1. 1972г.-появл. 8-бит микропроц. Intel 8008. 1974г.-Intel 8880. 1979г.-Intel 8086 с 4,77; 8; 10 МГц.

1981г.- IBM MSDOS 1.0. 1981г. принцип открытой архитектуры. 1982г.-Intel 80286 (16МГц, 16МБ ОЗУ, 1Гб) 1985г.-Inyel 80386(40МГц) 1989г.-Intel 80486(66МГц) 1991г.-MSDOS 5.0. 1992г.-Intel 80486X4 (100МГц) 1992г.-Windows 3.1. 1995г.-Windows 95.

## 2. 1 поколение. Компьютеры на электронных лампах (194х-1955)

## Быстродействие: несколько десятков тысяч операций в секунду.

## 2 поколение. Компьютеры на транзисторах (1955-1965)

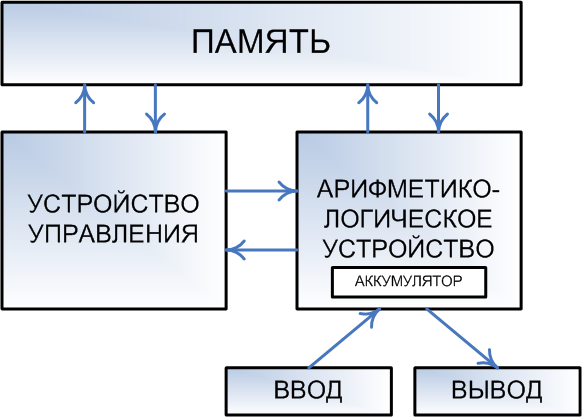
Быстродействие: сотни тысяч операций в секунду.

## 3 поколение. Компьютеры на интегральных схемах (1965-1980)

## Быстродействие: миллионы операций в секунду.

## 4 поколение. Компьютеры на больших (и сверхбольших) интегральных схемах (1980-…)

## Быстродействие: сотни миллионов операций в секунду.

3. **Архитектура фон Неймана** (модель фон Неймана, [Принстонская](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%81%D1%82%D0%BE%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D1%83%D0%BD%D0%B8%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%82%D0%B5%D1%82) архитектура) — широко известный принцип совместного хранения [команд](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) и [данных](https://ru.wikipedia.org/wiki/%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5_(%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D0%BA%D0%B0)) в [памяти компьютера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C). 

**Принцип однородности памяти**

Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Распознать их можно только по способу использования; то есть одно и то же значение в ячейке памяти может использоваться и как данные, и как команда, и как адрес в зависимости лишь от способа обращения к нему. Это позволяет производить над командами те же операции, что и над числами, и, соответственно, открывает ряд возможностей. Так, циклически изменяя адресную часть команды, можно обеспечить обращение к последовательным элементам массива данных. Такой приём носит название модификации команд и с позиций современного программирования не приветствуется. Более полезным является другое следствие принципа однородности, когда команды одной программы могут быть получены как результат исполнения другой программы. Эта возможность лежит в основе трансляции — перевода текста программы с языка высокого уровня на язык конкретной вычислительной машины.

**Принцип адресности**

Структурно основная память состоит из пронумерованных ячеек, причём процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые словами, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек — адреса.

**Принцип программного управления**

Все вычисления, предусмотренные алгоритмом решения задачи, должны быть представлены в виде программы, состоящей из последовательности управляющих слов — команд. Каждая команда предписывает некоторую операцию из набора операций, реализуемых вычислительной машиной. Команды программы хранятся в последовательных ячейках памяти вычислительной машины и выполняются в естественной последовательности, то есть в порядке их положения в программе. При необходимости, с помощью специальных команд, эта последовательность может быть изменена. Решение об изменении порядка выполнения команд программы принимается либо на основании анализа результатов предшествующих вычислений, либо безусловно.

4. По наиболее распространенной классификации все языки программирования, в соответствии с тем, в каких терминах необходимо описать задачу, делят на языки низкого и высокого уровня.

Если язык близок к естественному языку программирования, то он называется языком высокого уровня, если ближе к машинным командам, – языком низкого уровня.

В группу языков низкого уровня входят машинные языки и языки символического кодирования: Автокод, Ассемблер. Операторы этого языка – это те же машинные команды, но записанные мнемоническими кодами, а в качестве операндов используются не конкретные адреса, а символические имена. Все языки низкого уровня ориентированы на определенный тип компьютера, т. е. являются машинно–зависимыми.

Машинно–ориентированные языки – это языки, наборы операторов и изобразительные средства которых существенно зависят от особенностей ЭВМ (внутреннего языка, структуры памяти и т.д.).

К языкам программирования высокого уровня относят **Фортран** (переводчик формул – был разработан в середине 50–х годов программистами фирмы IBM и в основном используется для программ, выполняющих естественно – научные и математические расчеты), **Алгол**, **Кобол** (коммерческий язык – используется, в первую очередь, для программирования экономических задач), **Паскаль**, **Бейсик** (был разработан профессорами Дармутского колледжа Джоном Кемени и Томасом Курцом.), **Си** (Деннис Ритч – 1972 году), **Пролог** (в основе языка лежит аппарат математической логики) и т.д.

Эти языки машинно–независимы, т.к. они ориентированы не на систему команд той или иной ЭВМ, а на систему операндов, характерных для записи определенного класса алгоритмов. Однако программы, написанные на языках высокого уровня, занимают больше памяти и медленнее выполняются, чем программы на машинных языках.

Программу, написанную на языке программирования высокого уровня, ЭВМ не понимает, поскольку ей доступен только машинный язык. Поэтому для перевода программы с языка программирования на язык машинных кодов используют специальные программы – трансляторы.

5.   
Данные и программы во время работы ПЭВМ размещаются в оперативной памяти, которая представляет собой последовательность пронумерованных ячеек. По указанному номеру процессор находит нужную ячейку, поэтому номер ячейки называется ее адресом. Минимальная адресованная ячейка (согласно стандарту IBM), с точки зрения программиста, состоит из 8 двоичных позиций, т.е. в каждую позицию могут быть записаны либо 0, либо 1. Объем информации, который помещается в одну двоичную позицию, называется ***битом***. Объем информации, равный 8 битам, называется ***байтом***.

Таким образом, в одной ячейке из 8 двоичных разрядов помещается объем информации в один байт. Поэтому объем памяти принято оценивать количеством байт (210 байт = 1024 байт = 1 Кб, 210 Кб = 1048576 байт = 1 Мб).

Для помещения данных в такие ячейки производится их запись с помощью нулей и единиц (кодирование). При кодировании каждый символ, введенный с клавиатуры, заменяется последовательностью из 8 двоичных разрядов в соответ­ствии со стандартной кодовой таблицей, т.е. символ занимает один байт. Например, в соответствии с таблицей кодов ASCII D ® 01000100; F ® 00100110; 4 ® 00110100; ? ® 0011110.

При кодировании числа преобразуются в двоичное представление. Например,

2 = 1×21+ 0×20= 102; 5 = 1×22+ 0×21+ 1×20= 1012; 256 = 1×28= 1000000002.

При работе с числами различают:

1) ***целые***: ±*n*;

2) ***вещественные***:

- с фиксированной десятичной точкой: ±*n*.*m*;

- с плавающей десятичной точкой (экспоненциальная форма): ±*n*.*m*E±*p*, где *n*, *m* - целая и дробная части числа, *р* - порядок; ±0.xxxE±p - нормализованный вид.

С увеличением числа количество разрядов для его представления в двоичной системе резко возрастает, поэтому для размещения большого числа выделяется несколько подряд расположенных однобайтных ячеек. В этом случае адресом такой расширенной ячейки является адрес первого байта. Один бит такой ячейки выделяется под знак числа. Числа, размещенные таким образом - целые.

Для хранения вещественных чисел их предварительно приводят к нормализованному виду. Например, 35,6 = 0.356×10+2, где 0.356 – ман­тисса, +2 – порядок. После этого переводят порядок и мантиссу в двоичную систему. Такое число запоминается в комбинированной ячейке, один байт которой содержит порядок, несколько других содержат мантиссу. Числа, размещенные таким образом - вещественные.

Программа – это последовательность команд (инструкций), которые помещаются в памяти и выполняются процессором в указанном порядке.   
  
Команда размещается в комбинированной ячейке следующим образом. Первый байт содержит код операции (КОП) (например + или – или \*), которую необходимо выполнить над содержимым ячеек памяти. В одной, двух или трех ячейках (операндах команды) по 2 или 4 байта содержатся адреса ячеек (А1, А2, А3), над которыми нужно выполнить указанную операцию. Номер первого байта команды называется ее адресом. Последовательность из этих команд называется программой в машинных кодах (рис. 2).

**6.Программные модули**

Про­грам­мист пишет программу на ***языке высокого уровня***, т.е. наиболее удобном для записи алгоритма решения определенного класса задач. Исходный текст программы, введенный с помощью клавиатуры в память компьютера - ***исходный модуль*** (sourse code, в языке Си имеет расширение \*.cpp).

**Транслятор** - программа, осуществляющая перевод текстов с одного языка на другой, т.е. транслятор переводит программу с входного языка системы программирования на машинный язык ЭВМ, на которой функционирует данная система или будет функционировать разрабатываемая программа; либо на промежуточный язык программирования, уже реализованный или подлежащий реализации. Одной из разновидностей транслятора является ***компилятор***, обеспечивающий перевод программ с языка высокого уровня (приближенного к человеку) на язык более низкого уровня (близкий к ЭВМ), или машинозависимый язык.

Интерпретатор представляет собой программный продукт, выполняющий созданную программу путем одновременного анализа и реализации предписанных действий. При использовании интерпретатора отсутствует разделение на две стадии - перевод и выполнение.   
Большинство трансляторов языка Си, с которыми мы будем работать - компиляторы.   
Результат обработки исходного модуля компилятором - объектный модуль (object code, в языке Си имеет расширение \*.obj). Он не может быть выполнен, т.е. это незавершенный вариант машинной программы, т.к., например, к нему должны быть присоединены модули стандартных библиотек. Здесь компилятор (compiler) - вид транслятора, представляющего программу-переводчик исходного модуля в язык машинных команд.   
Исполняемый (абсолютный, загрузочный) модуль создает вторая специальная программа - «компоновщик». Ее еще называют редактором связей (Linker). Она и создает модуль, пригодный для выполнения на основе одного или нескольких объектных модулей.   
Загрузочный модуль (Load module, расширение \*.exe) – это программный модуль, представленный в форме, пригодной для загрузки его в память и выполнения.

7. **Этапы решения задач на компьютере:**

**1. Постановка задачи:**

• сбор информации о задаче;

• формулировка условия задачи;

• определение конечных целей решения задачи;

• определение формы выдачи результатов;

• описание данных (их типов, диапазонов величин, структуры и т. п.).

**2. Анализ и исследование задачи, модели:**

• анализ существующих аналогов;

• анализ технических и программных средств;

• разработка математической модели;

• разработка структур данных.

**3. Разработка алгоритма:**

• выбор метода проектирования алгоритма;

• выбор формы записи алгоритма (блок-схемы, псевдокод и др.);

• выбор тестов и метода тестирования;

• проектирование алгоритма.

**4. Программирование:**

• выбор языка программирования;

• уточнение способов организации данных;

• запись алгоритма на выбранном языке

программирования.

**5. Тестирование и отладка:**

• синтаксическая отладка;

• отладка семантики и логической структуры;

• тестовые расчеты и анализ результатов тестирования;

• совершенствование программы.

**6. Анализ результатов решения задачи и уточнение в случае необходимости математической модели с повторным выполнением этапов 2-5.**

**7. Сопровождение программы:**

• доработка программы для решения конкретных задач;

• составление документации к решенной задаче, к математической модели, к алгоритму, к программе, к набору тестов, к использованию.

