1

1. Истортя создания ЭВМ .

Первые гражданские ЭВМ Z1 и Z2 были созданы в конце 1930-х годов в Германии.

1941 год — Конрад Цузе создал вычислительную машину Z3, которая имела все свойства современного компьютера.

1942 год — в Университете штата Айова Джон Атанасов и его аспирант Клиффорд Берри создали (а точнее — разработали и начали монтировать) первую в США электронную цифровую вычислительную машину. Хотя эта машина так и не была завершена в связи с уходом Атанасова на войну, она, как пишут историки, оказала большое влияние на Джона Мокли, который, спустя четыре года создал первую в мире ЭВМ ЭНИАК.

В начале 1943 года успешные испытания прошла первая американская вычислительная машина Марк I, предназначенная для выполнения сложных баллистических расчётов для ВМС США.

В конце 1943 года заработала английская вычислительная машина специального назначения «Колосс». Машина работала над расшифровкой секретных кодов нацистской Германии.

В 1944 году Конрад Цузе разработал ещё более быструю вычислительную машину — Z4.

1946 год стал годом создания первой американской гражданской универсальной электронной цифровой вычислительной машины ЭНИАК.

В 1950 году в Киеве под руководством академика С. А. Лебедева был создан первый советский сверхвычислитель МЭСМ, на следующий год в Москве под руководством И. С. Брука была создана ЭВМ М-1.

С 1962 года ЭВМ применяются на космических кораблях Союз и Л-1 (облёт Луны).

1967 стал годом, посвящённым формальным методам проектирования электронных вычислительных машин под руководством Глушкова.

30 октября 1967 года в СССР произведена первая в мире полностью автоматическая стыковка двух космических аппаратов (беспилотных кораблей «Союз» под названиями «Космос-186» и «Космос-188»[2]

В 1969 году ЭВМ ракеты Н-1 обрабатывал данные с более чем 13 тысячи датчиков ракеты.

2

Поколения компьютеров.

I поколение компьютеров

Появились в 1946 году. К особенностям этих компьютеров относится применение вакуумно-ламповой технологии. Для ввода-вывода данных использовались перфоленты и перфокарты , магнитные ленты и печатающие устройства. В компьютерах первого поколения была реализована концепция хранимой программы. Компьютеры данного поколения сумели зарекомендовать себя в прогнозировании погоды , энергетических задач , Задач военного характера и других сложнейших операциях.

II поколение компьютеров

Появились в 1955 году. В них вместо ламп использовались транзисторы. Они стали более надежными, быстродействие их повысилось, потребление энергии уменьшилось. С появлением памяти на магнитных сердечниках цикл ее работы уменьшился до десятков микросекунд. Главный принцип структуры - централизация. Для компьютеров этого поколения характерно использование первых языков программирования высокого уровня, которые получили свое развитие в компьютерах следующего поколения.

III поколение компьютеров

Появились в 1964 году. Они проектировались на основе интегральных схем малой степени интеграции. Появились операционные системы , которые стали брать на себя задачи управления памятью , устройствами ввода-вывода и другими ресурсами ; стало возможным мультипрограммирование. В начале 60-х гг. группой разработчиков фирмы IBM был введен термин "архитектура компьютера". К концу 60-х гг. появились мини-компьютеры. Экономичность мини-компьютеров быстро расширила сферу их применения : управление, передача данных , автоматизация научных экспериментов и т. д. В рамках рассматриваемого поколения в 1971 году появился первый микропроцессор, как неожиданный результат работы фирмы Intel над схемами калькуляторов.

IV поколение компьютеров

Появились в 1975 г. с изобретением больших и сверхбольших интегральных схем. В компьютерах этого поколения стали использоваться быстродействующие системы памяти на интегральных схемах емкостью несколько мегабайт. Появление в середине 70-х первых персональных компьютеров предоставило индивидуальному пользователю такие же вычислительные ресурсы, какими в 60-е годы обладали большие компьютеры. К концу 80-х четк

о определилось существование двух классов компьютеров, определяющих развитие компьютерного мира:

суперкомпьютеров , имеющих многопроцессорную архитектуру и использующих принципы параллелизма, и персональных компьютеров .

V поколение компьютеров

Появились в 1990 г. Главный упор при создании компьютеров сделан на их "интеллектуальность". Внимание акцентируется на архитектуре, ориентированной на обработку знаний. Обработка знаний - это одна из областей практического применения искусственного интеллекта, предполагающая использование и обработку компьютером знаний, которыми владеет человек для решения проблем и принятия решений.

3

Архитектура Фон – Неймана. Принципы фон Неймана.

Архитектура фон Неймана (модель фон Неймана, Принстонская архитектура) — широко известный принцип совместного хранения команд и данных в памяти компьютера. Вычислительные машины такого рода часто обозначают термином «машина фон Неймана», однако соответствие этих понятий не всегда однозначно. В общем случае, когда говорят об архитектуре фон Неймана, подразумевают принцип хранения данных и инструкций в одной памяти.

Принципы фон Неймана 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.

4

4.Языки программирования и их классификация.

. Классификация языков программирования.

Существуют различные классификации языков программирования.

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. 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, Алгол, Кобол, Паскаль, Бейсик, Си, Пролог 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

5. Размещение данных и программ в памяти ПЭВМ.

ных позиций, т.е. в каждую позицию могут быть записаны либо 0, либо 1. Объем информации, который помещается в одну двоичную позицию, называ ется бит. Объем информации равный 8 бит называется байтом.

Таким образом, в одной ячейке из 8 двоичных разрядов помещается объем информации в один байт. Поэтому объем памяти принято оценивать количеством байт (1024 Байт = 1 Килобайт, 1024 Килобайт = 1048576 Байт = 1 Мегабайт, 1024 Мегабайт = 1073741824 Байт = 1 Гигабайт,).

Для помещения данных в такие ячейки производится их запись с помо щью нулей и единиц (кодирование). При кодировании каждый символ, до пускаемый на клавиатуре, заменяется последовательностью из 8 двоичных разрядов в соответствии со стандартной кодовой таблицей. (т.е. один символ размещается в одном байте). Например, в соответствии с таблицей кодов ASCII D=(01000100); F=(00100110); 4=(00110100).

При кодировании чисел они предварительно преобразуются в двоичное представление. Например

2=1-21+0-2°=102; 5=1-22+0-21+1-2°=1012; 256=1 •28=1000000002.

Заметим, что с увеличением числа количество разрядов для его пред ставления 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 пер вого байта. Один бит такой ячейки выделяется под знак числа. Числа, разме щенные таким естественным образом, называются числами с фиксированной запятой, или просто целыми.

Для хранения дробных чисел, или слишком больших, их предваритель-

+2

но приводят к нормализованному виду. Например, -35,6 = -0.356-10 , где — 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, называются числами с плавающей за пятой или просто действительными.

Программа - это последовательность команд, 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 \*), кото рую 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. Но мер первого байта команды называется ее адресом. Последовательность

из этих команд называется программой в машинных кодах.

Программа

Команды

КОП

А1

А2

А3

КОП

А3

А4

КОП

А5

Составление программ в современных компьютерах автоматизировано. Программист пишет программу на специальном языке высокого уровня, т.е. наиболее удобном для записи алгоритма решения определенного класса за дач. С помощью символов клавиатуры вводит ее текст в память компьютера. Далее происходит перевод алгоритма в команды машины (трансляция) под ключение необходимых стандартных программ (компоновка) и лишь после этого выполнение. Заметим, что полученная программа в кодах машины мо жет быть записана на диск и для ее многоразового выполнения уже не требу ется этапов трансляции и компоновки.

6

6.Программные модули. Понятие транслятора, компилятора и интерпретатора.

Трансляторы предназначены для преобразования программ, написанных на языках программирования, в программы на машинном языке.

В качестве входной информации трансляторы применяют исходные модули и формируют в результате своей работы объектные модули, являющиеся входной информацией для редактора связей.

Объектный модуль содержит текст программы на машинном языке и дополнительную информацию, обеспечивающую как настройку модуля по месту его загрузки в оперативную память, так и объединение этого модуля с другими независимо оттранслированными модулями в единую программу.

Объектный модуль, как правило, имеет расширение obj.

Трансляторы делятся на два класса: компиляторы и интерпретаторы.

Компилятор переводит весь исхо

дный модуль на машинный язык.

При компиляции одна и та же программа имеет несколько представлений - в виде текста и в виде выполняемого файла.

Интерпретатор последовательно переводит на машинный язык каждый оператор исходного модуля и сразу же выполняет его.

Основной недостаток интерпретатора - низкая скорость работы интерпретируемых программ (во время выполнения программы необходим перевод каждого оператора на машинный язык). Преимущество интерпретатора перед компилятором состоит в том, что программа пользователя имеет одно представление - в виде текста.

Соответственно говорят о компилируемых и интерпретируемых языках программирования.

Языки программирования Pascal, Object Pascal и С-подобные языки (С, С++, С#) являются компилируемыми.

Язык Java - ; пример интерпретируемого языка программирования

7

7. Этапы решения задач на ЭВМ.

Решение задач на компьютере включает в себя следующие основные этапы, часть из которых осуществляется без участия компьютера.

