

Следующий пример демонстрирует работу функции **strrev().**

**Поиск символа в строке.**

Для поиска символа в строке используются функции **strchr, strrchr, strspn, strcspn и strpbrk**.

**Функция char\*  strchr ( const  char  \*str, int  c );**

ищет первое вхождение символа, заданного параметром c, в строку str. В случае успеха функция **возвращает указатель на первый найденный символ, а в случае неудачи – NULL.**

**Функция \*  strrchr ( const  char  \*str, int  c );**

ищет последнее вхождение символа, заданного параметром c, в строку str. В случае успеха функция возвращает указатель на последний найденный символ, а в случае неудачи – NULL.

**Функция size\_t  strspn ( const  char  \*str1, const  char  \*str2 );**

возвращает индекс первого символа из строки  str1, который не входит в строку str2.

 Функция \_t  strcspn ( const  char  \*str1, const  char  \*str2 );

возвращает индекс первого символа из строки  str1, который входит в строку str2.

 Функция char\*  strpbrk ( const  char  \*str1, const  char  \*str2 );

находит первый символ в строке str1, который равен одному из символов в строке str2. В случае успеха функция возвращает указатель на этот символ, а в случае неудачи – NULL.

**Поиск подстрок**.

Стандартная библиотека предлагает воспользоваться функцией **strstr ()**.

Функция **strstr ()** описана следующим образом:

**char\* strstr{const char\* str, const char\* substr)**

**47.Функции преобразования типа Функции printf() и scanf(). Использование функции printf(). Модификаторы спецификации преобразования, используемые в функции printf(). Использование функции printf() для преобразования данных. Применение функции scanf().**

**Функции printf() и scanf().**

Функции **printf()** **и scanf()** дают нам возможность взаимодействовать с программой. Мы называем их функциями ввода-вывода.

**Модификаторы спецификации преобразования, используемые в функции printf().**

Мы можем несколько расширить основное определение спецификации преобразования, поместив **модификаторы** между знаком **% и символом, определяющим тип преобразования**. В приводимой ниже таблице дан список тех символов, которые вы имеете право туда поместить.

int main()

{

printf("/%ld/\n", 336);

printf("/%5.3d/\n", 336);

printf("/%10d/\n", 336);

printf("/%-10d/\n", 336);}

**Использование функции printf() для преобразования данных.**

мы можем использовать спецификации, применяемые для функции **printf() с** целью **преобразования десятичных чисел в восьмеричные или шестнадцатеричные и наоборот**. Или же если вы захотите напечатать данные в желаемом для вас виде, то необходимо указать спецификацию **%d для получения десятичных чисел, %о — для восьмеричных, а %х— для шестнадцатеричных.** При этом не имеет ни малейшего значения, в какой форме число первоначально появилось в программе.

**Применение функции scanf().**

Так же как для функции **printf(),** для функции **scanf()** указываются управляющая строка и следующий за ней список аргументов. Функция printf() использует имена переменных, константы и выражения, в то время как функция scanf() — только указатели на переменные. К счастью, при применении этой функции мы ничего не должны знать о таких указателях. Необходимо помнить только два правила:

1. Если вам нужно ввести некоторое значение и присвоить его переменной одного из основных типов, то перед именем переменной требуется писать символ &.

2. Если вы хотите ввести значение строковой переменной, использовать символ & не нужно.

**48.Основные операции. Операция присваивания: =. Операция сложения: +. ерациявычитания: -. ООперация изменения знака: -. Операция умножения: \*. Операция деления: /.**

Основные арифметические операции: сложения, вычитания, умножения, деления. Операции в языке Си применяются для представления арифметических действий.

**Операция присваивания "="**

Он означает *операцию присваивания* некоторого значения.

= **операцией присваивания**

**Операция сложения: +**

Выполнение операции **+** приводит к сложению двух величин, стоящих слева и справа от этого знака.

**Операция + называется "бинарной", или "диадической**". Эти названия отражают тот факт, что она имеет дело с **двумя *операндами*.**

**Операция вычитания: -**

Выполнение операции вычитания приводит к вычитанию числа, расположенного справа от знака -, из числа, стоящего слева от этого знака.

**Операция изменения знака: -**

Знак минус используется для указания или изменения алгебраического знака некоторой величины.

**get = -teg;**

Когда знак используется подобным образом, данная операция называется **"унарной".** Такое название указывает на то, что она имеет дело только с одним *операндом*.

**Операция умножения: \***

Операция умножения обозначается знаком \*.

**Операция деления: /**

В языке Си символ / указывает на операцию деления. Величина, стоящая слева от этого знака, делится на величину, расположенную справа от этого знака. Например, в результате выполнения оператора

**l = 126.0 / 2.0;**

**переменной l будет присвоено значение 63.0**. Над данными целого типа операция деления производится не так, как над данными с плавающей точкой: в первом случае результат будет целым числом, а во втором - числом с плавающей точкой. В языке Си принято правило, согласно которому дробная часть у результата деления целых чисел отбрасывается. Это действие называется "усечением".

**49.Основные операции. Поразрядные операции. Поразрядное И (AND). Поразрядное ИЛИ (OR). Поразрядное исключающее ИЛИ (OR).**

**В С++ существуют три поразрядные логические операции:**

1. поразрядное **И**, обозначение: &
2. поразрядное исключающее **ИЛИ**, обозначение:  ^
3. поразрядное включающее **ИЛИ**, обозначение:  |

Так же в С++ существуют [логические операции](http://cppstudio.com/obuchenie_cpp/logicheskie-operatsii): **ИЛИ** — ||; **И** — &&. У многих возникает вопрос, “Чем отличаются операции: & и &&; | и || ?”. Ответ на этот вопрос можно получить, если понять принцип работы поразрядных логических операций. Сразу могу сказать одно, что логические операции && и || используются только для построения логических условий. Тогда как поразрядные логические операции применяются в бинарной арифметике. Данные операции работают с битами ячеек памяти, причём операнды и результат могут быть заданы в другой форме, например, в десятичной.  Дальше рассмотрим каждую из операций подробно.

**Поразрядная логическая операция И.**  
Обозначения: X, Y – операнды; F – результат выполнения логической операции



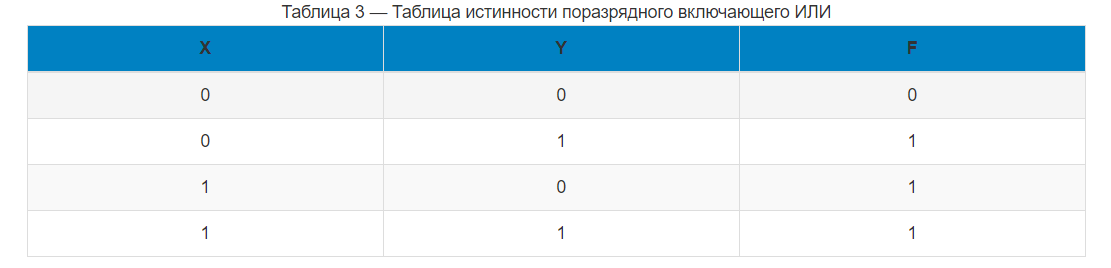
Из таблицы истинности видно, что результат будет нулевым, если хотя бы один из битов 0. Если оба бита равны 1, то результат равен 1.

**Поразрядное исключающее ИЛИ.**



Из таблицы истинности видно, что результат будет нулевым, если оба бита будут равны, во всех остальных случаях результат равен 1.

**Поразрядное включающее ИЛИ.**



Из таблицы истинности видно, что результат будет нулевым, если оба бита будут равны 0, во всех остальных случаях результат равен 1.

**50. Основные операции. Поразрядный сдвиг влево и вправо. Операции отношения и ллогические операции. Условная операция (?:). Операция запятая (,). Порядок выполнения операций.**

**Операции сдвига влево и вправо**

**Сдвиг влево (‹‹)**

Задача данного оператора cдвинуть биты своего операнда на какое-то количество влево, начиная с младшего правого бита. Пустые места после сдвига заполняются нулями. Если размера числа будет недостаточно чтобы сохранить сдвинутые биты в левую сторону, они будут утеряны.  
**Сдвиг вправо (››)**

Задача данного оператора cдвинуть биты своего операнда на какое-то количество вправо, начиная со старшего левого бита. Сдвиг вправо числа типа unsigned всегда заполняет освободившиеся биты нулями. Сдвиг вправо числа со знаком в некоторых системах приводи к заполнению этих битов значениями знакового бита, а в других нулями. Все младшие биты будут утеряны.