**Алгоритм обладает следующими свойствами:**

1. Дискретность. Это свойство состоит в том, что алгоритм должен представлять процесс решения задачи как последовательное выполнение простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, т.е. преобразование исходных данных в результат осуществляется во времени дискретно.

2. Определенность. Каждое правило алгоритма должно быть четким, однозначным.

3. Результативность. Алгоритм должен приводить к решению за конечное число шагов.

4. Массовость. Алгоритм решения задачи разрабатывается в общем виде, т.е. он должен быть применим для некоторого класса задач, различающихся лишь исходными данными.

5. Правильность. Алгоритм правильный, если его выполнение дает правильные результаты решения поставленной задачи.

**Программа -**последовательность инструкций, предназначенная для исполнения устройством управления вычислительной машины.

**Тестирование** - процесс выполнения программ с целью обнаружения факта наличия ошибок.

8. Алгоритмом называется точная инструкция исполнителю в понятной для него форме, определяющая процесс достижения поставленной цели на основе имеющихся исходных данных за конечное число шагов.

 Основными свойствами алгоритмов являются:

 1. Универсальность (массовость) - применимость алгоритма к различным наборам исходных данных.

 2. Дискретность - процесс решения задачи по алгоритму разбит на отдельные действия.

 3. Однозначность - правила и порядок выполнения действий алгоритма имеют единственное толкование.

 4. Конечность - каждое из действий и весь алгоритм в целом обязательно завершаются.

 5. Результативность - по завершении выполнения алгоритма обязательно получается конечный результат.

 6. Выполнимость - результата алгоритма достигается за конечное число шагов.

 Алгоритм считается правильным, если его выполнение дает правильный результат. Соответственно алгоритм содержит ошибки, если можно указать такие допустимые исходные данные или условия, при которых выполнение алгоритма либо не завершится вообще, либо не будет получено никаких результатов, либо полученные результаты окажутся неправильными.

 Выделяют три крупных класса алгоритмов:

 - вычислительные алгоритмы, работающие со сравнительно простыми видами данных, такими как числа и матрицы, хотя сам процесс вычисления может быть долгим и сложным;

 - информационные алгоритмы, представляющие собой набор сравнительно простых процедур, работающих с большими объемами информации (алгоритмы баз данных);

 - управляющие алгоритмы, генерирующие различные управляющие воздействия на основе данных, полученных от внешних процессов, которыми алгоритмы управляют.

**Способы записи алгоритмов**Для записи алгоритмов используют самые разнообразные средства. Выбор средства определяется типом исполняемого алгоритма. Выделяют следующие основные способы записи алгоритмов:  
- вербальный, когда алгоритм описывается на человеческом языке;  
- символьный, когда алгоритм описывается с помощью набора символов;  
- графический, когда алгоритм описывается с помощью набора графических изображений.  
Общепринятыми способами записи являются графическая запись с помощью блок-схем и символьная запись с помощью какого-либо алгоритмического языка.  
Описание алгоритма с помощью блок схем осуществляется рисованием последовательности геометрических фигур, каждая из которых подразумевает выполнение определенного действия алгоритма. Порядок выполнения действий указывается стрелками. Написание алгоритмов с помощью блок-схем регламентируется ГОСТом.

В зависимости от последовательности выполнения действий в алгоритме выделяют алгоритмы линейной, разветвленной и циклической структуры.

9. **Под алгоритмом понимается**набор правил, указывающих, какие действия и в каком порядке надо выполнять, чтобы за конечное число шагов решить задачу.

Примеры алгоритмов: решение алгебраического уравнения, алгоритм перехода улицы и т.д.

**Свойства алгоритмов:**

1. **Точность-**на каждом определенном шаге алгоритма точно известно, что нада делать.
2. **Искренность-** каждый шаг алгоритма должен указывать только одно конкретное действие и исполнитель должен выполнить его целиком.
3. **Массовость-** с помощью одного и того же алгоритма можно решать однотипные задачи и делать это неоднократно.
4. **Результативность-** Исполнение алгоритма сводится к выполнению конечного числа действий и всегда это неоднократно.<http://csaa.ru/osnovnye-principy-razrabotki-i-analiza-algoritmov/>

11. Типовые алгоритмические структуры. В рамках структурного программирования задачи, имеющие алгоритмическое решение, могут быть описаны с использованием следующих алгоритмических структур:

* **Следование**. Предполагает последовательное выполнение команд сверху вниз. Если алгоритм состоит только из структур следования, то он является линейным.
* **Ветвление**. Выполнение программы идет по одной из двух, нескольких или множества ветвей. Выбор ветви зависит от условия на входе ветвления и поступивших сюда данных.
* **Цикл**. Предполагает возможность многократного повторения определенных действий. Количество повторений зависит от условия цикла.
* **Функция (подпрограмма)**. Команды, отделенные от основной программы, выполняются лишь в случае их вызова из основной программы (из любого ее места). Одна и та же функция может вызываться из основной программы сколь угодно раз.

12.Алфавит, лексемы, разделители   
Алфавит. Алфавит любого языка программирования — это набор допустимых символов этого языка. В языке С++ используются следующие символы:   
• • прописные и строчные латинские буквы: АВСОЕЕСНПКЕМЫОРОЯЗТиУХУХУг; аЬсбеГ§Ь1]к1тпоряг81иушху2;   
• • десятичные цифры: 012345678 9;   
• • знак подчеркивания;   
• • буквы русского алфавита (для комментариев и вывода сообщений на экран);   
• • специальные символы:   
Лексемы языка С++. Лексема — это логически выделенная единица языка, воспринимаемая как единое целое компилятором и программистом.   
Лексемы бывают следующих видов:

• идентификаторы;   
• ключевые слова; • константы;   
• знаки операций;   
• разделители.

13. Идентификаторы (имена) служат для обозначения объектов программы (переменных, констант, меток, функций и т. д.).   
Ключевые (служебные) слова — это такие лексемы, которые используются системой программирования для своих специальных целей (имена операторов, директив препроцессора и т. д.). Препроцессор — специальная служебная программа, работающая перед компилятором и используемая для вставки библиотечных файлов, определения констант и т. д. Директива — это команда препроцессору выполнить то или иное действие.   
Константа — это лексема, представляющая изображение фиксированного числового, строкового или символьного значения, например: 100, 3.14159, «Здравствуй, мир!», 'У.   
Знаки операций — это лексемы, используемые при вычислении выражений, которые определяют порядок и правила вычисления значения и могут содержать имена переменных, константы, знаки операций, имена функций и скобки для определения порядка вычислений.   
Разделители используются для отделения друг от друга (разделения) лексем языка. Например, запятая разделяет элементы списка.   
При анализе выражения или оператора компилятор воспринимает их как последовательность лексем (табл. 3.2).   
14.

15. Программа на языке Си состоит из одной или более подпрограмм, называемых ***функциями***.  
  
Язык Си является блочно-структурированным. Каждый блок заключается в фигурные скобки {}.  
  
Основным блоком в программе консольного приложения на языке Си является главная функция, имеющая имя main().  
  
Каждое действие в языке Си заканчивается символом «точка с запятой» — ;. В качестве действия может выступать вызов функции или осуществление некоторых операций.  
  
***Имя функции*** — это коллективное имя группы описаний и операторов,  
заключенных в блок (фигурные скобки). За именем функции в круглых скобках указываются параметры функции.

## Комментарии в языке Си

В языке Си  для комментариев используются символы  
  
**/\* — начало комментария;  
\*/ — конец комментария.**  
  
Вся последовательность, заключенная между этими символами, является комментарием.  
  
Это удобно для написания **многострочных комментариев**:

int a;   /\* целая  
переменная \*/

Многострочные комментарии также удобно использовать при отладке для сокрытия от выполнения части кода.  
  
В дополнение к этому, для написания коротких комментариев могут использоваться символы //. При этом комментарием является все, что расположено после символов // и до конца строки:

float b; // вещественная переменная

## Главная функция

При выполнении консольного приложения, написанного на языке Си, операционная система компьютера передаёт управление функции с именем main(). Функцию main() нельзя вызывать из других функций программы, она является управляющей.  
  
Следующие за именем функции круглые скобки предназначены для указания параметров (аргументов), которые передаются в функцию при обращении к ней. В данном случае операционная система не передаёт в функцию main() никаких аргументов, поэтому список аргументов в круглых скобках пустой.  
  
Главную функцию можно записать по-разному:

* int main()
* void main().

Перед именем функции указывается тип возвращаемого значения. При обращении к главной функции значение возвращается операционной системе. Последняя запись не будет возвращать значения. Однако void main() — не совсем корректная запись, так как сообщает компилятору, что функция main() не возвращает никакого значения.  
  
При этом запись int main() сообщает компилятору о возвращении целочисленного значения, которое необходимо операционной системе и сообщает ей о том, что программа завершилась корректно. Если же это значение не возвращено, то операционная система понимает, что программа завершилась в аварийном режиме.  
Для возврата целочисленного значения перед завершением функции добавляется строка

return 0; // вещественная переменная

В фигурные скобки заключены описания и операторы.  
  
В общем случае программа может содержать несколько функций. Каждая функция имеет список передаваемых в нее параметров, указанный в круглых скобках, и набор операций, заключенных в блок, ограниченный фигурными скобками.

## 16. Типы данных

В языке С++ все переменные имеют определенный тип данных. Например, переменная, имеющая целочисленный тип не может содержать ничего кроме целых чисел, а переменная с плавающей точкой — только дробные числа.

**Тип данных** присваивается переменной при ее объявлении или инициализации. Ниже приведены основные типы данных языка C++, которые нам понадобятся.

### Основные типы данных в C++

* **int** — целочисленный тип данных.
* **float** — тип данных с плавающей запятой.
* **double** — тип данных с плавающей запятой двойной точности.
* **char** — символьный тип данных.
* **bool** — логический тип данных.

## 17.Объявление переменной

Объявление переменной в C++ происходит таким образом: сначала указывается тип данных для этой переменной а затем название этой переменной.

### Пример объявления переменных

**int** a; *// объявление переменной a целого типа.*

**float** b; *// объявление переменной b типа данных с плавающей запятой.*

**double** c = 14.2; *// инициализация переменной типа double.*

**char** d = 's'; *// инициализация переменной типа char.*

**bool** k = true; *// инициализация логической переменной k.*

* Заметьте, что в C++ **оператор присваивания** **(=)** — не является знаком равенства и не может использоваться для сравнения значений. Оператор равенства записывается как «двойное равно» — ==.

18. Рассмотрим, какие преобразования применяет компилятор при арифметических операциях:

1. Если один из операндов имеет тип **long double**, то второй операнд тоже будет преобразован в тип **long double**
2. Если предыдущий пункт не выполняется и если один из операндов имеет тип **double**, то второй операнд тоже будет преобразован к типу **double**
3. Если предыдущий пункт не выполняется и если один из операндов имеет тип **float**, то второй операнд тоже будет преобразован к типу **float**
4. Если предыдущий пункт не выполняется и если один из операндов имеет тип **unsigned long int**, то второй операнд тоже будет преобразован к типу **unsigned long int**
5. Если предыдущий пункт не выполняется и если один из операндов имеет тип **long**, то второй операнд тоже будет преобразован к типу **long**
6. Если предыдущий пункт не выполняется и если один из операндов имеет тип **unsigned**, то второй операнд тоже будет преобразован к типу **unsigned**
7. Если предыдущий пункт не выполняется то оба операнда приводятся к типу **int**

### Операция преобразования

С помощью специальной операции преобразования мы можем явным образом привести данные к нужному типу. Например:

В выражении int c = a / b; результат деления будет целочисленный - 2, при котором дробная часть будет отброшена, так как оба операнда операции представляют целые числа.