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

• составление документации к решенной задаче, к математической модели, к алгоритму, к программе, к набору тестов, к использованию. 2. Категории специалистов, занятых разработкой и эксплуатацией программного обеспечения.

8

Алгоритм. Способы записи алгоритмов.

алгори́тм — конечная совокупность точно заданных правил решения некоторого класса задач или набор инструкций, описывающих порядок действий исполнителя для решения определённой задачи. В старой трактовке вместо слова «порядок» использовалось слово «последовательность», но по мере развития параллельности в работе компьютеров слово «последовательность» стали заменять более общим словом «порядок». Независимые инструкции могут выполняться в произвольном порядке, параллельно, если это позволяют используемые исполнители

Различают три способа записи алгоритмов:

Вербальный – это случай, когда алгоритм описывается обычным языком (понятным человеку).

Графический – случай, когда алгоритм описывается с помощью графических обозначений. Это есть наиболее распространенный способ записи алгоритма, который обозначается в виде блок-схем.

Символьный – описание алгоритма с помощью набора .

9

Алгоритм. Свойства алгоритма. Принципы построения алгоритмов при

решении сложных задач. Пример алгоритма.

Пример алгоритма : Алгоритм "Как открыть дверь?"

**1.**Достать ключ из кармана.**2.**Вставить ключ в замок**3.**Повернуть ключ .**4.** Вынуть ключ.

алгори́тм — конечная совокупность точно заданных правил решения некоторого класса задач или набор инструкций, описывающих порядок действий исполнителя для решения определённой задачи. В старой трактовке вместо слова «порядок» использовалось слово «последовательность», но по мере развития параллельности в работе компьютеров слово «последовательность» стали заменять более общим словом «порядок». Независимые инструкции могут выполняться в произвольном порядке, параллельно, если это позволяют используемые исполнители

Свойства алгоритма:

-дискретность-последовательность решения (процесс) задач должен быть разбит на последовательность отдельных шагов.

-понятность-алгоритм обязательно должен быть понятен исполнителю. В связи с этим алгоритм нужно разрабатывать с ориентацией на опр-ого исполнителя, т.е. в алгоритм можно включать команды из систем команд данного исполнителя.

-детерминированность — будучи понятным, алгоритм не должен содержать команды, смысл кот-ых может восприниматься неоднозначно. Нарушение составителями алгоритмов этих требований приводит к тому, что одна и та же программа после выполнения разными исполнителями дает не одинаковые результаты.

-результативность –состоит в том, что при точном исполнении всех команд алгоритма, процесс решения задач должен прекратиться за конечное число шагов и при этом должен быть получен опред-ый при постановке задач результат.

-массовость- пригодность алгоритма для решения задач некоторого класса.

Способы записи алгоритма:

-словесный – способ на естественном языке.

-графический-описания алгоритма с помощью схем.

Процесс выполнения операций или групп операций

ввод исходных данных, вывод результата

Решение-выбор направления выполнения

Модификация-выполнение операций , меняющих команды или группы команд, изменяющих программ.

Соединители линий на одной странице.

Межстраничные соединители.

-язык программирования –удобен для ввода в комп-р.

-псевдокод-это язык, к-ый использует структуру и синтексис достаточно формализованного языка и одновременно допускает конструкции естеств. Языка.

Виды алгоритмов и основные принципы составления алгоритмов.

-Линейный – алгоритм, в кот-ом команды выполняются последовательно друг за другом в порядке их естественного следования независимо от каких-либо условий. S1, s2 , S3…Sn

-ветвящийся ( разветвящийся) — это процесс, в кот-ом его реализация происходит по одному из нескольких заранее предусмотренных направлений, в зависимости от исходных данных или промежуточных результатов.

Полная условная конструкция (полное ветвление)

Неполное условная конструкция

Выбор из нескольких

-циклический – алгоритм, в кот-ом последовательность может выполняться более 1 раза.

Цикл с параметром

Цикл с предусловием. Может не выполниться ни разу. В теле цикла обязательно нах-ся оператор, к-ый изменяет значение переменной, входящей в блок Q.

Цикл с постусловием. Выполняется хоть один раз.

Основные принципы алгоритмизации:

1. Выявить исходные данные, результаты и назначить им имена.

2. Метод решения задач.

3. Разбить метод решения задач на этапы.

4. При граф-ом представлении алгоритма каждый этап в виде соответствующего блока –схемы алгоритма и указать линиями связи порядок их выполнения.

5. В полученной схеме при любом варианте вычислений.

— предусмотреть выдачу результатов или сообщений об их отсутствии.

-обеспечить возможности после выполнение любой операции так или иначе перейти к блоку конец.

10

Правила применения символов и выполнения блок-схем. Описание символов

4.1. Правила применения символов

4.1.1. Символ предназначен для графической идентификации функции, которую он отображает, независимо от текста внутри этого символа.

4.1.2. Символы в схеме должны быть расположены равномерно. Следует придерживаться разумной длины соединений и минимального числа длинных линий.

4.1.3. Большинство символов задумано так, чтобы дать возможность включения текста внутри символа. Формы символов, установленные настоящим стандартом, должны служить руководством для фактически используемых символов. Не должны изменяться углы и другие параметры, влияющие на соответствующую форму символов. Символы должны быть, по возможности, одного размера.

Символы могут быть вычерчены в любой ориентации, но, по возможности, предпочтительной является горизонтальная ориентация. Зеркальное изображение формы символа обозначает одну и ту же функцию, но не является предпочтительным.

4.1.4. Минимальное количество текста, необходимого для понимания функции данного символа, следует помещать внутри данного символа. Текст для чтения должен записываться слева направо и сверху вниз независимо от направления потока.

Правила построения блок–схемы алгоритма

+

выявить исходные данные, результаты и назначить им имена;

выбрать порядок решения задачи;

разбить решение на этапы;

изобразить каждый этап своим блоком.

предусмотреть вывод результатов;

обеспечить выход (переход к блоку «конец».

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

Синтаксис языка СИ. Алфавит. Лексемы.

Алфавит языка Си включает:

- прописные и строчные буквы латинского алфавита, а также знак подчеркивания (код ASCII 95);

- арабские цифры от 0 до 9;

- специальные символы:

++(плюс) –(минус) \*(звездочка) /(дробная черта) =(равно) >(больше) <(меньше) ;(точка с запятой) &(амперсант) [ ](квадратные скобки) { }(фигурные скобки) ()(круглые скобки) \_(знак подчеркивания) .(точка) ,(запятая) :(двоето­чие) #("решетка") %(процент) ~(поразрядное отрицание) ?(знак вопроса) !(восклица­­тельный знак) \(обратный слеш).

- пробельные (разделительные) символы: пробел, символы табуляции, перевода строки, возврата каретки.

Лексемы

Из символов алфавита формируются лексемы языка – минимальные значимые единицы текста в программе:

- идентификаторы;

- ключевые (зарезервированные) слова;

- знаки операций;

- константы;

- разделители (скобки, точка, запятая, пробельные символы).

+Границы лексем определяются другими лексемами, такими, как разделители или знаки операций, а также комментариями.

13

13. Идентификаторы. Ключевые слова.

Идентификатор — это имя программного объекта. В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, sysop, SySoP и SYSOP — три различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы внутри имен не допускаются. Длина идентификатора по стандарту не ограничена, но некоторые компиляторы и компоновщики налагают на нее ограничения. Идентификатор создается на этапе объявления переменной, функции, типа и т. п., после этого его можно использовать в последующих операторах программы. Идентификатор не должен совпадать с ключевыми словами и именами используемых стандартных объектов языка.

Ключевые слова — это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Например: for, if, do, int, double, while, auto.

14

14. Литералы (константы). Комментарии. Директивы.

Константы — это данные, не изменяющие своего значения в ходе выполнения программы. В языке СИ можно выделить четыре вида констант: целые константы, константы с плавающей запятой, символьные константы и строковые литералы.

Комментарий — это строка (или несколько строк) текста, которая вставляется в исходный код для объяснения того, что делает код. В языке C++ есть 2 типа комментариев: однострочные и многострочные. Однострочные комментарии — это комментарии, которые пишутся после символов //. Они пишутся в отдельных строках и всё, что находится после этих символов комментирования, — игнорируется компилятором, например:

std::cout << «Hello, world!» << std::endl; // всё, что находится справа от двойного слеша, - игнорируется компилятором

Многострочные комментарии — это комментарии, которые пишутся между символами /\* \*/. Всё, что находится между звёздочками, — игнорируется компилятором:

/\* Это многострочный комментарий.

Эта строка игнорируется

и эта тоже. \*/

Препроцессор — это специальная программа, являющаяся частью компилятора языка Си. Она предназначена для предварительной обработки текста программы. Препроцессор позволяет включать в текст программы файлы и вводить макроопределения.

Работа препроцессора осуществляется с помощью специальных директив (указаний). Они отмечаются знаком решетка #. По окончании строк, обозначающих директивы в языке Си, точку с запятой можно не ставить.