**Операции отношений и логические операции.**

**Логические операции**

Перечень логических операций в порядке убывания относительного приоритета и их обозначения:

! - отрицание (логическое НЕТ);

&& - коньюнкция (логическое И);

|| - дизьюнкция (логическое ИЛИ).

**Операции сравнения**

== - равно или эквивалентно;

!= - не равно;

< - меньше;

<= - меньше либо равно;

> - больше;

>= - больше либо равно.

***Операция*«*,*»*(запятая)***

Данная операция используется для организации заданной последовательности вычисления выражений (обычно используется там, где по синтаксису допустима только одна операция, а необходимо разместить две и более, например, в операторе *for*). Форма записи:

***выражение*\_1, …, *выражение*\_*N*;**

***Условная операция «? :»***

Если одно и то же выражение (или переменная) вычисляется по-разному в зависимости от некоторого условия, вместо оператора *if* можно использовать более короткую запись - *условную операцию*. Эта операция – ***тернарная***, т.е. в ней участвуют три операнда. Формат написания условной операции следующий:

***Выражение*1?*выражение*2 :*выражение*3;**

если *выражение*1 (условие) отлично от нуля (истинно), то результатом операции является значение *выражения*2, в противном случае – значение *выражения*3. Каждый раз вычисляется только одно из выражений 2 или 3.

**51.Дополнительные операции. Операция деления по модулю: %. Операции увеличения и уменьшения: ++. Операция уменьшения: --. Старшинство операций.**

В языке Си имеется около сорока операций. Те операции, которые мы рассмотрели, являются наиболее общеупотребительными. Рассмотрим еще три операции, наиболее используемые программистами.

**Операция деления по модулю: %**

Эта операция используется в целочисленной арифметике. Ее результатом является остаток от деления целого числа, стоящего слева от знака операции, на число, расположенное справа от нее.

**Операция увеличения: ++**

Операция увеличения осуществляет следующее простое действие: она увеличивает значение своего ***операнда*** на единицу. Существуют две возможности использования данной операции: первая, когда символы ++ находятся слева от переменной ( *операнда* ), - **"префиксная" форма**, и вторая, когда символы ++ стоят справа от переменной, - **"постфиксная" форма**. Эти две формы указанной операции различаются между собой только тем, в какой момент осуществляется увеличение *операнда*. Префиксная форма изменяет значение *операнда* перед тем, как *операнд* используется. Постфиксная форма изменяет значение после того как *операнд* использовался.

**Операция уменьшения: --**

Каждой операции увеличения соответствует некоторая операция уменьшения, при этом вместо символов ++ мы используем --. Когда символы -- находятся слева от *операнда* - **"префиксная" форма** операции уменьшения. Если символы -- стоят справа от *операнда* - это **"постфиксная" форма** операции уменьшения.

**Старшинство операций**

**52. Выражения и операторы. Выражения. Операторы. Составные операторы (блоки).**

**Выражения**

Выражения представляют собой множество данных связанных между собой операциями - особыми операторами языка, возвращающих некоторое значение. Аргументы операций называют операндами. Большинство операций либо унарные (с одним операндом) или бинарные (с двумя операндами). Также операции характеризуются приоритетом (старшинством) выполнения в выражении.

**Операторы**

Операторы программы на языке С управляют процессом ее выполнения. В языке С, как и в других языках программирования, имеется ряд операторов, с помощью которых можно выполнять циклы, указывать другие операторы для выполнения и передавать управление на другой участок программы

Операторы языка С состоят из ключевых слов, выражений и других операторов. В операторах языка С появляются следующие ключевые слова:

**Составной оператор**

К составным операторам относят собственно составные операторы и блоки. В обоих случаях это последовательность операторов, заключенная в фигурные скобки. Блок отличается от составного оператора наличием определений в теле блока. Например:

**53.Ввод и вывод одного символа: функции getchar() и putchar(). Буферы. Чтение данных. Чтение строки. Чтение файла.**

**Ввод и вывод одного символа: функции getchar() и putchar().**

Функция getchar() получает один символ, поступающий с пульта терминала (и поэтому имеющий название), и передает его выполняющейся в данный момент программе. Функция putchar() получает один символ, поступающий из программы, и пересылает его для вывода на экран.

#include <stdio.h>

void main()

{

char ch;

ch = getchar();

putchar (ch);

}

**Буферы.**

При выполнении данной программы вводимый символ в одних вычислительных системах немедленно появляется на экране («эхо-печать»), в других же ничего не происходит до тех пор, пока вы не нажмете клавишу [ввод]. Первый случай относится к так называемому «небуферизованному» (прямому) вводу, означающему, что вводимый символ оказывается немедленно доступным ожидающей программе. Например, работа программы в системе, использующей буферизованный ввод, будет выглядеть следующим образом:

**Вот длинная входная строка. [ввод]**

**В**

В системе с небуферизованным вводом отображение на экране символа В произойдет сразу, как только вы нажмете соответствующую клавишу. Результат ввода-вывода при этом может выглядеть, например, так:

**В Вот длинная входная строка.**

Символ В, появившийся на второй позиции данной строки, — это непосредственный результат Зачем нужны буферы? Во-первых, оказывается, что передачу нескольких символов в виде одного блока можно осуществить гораздо быстрее, чем передавать их последовательно по одному. Во-вторых, если при вводе символов допущена ошибка, вы можете воспользоваться корректирующими средствами терминала, чтобы ее исправить. И когда в конце концов вы нажмете клавишу [ввод], будет произведена передача откорректированной строки.

**Чтение данных.**

#include <stdio.h>

#define STOP '\*'

void main(void)

{

char ch;

ch = getchar(); // строка 9

while (ch != STOP) { // строка 10

putchar(ch); // строка 11

ch = getchar(); //строка 12

}

}

При первом прохождении тела цикла функция putchar() получает значение своего apгумента в результате выполнения оператора, расположенного в строке 9; в дальнейшем, вплоть до завершения работы цикла, значением этого аргумента является символ, передаваемый программе функцией getchar(), расположенной в строке 12. Мы ввели новую операцию отношения !=, смысл которой выражается словами «не равно». В результате всего этого цикл while будет осуществлять чтение и печать символов до тех пор, пока не поступит признак STOP. Мы могли бы опустить в программе директиву #define и использовать лишь символ \* в операторе while, но наш способ делает смысл данного знака более очевидным.

**Чтение строки.**

Одной из возможностей является использование символа «новая строка» (\n). Для этого нужно лишь переопределить признак STOP:

**#define STOP '\n'**

**Чтение файла.**

Каким может быть идеальный признак STOP? Это должен быть такой символ, который обычно не используется в тексте и, следовательно, не приводит к ситуации, когда он случайно встретится при вводе, и работа программы будет остановлена раньше, чем мы хотели бы.

. Файлом можно назвать участок памяти, в который помещена некоторая информация. Обычно файл хранится в некоторой долговременной памяти, например, на гибких или жестких дисках, или на магнитной ленте. Чтобы отмечать, где кончается один файл и начинается другой, полезно иметь специальный символ, указывающий на конец файла. Это должен быть символ, который не может появиться где-нибудь в середине файла, точно так же как выше нам требовался символ, обычно не встречающийся во вводимом тексте. Решением указанной проблемы служит введение специального признака, называемого «End-of-File» (конец файла), или EOF, для краткости. Выбор конкретного признака EOF зависит от типа системы: он может состоять даже из нескольких символов. Но такой признак всегда существует, и компилятор с языка Си, которым вы пользуетесь, конечно же знает, как такой признак действует в вашей системе.

Каким образом можно воспользоваться символом EOF? Обычно его определение содержится в файле **<stdio.h>.** Общеупотребительным является определение:

**Отметим следующие моменты:**

1. Нам не нужно самим определять признак EOF, поскольку заботу об этом берет на себя файл <stdio.h>.