В выражении double d = a / b; результат деления будет представлять вещественное число - 2.00000, но так как оба операнда являются целыми числами, то опять же результат операции будет представлять целое число 2, и только поле выполнения деления произойдет присвоение результата переменной d с приведением значения 2 от типа int к типу double.

В выражении double e = (double)a / (double)b применяется явное преобразование данных к типу double, поэтому и результат деления будет представлять вещественное число - 2.50000.

Для выполнения операции приведении в скобках указывается тот тип, к которому надо привести значения

19.***Константа*** — это ограниченная последовательность символов алфавита языка, представляющая собой изображение фиксированного (неизменяемого) объекта.  
  
Константы бывают числовые, символьные и строковые. Числовые константы делятся на целочисленные и вещественные.

## Целочисленные константы

Целочисленные данные в языке Си могут быть представлены в одной из следующих [систем счисления](https://prog-cpp.ru/comp-data/):

Десятичные Восьмеричные Шестнадцатеричные

**Вещественные константы**

Константа с плавающей точкой (вещественная константа) всегда представляется числом с плавающей точкой двойной точности, т. е. как имеющая тип double, и состоит из следующих частей:

* целой части — последовательности цифр;
* точки — разделителя целой и дробной части;
* дробной части — последовательности цифр;
* символа экспоненты е или E;
* экспоненты в виде целой константы (может быть со знаком).

Любая часть (но не обе сразу) из нижеследующих пар может быть опущена:

* целая или дробная часть;
* точка или символ е (Е) и экспонента в виде целой константы.

## Символьные константы

Символьная константа — это один символ, например: ‘z’. В качестве символьных констант также могут использоваться управляющие коды, не имеющие графического представления. При этом код управляющего символа начинается с символа ‘\’ (обратный слеш).

## Как правило, нажатие клавиши **Enter** генерирует сразу два управляющих символа — перевод строки (0x0A) и возврат каретки (0x0D). Все символьные константы имеют тип char и занимают в памяти 1 байт. Значением символьной константы является числовое значение её внутреннего кода.

## Строковые константы

Строковая константа — это последовательность символов, заключенная в кавычки, например:

«Это строковая константа»

Кавычки не входят в строку, а лишь ограничивают её. Технически строковая константа представляет собой массив символов, и по этому признаку может быть отнесена к разряду сложных объектов языка Си.  
  
В конце каждой строковой константы компилятор помещает ‘\0’ (нуль-символ), чтобы программе было возможно определить конец строки. Такое представление означает, что размер строковой константы не ограничен каким-либо пределом, но для определения длины строковой константы её нужно полностью просмотреть.  
  
Поскольку строковая константа состоит из символов, то она имеет тип char. Количество ячеек памяти, необходимое для хранения строковой константы на 1 больше количества символов в ней (1 байт используется для хранения нуль-символа).  
  
Символьная константа ‘x’ и строка из одного символа «x» — не одно и то же. Символьная константа — это символ, используемый для числового представления буквы x, а строковая константа «x» содержит символ ‘x’ и нуль-символ ‘\0’ и занимает в памяти 2 байта. Если в программе строковые константы записаны одна за другой через разделители, то при выполнении программы они будут размещаться в последовательных ячейках памяти.

20. Операторы производят свои действия над переменными, константами и выражениями. В предыдущей теме были рассмотрены переменные, которые представляют собой именованные ячейки памяти, предназначенные для хранения и обработки данных. Для работы с переменными и константами предназначены операции.

*Операция* – это символ, представляющий собой некоторую операцию, производимую с данными. В языке Си существуют несколько видов операций: арифметические операции, операции сравнения, битовые и логические операции. Все эти операции будут рассмотрены ниже.

Переменные и константы, над которыми производится операция, называются *операндами*. Операция, которая воздействует на два операнда, называется *бинарной*, а операция, воздействующая на один операнд – *унарной*. В математике также используется термин *n-арная* операция, если производится действие над ***n*** операндами.

Большинство операций являются бинарными: сложение, умножение, деление, различные варианты операций сравнения и т.д. Унарной операцией является «минут», например, ***-7***.

Комбинация операций и операндов называется *выражением*. Простейшее выражение может состоять только из одного операнда, а выражение, являющееся частью другого, называется *подвыражением*.

Приоритет операций.

Всем наверно с детства знакома задачка с подковыркой, когда опоненту говорят сколько будет два плюс два умножить на два. Многие забывают что в математике сначала надо умножать, а потом только прибавлять, и потому ответ не 8, а 6.

В языке программирования Си, тоже есть определенны порядок выполнения арифметических операций. Он очень похож на тот, который мы используем в обычной жизни.

Операции в скобках

Функции (типа sqrt, cos и другие)

Умножение, деление, остаток от деления (слева направо, как в обычной жизни)

Сложение вычитание (слева направо, как в обычной жизни)

Для примера рассмотрим следующее выражение записанное на языке Си:

Приоритет арифметических операций

Над каждым действием вы видите каким оно выполняется по счету.

21. Наиболее часто используемым классом операций языка Си являются арифметические операции. К ним относятся операции сложения, вычитания, умножения, деления, взятия остатка от деления и операция изменения знака.

|  |  |  |
| --- | --- | --- |
| + | Сложение | z = x+y |
| - | Вычитание | z = x-y |
| - | Изменение знака | z = -x |
| \* | Умножение | z = x\*y |
| / | Деление | z = x/y |
| % | Остаток от деления | z = x%y |

Для сокращённой записи выражений в языке программирования С++ есть специальные операции, которые называются операциями присваивания. Рассмотрим фрагмент кода, с использованием операции присваивания.

|  |  |
| --- | --- |
|  | int value = 256;  value = value + 256; // обычное выражение с использованием двух операций:  = и +  value += 256; // сокращённое эквивалентное выражение с использованием операции  присваивания |

В **строке 2** переменной value присваивается значение 512, полученное в результате суммы значения содержащегося в переменной value с числом 256. В строке 3 выражение выполняет аналогичную операцию, что и в строке 2, но выражение записано в упрощённом виде. В этом выражении присутствует операция присваивания со знаком плюс +=. Таким образом, операция += суммирует значение переменной value co значением, которое находится правее:  256,  и присваивает результат суммы этой же переменной. Как видно из примера оператор в строке 3 короче оператора в строке 2, хоть и выполняет аналогичную операцию. Так что, если некоторую переменную нужно изменить, то рекомендуется использовать операции присваивания.

В С++ существует  пять операций присваивания, не считая основную операцию присваивания: =.

1. += операция присваивания-сложения;
2. -= операция присваивания-вычитания;
3. \*= операция присваивания-умножения;
4. /= операция присваивания-деления;
5. %= операция присваивания-остатка от деления;

Договоримся называть операции присваивания через дефис,  чтобы было понятно о какой именно операции идёт речь. В таблице 1 наглядно показаны

примеры использования операторов присваивания в языке программирования С++.

22. Операции сравнения предназначены для определения отношения между двумя переменными или между переменной и константой. Например, ***a<b*** или ***a>10***. Результатом операции сравнения является значение типа bool, т.е. true или false. Операции сравнения отвечают «Да» или «Нет» на вопрос «Связаны ли элементы указанным отношением?»

|  |  |  |
| --- | --- | --- |
| **Опера-ция** | **Описание** | **Пример** |
| < | Меньше | x<y< p=""></y<> |
| > | Больше | x>y |
| <= | Меньше либо равно | x<=y |
| >= | Больше либо равно | x>=y |
| == | Равно | x==y |
| != | Не равно | x!=y |

**Логические операции**

Логические операции служат для соединения выражений, которые содержат операции сравнения.

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Пример** |
| && | Логическое «и» | 10<20<="" p=""> |
| || | Логическое «или» | X%3==0 || y%7==0 |
| ! | Логическое «не» | !(x==10) |

Результаты логических выражений определяются согласно так называемой *таблице истинности*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **y** | **x&&y** | **x||y** | **!x** |
| false | false | false | false | true |
| false | true | false | true | true |
| true | false | false | true | false |
| true | true | true | true | false |

Значения true и false являются своеобразными константами, созданными для того, чтобы программисту-человеку было проще с ним управляться, однако с точки зрения компьютера эти константы равны ***1*** и ***0*** соответственно.

Оператор «запятая» используется для связки нескольких выражений. Левая сторона оператора «запятая» всегда вычисляется как void (то есть не выдающее значения). Это означает, что значение выражения, находящегося с правой стороны, станет значением разделенного запятыми выражения. Например:  
  
х = (у = 3, у + 1);  
  
Сначала присваивается 3 переменной у, а затем 4. переменной х. Скобки необходимы, поскольку оператор «запятая» имеет более низкий приоритет по сравнению с оператором присваивания.

Оператор «запятая» вызывает выполнение последовательности действий. Когда он используется с правой стороны оператора присваивания, то присваиваться будет значение последнего выражения, стоящего в разделенном запятыми списке. Ниже приведен еще один пример:  
  
у = 10;  
  
х = (у = у - 5, 25 / у);  
  
После выполнения х получит значение 5, поскольку исходным значением у было 10, а затем оно уменьшилось на 5. Затем 25 поделили на полученное 5 и получили результат.

Об операторе «запятая» следует думать как об обычном слове «и» в нормальном русском языке, когда оно используется в выражении «сделай это, и это, и это».

23.Битовые операции — это тестирование, установка или сдвиг битов в байте или слове, которые соответствуют стандартным типам языка С char и int. Битовые операторы не могут использоваться с float, double, long double, void и другими сложными типами. Таблица содержит имеющиеся операторы.

| **Оператор** | **Действие** |
| --- | --- |
| & | И |
| | | ИЛИ |
| ^ | Исключающее ИЛИ |
| ~ | Дополнение |
| >> | Сдвиг вправо |
| << | Сдвиг влево |
| Таблица: Битовые операторы | |

Битовые операторы И, ИЛИ, НЕ используют ту же таблицу истинности, что и их логические эквиваленты, за тем исключением, что они работают побитно. Исключающее ИЛИ имеет следующую таблицу истинности:

|  |  |  |
| --- | --- | --- |
| р | q | p^q |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Как следует из таблицы, исключающее ИЛИ выдает истину, если только один из операндов истинен. В противном случае получается ложь.

Битовые операторы наиболее часто применяются при разработке драйверов устройств, например программ для модемов, дисков и принтеров, поскольку битовые операторы могут использоваться для выключения некоторых битов, например четности. (Бит четности используется для подтверждения того, что остальные биты в байте не изменялись. Он, как правило, является старшим битом в байте.)

Операторы сдвига >> и << сдвигают биты в переменной вправо и влево на указанное число. Общий вид оператора сдвига вправо:

*переменная >> число сдвигов*

а общий вид оператора сдвига влево:

*переменная << число сдвигов*

Помните, что сдвиг — это не то же самое, что и вращение, то есть биты, сдвигающиеся на один конец, не появляются с другого. Сдвинутые биты теряются, а с другого конца появляются нули. В том случае, если вправо сдвигается отрицательное число, слева появляются единицы (поддерживается знаковый бит).