Основные директивы препроцессора

#include — вставляет текст из указанного файла

#define — задаёт макроопределение (макрос) или символическую константу

#undef — отменяет предыдущее определение

#if — осуществляет условную компиляцию при истинности константного выражения

#ifdef — осуществляет условную компиляцию при определённости символической константы

#ifndef — осуществляет условную компиляцию при неопределённости символической константы

#else — ветка условной компиляции при ложности выражения

#elif — ветка условной компиляции, образуемая слиянием else и if

#endif — конец ветки условной компиляции

#line — препроцессор изменяет номер текущей строки и имя компилируемого файла

#error — выдача диагностического сообщения

#pragma — действие, зависящее от конкретной реализации компилятора.

15

15. Структура программы на языке СИ.

Программа на языке Cи состоит из функций, описаний и директив препроцессора.

Одна из функций должна иметь имя main. Выполнение программы начинается с первого оператора этой функции.

Простейшее определение функции имеет следующий формат:

тип возвращаемого\_значения имя (параметры) {операторы, составляющие тело функции}

Как правило, функция используется для вычисления какого-либо значения, поэтому перед именем функции указывается его тип.

Если функция не должна возвращать значение, указывается тип void.

Функции не могут быть вложенными. Тело функции является блоком и заключается в фигурные скобки.

Программа может состоять из нескольких модулей (исходных файлов).

16

16. Основные типы данных.

bool: логический тип. Может принимать одну из двух значений true (истина) и false (ложь). Размер занимаемой памяти для этого типа точно не определен.

char: представляет один символ в кодировке ASCII. Занимает в памяти 1 байт (8 бит). Может хранить любое значение из диапазона от -128 до 127, либо от 0 до 255

signed char: представляет один символ. Занимает в памяти 1 байт (8 бит). Может хранить любой значение из диапазона от -128 до 127

unsigned char: представляет один символ. Занимает в памяти 1 байт (8 бит). Может хранить любой значение из диапазона от 0 до 255

wchar\_t: представляет расширенный символ. На Windows занимает в памяти 2 байта (16 бит), на Linux - 4 байта (32 бита). Может хранить любой значение из диапазона от 0 до 65 535 (при 2 байтах), либо от 0 до 4 294 967 295 (для 4 байт)

char16\_t: представляет один символ в кодировке Unicode. Занимает в памяти 2 байта (16 бит). Может хранить любой значение из диапазона от 0 до 65 535

char32\_t: представляет один символ в кодировке Unicode. Занимает в памяти 4 байта (32 бита). Может хранить любой значение из диапазона от 0 до 4 294 967 295

short: представляет целое число в диапазоне от –32768 до 32767. Занимает в памяти 2 байта (16 бит).

Данный тип также имеет синонимы short int, signed short int, signed short.

unsigned short: представляет целое число в диапазоне от 0 до 65535. Занимает в памяти 2 байта (16 бит). Данный тип также имеет синоним unsigned short int.

int: представляет целое число. В зависимости от архитектуры процессора может занимать 2 байта (16 бит) или 4 байта (32 бита). Диапазон предельных значений соответственно также может варьироваться от –32768 до 32767 (при 2 байтах) или от −2 147 483 648 до 2 147 483 647 (при 4 байтах). Но в любом случае размер должен быть больше или равен размеру типа short и меньше или равен размеру типа long

Данный тип имеет синонимы signed int и signed.

unsigned int: представляет положительное целое число. В зависимости от архитектуры процессора может занимать 2 байта (16 бит) или 4 байта (32 бита), и из-за этого диапазон предельных значений может меняться: от 0 до 65535 (для 2 байт), либо от 0 до 4 294 967 295 (для 4 байт).

В качестве синонима этого типа может использоваться unsigned

long: представляет целое число в диапазоне от −2 147 483 648 до 2 147 483 647. Занимает в памяти 4 байта (32 бита). У данного типа также есть синонимы long int, signed long int и signed long

unsigned long: представляет целое число в диапазоне от 0 до 4 294 967 295. Занимает в памяти 4 байта (32 бита) Имеет синоним unsigned long int.

long long: представляет целое число в диапазоне от −9 223 372 036 854 775 808 до +9 223 372 036 854 775 807. Занимает в памяти, как правило, 8 байт (64 бита). Имеет синонимы long long int, signed long long int и signed long long.

unsigned long long: представляет целое число в диапазоне от 0 до 18 446 744 073 709 551 615. Занимает в памяти, как правило, 8 байт (64 бита). Имеет синоним unsigned long long int.

float: представляет вещественное число ординарной точности с плавающей точкой в диапазоне +/- 3.4E-38 до 3.4E+38. В памяти занимает 4 байта (32 бита)

double: представляет вещественное число двойной точности с плавающей точкой в диапазоне +/- 1.7E-308 до 1.7E+308. В памяти занимает 8 байт (64 бита)

long double: представляет вещественное число двойной точности с плавающей точкой не менее 8 байт (64 бит). В зависимости от размера занимаемой памяти может отличаться диапазон допустимых значений.

void: тип без значения

17

Целый тип. Внутреннее представление в языке Си. Переполнение.

Размер типа int не определяется стандартом, а зависит от компьютера и компилятора. Для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного — 4 байта.

Спецификатор short перед именем типа указывает компилятору, что под число требуется отвести 2 байта независимо от разрядности процессора. Спецификатор long означает, что целая величина будет

занимать 4 байта. Таким образом, на 16-разрядном компьютере эквиваленты int и short int, а на 32-разрядном — int и long int.

Внутреннее представление величины целого типа — целое число в двоичном коде. При использовании спецификатора signed старший бит числа интерпретируется как знаковый (0 — положительное число, 1 — отрицательное). Спецификатор unsigned позволяет представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа. Таким образом, диапазон значений типа int зависит от спецификаторов. При использовании различных спецификаторов можно получить диапазон значений, лежащий в пределах от (-2)31до 231-1.

По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор signed можно опускать.

Переполнение — ситуация в компьютерной арифметике, при которой вычисленное в результате операции значение не может быть помещено в n-битный целочисленный тип данных.

18

Типы с плавающей (фиксированной) точкой. Внутреннее представление в

языке Си.

В СИ существуют два типа данных с плавающей точкой: float и double. Типы данных с плавающей точкой предназначены для хранения чисел с плавающей точкой. Типы данных float и double могут хранить как положительные, так и отрицательные числа с плавающей точкой. У типа данных float размер занимаемой памяти в два раза меньше, чем у типа данных double, а значит и диапазон принимаемых значений тоже меньше. Если тип данных float объявить с приставкой long, то диапазон принимаемых значений станет равен диапазону принимаемых значений типа данных double. В основном, типы данных с плавающей точкой нужны для решения задач с высокой точностью вычислений, например, операции с деньгами.

Внутреннее представление вещественного числа состоит из 2 частей: мантиссы и порядка. В IBM-совместимых ПК величины типа float занимают 4 байта, из которых один разряд отводится под знак мантиссы, 8 разрядов под порядок и 24 – под мантиссу.

Величины типы double занимают 8 байтов, под порядок и мантиссу отводятся 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка его диапазон.

19

Объявление объектов.

Первый шаг при создании объектов – объявление одной или нескольких **объектных переменных.** Аналогично обычным переменным объектные переменные можно создавать либо статически, либо динамически. Оба способа создания переменных реализованы только в С++. В остальных языках используется **только динамическое** создание объектов. Объекты адресуются своими **указателями,** хотя явно этого не видно, т.к. разрешено только сравнивать указатели на равенство и присваивать указателям допустимые значения.

Рассмотрим сначала объявление и создание объектов в С++.

Для статического создания объекта достаточно объявить одну или несколько переменных соответствующего классового типа:

**Circle** Circ1, Circ2;

При обработке этой строки компилятор включает в создаваемый код обращение к конструктору без параметров, поэтому объекты создаются с заранее заданными значениями свойств. При необходимости вместо конструктора без параметров можно вызвать конструктор с параметрами, например, следующим образом:

**Circle** Circ3 (50, 50, 20);

При обработке этой строки компилятор анализирует входные значения (их количество и тип) и включает код для вызова подходящего конструктора. После такого неявного создания объектов можно эти объекты использовать, обращаясь к их методам:

Circ1.Show ( ); // отобразить объект Circ1

Circ3.MoveTo (100, 100); // переместить объект Circ3

int r = Circ3.GetR ( ); // запросить радиус объекта Circ3

Для динамического создания объектов, прежде всего, необходимо объявить ссылочную переменную:

**Circle** \*pCirc1;

При обработке этой строки компилятор выделяет память только для хранения адреса будущего объекта. После этого записывается строка для динамического создания объекта, причем для выделения памяти используется стандартная функция **new,** a для инициализации выделенной области явно вызывается один из конструкторов класса:

pCirc1 = **new** Circle ( );

При необходимости объявление объектной переменной и вызов конструктора можно объединить в одной строке:

**Circle** \*pCirc1 = **new** Circle ( );

20

Преобразование типов. Приведение типов.

Любые операнды типа char, unsigned char или short преобразуются к типу int по правилам:

char расширяется нулем или знаком в зависимости от умолчания для char;

unsigned char расширяется нулем;

signed char расширяется знаком;

short, unsigned short и enum при преобразовании не изменяются.