2. Мы можем не интересоваться фактическим значением символа EOF, поскольку директива #define, имеющаяся в файле <stdio.h>, позволяет нам использовать его символическое представление.

3. При работе с данной программой, когда символы вводятся с клавиатуры, необходимо уметь вводить признак EOF. He думайте, что вы можете просто указать буквы E-O-F или число -1. (Число -1 служит эквивалентом кода ASCII данного символа, а не самим этим символом.) Вместо этого вам необходимо узнать, какое представление используется в вашей системе. В большинстве реализаций операционной системы UNIX, например, ввод знака [CTRL/D] (нажать на клавишу [D], держа нажатой клавишу [CTRL]) интерпретируется как признак EOF. Во многих микрокомпьютерах для той же цели используется знак [CTRL/Z].

**54.Переключение и работа с файлами. Операционная система UNIX. Переключение вывода. Переключение ввода. Комбинированное переключение.**

**Переключение и работа с файлами.**

**Операция переключения** — это средство ОС UNIX, а не самого языка Си. Но она оказалась настолько полезной, что при переносе компилятора с языка Си на другие вычислительные системы часто вместе с ним переносится в какой-то форме и эта операция. Даже если вы не работаете в среде ОС UNIX существует большая вероятность того, что вы в той или иной форме сможете воспользоваться операцией переключения. Мы обсудим сначала возможности этой операции в ОС UNIX, а затем и в других системах.

**Операционная система UNIX.**

**Переключение вывода.**

**Символ <** служит обозначением операции переключения, используемой в **ОС UNIX**. Выполнение указанной операции приводит к тому, что содержимое файла **words** будет направлено в файл с именем **getput4**. Сама программа ввод-вывод4 не знает (и не должна знать), что входные данные поступают из некоторого файла, а не с терминала; на ее вход просто поступает поток символов, она читает их и последовательно, по одному выводит на печать до тех пор, пока не встретит признак EOF. В операционной системе UNIX файлы и устройства ввода-вывода в логическом смысле представляют собой одно и то же, поэтому теперь файл для данной программы является устройством ввода-вывода. Если вы попробуете ввести команду

**getput4 < words**

**Переключение ввода.**

Теперь предположим, вам хочется, чтобы слова, вводимые с клавиатуры, попадали в файл с именем **mywords.** Для этого вы должны ввести команду

**getput4 > mywords**

и начать ввод символов. Символ > служит обозначением еще одной операции переключения, используемой в ОС UNIX. Ее выполнение приводит к тому, что создается новый файл с именем mywords, а затем результат работы программы ввод-вывод4, представляющий собой копию вводимых символов, направляется в данный файл. Если файл с именем mywords уже существует, он обычно уничтожается, и вместо него создается новый.

**Комбинированное переключение.**

Предположим теперь, что вы хотите создать копию файла **mywords** и назвать ее **savewords**. Введите для этого команду

**getput4 < mywords >savewords**

 и требуемое задание будет выполнено. Команда

**getput4 >savewords < mywords**

приведет к такому же результату, поскольку порядок указания операций переключения не имеет значения.

**Нельзя использовать в одной команде один и тот же файл и для ввода и для вывода одновременно.**

**55. Выбор вариантов. Операции отношения и выражения. Понятие "истина".**

**Выбор вариантов.**

**Операции отношения и выражения.**

Операции отношения используются для сравнений. Мы уже использовали ранее некоторые из них, а сейчас приведем полный список операций отношения, применяемых при программировании на языке Си.

**Операции сравнения(отношения)**

== - равно или эквивалентно;

!= - не равно;

< - меньше;

<= - меньше либо равно;

> - больше;

>= - больше либо равно.

Операции отношения применяются при формировании условных выражений, используемых в операторах if и while. Указанные операторы проверяют, истинно или ложно данное выражение. Ниже приводятся четыре не связанные между собой оператора

**Понятие «истина».**

во-первых, что выражение в Си всегда имеет значение. Это утверждение остается верным даже для условных выражений Оказывается, в языке Си значение «истина» — это 1, а «ложь» — 0.

**56.Условные операторы. Оператор if. Оператор if-else. Вложенные операторы if-else.**

**Условные операторы.**

В языке С имеется четыре основных условных оператора: **if, if-else,? и switch**. Перед тем, как обсудить их отдельно, необходимо отметить одно общее правило.

Большинство условных операторов можно использовать для выборочного выполнения или одной программной строки, или нескольких логически связанных строк (называемых блоком). Если условный оператор связан только с одной строкой исполняемого кода, эту строку заключать в фигурные скобки ({ }) не нужно. Если же условный оператор относится к блоку выполняемых операторов, то для связи блока с проверяемым условием фигурные скобки необходимы. По этой причине в операторе switch необходимы открывающая и закрывающая фигурные скобки.

**Оператор if.**

Оператор if используется для условного выполнения фрагмента кода. Простейший вид оператора if следующий:

**if (выражение)**

**действие\_если\_истина;**

 Обратите внимание на то, что выражение должно заключаться в круглые скобки. Если выражение ложно, действие\_если\_истина не выполняется, а выполняется оператор, следующий за невыполненным действием.

**if (выражение)**

**{**

**действие\_если\_истина1;**

**…**

**действие\_если\_истина4;**

**}**

**Оператор if-else.**

Оператор if-else нужен для того, чтобы программа выполнила два разных действия в зависимости от истинности некоторого выражения. В простейшем случае оператор if-else выглядит следующим образом:

**if (выражение)**

**действие\_если\_истина;**

**else**

**действие\_если\_ложь;**

Любое из выражений — **выражение\_если\_истина, выражение\_если\_ложь, или оба вместе** — могут быть составным оператором или блоком, заключенным в фигурные скобки. Синтаксис этих трех вариантов очень прост:

**Вложенные операторы if-else.**

**57.Условные операторы. Операторы if-else-if. Условный оператор ?. Оператор switch. Совместное использование операторов if-else-if и switch.**

**Операторы if-else-if.**

Комбинация операторов **if-else-if** часто используется для выполнения многочисленных последовательных сравнений. В общем виде они выглядят следующим образом:

**if(выражение 1)**

**действие\_если\_условие1\_истина;**

**else if(выражение2)**

**действие\_если\_условие2\_истина;**

**else if(выражение3)**

**действие\_если\_условиеЗ\_истина;**

**Существует правило, которое гласит, что else соответствует ближайшему if, кроме тех случаев, когда имеются фигурные скобки.**

**Условный оператор ?.**

Условный оператор ? позволяет кратко записать условие проверки. Соответствующие действия выполняются в зависимости от вычисленного значения выражение\_условие: истина или ложь. Этот оператор можно использовать вместо эквивалентного оператора if-else. Условный оператор имеет следующий синтаксис:

**Выражение\_условие ? действие\_если\_истина : действие\_если\_ложь;**

Оператор ? иногда называют троичным оператором, так как он требует трех операндов.

**Оператор switch.**Часто необходимо сравнить некоторую переменную или выражение с несколькими значениями. Для этого можно использовать либо вложенные операторы **if-else-if**, либо — оператор **switch**.

**switch (общее\_выражение) {**

**case константа1:**

**операторы1;**

**break;**

**case константа2:**

**операторы2;**

**break;**

**…**

**default: операторы;**

**Совместное использование операторов if-else-if и switch.**

**58.Оператор цикла. Цикл while. Завершение цикла while.**

**Оператор цикла.**

В языке С имеется стандартный набор операторов цикла: for, while и do-while Вас может, однако, удивить то, каким способом программа выходит из цикла Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. оператора break или оператора continue.

**Цикл while.**

Так же как и цикл for, в С цикл while является циклом с предусловием. Это означает, что в программе проверка\_условия осуществляется до выполнения оператора или операторов, входящих в тело цикла. Благодаря этому, циклы с предусловием могут либо не выполняться вообще, либо выполняться множество раз. В С синтаксис цикла while следующий:

**while(проверка\_условия)**

**оператор;**

В циклах while с несколькими операторами необходимы фигурные скобки:

**while(проверка\_условия) {**

**оператор1;**

**оператор2;**

**оператор3;**

**операторn;**

**Завершение цикла while.**

При построении цикла while вы должны включить в него какие-то конструкции, изменяющие величину проверяемого выражения так, чтобы в конце концов оно стало ложным. В противном случае выполнение цикла никогда не завершится.