Операции битового сдвига могут быть полезны при декодировании информации от внешних устройств  и для чтения информации о статусе. Операторы битового сдвига могут также использоваться для выполнения быстрого умножения и деления целых чисел. Сдвиг влево равносилен умножению на 2, а сдвиг вправо -  делению на 2, как показано в таблице.

|  | **Битовое представление х после выполнения каждого оператора** | **Значение х** |
| --- | --- | --- |
| char х; |  |  |
| x = 7; x = x << 1; x = x << 3; x =  x << 2; х = х >> 1; x = x >> 2; | 00000111 00001110 01110000 11000000 01100000 00011000 | 7 14 112 192 96 24 |
| Каждый сдвиг влево приводит к умножению на 2. Обратим внимание, что после сдвига х << 2 информация теряется, поскольку биты сдвигаются за конец байта. Каждый сдвиг вправо приводит к делению на 2. Обратим внимание, что деление не вернуло потерянные биты. | | |

Оператор дополнение, ~, инвертирует состояние каждого бита указанной переменной, то есть 1 устанавливается в 0, а 0 — в 1.

24. Математические вычисления не ограничиваются лишь арифметическими действиями. Кроме них, можно ещё встретить корни, модули, логарифмы, тригонометрические функции и пр. Научимся же использовать подобные функции в своих программах.

Для использования математических функций нужно подключить заголовочный файл **math.h**. В ней определено много различных функций, но мы пока рассмотрим следующие:

Некоторые математические функции

**fabs(x)** модуль числа x  
**sqrt(x)** квадратный корень из числа x  
**sin(x)** синус числа x (х в радианах)  
**cos(x)** косинус числа x (х в радианах)  
**pow(x, y)** вычисление xy  
**exp(x)** вычисление ex  
**log(x)** натуральный логарифм числа x  
**log10(x)** десятичный логарифм числа x

Два важных момента.

* Все функции возвращают значение типа **double**.
* Параметры функций – вещественные числа(**double**), но можно передавать и целые числа. При этом произойдёт неявное преобразование типа. Компилятор из целого числа, например 3, сделает вещественное 3.0.

## stdlib.h

**stdlib.h** — заголовок (заголовочный файл) **стандартной библиотеки общего назначения** языка [Си](https://dic.academic.ru/dic.nsf/ruwiki/7630), который содержит в себе функции, занимающиеся выделением памяти, контроль процесса выполнения программы, преобразования типов и другие. Заголовок вполне совместим с [C++](https://dic.academic.ru/dic.nsf/ruwiki/2465) и в C++ известен как cstdlib. Название «stdlib» расшифровывается как «standard library» (стандартная библиотека).

Члены stdlib.h можно разделить на следующие категории: преобразования типов, управление памятью, контроль процесса, сортировка и поиск, математика.

|  |  |
| --- | --- |
| **Имя** | **Описание** |
| **Преобразование типов** | |
| [atof](https://dic.academic.ru/dic.nsf/ruwiki/1350414) | строка в число двойной точности (НЕ float) |
| [atoi](https://dic.academic.ru/dic.nsf/ruwiki/1350415) | строка в целое число |
| atol | строка в длинное целое число |
| [strtod](https://dic.academic.ru/dic.nsf/ruwiki/1358264) | строка в число двойной точности (double) |
| strtol | строка в длинное целое число |
| strtoul | строка в беззнаковое длинное целое число (unsigned long int) |
| **Генерация псевдослучайных последовательностей** | |
| rand | генерирует псевдослучайное значение |
| srand | устанавливает начальное значение генератора псевдослучайных чисел |
| **Выделение и освобождение памяти** | |
| [malloc](https://dic.academic.ru/dic.nsf/ruwiki/175292) [calloc](https://dic.academic.ru/dic.nsf/ruwiki/719225) realloc | выделяет память из кучи |
| free | освобождает память обратно в кучу |
| **Контроль процесса выполнения программы** | |
| abort | некорректное завершение выполнения |
| atexit | регистрирует [обратный вызов](https://dic.academic.ru/dic.nsf/ruwiki/581734) функции для выхода из программы |
| exit | завершает выполнение программы |
| getenv | извлекает переменные окружения |
| system | выполняет внешнюю команду |
| **Сортировка и поиск** | |
| bsearch | [двоичный поиск](https://dic.academic.ru/dic.nsf/ruwiki/83374) в массиве |
| qsort | сортировка массива |
| **Математика** | |
| abs labs | [абсолютная величина](https://dic.academic.ru/dic.nsf/ruwiki/23718) |
| div ldiv | деление целых чисел |
| **Многобайтовые операции/ широкие символы** | |
| mblen | размер многобайтовых символов[[1]](https://dic.academic.ru/dic.nsf/ruwiki/1358192#cite_note-0) |
| mbtowc, wctomb, mbstowcs, wcstombs | преобразование многобайтовых и [широких](https://dic.academic.ru/dic.nsf/ruwiki/754412) символов[[2]](https://dic.academic.ru/dic.nsf/ruwiki/1358192#cite_note-1) |

## 25. Функция стандартного вывода printf()

Функция printf() является функцией стандартного вывода. С помощью этой функции можно вывести на экран монитора строку символов, число, значение переменной...

Функция printf() имеет прототип в файле stdio.h  
int printf(char \*управляющая строка, ...);

В случае успеха функция printf() возвращает число выведенных символов.

Управляющая строка содержит два типа информации: символы, которые непосредственно выводятся на экран, и спецификаторы формата, определяющие, как выводить аргументы.

Функция printf() это функция форматированного вывода. Это означает, что в параметрах функции необходимо указать формат данных, которые будут выводиться. Формат данных указывается спецификаторами формата. Спецификатор формата начинается с символа % за которым следует код формата.

**Спецификаторы формата:**

|  |  |
| --- | --- |
| %с | символ |
| %d | целое десятичное число |
| %i | целое десятичное число |
| %e | десятичное число в виде x.xx e+xx |
| %E | десятичное число в виде x.xx E+xx |
| %f | десятичное число с плавающей запятой xx.xxxx |
| %F | десятичное число с плавающей запятой xx.xxxx |
| %g | %f или %e, что короче |
| %G | %F или %E, что короче |
| %o | восьмеричное число |
| %s | строка символов |
| %u | беззнаковое десятичное число |
| %x | шестнадцатеричное число |
| %X | шестнадцатеричное число |
| %% | символ % |
| %p | указатель |
| %n | указатель |

Кроме того, к командам формата могут быть применены модификаторы l и h.

|  |  |
| --- | --- |
| %ld | печать long int |
| %hu | печать short unsigned |
| %Lf | печать long double |

В спецификаторе формата, после символа % может быть указана точность (число цифр после запятой). Точность задаётся следующим образом: %.n<код формата>. Где n - число цифр после запятой, а <код формата> - один из кодов приведённых выше.

Вы находитесь здесь: [Главная страница](http://www.realcoding.net/) → [C/С++](http://www.realcoding.net/articles/c/s)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

# Функции стандартного ввода/вывода (printf, scanf)

**Автор: Бардин П.Б.  
[http://proger.ru](http://proger.ru/)**

## Введение

Стандартная библиотека C/C++ включает ряд функций для чтения и записи на консоли (клавиатура и монитор). Эти функции читают и пишут данные как простой поток символов.

Понятие потока (stream), использованное в программировании, тесно связано с обычным, бытовым пониманием этого слова. Поток ввода можно сравнить с трубой, по которой вода (информация) поступает в бассейн (память компьютера), поток вывода - с трубой, по которой вода выходит из бассейна. Важной особенностью этой трубы является то, что в каждый момент времени данные могут двигаться только в одном направлении. Даже если одна и та же труба используется для ввода и вывода, это не может происходить одновременно: для переключения направления потока его нужно остановить, выполнить некие действия и лишь затем направить поток в обратном направлении. Другой особенностью потока является то, что он почти никогда не иссякает. Иногда он высыхает, но этот период не может быть долгим, если система функционирует нормально.

## Функция стандартного вывода printf()

Функция printf() является функцией стандартного вывода. С помощью этой функции можно вывести на экран монитора строку символов, число, значение переменной...

Функция printf() имеет прототип в файле stdio.h  
int printf(char \*управляющая строка, ...);

В случае успеха функция printf() возвращает число выведенных символов.

Управляющая строка содержит два типа информации: символы, которые непосредственно выводятся на экран, и спецификаторы формата, определяющие, как выводить аргументы.

Функция printf() это функция форматированного вывода. Это означает, что в параметрах функции необходимо указать формат данных, которые будут выводиться. Формат данных указывается спецификаторами формата. Спецификатор формата начинается с символа % за которым следует код формата.

**Спецификаторы формата:**

|  |  |
| --- | --- |
| %с | символ |
| %d | целое десятичное число |
| %i | целое десятичное число |
| %e | десятичное число в виде x.xx e+xx |
| %E | десятичное число в виде x.xx E+xx |
| %f | десятичное число с плавающей запятой xx.xxxx |
| %F | десятичное число с плавающей запятой xx.xxxx |
| %g | %f или %e, что короче |
| %G | %F или %E, что короче |
| %o | восьмеричное число |
| %s | строка символов |
| %u | беззнаковое десятичное число |
| %x | шестнадцатеричное число |
| %X | шестнадцатеричное число |
| %% | символ % |
| %p | указатель |
| %n | указатель |

Кроме того, к командам формата могут быть применены модификаторы l и h.

|  |  |
| --- | --- |
| %ld | печать long int |
| %hu | печать short unsigned |
| %Lf | печать long double |

В спецификаторе формата, после символа % может быть указана точность (число цифр после запятой). Точность задаётся следующим образом: %.n<код формата>. Где n - число цифр после запятой, а <код формата> - один из кодов приведённых выше.

Например, если у нас есть переменная x=10.3563 типа float и мы хотим вывести её значение с точностью до 3-х цифр после запятой, то мы должны написать:

printf("Переменная x = %.3f",x);

Результат:  
Переменная x = 10.356

Вы также можете указать минимальную ширину поля отводимого для печати. Если строка или число больше указанной ширины поля, то строка или число печатается полностью.  
   
Например, если вы напишите:

printf("%5d",20);

то результат будет следующим:  
   20

Обратите внимание на то, что число 20 напечаталось не с самого начала строки. Если вы хотите чтобы неиспользованные места поля заполнялись нулями, то нужно поставить перед шириной поля символ 0.  
  
Например:

printf("%05d",20);

Результат:  
00020

Кроме спецификаторов формата данных в управляющей строке могут находиться управляющие символы:

|  |  |
| --- | --- |
| \b | BS, забой |
| \f | Новая страница, перевод страницы |
| \n | Новая строка, перевод строки |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |
| \" | Двойная кавычка |
| \' | Апостроф |
| \\ | Обратная косая черта |
| \0 | Нулевой символ, нулевой байт |
| \a | Сигнал |
| \N | Восьмеричная константа |
| \xN | Шестнадцатеричная константа |
| \? | Знак вопроса |

Чаще всего вы будете использовать символ \n. С помощью этого управляющего символа вы сможете переходить на новую строку. Посмотрите примеры программ и вы всё поймёте.

## 26. Функция стандартного ввода scanf()

Функция scanf() - функция форматированного ввода. С её помощью вы можете вводить данные со стандартного устройства ввода (клавиатуры). Вводимыми данными могут быть целые числа, числа с плавающей запятой, символы, строки и указатели.

Функция scanf() имеет следующий прототип в файле stdio.h:  
int scanf(char \*управляющая строка);

Функция возвращает число переменных которым было присвоено значение.

Управляющая строка содержит три вида символов: спецификаторы формата, пробелы и другие символы. Спецификаторы формата начинаются с символа %.