Затем любые два операнда становятся либо int, либо float, double или long double.

Преобразование других типов данных осуществляется следующим образом:

Если один из операндов имеет тип long double, то другой преобразуется к типу long double.

Если один из операндов имеет тип double, то другой преобразуется к типу double.

Если один из операндов имеет тип float, то другой преобразуется к типу float.

Иначе, если один из операндов имеет тип unsigned long, то другой преобразуется к типу unsigned long.

Иначе, если один из операндов имеет тип long, то другой преобразуется к типу long.

Иначе, если один из операндов имеет тип unsigned, то другой преобразуется к типу unsigned.

Иначе оба операнда должны иметь тип int.

Тип результата тот же, что и тип участвующих в выражении операндов.

Любые типы можно преобразовывать в любые другие:

double I = (double)15 / 2;

21

Константы в программах (целочисленные, вещественные, символьные, строковые)

Символическая константа - это имя, значение которого не может быть изменено в его области видимости. Символьная константа - это символ, заключенный в одинарные кавычки; например, "a" или "0". Константа – это именованная область памяти, значение которой не может изменяться в процессе выполнения программы.

Константы — это данные, не изменяющие своего значения в ходе выполнения программы. В языке СИ можно выделить четыре вида констант: целые константы, константы с плавающей запятой, символьные константы и строковые литералы.

Целочисленная константа представляет собой число, записываемое явным образом, например, 212 или 1776. В языке С++, как и в С, целые числа можно записывать в трех различных системах счисления: с основанием 10 (наиболее распространенная форма), с основанием 8 (старая запись в системах семейства Unix) и с основанием 16 (излюбленная форма компьютерных хакеров).

Строковая константа - это массив символов, заключенный в кавычки. Она имеет тип string. Если необходимо ввести в строку двойную кавычку ("), то перед ней надо поставить символ обратной косой черты (\). В строку могут быть введены любые специальные символьные константы, перед которыми стоит символ обратной косой черты (\). Длина строковой константы - от 0 до 255 символов. Если длина строковой константы превосходит максимальную, лишние символы справа отбрасываются, и компилятор выдает соответствующее предупреждение.

**22**

**22. Операции и выражения. Унарные, бинарные и тернарные операции. Приоритет в выражениях. Приоритет операций.**

Операции используются для управления данными

|  |  |  |
| --- | --- | --- |
| Приоритет | Оператор | Описание |
| 1 | :: | Область видимости |
| 2 | Х++, х-- | Постинкремент |
| () | Вызов функции |
| [] | Обращение к массиву по индексу |
| . | Выбор элемента по ссылке |
| -> | Выбор элемента по указателю |
| 3 | ++х, --х | Прединкремент |
| +, - | Унарный плюс, унарный минус |
| !, ~ | Логическое «НЕ», побитовое «НЕ» |
| (type) | Приведение к типу |
| \* | Разыменование |
| & | Амперсант |
| sizeof | Размер |
| new[] | Динамическое выделение памяти |
| delete[] | Динамическое освобождение памяти |
| 4 | \*, /, % | Умножение, деление, остаток |
| 5 | +, - | Плюс, минус |
| 6 | <<, >> | Побитовый сдвиг влево, вправо |
| 7 | <, <=, >=, > | Операции сравнения |
| 8 | ==, != | Операции сравнения |
| 9 | & | Побитовое «И» |
| 10 | ^ | Исключающее «ИЛИ» |
| 11 | | | Побитовое «ИЛИ» |
| 12 | && | Логическое «И» |
| 13 | || | Логическое «ИЛИ» |
| 14 | ?: | Тернарное условие |
| 15 | = | Прямое присваивание |
| +=, -= | Присваивание с суммированием, разностью |
| \*=, /=, % | Присваивание с умножением, делением, остатка от деления |
| <<=, >>= | Присваивание с побитовым сдвигом |
| |=, ^=, &= | Присваивание с побитовыми операциями |
| 16 | , | Запятая |

23

23. Операции и выражения. Арифметические операции. Операция присваивания и ее сокращенная запись.

Арифметические операции

Операндами могут быть константы, переменные, функции, элементы массива, указатели, любые арифметические выражения.

Порядок выполнения:

1. Выражения в круглых скобках

2. Функция

3. \*, /

4. +, -

Унарные +, - обладают самым высоким приоритетом.

Операции присваивания

Операции присваивания присваивают значение правого операнда левому

− Изменяет значение в нескольких операндах(i=j=k=0)

− Операции присваивания, которые сочетаются с математическими операциями (+=, -= и т.д.)

24

24. Операции и выражения. Операции сравнения. Логические операции.

Операция запятая.

Операции сравнения

Больше > x > y true, если x больше y, в противном случае — false

Меньше < x < y true, если x меньше y, в противном случае — false

Больше или равно >= x >= y true, если x больше/равно y, в противном случае — false

Меньше или равно <= x <= y true, если x меньше/равно y, в противном случае — false

Равно == x == y true, если x равно y, в противном случае — false

Не равно != x != y true, если x не равно y, в противном случае — false

Логические операции

Логическое НЕ ! !x true, если x — false и false, если x — true

Логическое И && x && y true, если x и y — true, в противном случае — false

Логическое ИЛИ || x || y true, если x или y — true, в противном случае — false

Оператор «запятая» используется для связки нескольких выражений.

х = (у = 3, у + 1); //Сначала присваивается 3 переменной у, а затем 4.

25

25. Операции и выражения. Побитовые логические операции. Операции над битами.

& И

| ИЛИ

^ Исключающее ИЛИ

~ Побитовое НЕ

>> Сдвиг вправо

<< Сдвиг влево

26

Стандартные математические функции. Файлы math.h, stdlib.h

**Стандартные математические функции**

Стандартные математические функции существуют для облегчения программирования математических вычислений; функции работают со значениями, представленными в форме с плавающей точкой.*Основные математические функции* объявлены в *include*-файле ***<math.h>***.и stdlib.h

|  |  |
| --- | --- |
| ФУНКЦИЯ | ИСПОЛЬЗОВАНИЕ |
| acos(x) | вычисляет *arccos x* |
| asin(x) | вычисляет *arcsin x* |
| atan(x) | вычисляет *arctg x* |
| atan2(y,x) | вычисляет *arctg(y/x)* |
| ceil(x) | находит целую часть *x* |
| cos(x) | вычисляет *cos x* |
| exp(x) | вычисляет экспоненциальную функцию |
| fabs(x) | вычисляет абсолютное значение *x* |
| floor(x) | находит наибольшее целое, которое меньше или равно *x* |
| fmod(x,y) | находит остаток с плавающей точкой от деления *x* на *у* |
| log(x) | вычисляет натуральный логарифм *х* |
| log10(x) | вычисляет десятичный логарифм *х* |
| modf(x,&n) | разделяет *х* на целую (возвращаемое значение) и дробную (*n*) части |
| pow(x,y) | вычисляет *х* в степени *y* |
| sin(x) | вычисляет *sin x* |
| sqrt(x) | находит квадратный корень из *х* |
| tan(x) | вычисляет *tg x* |

27

Функции форматного ввода. Управляющая строка. Управляющие символы. Спецификаторы формата.

Форматный ввод данных осуществляется с помощью функцийscanf, fscanf,sscanfиcscanf. Все названные функции вводят значения, интерпретируют их в соответствии с "типом форматов, преобразуют их'в соответствии с типом аргументов и присваивают введенные значения переменным. Все функции типаscanfвводят последовательно значения из полей входного потока, интерпретируя их в соответствии с их форматом.

Управля́ющие си́мволы — символы в кодировке, которым не приписано графическое представление, но которые используются для управления устройствами, организации передачи данных и других целей. Управляющие символы ASCII предназначались для управления работой телетайпов и видеотерминалов и вводились на них сочетаниями с клавишей Ctrl, которая обнуляла в коде введённой клавиши бит 6.

**printf***(<управляющая\_строка>, <аргумент\_1>, <аргумент\_2>, ... );*

*Управляющая\_строка*содержит объекты трех типов: обычные символы, которые просто выводятся на экран, спецификации преобразования, каждая из которых вызывает вывод на экран значения очередного аргумента из последующего списка и управляющие символы-константы. Каждая *спецификация преобразования*, как и в функции форматного ввода, начинается со знака *%* и заканчивается также *символом преобразования*. Если после знака *%* записан не символ, то он выводится на экран. Функция *printf*использует управляющую строку, чтобы определить, сколько всего аргументов и какие их типы.

Перед спецификаторами формата ввода ставится знак %, и они сообщают функции scanf() о типе далее читаемых данных. Эти спецификаторы перечислены в таблице. Спецификаторы формата рассматриваются слева направо, и в таком же порядке с ним сопоставляются аргументы из списка аргументов.

| **Код** | **Значение** |
| --- | --- |
| %c %d %i %е %f %g %о %s %x %р %n %u %[] | Читает одиночные символы Читает десятичное число Читает десятичное число Читает число с плавающей запятой Читает число с плавающей запятой Читает число с плавающей запятой Читает восьмеричное число Читает строку Читает шестнадцатиричное число Читает указатель Получает целочисленное значение, равное числу прочитанных символов Читает беззнаковое целое Сканирует множество символов |
| Таблица: Спецификаторы формата функции scanf() | |

28

Функции форматного вывода. Управляющая строка. Спецификаторы формата.