**59.Оператор цикла. Цикл do-while. Цикл for.**

**Оператор цикла.**

В языке С имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторый других языках высокого уровня циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В С можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

**Цикл do-while.**

Цикл do-while отличается от циклов for и while тем, что это — цикл с постусловием. Другими словами: цикл всегда выполняется хотя бы один раз, после чего в конце первого прохода проверяется условие продолжения цикла. В отличие от этого циклы for и while могут не выполняться вообще, или выполняться множество раз в зависимости от значения переменной управления циклом. Поскольку циклы do-while выполняются, по меньшей мере, один раз, их лучше использовать тогда, когда нет сомнений о вхождении в определенный цикл.

Синтаксис цикла do-while следующий:

**do**

**действие;**

**while(проверка\_условия) ;**

**do {**

**действие1;**

**действие2;**

**действиеЗ;**

**действиеn;**

**} while(проверка\_условия) ;**

**Цикл for.**

В цикле for распространен для «математических повторений. Вот пример его записи:

 for(count = 1; count <= NUMBER; count ++ )

printf(" Будь моим Валентином!\n");

В круглых скобках содержатся три выражения, разделенные символом «точка с запятой». Первое из них служит для инициализации счетчика. Она осуществляется только один раз — когда цикл for начинает выполняться. Второе выражение — для проверки условия; она производится перед каждым возможным выполнением тела цикла. Когда выражение становится ложным (или в общем случае равным нулю), цикл завершается. Третье выражение вычисляется в конце каждого выполнения тела цикла.

**60.Оператор цикла. Операция "запятая" в цикле for. Гибкость конструкции for. Философ Зенон и цикл for.**

**Оператор цикла.**

В языке С имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторый других языках высокого уровня циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В С можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

**Операция «запятая» в цикле for.**

Операция «запятая» увеличивает гибкость использования цикла for, позволяя включать в его спецификацию несколько инициализирующих или корректирующих выражений. #define FIRST 20

#define NEXT 17

main()

{

int ounces, cost;

printf(" Фунции стоимость\n" );

for(ounces = 1, cost = FIRST; ounces < 16; ounces++, cost + = NEXT)

printf(" %3d %7d\n" , ounces, cost);}

**Гибкость конструкции for**

1. Можно применять операцию уменьшения для счета в порядке убывания вместо счета в порядке возрастания.

 2. При желании вы можете вести счет двойками, десятками и т. д.

3. Можно вести подсчет с помощью символов, а не только чисел:

 4. Можно проверить выполнение некоторого произвольного условия, отличного от условия, налагаемого на число итераций. В нашей программе таблица кубов вы могли бы заменить спецификацию

 Это было бы целесообразно в случае, если бы нас больше занимало ограничение максимального значения диапазона кубов чисел, а не количества итераций.

 5. Можно сделать так, чтобы значение некоторой величины возрастало в геометрической, а не в арифметической прогрессии, т. е. вместо прибавления фиксированного значения на каждом шаге цикла, выполнялось бы умножение:

 6. В качестве третьего выражения можно использовать любое правильно составленное выражение. Какое бы выражение вы ни указали, его значение будет меняться при каждой итерации.

  7. Можно даже опустить одно или более выражений (но при этом нельзя опустить символы «точка с запятой»). Необходимо только включить в тело цикла несколько операторов, которые в конце концов приведут к завершению его работы.

 8. Первое выражение не обязательно должно инициализировать переменную. Вместо этого, например, там мог бы стоять оператор printf() некоторого специального вида. Необходимо помнить только, что первое выражение вычисляется только один раз перед тем, как остальные части цикла начнут выполняться.

Философ Зенон известен своими знаменитыми парадоксами (апориями). Посмотрим, как с помощью операции «запятая» можно разрешить старый парадокс. Греческий философ Зенон утверждал, что пущенная стрела никогда не достигнет цели. Сначала, говорил он, стрела пролетит половину расстояния до цели. После этого ей останется пролететь половину всего расстояния, но сначала она должна будет пролететь половину того, что ей осталось пролететь, и т. д. до бесконечности. Поскольку расстояние полета разбито на бесконечное число частей, для достижения цели стреле может потребоваться бесконечное время. Мы сомневаемся, однако, что Зенон вызвался бы стать мишенью для стрелы, полагаясь только на убедительность своего аргумента.

Применим количественный подход и предположим, что за одну секунду полета стрела пролетает первую половину расстояния. Тогда за последующую 1/2 секунды она пролетит половину того, что осталось от половины, за 1/4 — половину того, что осталось после этого, и т д. Полное время полета представляется в виде суммы бесконечного ряда 1 + 1/2 + 1/4 + 1/8 + 1/16 + ...

**61.Оператор цикла. Вложенные циклы. Алгоритмы и псевдокод.**

**Оператор цикла.**

В языке С имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторый других языках высокого уровня циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В С можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

**Вложенные циклы.**

Вложенным называется цикл, находящийся внутри другого цикла.

Под **алгоритмом** понимали конечную последовательность точно сформулированных правил, которые позволяют решать те или иные классы задач. Такое определение алгоритма не является строго математическим, так как в нем не содержится точной характеристики того, что следует понимать под классом задач и под правилами их решения.

**Свойства алгоритмов**.

Каждое указание алгоритма предписывает исполнителю выполнить одно конкретное законченное действие. Исполнитель не может перейти к выполнению следующей операции, не закончив полностью выполнения предыдущей. Предписания алгоритма надо выполнять последовательно одно за другим, в соответствии с указанным порядком их записи. Выполнение всех предписаний гарантирует правильное решение задачи.

достижению цели. Разделение выполнения решения задачи на отдельные операции (выполняемые исполнителем по определенным командам) — важное свойство алгоритмов, называемое **дискретностью**.

Алгоритм, составленный для конкретного исполнителя, должен включать только те команды, которые входят в его систему команд. Это свойство алгоритма называется **понятностью**.

Еще одно важное требование, предъявляемое к алгоритмам, — **результативность (или конечность)** алгоритма. Оно означает, что исполнение алгоритма должно закончиться за конечное число шагов.

**Формы записи алгоритмов**

· записан на естественном языке (примеры записи алгоритма на естественном языке приведены при определении понятия алгоритма);

· изображен в виде блок-схемы;

· записан на алгоритмическом языке.

**Базовые структуры алгоритмов** — это определенный набор блоков и стандартных способов их соединения для выполнения типичных последовательностей действий. К основным структурам относятся следующие: **линейные** , **разветвляющиеся** , **циклические** .

**Линейными** называются алгоритмы, в которых действия осуществляются последовательно друг за другом.

**Разветвляющимся** называется алгоритм, в котором действие выполняется по одной из возможных ветвей решения задачи, в зависимости от выполнения условий.

**Циклическим** называется алгоритм, в котором некоторая часть операций (тело цикла — последовательность команд) выполняется многократно.

**Псевдокод**

Компактный язык описания алгоритмов, использующий ключевые слова императивных языков программирования, но опускающий несущественные подробности и специфический синтаксис. Псевдокод обычно опускает детали, несущественные для понимания алгоритма человеком. Такими несущественными деталями могут быть описания переменных, системно-зависимый код и подпрограммы. Главная цель использования псевдокода - обеспечить понимание алгоритма человеком, сделать описание более воспринимаемым, чем исходный код на языке программирования.

**62.Оператор цикла. Оператор break. Оператор continue. Совместное использование операторов break и continue.**

**Оператор цикла.**

В языке С имеется стандартный набор операторов цикла: for, while и do-while (называемый в некоторый других языках высокого уровня циклом repeat-until). Вас может, однако, удивить то, каким способом программа выходит из цикла. В С можно изменить порядок выполнения цикла четырьмя способами. Естественно, все циклы заканчиваются при выполнении заданного проверочного условия. Однако, в С цикл может также закончиться по некоторому заданному условию ошибки при помощи операторов break или exit. Кроме этого, в циклах может быть собственная управляющая логика, изменяемая при помощи оператора break или оператора continue.