**Спецификаторы формата:**

|  |  |
| --- | --- |
| %c | чтение символа |
| %d | чтение десятичного целого |
| %i | чтение десятичного целого |
| %e | чтение числа типа float (плавающая запятая) |
| %h | чтение short int |
| %o | чтение восьмеричного числа |
| %s | чтение строки |
| %x | чтение шестнадцатеричного числа |
| %p | чтение указателя |
| %n | чтение указателя в увеличенном формате |

При вводе строки с помощью функции scanf() (спецификатор формата %s), строка вводиться до первого пробела!! т.е. если вы вводите строку "Привет мир!" с использованием функции scanf()

char str[80];        // массив на 80 символов  
scanf("%s",str);

то после ввода результирующая строка, которая будет храниться в массиве str будет состоять из одного слова "Привет". ФУНКЦИЯ ВВОДИТ СТРОКУ ДО ПЕРВОГО ПРОБЕЛА! Если вы хотите вводить строки с пробелами, то используйте функцию

char \*gets( char \*buf );

С помощью функции gets() вы сможете вводить полноценные строки. Функция gets() читает символы с клавиатуры до появления символа новой строки (\n). Сам символ новой строки появляется, когда вы нажимаете клавишу enter. Функция возвращает указатель на buf. buf - буфер (память) для вводимой строки.

Хотя gets() не входит в тему этой статьи, но всё же давайте напишем пример программы, которая позволяет ввести целую строку с клавиатуры и вывести её на экран.

#include <stdio.h>  
  
void main(void)  
{  
    char buffer[100];       // массив (буфер) для вводимой строки  
  
    gets(buffer);            // вводим строку и нажимаем enter  
    printf("%s",buffer);    // вывод введённой строки на экран  
}

Ещё одно важное замечание! Для ввода данных с помощью функции scanf(), ей в качестве параметров нужно передавать адреса переменных, а не сами переменные. Чтобы получить адрес переменной, нужно поставить перед именем переменной знак &(амперсанд). Знак & означает взятие адреса.

Что значит адрес? Попробую объяснить. В программе у нас есть переменная. Переменная хранит своё значение в памяти компьютера. Так вот адрес, который мы получаем с помощью & это адрес в памяти компьютера где храниться значение переменной.

Давайте рассмотрим пример программы, который показывает нам как использовать &

#include <stdio.h>  
  
void main(void)  
{  
    int x;  
  
    printf("Введите переменную x:");  
    scanf("%d",&x);  
    printf("Переменная x=%d",x);  
}

Теперь давайте вернёмся к управляющей строке функции scanf(). Ещё раз:

int scanf(char \*управляющая строка);

Символ пробела в управляющей строке дает команду пропустить один или более пробелов в потоке ввода. Кроме пробела может восприниматься символ табуляции или новой строки. Ненулевой символ указывает на чтение и отбрасывание этого символа.

Разделителями между двумя вводимыми числами являются символы пробела, табуляции или новой строки. Знак \* после % и перед кодом формата (спецификатором формата) дает команду прочитать данные указанного типа, но не присваивать это значение.

Например:

scanf("%d%\*c%d",&i,&j);

при вводе 50+20 присвоит переменной i значение 50, переменной j - значение 20, а символ + будет прочитан и проигнорирован.

В команде формата может быть указана наибольшая ширина поля, которая подлежит считыванию.

Например:

scanf("%5s",str);

указывает необходимость прочитать из потока ввода первые 5 символов. При вводе 1234567890ABC массив str будет содержать только 12345, остальные символы будут проигнорированы. Разделители: пробел, символ табуляции и символ новой строки - при вводе символа воспринимаются, как и все другие символы.

Если в управляющей строке встречаются какие-либо другие символы, то они предназначаются для того, чтобы определить и пропустить соответствующий символ. Поток символов 10plus20 оператором

scanf("%dplus%d",&x,&y);

присвоит переменной x значение 10, переменной y - значение 20, а символы plus пропустит, так как они встретились в управляющей строке.

Одной из мощных особенностей функции scanf() является возможность задания множества поиска (scanset). Множество поиска определяет набор символов, с которыми будут сравниваться читаемые функцией scanf() символы. Функция scanf() читает символы до тех пор, пока они встречаются в множестве поиска. Как только символ, который введен, не встретился в множестве поиска, функция scanf() переходит к следующему спецификатору формата. Множество поиска определяется списком символов, заключённых в квадратные скобки. Перед открывающей скобкой ставиться знак %. Давайте рассмотрим это на примере.

#include <stdio.h>  
  
void main(void)  
{  
    char str1[10], str2[10];  
    scanf("%[0123456789]%s", str1, str2);  
    printf("\n%s\n%s",str1,str2);  
}  
Введём набор символов:  
12345abcdefg456  
  
На экране программа выдаст:  
12345  
abcdefg456

При задании множества поиска можно также использовать символ "дефис" для задания промежутков, а также максимальную ширину поля ввода.

scanf("%10[A-Z1-5]", str1);

Можно также определить символы, которые не входят в множество поиска. Перед первым из этих символов ставиться знак ^. Множество символов различает строчные и прописные буквы.

Напомню, что при использовании функции scanf(), ей в качестве параметров нужно передавать адреса переменных. Выше был написан код:

char str[80];        // массив на 80 символов  
scanf("%s",str);

Обратите внимание на то, что перед str не стоит символ &. Это сделано потому, что str является массивом, а имя массива - str является указателем на первый элемент массива. Поэтому знак & не ставиться. Мы уже передаем функции scanf() адрес. Ну проще говоря str это адрес в памяти компьютера где будет храниться значение первого элемента массива.

# 27. 12. Синтаксис и семантика оператора выражения

Синтаксис оператора:

любое выражение, после которого поставлен символ «точка с запятой», воспринимается компилятором как отдельный оператор

Семантика оператора:

производится вычисление значения выражения, после чего полученное значение удаляется

При вычислении выражения могут возникать побочные эффекты в виде изменения значений некоторых переменных

Именно эти побочные эффекты и являются целью использования оператора-выражения

Например:

x = y + 6; x ++ < -- y; a && b <= (x \*= 4); x \* x + 8;

Пустой оператор не выполняет никаких действий и может быть исполь­зован для передачи управления в конец составного оператора. Пустой оператор записывается в виде точки с запятой.

Например, пустыми операторами будут следующие записи:

* 1. **; ;** 2) <метка>**: ;**
* **Составной оператор**
* Синтаксис:
* {
* [<*объявление*>]
* .
* .
* .
* [**<***оператор***>**]
* }
* Действие:
* Действие составного оператора заключается в последовательном выполнении содержащихся в нем операторов, за исключением тех случаев, когда какой-либо оператор явно передает управление в другое место программы.
* В начале составного оператора могут содержаться объявления (см. разделы 3.6, 3.6.2). Они служат для определения переменных, локальных для данного блока, либо для распространения на данный блок области действия глобальных объектов

28. Рассмотрим третью [алгоритмическую структуру](https://prog-cpp.ru/algorithm-structure/) — цикл.  
***Циклом*** называется блок кода, который для решения задачи требуется повторить несколько раз.  
  
Каждый цикл состоит из

* блока проверки условия повторения цикла
* тела цикла

Цикл выполняется до тех пор, пока блок проверки условия возвращает истинное значение.  
Тело цикла содержит последовательность операций, которая выполняется в случае истинного условия повторения цикла. После выполнения последней операции тела цикла снова выполняется операция проверки условия повторения цикла. Если это условие не выполняется, то будет выполнена операция, стоящая непосредственно после цикла в коде программы.  
  
В языке Си следующие виды циклов:

* while — цикл с предусловием;
* do…while — цикл с постусловием;
* for — параметрический цикл (цикл с заданным числом повторений).

## Цикл с предусловием while

Общая форма записи

while (Условие)  
{  
  БлокОпераций;  
}

Если **Условие** выполняется (выражение, проверяющее **Условие**, не равно нулю), то выполняется **БлокОпераций**, заключенный в фигурные скобки, затем **Условие** проверяется снова.  
Последовательность действий, состоящая из проверки **Условия** и выполнения **БлокаОпераций**, повторяется до тех пор, пока выражение, проверяющее **Условие**, не станет ложным (равным нулю). При этом происходит выход из цикла, и производится выполнение операции, стоящей после оператора цикла.

**Цикл с постусловием do...while**

Общая форма записи  do {  
  БлокОпераций;  
} while (Условие);

Цикл do...while — это цикл с постусловием, где истинность выражения, проверяющего **Условие** проверяется после выполнения **Блока Операций**, заключенного в фигурные скобки. Тело цикла выполняется до тех пор, пока выражение, проверяющее **Условие**, не станет ложным, то есть тело цикла с постусловием выполнится хотя бы один раз.  
  
Использовать цикл do...while лучше в тех случаях, когда должна быть выполнена хотя бы одна итерация, либо когда инициализация объектов, участвующих в проверке условия, происходит внутри тела цикла.

## Параметрический цикл for

Общая форма записи

for (Инициализация; Условие; Модификация)  
{  
  БлокОпераций;  
}  
  
for — параметрический цикл (цикл с фиксированным числом повторений). Для организации такого цикла необходимо осуществить три операции:

* **Инициализация** - присваивание параметру цикла начального значения;
* **Условие** - проверка условия повторения цикла, чаще всего - сравнение величины параметра с некоторым граничным значением;
* **Модификация** - изменение значения параметра для следующего прохождения тела цикла.

Эти три операции записываются в скобках и разделяются точкой с запятой **;**;. Как правило, параметром цикла является целочисленная переменная.  
**Инициализация** параметра осуществляется только один раз — когда цикл for начинает выполняться.

**Итерация** — повторение какого-либо действия.

29.Условный оператор if может использоваться в форме ***полной*** или ***неполной*** развилки.

| **Неполная развилка** | **Полная развилка** | |
| --- | --- | --- |
| if (Условие) {   БлокОпераций1; } | if (Условие) {   БлокОпераций1; } else {   БлокОпераций2; } |

**Тернарные операции**

***Тернарная условная операция*** имеет 3 аргумента и возвращает свой второй или третий операнд в зависимости от значения логического выражения, заданного первым операндом. Синтаксис тернарной операции в языке Си

Условие ? Выражение1 : Выражение2;

Если выполняется **Условие**, то тернарная операция возвращает **Выражение1**, в противном случае - **Выражение2**.  
Тернарные операции, как и операции условия, могут быть вложенными. Для  разделения вложенных операций используются круглые скобки.

## 30. Оператор ветвления switch (оператор множественного выбора)

Оператор if позволяет осуществить выбор только между двумя вариантами. Для того, чтобы производить выбор одного из нескольких вариантов необходимо использовать вложенный оператор if. С этой же целью можно использовать оператор ветвления switch.  
  
Общая форма записи

switch (ЦелоеВыражение)  
{  
  case Константа1: БлокОпераций1;  
    break;  
  case Константа2: БлокОпераций2;  
    break;  
  . . .  
  case Константаn: БлокОперацийn;  
    break;  
  default: БлокОперацийПоУмолчанию;  
    break;  
}

Оператор ветвления switch выполняется следующим образом:

* вычисляется **ЦелоеВыражение** в скобках оператора switch;
* полученное значение сравнивается с метками (**Константами**) в опциях case, сравнение производится до тех пор, пока не будет найдена метка, соответствующая вычисленному значению целочисленного выражения;
* выполняется **БлокОпераций** соответствующей метки case;
* если соответствующая метка не найдена, то выполнится **БлокОперацийПоУмолчанию**, описанный в опции default.

Альтернатива default может отсутствовать, тогда не будет произведено никаких действий.  
Опция break; осуществляет выход из оператора switch и переход к следующему за ним оператору. При отсутствии опции break будут выполняться все операторы, начиная с помеченного данной меткой и кончая оператором в опции default.  
  
Константы в опциях case должны быть целого типа (могут быть символами).