Форматный вывод данных осуществляется с помощью функцийprintf,fprintf, sprintf

**printf***(<управляющая\_строка>, <аргумент\_1>, <аргумент\_2>, ... );*

*Управляющая\_строка*содержит объекты трех типов: обычные символы, которые просто выводятся на экран, спецификации преобразования, каждая из которых вызывает вывод на экран значения очередного аргумента из последующего списка и управляющие символы-константы. Каждая *спецификация преобразования*, как и в функции форматного ввода, начинается со знака *%* и заканчивается также *символом преобразования*. Если после знака *%* записан не символ, то он выводится на экран. Функция *printf*использует управляющую строку, чтобы определить, сколько всего аргументов и какие их типы.

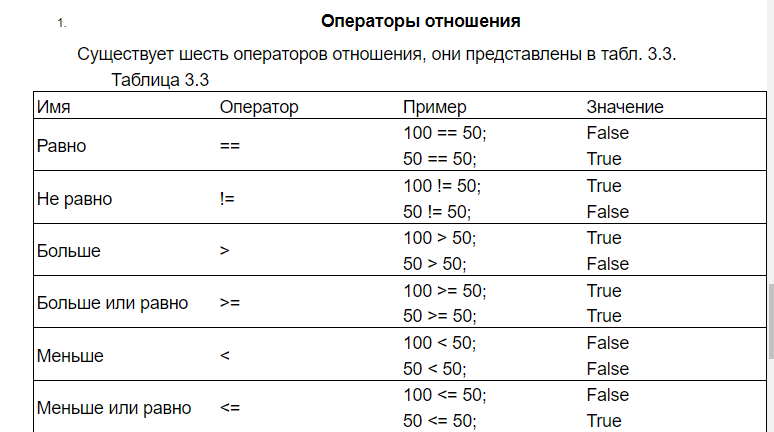
Перед спецификаторами формата ввода ставится знак %, и они сообщают функции scanf() о типе далее читаемых данных. Эти спецификаторы перечислены в таблице. Спецификаторы формата рассматриваются слева направо, и в таком же порядке с ним сопоставляются аргументы из списка аргументов.

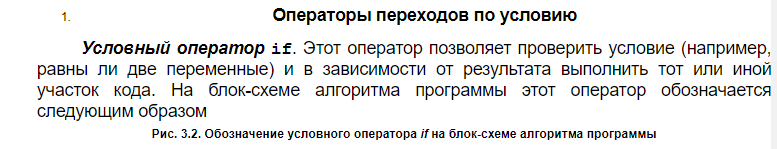
| **Код** | **Значение** |
| --- | --- |
| %c %d %i %е %f %g %о %s %x %р %n %u %[] | Читает одиночные символы Читает десятичное число Читает десятичное число Читает число с плавающей запятой Читает число с плавающей запятой Читает число с плавающей запятой Читает восьмеричное число Читает строку Читает шестнадцатиричное число Читает указатель Получает целочисленное значение, равное числу прочитанных символов Читает беззнаковое целое Сканирует множество символов |
| Таблица: Спецификаторы формата функции scanf() | |

29

Категории операторов. Синтаксис и семантика операторов. Пустой оператор. Составной оператор.

В языке С++ операторы управляют последовательностью выполнения выражений, возвращают результаты вычислений или ничего не делают (пустые операторы). Операторы последовательного действия выполняют определенные действия над операндами – одно за другим  
x=a+b; Здесь a и b – операнды, а x – результат операции.  
Оператор присвоения (=) присваивает результаты операций, выполняемых над операндами, расположенными справа от знака равенства, операнду, находящемуся слева от него.

Арифметические операторы. Существуют пять арифметических операторов: сложения (+), вычитания (-), умножения (\*), целочисленного деления (/) и деления по модулю (%).  
  
Оператор инкремента (++) увеличивает значение переменной на единицу, а оператор декремента (--) уменьшает его на единицу.  


  
Любая группа операторов, размещенных между словами BEGIN и END (иначе, операторные скобки), рассматривается как один - составной оператор. Пустой оператор - это оператор, который не предусматривает выполнения никаких действий.

30

Операторы цикла. Цель использования операторов цикла. Итерация. Параметры цикла.

циклы нужны для экономии памяти отводимой под програмные инструкции при необходимости повторять одни и те же действия много раз.  
например, надо заполнить подряд идущие 1000 ячеек памяти. можно написать тысячу команд, которые будут записывать каждая в свою ячейку. а можно организовать цикл и команд потребуется на три порядка меньше.

**Оператор цикла**– это инструкция, предписывающая циклическую операцию.

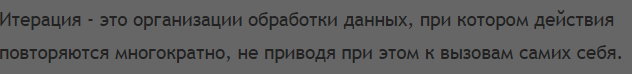
В С++ используется три вида операторов цикла:

– ***for****(параметрический, с предусловием),*

– ***while****(с предусловием),*

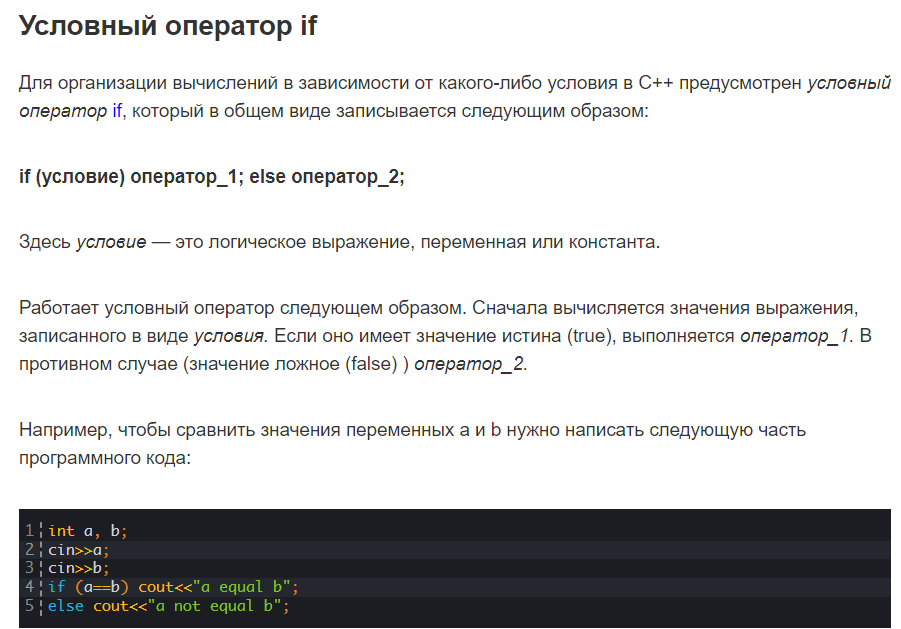
– ***do…while****(с послеусловием).*

Параметр цикла - переменная, управляющая выполнением цикла. Параметр цикла изменяет свое значение при каждом проходе тела цикла.(ЕСЛИ ЧТО ПРИВЕСТИ ПРИМЕР С ПЕРЕМЕННОЙ I В ЦИКЛЕ ФОР)

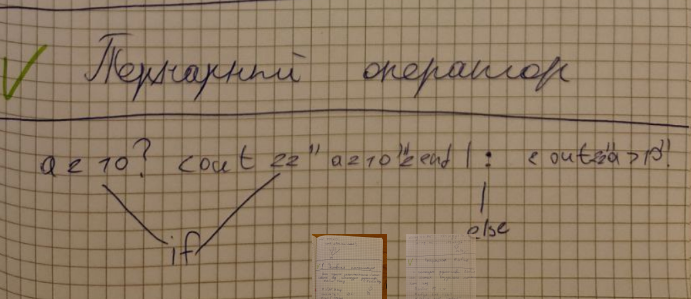


31

Условные операторы. Тернанрая операция.







32

Оператор выбора альтернатив switch. Операторы передачи управления (goto, continue, break, return)

# **Операторы передачи управления goto,continue,break,return.**

Формально простыми операторами передачи управления являются:

– безусловный переход ***goto***(перейти на . . . );

– переход к следующему шагу цикла ***continue***;

– ***break***– выход из цикла и оператора ***switch***;

– оператор возврата из функции ***return***.

Оператор безусловного перехода

***goto*** ***Метка*** ;

+передает управление оператору с указанной ***Меткой***. ***Метка*** – идентификатор, записанный по синтаксису языка Си с символом «двоеточие» после него. Например, пустой оператор «**;**» с меткой ***m*1** :

Общая форма записи оператора switch следующа

switch (***выражение***)

{

case *константа1*:

*последовательность\_операторов1*;

break;

case *константа2*:

*последовательность*\_*операторов2*;

break;

...

case *константаN*:

*последовательность\_операторовN*;

break;

default

*последовательность\_операторов*;

}

33

Массивы. Объявления. Инициализация. Доступ к элементам массива. Действия над элементами массива.