В теле любого цикла можно использовать **оператор break**, который позволяет выйти из цикла, не завершая его.

**Оператор continue**

В С существует небольшое различие между операторами **break и continue**.. В отличие от break, оператор continue приводит к игнорированию всех следующих за ним операторов, однако не препятствует инкременту переменной управления циклом или выполнению проверки условия продолжения цикла. Другими словами: если переменная управления циклом продолжает отвечать условию выполнения цикла, то цикл повторяется.

**while(<условие>)**

**continue;**

**Совместное использование операторов break и continue.**

Операторы break и continue можно использовать вместе для решения некоторых интересных задач программирования.

**63.Оператор goto. Использование goto.**

**Оператор goto.**

goto - оператор условного перехода.Оператор goto — одно из важнейших средств Бейсика и Фортрана — также реализован и в Си. Однако на этом языке в отличие от двух других можно программировать, совершенно не используя указанное средство. Керниган и Ритчи считают оператор goto «чрезвычайно плохим» средством и предлагают «применять его как можно реже или не применять совсем».

Оператор goto состоит из двух частей — ключевого слова goto и имени метки. Имена меток образуются по тем же правилам, что и имена переменных. Приведем пример записи оператора

**goto part2;**

 Чтобы этот оператор выполнился правильно, необходимо наличие другого оператора, имеющего метку part2; в этом случае запись оператора начинается с метки, за которой следует двоеточие.

**part2: printf(" Уточненный анализ:\n");**

**Использование goto.**

В принципе вы никогда не обязаны пользоваться оператором goto при программировании на Си. Ниже вкратце приводится несколько знакомых вам ситуаций, реализуемых с помощью goto, а затем показывается, как это можно осуществить другими средствами, в большей степени соответствующими духу языка Си.

**64.Оператор exit(). Оператор atexit(). Сравнение циклов.**

**Оператор exit().**

В некоторых случаях программу необходимо закончить до того, как выполнились все ее операторы или условия. Для таких особых случаев в С имеется библиотечная функция exit(). Эта функция может иметь один целочисленный аргумент, называемый статусом. Операционные системы UNIX и MS-DOS интерпретируют значение статуса, равное нулю, как успешное завершение программы, а все ненулевые значения статуса говорят о различного вида ошибках.

**Оператор atexit().**

Всякий раз, когда программа вызывает функцию exit() или происходит нормальное завершение программы, можно также вызывать любую зарегистрированную "функцию выхода", занесенную в atexit().

# 

**65.Массивы. Понятие массив. Массивы в С. Объявление массивов. Доступ к элементам массива. Размещение массивов в памяти. Проблема ввода.**

**Массивы.**

**Массив** — это набор переменных, имеющих одно и то же базовое имя и отличающихся одна от другой числовым признаком. Перечисление элементов массива начинается с 0, а не с 1.

**Понятие массив.**

Можно рассматривать **массивы** как переменные, содержащие несколько элементов одного типа. Доступ к каждому отдельному элементу данных осуществляется при помощи индекса этой переменной. В языке С массивы не являются стандартным типом данных; они представляют собой составной тип, созданный на основе других типов данных. В С возможно создавать массивы из любых типов переменных: символов, целых, чисел двойной длины, массивов, указателей, структур и так далее. В общих чертах концепции массивов и способы их использования в С и C++ совпадают.

**Массивы в С.**

Массивы имеют четыре основных характеристики:

· Отдельные объекты данных в массиве называются элементами.

· Все элементы массива должны иметь одинаковый тип данных.

· Все элементы располагаются в памяти компьютера последовательно, и индекс первого элемента равен нулю.

· Имя массива является постоянным значением, представляющим собой адрес первого элемента массив

**Объявление массивов.**

Ниже приведены примеры объявлений массивов:

int iarray[12]; /\* массив из 12 целых чисел \*/

char carray[20]; /\* массив из 20 символов \*/

**Проблема ввода.**

Язык Си предоставляет много средств для структурирования программ. С помощью операторов while и for реализуются циклы с предусловием. Второй оператор особенно подходит для циклов, включающих в себя инициализацию и коррекцию переменной. Использование операции «запятая» в цикле for позволяет инициализировать и корректировать более одной переменной. Для тех редких случаев, когда требуется использовать цикл с постусловием, язык Си предоставляет оператор do while. Операторы break, continue и goto обеспечивают дополнительные возможности управления ходом выполнения программы.

**66.Массивы. Инициализация массивов. Инициализация по умолчанию. Явная инициализация. Инициализация безразмерных массивов.**

**Инициализация массивов.**

Имеется три способа инициализации массивов:

· По умолчанию во время их создания. Применимо только к глобальным и статическим (static) массивам.

· Явно во время создания при помощи констант инициализации.

· Во время выполнения программы при присваивании или копировании данных в массив.

 Для инициализации массива во время создания можно использовать только константы. Если элементы массива должны получать значения из переменных, то в программном коде должны быть явные операторы инициализации массива.

**Инициализация по умолчанию.**

В стандарте ANSI С оговорено, что массивы бывают либо глобальные (описанные вне main() и любых других функций), либо автоматические static (статические, но описанные после какой-либо открывающей скобки), и при отсутствии инициализирующей информации они всегда получают значения двоичных нулей. В С числовые массивы инициализируются значением ноль. (Массивы указателей получают начальные значения null)

**Явная инициализация.**

Аналогично описаниям и инициализации переменных типов **int, char, float, double** и других вы можете инициализировать массивы. Стандарт ANSI С позволяет задавать и инициализирующие значения любого массива, глобального или иного, описанного в любой части программы. Следующий фрагмент кода иллюстрирует описание и инициализацию массивов:

 int iarray[3] = {-1,0,1};

**Безразмерные массивы**

Для большинства компиляторов безразлично — указываете ли вы размер массива, или задаете список фактических значений; важно указать либо то, либо другое. Например: в программах часто описывается набор собственных сообщений об ошибках. Это можно сделать двумя способами. Вот первый из них:

 char e1[12] = "Ошибка чтения\n";

При таком подходе необходимо считать количество символов в строке, не забывая добавить 1, чтобы учесть невидимый null-символ окончания строки \0. Это, в лучшем случае, очень утомительный для глаз метод, который чреват многими ошибками. При втором способе, показанном ниже, используются безразмерные массивы, и язык С автоматически определяет их размер:

char e2[] = "Ошибка записи\n";

Всякий раз, когда встречается оператор инициализации массива и размер массива не указан, компилятор автоматически выделяет достаточно места для всех указанных данных.

**67.Массивы. Инициализация массивов и классы памяти. Вычисление размера массива (sizeof()). Выход индекса за пределы массива.**

**Инициализация массивов и классы памяти.**

Для хранения данных, необходимых программе, часто используют массивы.

Внешние и статические массивы можно инициализировать. Автоматические и регистровые массивы инициализировать нельзя.

Прежде чем попытаться инициализировать массив, давайте посмотрим, что там находится, если мы в него ничего не записали. Если ничего не засылать в массив перед началом работы с ним, то внешние, статические и автоматические массивы инициализируются для числовых типов нулем и '\0' (null) для символьных типов, а регистровые массивы содержат какой-то мусор, оставшийся в этой части памяти

**Вычисление размера массива (sizeof()).**

Как вы уже знаете, операция **sizeof()** возвращает физический размер в байтах того объекта, к которому она применяется. Ее можно использовать с объектами любых типов, за исключением битовых полей. Часто операция sizeof() применяется для определения физического размера переменной в тех случаях, когда размер переменных этого типа может меняться от одного компьютера к другому. Вы уже видели, что в зависимости от используемой системы целые числа могут иметь длину 2 или 4 байта.

**Выход индекса за пределы массива.**

Вам, наверное, знакомо выражение: "За все нужно платить". Это справедливо и для массивов в С. "Все" — это быстрое выполнение кода программы, а "плата" — отсутствие проверки границ массива. Напомним: поскольку язык С был разработан для замены ассемблера, в целях обеспечения компактности кода, ошибки такого рода компилятор не проверяет. Поскольку компилятор никак не определяет ошибочные ситуации, вы должны быть очень осторожны при работе с граничными значениями индекса массива.