## Оператор goto

## Имеющийся в C# **оператор goto** представляет собой оператор безусловного перехода. Когда в программе встречается оператор goto, ее выполнение переходит непосредственно к тому месту, на которое указывает этот оператор. Он уже давно "вышел из употребления" в программировании, поскольку способствует созданию "макаронного" кода. Хотя в некоторых случаях он оказывается удобным и дает определенные преимущества, если используется благоразумно. Главный недостаток оператора goto с точки зрения программирования заключается в том, что он вносит в программу беспорядок и делает ее практически неудобочитаемой. Но иногда применение оператора goto может, скорее, прояснить, чем запутать ход выполнения программы.

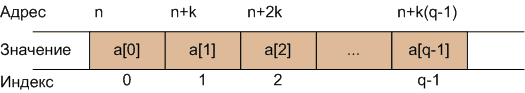
Для выполнения оператора goto требуется метка — действительный в C# идентификатор с двоеточием. Метка должна находиться в том же методе, где и оператор goto, а также в пределах той же самой области действия.

Оператор break   
С помощью оператора break можно специально организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку условия цикла. Когда в теле цикла встречается оператор break, цикл завершается, а выполнение программы возобновляется с оператора, следующего после этого цикла. Оператор break можно применять в любом цикле, предусмотренном в C#.

Оператор continue   
С помощью оператора continue можно организовать преждевременное завершение шага итерации цикла в обход обычной структуры управления циклом. Оператор continue осуществляет принудительный переход к следующему шагу цикла, пропуская любой код, оставшийся невыполненным. Таким образом, оператор continue служит своего рода дополнением оператора break.   
  
В циклах while и do-while оператор continue вызывает передачу управления непосредственно условному выражению, после чего продолжается процесс выполнения цикла. А в цикле for сначала вычисляется итерационное выражение, затем условное выражение, после чего цикл продолжается:

Оператор return   
Оператор return организует возврат из метода. Его можно также использовать для возврата значения. Имеются две формы оператора return: одна — для методов типа void, т.е. тех методов, которые не возвращают значения, а другая — для методов, возвращающих конкретные значения.   
31. ***Массив*** — это непрерывный участок памяти, содержащий последовательность объектов одинакового типа, обозначаемый одним именем.  
  
Массив характеризуется следующими основными понятиями:  
  
***Элемент массива (значение элемента массива)*** – значение, хранящееся в определенной ячейке памяти, расположенной в пределах массива, а также адрес этой ячейки памяти.  
Каждый элемент массива характеризуется тремя величинами:

* адресом элемента — адресом начальной ячейки памяти, в которой расположен этот элемент;
* индексом элемента (порядковым номером элемента в массиве);
* значением элемента.

***Адрес массива*** – адрес начального элемента массива.  
  
***Имя массива*** – идентификатор, используемый для обращения к элементам массива.  
  
***Размер массива*** – количество элементов массива  
  
***Размер элемента*** – количество байт, занимаемых одним элементом массива.  
  
Графически расположение массива в памяти компьютера можно представить в виде непрерывной ленты адресов.  


Представленный на рисунке массив содержит q элементов с индексами от 0 до q-1. Каждый элемент занимает в памяти компьютера k байт, причем расположение элементов в памяти последовательное.  
  
Адреса i-го элемента массива имеет значение

**n+k·i**

Адрес массива представляет собой адрес начального (нулевого) элемента массива. Для обращения к элементам массива используется порядковый номер (индекс) элемента, начальное значение которого равно 0. Так, если массив содержит q элементов, то индексы элементов массива меняются в пределах от 0 до q-1.

**Объявление и инициализация массивов**

Для объявления массива в языке Си используется следующий синтаксис:

**тип имя[размерность]={инициализация};**

***Инициализация*** представляет собой набор начальных значений элементов массива, указанных в фигурных скобках, и разделенных запятыми.

int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  // массив a из 10 целых чисел

Если количество инициализирующих значений, указанных в фигурных скобках, меньше, чем количество элементов массива, указанное в квадратных скобках, то все оставшиеся элементы в массиве (для которых не хватило инициализирующих значений) будут равны нулю. Это свойство удобно использовать для задания нулевых значений всем элементам массива.

int b[10] = {0}; // массив b из 10 элементов, инициализированных 0

Если массив проинициализирован при объявлении, то константные начальные значения его элементов указываются через запятую в фигурных скобках. В этом случае количество элементов в квадратных скобках может быть опущено.

int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

### При обращении к элементам массива индекс требуемого элемента указывается в квадратных скобках []. Операции с элементами массива.

Практически все операции с массивом следует проводить поэлементно в цикле. Для обработки элементов массива удобно использовать цикл ***for ...do***, а верхний индекс массивов определять как предварительно описанную константу. В этом случае все циклы по обработке массива будут заканчиваться значением этой константы. При изменении числа элементов массива, в программе достаточно изменить значение константы (т.к. все циклы зависят от константы).

Стандартные функции Low() и High() действуют для идентификаторов типа массива. Они возвращают нижние и верхние границы массива. Стандартная функция Length() возвращает количество элементов первого измерения массива (для матрицы возвращается число строк)

Доступ к каждому элементу массива осуществляется с помощью индекса, т.е. порядкового номера элемента массива. Когда мы хотим обратиться к элементу массива, надо указать имя массива и порядковый номер элемента: *a[1]*, *b[red]*, *c['z']*, *d['s']* . Если указано только имя массива - речь идет обо всем массиве (a, s и d и т.д.). Для массивов определена единственная операция – операция присваивания для однотипных массивов. Например, *s:=a* (такая операция означает, что все элементы массива *a* копируются в соответствующие элементы массива *s*). Все остальные операции определены для элементов согласно их типу. При обработке массива нужно последовательно обрабатывать все его элементы; при вводе массива необходимо последовательно вводить 1-й, 2-й и 3-й и т.д. элементы массива; аналогично и при выводе. Если статический массив создан, но значения назначены не во все элементы, неиспользованные элементы занимают память и содержат произвольные данные.

## 32. Многомерные массивы

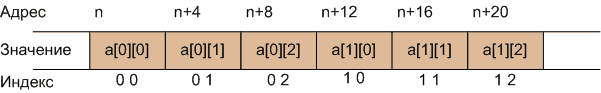
В языке Си могут быть также объявлены многомерные массивы. Отличие многомерного массива от одномерного состоит в том, что в одномерном массиве положение элемента определяется одним индексом, а в многомерном — несколькими. Примером многомерного массива является матрица.  
  
Общая форма объявления многомерного массива

тип имя[размерность1][размерность2]...[размерностьm];

Элементы многомерного массива располагаются в последовательных ячейках оперативной памяти по возрастанию адресов. В памяти компьютера элементы многомерного массива располагаются подряд, например массив, имеющий 2 строки и 3 столбца,

int a[2][3];

будет расположен в памяти следующим образом

  
  
Общее количество элементов в приведенном двумерном массиве определится как

**КоличествоСтрок \* КоличествоСтолбцов = 2 \* 3 = 6.**

Количество байт памяти, требуемых для размещения массива, определится как

**КоличествоЭлементов \* РазмерЭлемента = 6 \* 4 = 24 байта.**

**Инициализация многомерных массивов**

Значения элементов многомерного массива, как и в одномерном случае, могут быть заданы константными значениями при объявлении, заключенными в фигурные скобки {}. Однако в этом случае указание количества элементов в строках и столбцах должно быть обязательно указано в квадратных скобках [].

33. Указатели представляют собой объекты, значением которых служат адреса других объектов (переменных, констант, указателей) или функций. Указатели - это неотъемлемый компонент для управления памятью в языке Си.   
  
Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки \*. Например, определим указатель на объект типа int:   
int \*p;   
Пока указатель не ссылается ни на какой объект. Теперь присвоим ему адрес переменной:   
  
int x = 10; // определяем переменную   
int \*p; // определяем указатель   
p = &x; // указатель получает адрес переменной   
Указатель хранит адрес объекта в памяти компьютера. И для получения адреса к переменной применяется операция &. Эта операция применяется только к таким объектам, которые хранятся в памяти компьютера, то есть к переменным и элементам массива.   
  
Что важно, переменная x имеет тип int, и указатель, который указывает на ее адрес тоже имеет тип int. То есть должно быть соответствие по типу.   
  
Какой именно адрес имеет переменная x? Для вывода значения указателя можно использовать специальный спецификатор %p:   
  
В моем случае машинный адрес переменной x - 0060FEA8. Но в каждом отдельном случае адрес может быть иным. Фактически адрес представляет целочисленное значение, выраженное в шестнадцатеричном формате.   
То есть в памяти компьютера есть адрес 0x0060FEA8, по которому располагается переменная x. Так как переменная x представляет тип int, то на большинстве архитектур она будет занимать следующие 4 байта (на конкретных архитектурах размер памяти для типа int может отличаться). Таким образом, переменная типа int последовательно займет ячейки памяти с адресами 0x0060FEA8, 0x0060FEA9, 0x0060FEAA, 0x0060FEAB.

# Области применения указателей

Можно выделить следующие основные области применения указателей:

1. Использование указателей в качестве параметров функций для возврата результатов вычислений.
2. Работа с объектами, размещенными в динамической памяти.
3. Замена манипуляций со сложными объектами данных манипуляцией с адресами этих объектов.

## Нулевой указатель

Нулевой указатель – это такое значение указателя, которое не может быть адресом переменной или функции. Наиболее часто используемыми представлениями нулевого указателя как константы являются 0, 0LиNULL. Считается хорошим стилем программирования всем указателям, которые не работают с реальными объектами (переменными) или функциями, было присвоено значениеNULL

## Определение указателя

В определении указателя используется оператор разыменования “\*”. Упрощенный формат определения указателя имеет следующий вид:

*type*\**name*;

Здесь *type –*спецификация типа, определяющая тип объекта, адрес которого будет храниться в указателе;*name –*имя (идентификатор) указателя. Заметим, что иногда тип объект, на который установлен указатель, называют базовым типом. Как уже отмечалось, в записи определений используется синтаксис выражений. Продемонстрируем это положение на примере. Пусть имеется следующее определение указателя:

**int**\*ptr;

Здесь*type*– это***int***, аptr–*name*. Приведенное выше определение можно “прочитать”, используя синтаксис выражений, следующим образом. Применение оператора разыменования к имениptrпозволяет получить объект типа**int**. Отсюда следует тот факт, чтоptrявляется именем указателя

Во время определения указатель может быть инициализирован. В качестве инициализаторов указателя при его объявлении могут использоваться:

* Нулевой указатель.
* Указатель-переменная
* Указатель-выражение.

Приведем примеры инициализации указателей во время их определения.

/\* Примеры инициализации указателей при их определении \*/ int\* p = NULL; /\* Инициализация указателя p нулевым указателем\*/ int\* p2 = p; /\* Инициализация указателя p2 значением существующего указателя p\*/ int n = 5; int\* p3 = &n; /\* Инициализация указателя значением указателя-выражения &n \*/ /\* . . . . . . . . . . . . . . . . . . . . . . . . . . . \*/

34. Указатели поддерживают ряд операций: присваивание, получение адреса указателя, получение значения по указателю, некоторые арифметические операции и операции сравнения.   
Присваивание   
Указателю можно присвоить либо адрес объекта того же типа, либо значение другого указателя.   
Присвоение указателю адреса уже рассматривалось в прошлой теме. Для получения адреса объекта используется операция &:   
int a = 10;   
int \*pa = &a; // указатель pa хранит адрес переменной a   
При этом указатель и переменная должны иметь один и тот же тип, в данном случае это тип int.   
Присвоение указателю другого указателя:   
#include <iostream>   
using std::cout;   
using std::endl;   
  
int main()   
{   
int a = 10;   
int b = 2;   
  
int \*pa = &a;   
int \*pb = &b;   
  
cout << "Variable a: address=" << pa << "\t value=" << \*pa << endl;   
cout << "Variable b: address=" << pb << "\t value=" << \*pb << endl;   
  
pa = pb; // теперь указатель pa хранит адрес переменной b   
cout << "Variable b: address=" << pa << "\t value=" << \*pa << endl;   
  
return 0;   
}   
Когда указателю присваивается другой указатель, то фактически первый указатель начинает также указывать на тот же адрес, на который указывает второй указатель.   
  