**Объявление массива**

Массив - именованная группа данных одного типа, собранных под одним именем

[М\_К\_П](модификатор класса памяти) <тип><список\_массивов>

<тип> имя [P1][P2]...[Pn]

int arr[10] - массив целых чисел размером в 10 элементов.

int arr [i][j] /\* [строка][столбец] \*/

**Инициализация**

<тип>имя [Размер] = {список\_инициализаторов}

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; - одномерный

int [2][3] = {{1,2,3},{4,5,6}} - двухмерный

**Доступ к элементу массива**

Доступ к элементу массива осуществляется с помощью имени массива и индекса

Имя [номер\_элемента] - одномерный

Имя [P1][P2]...[Pn] - многомерный

а[0]=1

**Индексное выражение**

Индексное выражение arr[5] ссылается на шестой элемент массива, т.к. нумерация элементов идет с нуля.

Если необходимо обратиться к 10 элементу, надо написать аrr[9] или аrr+9.

34

Одномерный, двумерный и многомерный массив.

Массив — структура данных, хранящая набор значений, идентифицируемых по индексу или набору индексов, принимающих целые значения из некоторого заданного непрерывного диапазона. Одномерный массив можно рассматривать как реализацию абстрактного типа данных — вектор

### Многомерные массивы

Многомерный массив - это массив массивов, т. е. массив, элементами которого являются массивы. Размерность массива - это количество индексов, используемых для ссылки на конкретный элемент массива. Многомерные массивы объявляются точно так же, как и одномерные, только после имени массива ставится более одной пары квадратных скобок. Пример определения двухмерного массива (матрицы) с 10 строками и 30 столбцами:

int array[10][30];

Двумерный массив (прямоугольная таблица (матрица, набор векторов)) - это пример массива, в котором элементы нумеруются двумя индексами, элемент которого зависит от его местоположения в строке и в столбце.

В качестве номера (индекса) элемента массива используется выражение порядкового типа (чаще integer).

1. Для определения позиции элемента в двумерном массиве необходимы два индекса. Любой двумерный массив есть матрица, а матрица есть таблица. Поэтому удобно описывать двумерные массивы путем указания границ изменения индексов (номеров) строк и столбцов.

35

Указатели. Инициализация указателей. Примеры работы. Область применения.

Указатель - это какой-либо адрес в памяти вашего компьютера.

Можно выделить следующие основные области применения указателей:

+

1Использование указателей в качестве параметров функций для возврата результатов вычислений.

2Работа с объектами, размещенными в динамической памяти.

3Замена манипуляций со сложными объектами данных манипуляцией с адресами этих объектов.

# **Примеры использования указателей**

Пример 1. Объявить и инициализировать переменную типа float и указатель на нее. Вывести на экран содержимое области памяти, на которую указывает этот указатель.

#include <stdio.h>

int main()

{

float f=100.25;

float \*pi=&f;

printf(“Значение переменной f= %f \n”,f);

printf(“Адрес переменной f: pi=%p\n”,&f); //Форматная спецификация %p

//используется для печати значения указателя

printf(“Cодержимое ячейки по адресу pi %f”,\*pi);

36

Операторы адреса и разыменования. Операции над указателями. Область применения указателей.

Для получения адреса переменной используется операция адреса (&). Она представляет собой унарную операцию, результатом выполнения которой является адрес переменной в памяти – своего операнда. Используя операцию адреса (&), можно получить адрес любых переменной (как глобальных, так и локальных), массивов и других данных. Операция разыменования \* (или операция косвенной адресации) – унарная операция для доступа к значению переменной, расположенной по адресу операнда.

**Объявления и инициализация переменных** **указателей**

**int \*countPtr, count;**

объявляет переменную **countPtr**типа int \* (т.е. указатель на целое число) и читается как **<<countPtr**является указателем на целое число» или **<<countPtr**указывает на объект типа int\*. Однако переменная **count**объявлена как целое число, но не как указатель на целое число. Символ \* в объявлении относится только к **countPtr.**Каждая переменная, объявляемая как указатель, должна иметь перед собой знак звездочки (\*).

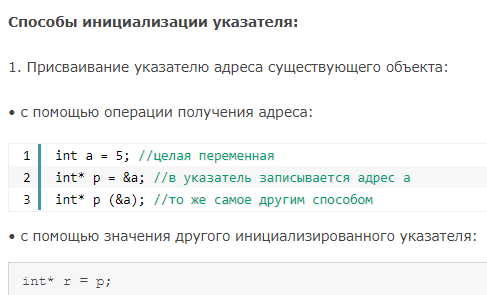
& или *операция адресации,*— унарная операция, которая возвращает адрес своего операнда. Например, если имеются объявления

**int у = 5; int \*yPtr;**

то оператор

+**yPtr = &y;**

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. 
4. 37
5. Указатели и const.
6. Мы также можем сделать указатель константным. **Константный указатель**— это указатель, значение которого не может быть изменено после инициализации. Для объявления константного указателя используется **ключевое слово const между звёздочкой и именем указателя**:

|  |  |
| --- | --- |
| 1  2 | int value = 7;  int \*const ptr = &value; |

1. Подобно обычным константным переменным, константный указатель должен быть инициализирован значением при объявлении. Это означает, что он всегда будет указывать на один и тот же адрес. В вышеприведенном примере ptr всегда будет указывать на адрес value (до тех пор, пока указатель не выйдет из [**области видимости**](https://ravesli.com/urok-52-svyazi-oblast-vidimosti-i-prodolzhitelnost/#toc-0) и не уничтожится):
2. Однако, поскольку переменная value, на которую указывает указатель, не является константой, то её значение можно изменить путем разыменования константного указателя:

38

# **Ссылки**

В C++ имеется несколько видоизмененная форма указателя, называемая ссылкой. Ссылка на некоторую переменную может рассматриваться как указатель, который при употреблении всегда разыменовывается. Однако для ссылки не требуется дополнительного пространства в памяти: она является просто другим именем или псевдонимом уже существующей объявленной или описанной переменной. Поэтому все, что делается со ссылкой, в действительности происходит с переменной, на которую она ссылается. Ссылка определяется в программе следующим образом:

тип &имя\_ссылки=имя\_переменной;

При этом переменная, на которую указывает (ссылается) ссылка, должна быть объявлена или описана до определения ссылки, например int t, &rt = t;. Оператор ссылки (&) выглядит так же, как оператор адреса, который используется для возвращения адреса при работе с указателями. Но это не одинаковые операторы. При определении ссылки необходимо ее инициализировать. Если объявить ссылку без инициализации, будет сгенерирована ошибка компиляции. Так как ссылка является псевдонимом своего адресата, то применение оператора взятия адреса к ссылке возвратит адрес ее адресата. Переназначить ссылку нельзя. Она ссылается на свой адресат и только на него. Ссылка не может быть нулевой.

39

# Применение указателей для работы с массивами. Одномерный динамический массив. Выделение/высвобождение памяти. ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ ПРИ РАБОТЕ С МАССИВАМИ

ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ ПРИ РАБОТЕ С МАССИВАМИ

     Попробуем написать функцию, использующую массивы, а затем перепишем ее, применяя указатели.

     Рассмотрим простую функцию, которая находит (или пытается найти) среднее значение массива целых чисел. На входе функции мы имеем имя массива и количество элементов. На выходе получаем среднее значение, которое передастся при помощи оператора **return**. Оператор вызова функции может выглядеть следующим образом:

printf("Среднее из заданных значений %d. ", mean(numbs,size));

/\* находит среднее значение массива из n целых чисел \*/

int mean(array, n);

int array[ ], n;

{

int index;

long sum; /\* Если целых слишком много, их можно

            суммировать в формате long int \*/

if(n > 0)

{

for(index = 0, sum = 0; index < n; index++)

    sum + = array[index];

return((int)(sum/n)); /\* возвращает int \* / }

else {

printf("Нет массива. ");

return(0); }

}

     Эту программу легко переделать, применяя указатели. Объявим **ра** указателем на тип **int**. Затем заменим элемент массива **array[index]** на соответствующее значение: **\*(ра + index)**.

/\* Использование указателей для нахождения

        среднего значения массива n целых чисел \*/

int mean(pa, n) int oра, n;

{

int index;

long sum; /\*Если целых слишком много,

    их можно суммировать в формате long int \*/

if(n > 0)

{

for(index=0, sum=0; index < n; index++)

sum + = \*(pa + index);

return((int)(sum/n)); /\* Возвращает целое \*/ }

else {

printf("Нет массива. ");

return(0); }

}

Это оказалось несложным, но возникает вопрос: должны ли мы изменить при этом вызов функции, в частности **numbs**, который был именем массива в операторе **mean(numbs, size)**? Ничего не нужно менять, поскольку имя массива является указателем. Как мы уже говорили в предыдущем разделе, операторы описания:

int ра[ ];

и

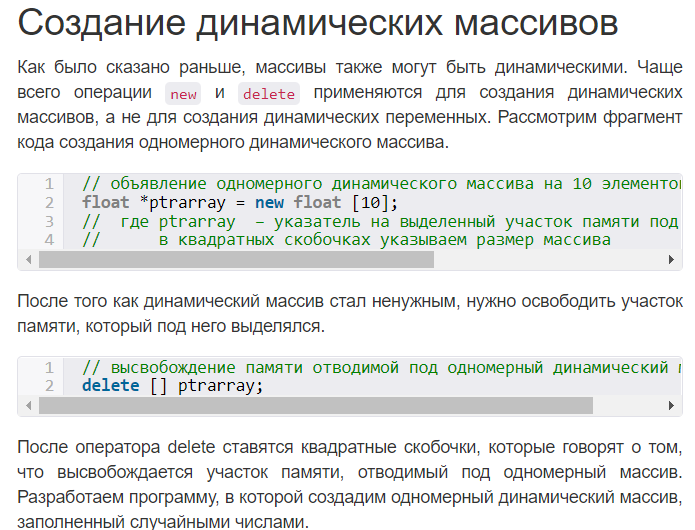
int \*ра;

идентичны по действию: оба объявляют ра указателем. В программе можно применять любой из них, хотя до сих пор мы использовали второй в виде **\*(ра + index)**.