**68. Массивы. Многомерные массивы. Инициализация двумерного массива.**

*Массив* - это структура данных, представленная в виде группы ячеек одного типа, объединенных под одним единым именем. Массивы используются для обработки большого количества однотипных данных. Имя массива является [указателем](http://cppstudio.com/obuchenie_cpp/ukazateli).

Одномерный массив — массив, с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив — это массив, у которого может быть только одна строка, и n-е количество столбцов. Столбцы в одномерном массиве — это элементы массива.

Инициализация одномерного массива выполняется в фигурных скобках после знака **равно**, каждый элемент массива отделяется от предыдущего запятой.

int a[] = { 4,8,9,5,-2,7,2,1,1,0,13,-1 };

//объявление и инициализация

Индекс ячейки – это целое неотрицательное число, по которому можно обращаться к каждой ячейке массива и выполнять какие-либо действия над ней (ячейкой).

С позволяет создавать многомерные массивы. Простейшим видом многомерного массива является двумерный массив. Двумерный массив - это массив одномерных массивов. Двумерный массив объявляется следующим образом:  
***тип имя\_массива[размер второго измерения][размер первого измерения];***  
Следовательно, для объявления двумерного массива целых с размером 10 на 20 следует написать:  
int d[10] [20] ; // объявление

int a[4][3] = { {4,8,9}, {5,-2,7}, {2,1,1}, {0,13,-1} };

//объявление и инициализация  
Посмотрим внимательно на это объявление. В противоположность другим компьютерным языкам, где размерности массива отделяются запятой, С помещает каждую размерность в отдельные скобки.

Двумерные массивы сохраняются в виде матрицы, где первый индекс отвечает за строку, а второй -за столбец. Это означает, что правый индекс изменяется быстрее левого, если двигаться по массиву в порядке расположения элементов в памяти. Левый индекс можно рассматривать как указатель на строку.

Число байт в памяти, требуемых для размещения двумерного массива, вычисляется следующим  образом:  
  
***число байт = размер второго измерения \* размер первого измерения \* sizeof (базовый тип)***

**69. Массивы. Массивы в качестве аргументов функций. Передача массивов функциям С. Передача массивов функциям C++.**

*Массив* — это набор переменных одного типа, имеющих одно и то же имя. Доступ к конкретному элементу массива осуществляется с помощью индекса. В языке С все массивы располагаются в отдельной непрерывной области памяти. Первый элемент массива располагается по самому меньшему адресу, а последний — по самому большому. Массивы могут быть одномерными и многомерными. *Строка* представляет собой массив символьных переменных, заканчивающийся специальным нулевым символом, это наиболее распространенный тип массива.

Одномерный массив — массив, с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив — это массив, у которого может быть только одна строка, и n-е количество столбцов.

С позволяет создавать многомерные массивы. Простейшим видом многомерного массива является двумерный массив. Двумерный массив - это массив одномерных массивов. Двумерный массив объявляется следующим образом:  
***тип имя\_массива[размер второго измерения][размер первого измерения];***  
Следовательно, для объявления двумерного массива целых с размером 10 на 20 следует написать:  
int d[10] [20] ; // объявление

int a[4][3] = { {4,8,9}, {5,-2,7}, {2,1,1}, {0,13,-1} };

//объявление и инициализация

**Вызов функций с помощью массивов**

Когда в качестве аргумента функции используется массив, то функции передается его адрес. В этом и состоит исключение по отношению к правилу, которое гласит, что при передаче параметров используется вызов по значению. В случае передачи массива функции ее внутренний код работает с реальным содержимым этого массива и вполне может изменить это содержимое.

Классическим примером передачи массивов в функции является стандартная библиотечная функция gets(). Хотя gets(), которая находится в вашей стандартной библиотеке, и более сложная, чем предлагаемая вам версия xgets(), но с помощью функции xgets() вы сможете получить представление о том, как работает gets().

Функцию xgets() следует вызывать с указателем char \*.

Так как в языке С нет встроенной проверки границ, программист должен сам позаботиться, чтобы в любом массиве, используемом при вызове xgets(), помещалось не менее 80 символов. Когда символы вводятся с клавиатуры, они сразу записываются в строку. Если пользователь нажимает клавишу <Backspase>, то счетчик t уменьшается на 1, а из массива удаляется последний символ, введенный перед нажатием этой клавиши. Когда пользователь нажмет <ENTER>, в конец строки запишется нуль, т.е. признак конца строки. Так как массив, использованный для вызова xgets(), модифицируется, то при возврате из функции в нем будут находиться введенные пользователем символы.

**70. Ввод и вывод строк. Строковые функции и символьные массивы. Динамическое выделение памяти. Функции gets(), puts(), fgets(), fputs() и sprintf(). Функции strcpy(), strcat(), strncmp() и strlen().**

| **Функция** | **Пояснение** |
| --- | --- |
| **strlen(**имя\_строки**)** | определяет длину указанной строки, без учёта нуль-символа |
| **Копирование строк** | |
| **strcpy(**s1,s2**)** | выполняет побайтное копирование символов из строки  s2 в строку s1 |
| **strncpy(**s1,s2, n**)** | выполняет побайтное копирование n символов из строки  s2 в строку s1. возвращает значения s1 |
| **Объединение строк** | |
| **strcat(**s1,s2**)** | объединяет строку s2 со строкой s1. Результат сохраняется в s1 |
| **strncat(**s1,s2,n**)** | объединяет n символов строки s2 со строкой s1. Результат сохраняется в s1 |
| **Сравнение строк** | |
| **strcmp(**s1,s2**)** | сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 С учётом регистра |
| **strncmp(**s1,s2**)** | сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 С учётом регистра |
| **stricmp(**s1,s2**)** | сравнивает строку s1 со строкой s2 и возвращает результат типа **int**: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 Без учёта регистра |
| **Функции преобразования** | |
| **atof(**s1**)** | преобразует строку s1 в тип double |
| **atoi(**s1**)** | преобразует строку s1 в тип int |
| **atol(**s1**)** | преобразует строку s1 в тип long int |

Строки в С++ представляются как массивы элементов типа char, заканчивающиеся нуль-терминатором \0 называются С строками или строками в стиле С.

\0  — символ нуль-терминатора.

Символьные строки состоят из набора символьных констант заключённых в двойные кавычки. При объявлении строкового массива необходимо учитывать наличие в конце строки нуль-терминатора, и отводить дополнительный байт под него.

Строка при объявлении  может быть инициализирована начальным значением, например, так:

|  |  |
| --- | --- |
| 1 | char string[10] = "abcdefghf"; |

Строка может содержать символы, цифры и специальные знаки. В С++ строки заключаются в двойные кавычки. Имя строки является константным указателем на первый символ. Разработаем программу, с использованием строк.

**71. Указатели. Определение переменных-указателей. Разыменование указателей. Объявление переменных-указателей. Простые операторы с указателями. Инициализация указателей. Неправильное использование операции определения адреса.**

**Указатель** – переменная, значением которой является адрес ячейки памяти. То есть указатель ссылается на блок данных  из области памяти, причём на самое его начало. Указатель может ссылаться на переменную или функцию. Для этого нужно знать адрес переменной или функции. Так вот, чтобы узнать адрес конкретной переменной в С++ существует унарная операция взятия адреса **&**. Такая операция извлекает адрес объявленных переменных, для того, чтобы его присвоить указателю.

Основной операцией при работе с указателями является получение доступа к значению, адрес которого хранится в указателе.  
Например:

int \*pn, n;  
\*pn = 5;  
n = \*pn;

Выражение \*pn имеет такой же смысл, как имя целой переменной. Операция <<\*>> называется разыменованием.  
Действие, обратное к разыменованию, позволяет получить адрес переменной по ее имени. Например,

pn = &n;

эта операция называется взятие адреса.