Нулевые указатели   
Нулевой указатель (null pointer) - это указатель, который не указывает ни на какой объект. Если мы не хотим, чтобы указатель указывал на какой-то конкретный адрес, то можно присвоить ему условное нулевое значение. Для создания нулевого указателя можно применять различные способы:   
  
int \*p1 = nullptr;   
int \*p2 = NULL;   
int \*p3 = 0;   
Ссылки на указатели   
Так как ссылка не является объектом, то нельзя определить указатель на ссылку, однако можно определить ссылку на указатель. Через подобную ссылку можно изменять значение, на которое указывает указатель или изменять адрес самого указателя:   
  
#include <iostream>   
int main()   
{   
int a = 10;   
int b = 6;   
int \*p = 0; // указатель   
int \*&pRef = p; // ссылка на указатель   
pRef = &a; // через ссылку указателю p присваивается адрес переменной a   
std::cout << "p value=" << \*p << std::endl; // 10   
\*pRef = 70; // изменяем значение по адресу, на который указывает указатель   
std::cout << "a value=" << a << std::endl; // 70   
  
pRef = &b; // изменяем адрес, на который указывает указатель   
std::cout << "p value=" << \*p << std::endl; // 6   
  
return 0;   
}   
Разыменование указателя   
Операция разыменования указателя представляет выражение в виде \*имя\_указателя. Эта операция позволяет получить объект по адресу, который хранится в указателе.   
#include <iostream>   
using std::cout;   
using std::endl;   
int main()   
{   
int a = 10;   
int \*pa = &a;   
int \*pb = pa;   
\*pa = 25;   
cout << "Value on pointer pa: " << \*pa << endl; // 25   
cout << "Value on pointer pb: " << \*pb << endl; // 25   
cout << "Value of variable a: " << a << endl; // 25   
  
return 0;   
}   
Через выражение \*pa мы можем получить значение по адресу, который хранится в указателе pa, а через выражение типа \*pa = значение вложить по этому адресу новое значение.   
  
И так как в данном случае указатель pa указывает на переменную a, то при изменении значения по адресу, на который указывает указатель, также изменится и значение переменной a.   
  
Адрес указателя   
Указатель хранит адрес переменной, и по этому адресу мы можем получить значение этой переменной. Но кроме того, указатель, как и любая переменная, сам имеет адрес, по которому он располагается в памяти. Этот адрес можно получить также через операцию &:   
int a = 10;   
int \*pa = &a;   
std::cout << "address of pointer=" << &pa << std::endl; // адрес указателя   
std::cout << "address stored in pointer=" << pa << std::endl; // адрес, который хранится в указателе - адрес переменной a   
std::cout << "value on pointer=" << \*pa << std::endl; // значение по адресу в указателе - значение переменной a   
Операции сравнения   
К указателям могут применяться операции сравнения >, >=, <, <=,==, !=. Операции сравнения применяются только к указателям одного типа и к значениям NULL и nullptr. Для сравнения используются номера адресов:   
#include <iostream>   
using std::cout;   
using std::endl;   
  
int main()   
{   
int a = 10;   
int b = 20;   
int \*pa = &a;   
int \*pb = &b;   
  
if(pa > pb)   
cout << "pa (" << pa << ") is greater than pb ("<< pb << ")" << endl;   
else   
cout << "pa (" << pa << ") is less or equal pb ("<< pb << ")" << endl;   
  
return 0;   
}   
Консольный вывод в моем случае:   
  
pa (0x60fe94) is greater than pb (0x60fe90)   
Приведение типов   
Иногда требуется присвоить указателю одного типа значение указателя другого типа. В этом случае следует выполнить операцию приведения типов с помощью операции (тип\_указателя \*):   
  
#include <iostream>   
int main()   
{   
char c = 'N';   
char \*pc = &c;   
int \*pd = (int \*)pc;   
void \*pv = (void\*)pc;   
std::cout << "pv=" << pv << std::endl;   
std::cout << "pd=" << pd << std::endl;   
  
return 0;   
}   
Для преобразования указателя к другому типу в скобках перед указателем ставится тип, к которому надо преобразовать. Причем если мы не можем просто создать объект, например, переменную типа void, то для указателя это вполне будет работать. То есть можно создать указатель типа void.   
  
Кроме того, следует отметить, что указатель на тип char (char \*pc = &c) при выводе на консоль система интерпретирует как строку:   
  
  
std::cout << "pc=" << pc << std::endl;   
Поэтому если мы все-таки хотим вывести на консоль адрес, который хранится в указателе типа char, то это указатель надо преобразовать к другому типу, например, к void\* или к int\*.

35. Ссылка (на неконстантное значение) объявляется с использованием амперсанда (&) между типом данных и именем ссылки:   
int value = 7; // обычная переменная   
int &ref = value; // ссылка на переменную value   
В этом контексте амперсанд не означает «оператор адреса», он означает «ссылка на».

Ссылки чаще всего используются в качестве [**параметров**](https://ravesli.com/urok-13-parametry-i-argumenty-funktsij/) в функциях. В этом контексте ссылка-параметр работает как псевдоним аргумента, и сам аргумент не копируется при передаче в параметр. Это, в свою очередь, улучшает производительность, если аргумент слишком большой или затратный для копирования.

### 36. Объявление одномерных динамических массивов

Под объявлением одномерного *динамического массива* понимают объявление указателя на переменную заданного типа для того, чтобы данную переменную можно было использовать как *динамический массив*.

*Синтаксис*:

Тип \* ИмяМассива;

ИмяМассива – *идентификатор* массива, то есть имя указателя для выделяемого *блока памяти*.

Тип – тип элементов объявляемого *динамического массива*. Элементами *динамического массива* не могут быть функции и элементы типа void.

Например:

int \*a;

double \*d;

В данных примерах a и d являются указателями на начало выделяемого участка памяти. Указатели принимают *значение* адреса выделяемой области памяти для значений типа int и типа double соответственно.

Таким образом, при динамическом распределении памяти для динамических массивов следует описать соответствующий *указатель*, которому будет присвоено *значение* адреса начала области выделенной памяти.

### Выделение памяти под одномерный динамический массив

Для того чтобы выделить *память* под одномерный *динамический массив* в языке С++ существует 2 способа.

1) *при помощи операции* new, которая выделяет для размещения массива участок динамической памяти соответствующего размера и не позволяет инициализировать элементы массива.

*Синтаксис*:

ИмяМассива = new Тип [ВыражениеТипаКонстанты];

ИмяМассива – *идентификатор* массива, то есть имя указателя для выделяемого *блока памяти*.

Тип – тип указателя на *массив*.

ВыражениеТипаКонстанты – задает количество элементов (*размерность) массива*. *Выражение* константного типа вычисляется на этапе компиляции.

Например:

int \*mas;

mas = new int [100]; /\*выделение динамической памяти

размером 100\*sizeof(int) байтов\*/

double \*m = new double [n]; /\*выделение динамической

памяти размером n\*sizeof(double) байтов\*/

long (\*lm)[4];

lm = new long [2] [4]; /\*выделение динамической памяти

размером 2\*4\*sizeof(long) байтов\*/

При выделении динамической памяти размеры массива должны быть полностью определены.

2) *при помощи библиотечной функции* malloc (calloc), которая служит для выделения динамической памяти.

*Синтаксис*:

ИмяМассива = (Тип \*) malloc(N\*sizeof(Тип));

или

ИмяМассива = (Тип \*) calloc(N, sizeof(Тип));

ИмяМассива – *идентификатор* массива, то есть имя указателя для выделяемого *блока памяти*.

Тип – тип указателя на *массив*.

N – количество элементов массива.

Например:

float \*a;

a=(float \*)malloc(10\*sizeof(float));

// или

a=(float \*)calloc(10,sizeof(float));

/\*выделение динамической памяти размером 10\*sizeof(float) байтов\*/

Так как *функция* malloc (calloc) возвращает *нетипизированный указатель* void \*, то необходимо выполнять преобразование полученного *нетипизированного указателя* в *указатель* объявленного типа.

### Освобождение памяти, выделенной под одномерный динамический массив

*Освобождение памяти*, выделенной под одномерный *динамический массив*, также осуществляется 2 способами.

1) *при помощи операции* delete, которая освобождает участок памяти ранее выделенной операцией new.

*Синтаксис*:

delete [] ИмяМассива;

ИмяМассива – *идентификатор* массива, то есть имя указателя для выделяемого *блока памяти*.

Например:

delete [] mas; /\*освобождает память, выделенную под

массив, если mas адресует его начало\*/

delete [] m;

delete [] lm;

Квадратные скобки [] сообщают оператору, что требуется освободить *память*, занятую всеми элементами, а не только первым.

2) *при помощи библиотечной функции* free, которая служит для освобождения динамической памяти.

*Синтаксис*:

free (ИмяМассива);

ИмяМассива – *идентификатор* массива, то есть имя указателя для выделяемого *блока памяти*.

Например:

free (a); //освобождение динамической памяти

### 37. Объявление двумерных динамических массивов

Под объявлением двумерного *динамического массива* понимают объявление двойного указателя, то есть объявление указателя на *указатель*.

*Синтаксис*:

Тип \*\* ИмяМассива;

ИмяМассива – *идентификатор* массива, то есть имя двойного указателя для выделяемого *блока памяти*.

Тип – тип элементов объявляемого *динамического массива*. Элементами *динамического массива* не могут быть функции и элементы типа void.

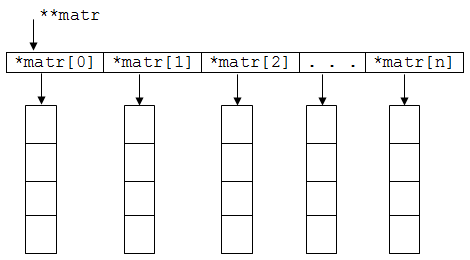
Например:

int \*\*a;

float \*\*m;

### Выделение памяти под двумерный динамический массив

При формировании двумерного *динамического массива* сначала выделяется *память* для массива указателей на одномерные массивы, а затем в цикле с параметром выделяется *память* под одномерные массивы. На [рис. 26.1](https://www.intuit.ru/studies/courses/648/504/lecture/11452?page=1#image.26.1) представлена схема динамической области памяти, выделенной под *двумерный массив*.



**Рис. 26.1.**Выделение памяти под двумерный массив

При работе с динамической памятью в языке С++ существует 2 способа выделения памяти под двумерный *динамический массив*.

1) *при помощи операции* new, которая позволяет выделить в динамической памяти участок для размещения массива соответствующего типа, но не позволяет его инициализировать.

*Синтаксис* выделения памяти под *массив* указателей:

ИмяМассива = new Тип \* [ВыражениеТипаКонстанты];

*Синтаксис* выделения памяти для массива значений:

ИмяМассива[ЗначениеИндекса] = new Тип [ВыражениеТипа

Константы];

ИмяМассива – *идентификатор* массива, то есть имя двойного указателя для выделяемого *блока памяти*.

Тип – тип указателя на *массив*.

ВыражениеТипаКонстанты – задает количество элементов (*размерность) массива*. *Выражение* константного типа вычисляется на этапе компиляции.