     Понятно ли вам, как работать с указателями? Указатель устанавливается на первый элемент массива, и значение, находящееся там, добавляется в **sum**. Затем указатель передвигается на следующий элемент (к указателю прибавляется единица), и значение, находящееся в нем, также прибавляется к **sum** и т. д. Это похоже на механизм работы с массивом, где индекс действует как стрелка часов, показывающая по очереди на каждый элемент массива.

Теперь у нас есть два подхода; какой же из них выбрать? Во-первых, хотя массивы и указатели тесно связаны, у них есть отличия. Указатели являются более общим и широко применяемым средством, однако многие пользователи (по крайней мере начинающие) считают, что массивы более привычны к понятны. Во-вторых, при использовании указателей у нас нет простого эквивалента для задания размера массива. Самую типичную ситуацию, в которой можно применять указатель, мы уже показали: это функция, работающая с массивом, который находится где-то в другой части программы. Мы предлагаем использовав любой из подходов по вашему желанию. Однако несомненное преимущество использования указателей в приведенном выше примере должно научить вас легко применять их, когда в этом возникает необходимость.

В С++ операции new и delete предназначены для динамического распределения памяти компьютера.  Операция new  выделяет память из области свободной памяти, а операция delete высвобождает выделенную память.  Выделяемая память, после её использования должна высвобождаться, поэтому операции new и delete используются парами. Даже если не высвобождать память явно, то она освободится ресурсами ОС по завершению работы программы. Рекомендую все-таки не забывать про операцию delete.



40

Двумерный динамический массив. Выделение/высвобождение памяти.

Для хранения информации можно использовать систему динамического выделения памяти. При таком способе память под данные выделяется по мере необходимости из объема свободной памяти, которая располагается между вашей программой (вместе с ее областью хранения постоянных данных) и стеком. Эта область памяти называется – кучей.

Динамическое выделение памяти осуществляется во время выполнения программы. Программа может создать столько переменных, сколько ей требуется в зависимости от ситуации.

ДВП обеспечивает значительное повышение гибкости, но имеет свои ограничения, т.к. память берется из кучи и может исчерпаться.

В С++ существуют два оператора для ДВП: new (выделяет память и возвращает указатель на ее начало) и delete (освобождает память, предварительно выведенную посредством оператора new).

*Синтаксис.*

x=new тип;

delete x;

*Выделение и удаление памяти под одномерные массивы.*

x=new тип-массива [размер];

delete []x;

При выделении памяти под массивы им нельзя присвоить начальное значение.

*Выделение и удаление памяти под двумерные массивы.*

// объявление двумерного динамического массива на 10 элементов:  
float \*\*ptrarray = new float\* [2]; // две строки в массиве  
for (int count = 0; count < 2; count++)  
ptrarray[count] = new float [5]; // и пять столбцов  
// где ptrarray – массив указателей на выделенный участок памяти под массив вещественных чисел типа float

// высвобождение памяти отводимой под двумерный динамический массив:

for (int count = 0; count < 2; count++)

delete [] ptrarray[count];

// где 2 – количество строк в массиве

41

Многомерные массивы и указатели на указатели.

Многомерные массивы - это массивы с более чем одним индексом. Многомерные массивы - это матрицы, имеющие строки и столбцы. Многомерные или многосвязные системы это системы, имеющие несколько управляемых величин, а также несколько задающих и возмущающих воздействий

Указатели на многомерные массивы в языке СИ — это массивы массивов, т.е. такие массивы, элементами которых являются массивы. При объявлении таких массивов в памяти компьютера создается несколько различных объектов. Например, при выполнении объявления двумерного массива int arr[4][3] в памяти выделяется участок для хранения значения переменной arr, которая является указателем на массив из четырех указателей. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа int, и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа int, каждый из которых состоит из трех элементов.

42

Поэтому **динамическое выделение памяти** C можно определить как процедуру, в которой размер структуры данных (например, массива) изменяется во время выполнения.

Динамическая память, функции malloc(), calloc(), realloc(), free()

**C malloc () метод**

Метод **«malloc»** или **«распределение памяти»** в C используется для динамического выделения одного большого блока памяти с указанным размером. Он возвращает указатель типа void, который может быть приведен к указателю любой формы.

**Синтаксис:**

ptr = (cast-type\*) malloc(byte-sisize)

**C calloc () метод**

Метод **«calloc»** или **«непрерывное распределение»** в C используется для динамического выделения указанного количества блоков памяти указанного типа. Он инициализирует каждый блок значением по умолчанию «0».

**Синтаксис:**

ptr = (cast-type\*)calloc(n, elemnt-size)

**C free () метод**

**«Свободный»** метод C используется для динамического **де-выделить** память. Память, выделенная с использованием функций malloc () и calloc (), не выделяется самостоятельно. Следовательно, метод free () используется всякий раз, когда происходит динамическое выделение памяти. Это помогает уменьшить потери памяти, освобождая ее.

**Синтаксис:**

free(ptr);

**C realloc () метод**

Метод **«realloc»** или **«перераспределение»** в C используется для динамического изменения выделения памяти ранее выделенной памяти. Другими словами, если памяти, ранее выделенной с помощью malloc или calloc, недостаточно, realloc может использоваться для **динамического перераспределения памяти** .

**Синтаксис:**

ptr = realloc(ptr, newSize);

where ptr is reallocated with 'newSize'.

43

Строки. Объявление. Инициализация. Действия над строками и элементами строки.

*string ИмяСтроки;*

Размер строки указывать не требуется, он будет автоматически определен при инициализации или присавивании.

 *string s=”ГУ-ВШЭ”;*

**Ввод\вывод**

gets(s); //ввод строки s

puts(s); // вывод строки s

Можно использовать для вывода строки

printf(“%s”,T);

При использовании для ввода строки с помощью

scanf(“%s”,T);

Необходимо учитывать, что разделителем элементов списка ввода является пробел, поэтому строка будет введена до первого пробела.

**Действия со строками:**

Копирование одной строки в другую, сравнение двух строк, поиск подстроки в строке, поиск символа в строке, соединение строк и т.д.

Все действия со строками выполняются

с помощью посимвольной обработки массива символов,

+с помощью функций, описанных в заголовочном файле <string.h>,

с помощью функций, описанных в заголовочном файле <string.h> и рабочих указателей.

44 Строки. Ввод/вывод строк.

Строка представляет собой массив символов, заканчивающийся нуль-символом. Нуль-символ — это символ с кодом, равным О, что записывается в виде управляющей последовательности. По положению нуль-символа определяется фактическая длина строки. Длина Строки определяется с помощью функции strlen, которая вычисляет длину, выполняя поиск нуль-символа.

Объявление и инициализация.

Строки реализуются посредством массивов символов

Как видите, есть несколько основных способов вывести строку на экран.

* использовать функцию **printf** со спецификатором **%s**
* использовать функцию **puts**
* использовать функцию **fputs**, указав в качестве второго параметра стандартный поток для вывода **stdout**.

ВВОД СТРОК

Единственный нюанс у функций **puts** и **fputs**. Обратите внимание, что функция **puts** переносит вывод на следующую строку, а функция **fputs** не переносит.

Как видите, с выводом всё достаточно просто.

Функция gets приостанавливает работу программы, читает строку символов, введенных с клавиатуры, и помещает в символьный массив, имя которого передаётся функции в качестве параметра.  
Завершением работы функции **gets** будет являться символ, соответствующий клавише ввод и записываемый в строку как нулевой символ.  
Заметили опасность? Если нет, то о ней вас любезно предупредит компилятор. Дело в том, что функция **gets** завершает работу только тогда, когда пользователь нажимает клавишу ввод. Это чревато тем, что мы можем выйти за рамки массива, в нашем случае — если введено более **20** символов.  
К слову, ранее ошибки переполнения буфера считались самым распространенным типом уязвимости. Они встречаются и сейчас, но использовать их для взлома программ стало гораздо сложнее.

Итак, что мы имеем. У нас есть задача: записать строку в массив ограниченного размера. То есть, мы должны как-то контролировать количество символов, вводимых пользователем. И тут нам на помощь приходит функция **fget**

45

Строки. Библиотека string.h. Прототипы функций для работы со строкой.