**72. Указатели. Указатели на массивы. Указатели и многомерные массивы. Указатели на указатели. Указатели на строки.**

**Указатель** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.  
Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).  
Указатель, как и любая переменная, должен быть объявлен.  
Общая форма объявления указателя

**тип \*ИмяОбъекта;**

Тип указателя— это тип переменной, адрес которой он содержит.  
Для работы с указателями в Си определены две операции:

операция \* (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Например,

Для указанного примера обращение к одним и тем же значениям переменной и адреса представлено в таблице

сhar c;    // переменная  
char \*p;  // указатель  
p = &c;   // p = адрес c

Необходимо помнить, что компиляторы высокого уровня поддерживают [прямой способ адресации](https://prog-cpp.ru/asm-memory/#front): младший байт хранится в ячейке, имеющей младший адрес.

**Указатель на массив:**

int\* myM;

myM = new int[10];

Возможно объявление переменной, которая содержит адрес другой переменной, которая, в свою очередь, также является указателем. Такая переменная может быть необходима, если в функции нужно изменить адрес какого-либо объекта. Однако наличие более двух звёздочек в объявлении переменной говорит, скорее всего, о плохом проектировании.

|  |
| --- |
| int \*\*ppi; |

**Строка** – это последовательность (массив) символов (типа [char](http://natalia.appmat.ru/c&c++/lezione1.php#l1_4)), которая заканчивается специальным символом – признаком конца строки. Это символ записывается как '\0' (не путайте с символом переноса строки '\n') и равен 0.

**73. Указатели. Арифметические операции с указателями. Арифметические операции с указателями и массивы. Операции с указателями.**

**Указатель** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.  
Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).  
Указатель, как и любая переменная, должен быть объявлен.  
Общая форма объявления указателя

**тип \*ИмяОбъекта;**

Тип указателя— это тип переменной, адрес которой он содержит.  
Для работы с указателями в Си определены две операции:

операция \* (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Необходимо помнить, что компиляторы высокого уровня поддерживают [прямой способ адресации](https://prog-cpp.ru/asm-memory/#front): младший байт хранится в ячейке, имеющей младший адрес.

Над адресами в C++ определены следующие арифметические операции:

* сложение и вычитание указателей с константой;
* вычитание одного указателя из другого;
* инкремент;
* декремент.

**Инкремент**перемещает указатель к следующему элементу массива, а **декремент***–* к предыдущему;

К указателям могут применяться только две арифметические операции: сложение и вычитание. Каждый раз, когда уменьшается - на предыдущий элемент. В случае указателей на символы это приводит к «обычной» арифметике. Все остальные указатели увеличиваются или уменьшаются на длину базового типа.

В большинстве случаев вычитание одного указателя из другого имеет смысл только тогда, когда оба указателя указывают на один объект, - как правило, массив. В результате вычитания получается число элементов базового типа, находящихся между указателями.. Нельзя умножать или делить указатели, нельзя складывать указатели, нельзя применять битовый сдвиг или маски к указателям, нельзя добавлять или вычитать типы float или double.

К указателям так же применимы *операции отношения ==*, !=, <,>,<=,>=. Иными словами, указатели можно сравнивать.

**74. Указатели. Применение к указателям оператора sizeof. Сложности при использовании операций ++ и --. Сравнение указателей. Переносимость указателей. Использование функции sizeof() с указателями в среде DOS.**

**Указатель** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект. Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).  
Указатель, как и любая переменная, должен быть объявлен.Общая форма объявления указателя **тип \*ИмяОбъекта;**

Тип указателя— это тип переменной, адрес которой он содержит.  
Для работы с указателями в Си определены две операции:

операция \* (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

int \*pi = new int[ 3 ];

size\_t pointer\_size = sizeof ( pi );

Здесь значением pointer\_size будет память под указатель в байтах (4 в 32-битных системах), а не массива ia.Применение sizeof к указателю позволяет узнать размер памяти, необходимой для хранения адреса.

Над адресами в C++ определены следующие арифметические операции:

* сложение и вычитание указателей с константой;
* вычитание одного указателя из другого;
* инкремент;
* декремент.

**Инкремент**перемещает указатель к следующему элементу массива, а **декремент***–* к предыдущему;

.К указателям так же применимы *операции отношения ==*, !=, <,>,<=,>=. Иными словами, указатели можно сравнивать. Однако, все эти утверждения верны, если речь идет об указателях, ссылающихся на один массив. В противном случае результат арифметических операций и операций отношения будет не определен.

**Переносимость указателей**.

В приведенных примерах адреса представлялись целыми числами. Вы можете предположить, что в С указатели имеют тип int. Это не так. Указатель содержит адрес переменной некоторого типа, однако сам указатель не относится ни какому простому типу данных вроде int, float и им подобному. В конкретной системе С указатель может копироваться в переменную типа int, а переменная int может копироваться в указатель; однако, язык С не гарантирует, что указатели могут храниться в переменных типа int. Для обеспечения переносимости программного кода таких операций следует избегать.Кроме того, разрешены не все арифметические операции с указателями. Например, запрещается складывать или перемножать два указателя, или делить один указатель на другой.

**75. Указатели. Указатели на функции. Динамическая память. Использование указателей типа void.**

**Указатель** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект. Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).  
Общая форма объявления указателя

**тип \*ИмяОбъекта;**

Тип указателя— это тип переменной, адрес которой он содержит.  
Для работы с указателями в Си определены две операции:

операция \* (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

*Указатель на функцию* можно использовать в нескольких важных случаях. Рассмотрим функцию qsort(). Одним из ее параметров является указатель на функцию. Адресуемая функция осуществляет необходимое сравнение, которое должно выполняться для элементов сортируемого массива. Применение в qsort() указателя на функцию вызвано тем, что процесс сравнения двух элементов может быть весьма сложным, и при помощи одного управляющего флага его представить нельзя. Функцию нельзя передать по значению — то есть, передать ее код.

*Динамическая память.*

При компиляции программы на С память компьютера разбивается на четыре области, содержащие код программы, все глобальные данные, стек и динамически распределяемую область памяти (иногда ее называют heap-куча). Динамическая память (heap) — это просто свободная область памяти, с которой работают при помощи функций динамического выделения памяти malloc() и free().

При вызове функции malloc() выделяется непрерывный блок памяти для указанного объекта и возвращается указатель на начало этого блока. Функция free() возвращает выделенный блок обратно в динамическую область памяти для повторного использования.

Аргумент, передаваемый функции malloc(), представляет собой объем необходимой памяти в байтах. В следующем фрагменте кода выделяется память для 300 чисел типа float:

Когда блок становится ненужным, его можно вернуть операционной системе, используя, следующий оператор:

float \*pf;

int inum\_floats = 300;

pf = (float \*) malloc(inum\_floats \* sizeof(float));

free((void \*) pf);

Для распределения динамической памяти в большинстве компиляторов С используются описанные выше библиотечные функции malloc() и free().*В C++* для выделения и освобождения динамической памяти используются операторы *new и delete*.

***Использование указателей типа void****.* Эти указатели используются в том случае, когда тип переменной не известен. Так как void не имеет типа, то к нему не применима операция разадресации (взятие содержимого) и адресная арифметика, так как неизвестно представление данных. Тем не менее, если мы работаем с указателем типа void, то нам доступны операции сравнения.Если необходимо работать с пустым указателем, то сначала нужно явно привести тип. Переменная не может иметь типа void, этот тип определён только для указателей. Пустые указатели нашли широкое применение при вызове функций. Можно написать функцию общего назначения, которая будет работать с любым типом.

Объявление указателя на неопределенный тип:void \*ptr;

Такому указателю может быть присвоен указатель на любой тип, но не наоборотДля последней операции необходимо явное приведение типа.

Над указателем неопределенного типа нельзя выполнять операцию разыменования без явного приведения типа.

**76. Указатели. Указатели и массивы. Функции, массивы и указатели. Использование указателей при работе с массивами. Строки (массивы типа char).**

**Указатель** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.  
Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).  
Указатель, как и любая переменная, должен быть объявлен.  
Общая форма объявления указателя

**тип \*ИмяОбъекта;**  
Тип указателя— это тип переменной, адрес которой он содержит.  
Для работы с указателями в Си определены две операции:

операция \* (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

**Указатели и массивы.**

Указатели позволяют нам работать с символическими адресами. Указатели позволяют эффективно организовать работу с массивами.Обозначение массива представляет собой просто скрытую форму использования указателей.