Например:

int n, m;//n и m – количество строк и столбцов матрицы

float \*\*matr; //указатель для массива указателей

matr = new float \* [n]; //выделение динамической памяти

под массив указателей

for (int i=0; i<n; i++)

matr[i] = new float [m]; //выделение динамической памяти

для массива значений

При выделении динамической памяти размеры массивов должны быть полностью определены.

2) *при помощи библиотечной функции* malloc (calloc), которая предназначена для выделения динамической памяти.

*Синтаксис* выделения памяти под *массив* указателей:

ИмяМассива = (Тип \*\*) malloc(N\*sizeof(Тип \*));

или

ИмяМассива = (Тип \*\*) calloc(N, sizeof(Тип \*));

*Синтаксис* выделения памяти для массива значений:

ИмяМассива[ЗначениеИндекса]=(Тип\*)malloc(M\*sizeof(Тип));

или

ИмяМассива[ЗначениеИндекса]=(Тип\*)calloc(M,sizeof(Тип));

ИмяМассива – *идентификатор* массива, то есть имя двойного указателя для выделяемого *блока памяти*.

Тип – тип указателя на *массив*.

N – количество строк массива;

M – количество столбцов массива.

Например:

int n, m;//n и m – количество строк и столбцов матрицы

float \*\*matr; //указатель для массива указателей

matr = (float \*\*) malloc(n\*sizeof(float \*));

//выделение динамической памяти под массив указателей

for (int i=0; i<n; i++)

matr[i] = (float \*) malloc(m\*sizeof(float));

//выделение динамической памяти для массива значений

Так как *функция* malloc (calloc) возвращает *нетипизированный указатель* void \*, то необходимо выполнять его преобразование в *указатель* объявленного типа.

### Освобождение памяти, выделенной под двумерный динамический массив

Удаление из динамической памяти двумерного массива осуществляется в порядке, обратном его созданию, то есть сначала освобождается *память*, выделенная под одномерные массивы с данными, а затем *память*, выделенная под *одномерные массив* указателей.

*Освобождение памяти*, выделенной под двумерный *динамический массив*, также осуществляется 2 способами.

1) *при помощи операции* delete, которая освобождает участок памяти ранее выделенной операцией new.

*Синтаксис* *освобождения памяти*, выделенной для массива значений:

delete ИмяМассива [ЗначениеИндекса];

*Синтаксис* *освобождения памяти*, выделенной под *массив* указателей:

delete [] ИмяМассива;

ИмяМассива – *идентификатор* массива, то есть имя двойного указателя для выделяемого *блока памяти*.

Например:

for (int i=0; i<n; i++)

delete matr [i];

//освобождает память, выделенную для массива значений

delete [] matr;

//освобождает память, выделенную под массив указателей

Квадратные скобки [] означают, что освобождается *память*, занятая всеми элементами массива, а не только первым.

2) *при помощи библиотечной функции* free, которая предназначена для освобождения динамической памяти.

*Синтаксис* *освобождения памяти*, выделенной для массива значений:

free (ИмяМассива[ЗначениеИндекса]);

*Синтаксис* *освобождения памяти*, выделенной под *массив* указателей:

free (ИмяМассива);

ИмяМассива – *идентификатор* массива, то есть имя двойного указателя для выделяемого *блока памяти*.

Например:

for (int i=0; i<n; i++)

free (matr[i]);

//освобождает память, выделенную для массива значений

free (matr);

//освобождает память, выделенную под массив указателей

38. При создании массива с фиксированными размерами под него выделяется определенная память. Например, пусть у нас будет массив с пятью элементами:

|  |  |
| --- | --- |
| 1 | double numbers[5] = {1.0, 2.0, 3.0, 4.0, 5.0}; |

Для такого массива выделяется память 5 \* 8 (размер типа double) = 40 байт. Таким образом, мы точно знаем, сколько в массиве элементов и сколько он занимает памяти. Однако это не всегда удобно. Иногда бывает необходимо, чтобы количество элементов и соответственно размер выделяемой памяти для массива определялись динамически в зависимости от некоторых условий. Например, пользователь сам может вводить размер массива. И в этом случае для создания массива мы можем использовать динамическое выделение памяти.

Для управления динамическим выделением памяти используется ряд функций, которые определены в заголовочном файле **stdlib.h**:

* **malloc()**. Имеет прототип

|  |  |
| --- | --- |
| 1 | void \*malloc(unsigned s); |

* Выделяет память длиной в s байт и возвращает указатель на начало выделенной памяти. В случае неудачного выполнения возвращает **NULL**
* **calloc()**. Имеет прототип

|  |  |
| --- | --- |
| 1 | void \*calloc(unsigned n, unsigned m); |

* Выделяет память для n элементов по m байт каждый и возвращает указатель на начало выделенной памяти. В случае неудачного выполнения возвращает **NULL**
* **realloc()**. Имеет прототип

|  |  |
| --- | --- |
| 1 | void \*realloc(void \*bl, unsigned ns); |

* Изменяет размер ранее выделенного блока памяти, на начало которого указывает указатель bl, до размера в ns байт. Если указатель bl имеет значение **NULL**, то есть память не выделялась, то действие функции аналогично действию **malloc**
* **free()**. Имеет прототип

|  |  |
| --- | --- |
| 1 | void \*free(void \*bl); |

* Освобождает ранее выделенный блок памяти, на начало которого указывает указатель bl.
* Если мы не используем эту функцию, то динамическая память все равно освободится автоматически при завершении работы программы. Однако все же хорошей практикой является вызов функции **free()**, который позволяет как можно раньше освободить память.

Рассмотрим применение функций на простой задаче. Длина массива неизвестна и вводится во время выполнения программы пользователем, и также значения всех элементов вводятся пользователем:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | #include <stdio.h>  #include <stdlib.h>    int main(void)  {      int \*block; // указатель для блока памяти      int n;      // число элементов массива      // ввод числа элементов      printf("Size of array=");      scanf("%d", &n);        // выделяем память для массива      // функция malloc возвращает указатель типа void\*      // который автоматически преобразуется в тип int\*      block = malloc(n \* sizeof(int));        // вводим числа в массив      for(int i=0;i<n; i++)      {          printf("block[%d]=", i);          scanf("%d", &block[i]);      }      printf("\n");        // вывод введенных чисел на консоль      for(int i=0;i<n; i++)      {          printf("%d \t", block[i]);      }        // освобождаем память      free(block);      return 0;  } |

Консольный вывод программы:

Size of array=5

block[0]=23

block[1]=-4

block[2]=0

block[3]=17

block[4]=81

23 -4 0 17 81

Здесь для управления памятью для массива определен указатель block типа **int**. Количество элементов массива заранее неизвестно, оно представлено переменной n.

Вначале пользователь вводит количество элементов, которое попадает в переменную n. После этого необходимо выделить память для данного количества элементов. Для выделения памяти здесь мы могли бы воспользоваться любой из трех вышеописанных функций: malloc, calloc, realloc. Но конкретно в данной ситуации воспользуемся функцией **malloc**:

|  |  |
| --- | --- |
| 1 | block = malloc(n \* sizeof(int)); |

Прежде всего надо отметить, что все три выше упомянутые функции для универсальности возвращаемого значения в качестве результата возвращают указатель типа **void \***. Но в нашем случае создается массив типа int, для управления которым используется указатель типа **int \***, поэтому выполняется неявное приведение результата функции malloc к типу **int \***.

В саму функцию malloc передается количество байтов для выделяемого блока. Это количество подсчитать довольно просто: достаточно умножить количество элементов на размер одного элемента n \* sizeof(int).

После выполнения всех действий память освобождается с помощью функции **free()**:

|  |  |
| --- | --- |
| 1 | free(block); |

Важно, что после выполнения этой функции мы уже не сможем использовать массив, например, вывести его значения на консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5 | free(block);  for(int i=0;i<n; i++)  {      printf("%d \t", block[i]);  } |

И если мы попытаемся это сделать, то получим неопределенные значения.

Вместо функции malloc аналогичным образом мы могли бы использовать функцию **calloc()**, которая принимает количество элементов и размер одного элемента:

|  |  |
| --- | --- |
| 1 | block = calloc(n, sizeof(int)); |

Либо также можно было бы использовать функцию **realloc()**:

|  |  |
| --- | --- |
| 1  2 | int \*block = NULL;  block = realloc (block, n \* sizeof(int)); |

При использовании realloc желательно (в некоторых средах, например, в Visual Studio, обязательно) инициализировать указатель хотя бы значением NULL.

Но в целом все три вызова в данном случае имели бы аналогичное действие:

|  |  |
| --- | --- |
| 1  2  3 | block = malloc(n \* sizeof(int));  block = calloc(n, sizeof(int));  block = realloc (block, n \* sizeof(int)); |

Теперь рассмотрим более сложную задачу - динамическое выделение памяти для двухмерного массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | #include <stdio.h>  #include <stdlib.h>    int main(void)  {      int \*\*table;    // указатель для блока памяти для массива указателей      int \*rows;      // указатель для блока памяти для хранения информации по строкам        int rowscount;  // количество строк      int d;      // вводимое число        // ввод количества строк      printf("Rows count=");      scanf("%d", &rowscount);        // выделяем память для двухмерного массива      table = calloc(rowscount, sizeof(int\*));      rows = malloc(sizeof(int)\*rowscount);      // цикл по строкам      for (int i = 0; i<rowscount; i++)      {          printf("\nColumns count for row %d=", i);          scanf("%d", &rows[i]);          table[i] = calloc(rows[i], sizeof(int));            for (int j = 0; j<rows[i]; j++)          {              printf("table[%d][%d]=", i, j);              scanf("%d", &d);              table[i][j] = d;          }      }      printf("\n");        // вывод введенных чисел на консоль      for (int i = 0; i<rowscount; i++)      {          printf("\n");            for (int j = 0; j<rows[i]; j++)          {              printf("%d \t", table[i][j]);          }          // освобождение памяти для одной строки          free(table[i]);      }        // освобождение памяти      free(table);      free(rows);        return 0;  } |

Переменная table представляет указатель на массив указателей типа **int\***. Каждый указатель table[i] в этом массиве представляет указатель на подмассив элементов типа **int**, то есть отдельные строки таблицы. А переменная table фактически представляет указатель на массив указателей на строки таблицы.

Для хранения количества элементов в каждом подмассиве определяется указатель rows типа **int**. Фактически он хранит количество столбцов для каждой строки таблицы.

Сначала вводится количество строк в переменную rowscount. Количество строк - это количество указателей в массиве, на который указывает указатель table. И кроме того, количество строк - это количество элементов в динамическом массиве, на который указывает указатель rows. Поэтому вначале необходимо для всех этих массивов выделить память:

|  |  |
| --- | --- |
| 1  2 | table = calloc(rowscount, sizeof(int\*));  rows = malloc(sizeof(int)\*rowscount); |

Далее в цикле осуществляется ввод количества столбцов для каждый строки. Введенное значение попадает в массив rows. И в соответствии с введенным значением для каждой строки выделяется необходимый размер памяти:

|  |  |
| --- | --- |
| 1  2 | scanf("%d", &rows[i]);  table[i] = calloc(rows[i], sizeof(int)); |

Затем производится ввод элементов для каждой строки.

В конце работы программы при выводе происходит освобождение памяти. В программе память выделяется для строк таблицы, поэтому эту память надо освободить:

|  |  |
| --- | --- |
| 1 | free(table[i]); |

И кроме того, освобождается память, выделенная для указателей table и rows:

|  |  |
| --- | --- |
| 1  2 | free(table);  free(rows); |

Консольный вывод программы:

Rows count=2

Columns count for 1=3

table[0][0]=1

table[0][1]=2

table[0][2]=3

Columns count for 2=2

table[1][0]=4

table[2][1]=5

1 2 3

4 5