Строка представляет собой массив символов, заканчивающийся нуль-символом. Нуль-символ — это символ с кодом, равным О, что записывается в виде управляющей последовательности. По положению нуль-символа определяется фактическая длина строки. Длина Строки определяется с помощью функции strlen, которая вычисляет длину, выполняя поиск нуль-символа.

Объявление и инициализация.

Строки реализуются посредством массивов символов

ФУНКЦИИ БИБЛИОТЕКИ STRING H

Сложение строк

char \*strcat(char \*s1, char \*s2);

Добавляет s2 к s1 и возвращает s1. В конец результирующей строки добавляется нуль-символ.

Поиск символа в строке

char \*strchr(char \*s, int ch);

Возвращает указатель на первое вхождение символа ch в строку s, если его нет, то возвращается NULL.

Сравнение строк

int \*strcmp(char \*s1, char \*s2);

Сравнивает строки и возвращает отрицательное (если s1 меньше s2), нулевое (если s1 равно s2) или положительное (если s1 больше s2) значение.

char \*strcoll(char \*sl, char \*s2);

Cравнивает строки аналогично strcmp, но учитывает установки локализации (информацию, зависящую от страны).

Копирование строк

char \*strcpy(char \*sl, char \*s2);

Функция копирует s2 в s1 и возвращает s1.

Поиск символов одной строки в другой

size\_t strcspn(char \*sl, char \*s2);

Возвращает значение индекса любого из символов из s2 в строке s1.

Поиск строки с описанием ошибки

char \*strerror(size\_t n);

Возвращает указатель на строку с описанием ошибки номер n.

Определение длины строки

size\_t strlen(char \*s);

+Функция возвращает длину строки (без учета нуль-символа).

Сложение строки с несколькими символами другой строки

46

Модульное программирование

Концепцию модульного программирования можно сформулировать в виде

нескольких понятий и положений:

· Функциональная декомпозиция задачи - разбиение большой задачи на ряд

более мелких, функционально самостоятельных подзадач - модулей. Модули

связаны между собой только по входным и выходным данным.

· Модуль - основа концепции модульного программирования. Каждый модуль

в функциональной декомпозиции предста

входом и одним выходом. Модульный подход позволяет безболезненно

производить модернизацию программы в процессе ее эксплуатации и

облегчает ее сопровождение. Дополнительно модульный подход позволяет

разрабатывать части программ одного проекта на разных языках

программирования, после чего с помощью компоновочных средств

объединять их в единый загрузочный модуль.

· Реализуемые решения должны быть простыми и ясными. Если назначение

модуля непонятно, то это говорит о том, что декомпозиция начальной или

промежуточной задачи была проведена недостаточно качественно. В этом

случае необходимо еще раз проанализировать задачу и, возможно, провести

дополнительное разбиение на подзадачи. При наличии сложных мест в

проекте их нужно подробнее документировать с помощью продуманной

системы комментариев. Этот процесс нужно продолжать до тех пор, пока

действительно не удастся добиться ясного понимания назначения всех

модулей задачи и их оптимального сочетания.

· Назначение всех переменных модуля должно быть описано с помощью

комментариев по мере их определения.

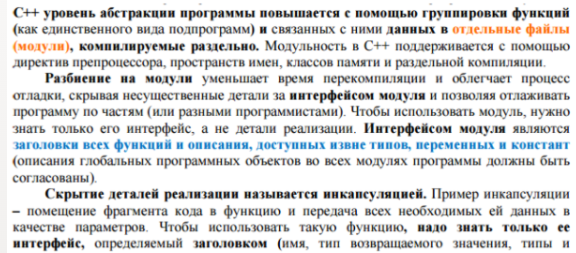
47

Подпрограмма — поименованная или иным образом идентифицированная часть компьютерной программы, содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы. В языках программирования для оформления и использования подпрограмм существуют специальные синтаксические средства

Подпрограммы. Виды подпрограмм.

Подпрограммы могут быть двух видов: подпрограмма без параметров и подпрограмма с параметрами. Обращение к подпрограмме может быть организовано из любого места основной программы или другой подпрограммы сколько угодно раз.

При работе с подпрограммами важными являются понятия формальных и фактических параметров. Формальные параметры — это идентификаторы входных данных для подпрограммы. Если формальные параметры получают конкретные значения, то они называются фактическими.



48

тип имя\_функции(тип имя\_парам1, тип имя\_парам2, ..., имя\_парамN);

Любая функция имеет тип, также, как и любая переменная.

Функция может возвращать значение, тип которого в большинстве случаев аналогично типу самой функции.

Если функция не возвращает никакого значения, то она должна иметь тип void (такие функции иногда называют процедурами)

При объявлении функции, после ее типа должно находиться имя функции и две круглые скобки — открывающая и закрывающая, внутри которых могут находиться один или несколько аргументов функции, которых также может не быть вообще.

после списка аргументов функции ставится открывающая фигурная скобка, после которой находится само тело функции.

Оператор вызова функций, возвращающих результат по ссылке, имеет следующий вид:

|  |
| --- |
| *имя\_функции (имя1, имя2, . . . , имяN);* |

Здесь *имя\_функции –*имя вызываемой функции;

*(имя, имя2, . . . , имяN)*– список *аргументов*, состоящий из имен как входных, так и выходных данных, записанных в порядке соответствующем списку параметров. В этом списке входные данные – реальные данные, имеющие свои значения.

49

Обмен информацией между функциями.

При работе программы функции должны обмениваться информацией. Это можно осуществить с помощью глобальных переменных, через параметры и через возвращаемое функцией значение.  
Использование глобальных переменных  
  
Глобальные переменные видны во всех функциях, где не описаны локальные переменные с теми же именами, поэтому использовать их для передачи данных между функциями очень легко. Тем не менее, это не рекомендуется, поскольку затрудняет отладку программы и препятствует помещению функций в библиотеки общего пользования. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.  
  
Если в теле функции имя глобальной переменной совпадает с именем локальной переменной, то все операции выполняются с локальной переменной. Локальная переменная в этом случае скрывает глобальную переменную. Для доступа к глобальной переменной необходимо применить операцию разрешения области видимости. Для этого перед переменной ставится префикс  :: . Пример:  
  
**#include <iostream.h>  
  
int Turn = 5; //объявление глобальной переменной  
  
int main ()  
  
{int Turn = 70; // объявление локальной переменной  
  
cout << Turn <<’\n’;  //вывод локального значения  
  
cout << ::Turn <<’\n’;  //вывод глобального значения  
  
return 0;  
  
}**  
  
В результате в две строки будет выведено два значения: 5 и 70.  
Использование возвращаемого значения  
  
Механизм возврата из функции в вызвавшую ее функцию реализуется оператором  
  
**return [ выражение ];**  
  
Функция может содержать несколько операторов return, это определяется потребностями алгоритма. Однако, после любого из этих операторов прекращается выполнение функции. Если функция описана как void, выражение не указывается и , кроме того, оператор return можно опускать.  
  
Выражение, указанное после return, неявно преобразуется к типу возвращаемого функцией значения и передается в точку вызова функции. Вместо выражения можно записать переменную или константу.  
  
Примеры:  
  
**int sum(int a, int b){ return (a + b);}  
  
int f1(){return 1;} // правильно  
  
void f2(){return 1;} // неправильно, f2 не должна возвращать значение  
  
double f3{return 1;} // правильно, 1 преобразуется к типу double**

50

Передача массивов в качестве параметров функции.

Когда массив используется в качестве аргумента функции, передается только адрес массива, а не копия всего массива. При вызове функции с именем массива в функцию передается указатель на первый элемент массива. (Надо помнить, что в С имена массивов без индекса - это указатели на первый элемент массива.) Параметр должен иметь тип, совместимый с указателем. Имеется три способа объявления параметра, предназначенного для получения указателя на массив

Параметр должен иметь тип, совместимый с указателем.

Способы. 1) Размерность как отдельный параметр void quart(int, float \*x)

2)оазмерность в фигурных скобках – void quart (int, float x[]);

3)voiddisplay(intnum[10]);

Важно понять, что при использовании массива в качестве аргумента функции происходит передача в функцию его адреса. Это означает, что код внутри функции действует и может изменять настоящее значение массива, используемого при вызове.

51

Указатель на функцию (function pointer) хранит адрес функции. По сути указатель на функцию содержит адрес первого байта в памяти, по которому располагается выполняемый код функции.

Самым распространенным указателем на функцию является ее имя. С помощью имени функции можно вызывать ее и получать результат ее работы.

Но также указатель на функцию мы можем определять в виде отдельной переменной с помощью следующего синтаксиса:

|  |  |
| --- | --- |
| 1 | тип (\*имя\_указателя) (параметры); |

Используем указатель на функцию:

#include <iostream>

**void** hello();

**void** goodbye();

**int** main()

{

**void** (\*message)();

    message=hello;

    message();

    message = goodbye;

    message();

**return** 0;

}

**void** hello()

{

    std::cout << "Hello, World" << std::endl;

}

**void** goodbye()

{

    std::cout << "Good Bye, World" << std::endl;

}