В нашей системе единицей адресации является байт, но тип int использует два байта, а тип float — четыре. Что произойдет, если вы скажете: «прибавить единицу к указателю?» Компилятор языка Си добавит единицу памяти. Для массивов это означает, что мы перейдем к адресу следующего элемента, а не следующего байта. Вот почему мы должны специально оговаривать тип объекта, на который ссылается указатель; одного адреса здесь недостаточно, так как машина должна знать, сколько байтов потребуется для запоминания объекта. (Это справедливо также для указателей на скалярные переменные; иными словами, при помощи операции \*pt нельзя получить значение.)

**Функции, массивы и указатели.**

Массивы можно использовать в программе двояко. Во-первых, их можно описать в теле функции. Во-вторых, они могут быть аргументами функции.

 Очевидно, что массив ages состоит из 50 элементов. А что можно сказать о массиве years? Оказывается, в программе нет такого массива. Описатель int years[]; создает не массив, а указатель на него.

Вот вызов нашей функции:

 ages — аргумент функции convert. Вы помните, что имя ages является указателем на первый элемент массива, состоящего из 50 элементов. Таким образом, оператор вызова функции передает ей указатель, т. е. адрес функции convert (). Это значит, что аргумент функции является указателем, и мы можем написать функцию convert () следующим образом:

Действительно, операторы

*int years [];*

*int \*years;*синонимы. Оба они объявляют переменную years указателем массива целых чисел. Однако главное их отличие состоит в том, что первый из них напоминает нам, что указатель years ссылается на массив.

**Использование указателей при работе с массивами.**

**Строки (массивы типа char).**

Многие строковые операции в С обычно выполняются с использованием указателей и арифметических операций с указателями для доступа к элементам символьного массива. Это обусловлено тем, что символьные массивы или строки, как правило, обрабатываются строго последовательно. Напоминаем, что все строки в С заканчиваются символом null (\0). Следующая программа на C++ иллюстрирует использование указателей с символьными массивами:

**ПРИМЕЧАНИЕ.**

Функция strlen() считает только символы. Она не включает в результат null - символ \0

**77. Указатели. Массивы указателей. Дополнительная информация об указателях на указатели. Массивы указателей на строки.**

**Указатель** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.  
Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).  
Указатель, как и любая переменная, должен быть объявлен.  
Общая форма объявления указателя

**тип \*ИмяОбъекта;**

Тип указателя— это тип переменной, адрес которой он содержит.  
Для работы с указателями в Си определены две операции:

операция \* (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

Например,

**Массивы указателей.**

Массив указателей — это такой массив, элементы которого являются указателями на другие объекты. Эти объекты, в свою очередь, могут быть указателями. Это означает, что можно иметь массив указателей, которые ссылаются на другие указатели.

Можно создавать массивы указателей. Для объявления массива целочисленных указателей из десяти элементов следует написать:  
int \*х[10];  
Для присвоения адреса целочисленной переменной var третьему элементу массива следует написать:  
х[2] = &var;  
Для получения значения var следует написать:  
\*х [2]

Если необходимо передать массив указателей в функцию, можно использовать метод, аналогичный передаче обычных массивов. Просто надо вызвать функцию с именем массива без индексов. Например, функция, получающая массив х, должна выглядеть следующим образом:

надо помнить, что q - это не указатель на целое, а массив указателей на целые. Следовательно, необходимо объявить параметр q как массив целых указателей. Он не может объявиться как простой целочисленный указатель, поскольку он не является им.

Типичным использованием массивов указателей является хранение сообщений об ошибках. Можно создать функцию, выводящую сообщение по полученному номеру

**78. Указатели. Ссылочный тип в C++ (reference type). Адрес в качестве возвращаемого значения функции. Передача параметров по ссылке и по значению. Использование встроенного отладчика. Использование ссылочного типа. Использование указателей и ссылок с ключевым**

**Указатель** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.Каждая переменная в памяти имеет свой адрес - номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).  
Указатель, как и любая переменная, должен быть объявлен.  
Общая форма объявления указателя

**тип \*ИмяОбъекта;**  
Тип указателя— это тип переменной, адрес которой он содержит.  
Для работы с указателями в Си определены две операции:

операция \* (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;

операция & (амперсанд) — позволяет определить адрес переменной.

**Ссылочный тип в C++ (reference type).**

Так же, как указатель, ссылочная переменная указывает на положение другой переменной, однако, так же как и в случае обычной переменной, для ссылки не требуется специальной операции разыменования. Синтаксис ссылочной переменной понятен:

int iresult\_a=5;

int& riresult\_a=iresult\_a; // правильно

int& riresult b; // неправильно: нет начального значения

Поскольку обе переменные указывают на одну и ту же ячейку памяти, они представляют собой, по сути, одну переменную.

Значение ссылочного типа должно быть задано при объявлении и не может меняться в процессе выполнения программы. После инициализации типа при объявлении, ссылка всегда указывает на одну и ту же ячейку памяти. Другими словами, ссылку можно считать указателем на ячейку памяти с постоянным адресом.

**Адрес в качестве возвращаемого значения функции.**

Когда функция возвращает адрес либо в виде переменной-указателя, либо в виде ссылки, пользователь получает некоторый адрес в памяти. Он может считать значение, находящееся по этому адресу, и, если тип указателя не объявлен как const, всегда может что-нибудь туда записать. Если функция возвращает адрес, то пользователь получает возможность читать и — в случае указателей, имеющих тип, отличный от const — обновлять локальные данные. Это важно для проектирования приложений.

**Передача параметров функции по ссылке и по значению**

В функцию параметры могут передаваться как по значению, так и по ссылке.

При передаче параметров по значению они при выходе из функции не изменятся.

При передаче же параметров по ссылке при выходе из функции значения могут измениться.

**void func(int &n, int &m) {**

**int tmp = n;**

**n = m;**

**m = tmp;**

**}**

После выполнения такой функции значения её параметров поменяются местами.

**Использование встроенного отладчика.**

Чтобы увидеть работу этой программы на C++ в реальных условиях, можно воспользоваться встроенным отладчиком. Проследим при помощи окна Trace изменения переменной pi.

**Использование ссылочного типа.**

можно выделить четыре основных причины использования ссылок C++:

· Ссылки упрощают восприятие программы, поскольку можно игнорировать детали процесса передачи параметра.

· Ссылки передают ответственность за передачу аргумента тому программисту, который пишет функцию, а не тому, который ею пользуется.

· Ссылки являются необходимой составной частью перегрузки операций.

· Ссылки используются при передаче функциям классов, при этом конструкторы и деструкторы не вызываются.

Ссылка представляет собой синоним имени, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

*тип & имя;*

где тип — это тип величины, на которую указывает ссылка, & — оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа, например:

int коl;  
int& pal = kol; // ссылка pal - альтернативное имя для коl  
const char& CR = '\n '; // ссылка на константу

Запомните следующие правила.

* Переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции, описана как extern или ссылается на поле данных класса.
* После инициализации ссылке не может быть присвоена другая переменная.
* Тип ссылки должен совпадать с типом величины, на которую она ссылается.
* Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений. Ссылки позволяют использовать в функциях переменные, передаваемые по адресу, без операции разадресации, что улучшает читаемость программы.

Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

**Использование указателей и ссылок с ключевым словом const**

// Указатель является константой

int\* const nl=unumber;

// Указатель указывает на константу (указываемое значение есть const)

const int\* n2=&count;

// И указатель, и указываемое значение // являются константами

const int\* const n3=&count;

// Указатели на строки // строка является константной

const char\* strl="text";

// Указатель на строку является константой

char\* const str2="text";

// Указатель и сама строка – константы

const char\* const str3="text";

/\* Массивы указателей на символы \*/// Символы являются константами

const char\* textl[]={"lnel","lne2","lne3"};

// Указатели являются константами

char\* const text2[]={"lnel","lne2","lne3"};

// Указатели и символы являются константами

const char\* const text3={"stl","st2","st3");