Министерство общего и профессионального образования Свердловской области

ГАПОУ СО «Асбестовский политехникум»

ЯЗЫК ПОГРАММИРОВАНИЯ C++

КОНСПЕКТ ЛЕКЦИЙ

2017 г.

Конспект лекций разработан на основе рабочей программы учебной дисциплины «Основы программирования» для студентов специальности *09.02.03 «Программирование в компьютерных системах»* и преподавателей.

Дисциплина «Основы программирования» является общепрофессиональной, устанавливающей базовый уровень знаний для освоения других общепрофессиональных и специальных дисциплин.

Цель дисциплины и конспектов подготовить студента к базовым основам профессиональной деятельности по разработке и модификации программного продукта.

Характерной особенностью данного пособия является систематизация рассматриваемых вопросов. Особое внимание уделено языку программирования С++. Приведено множество примеров, которые иллюстрируют излагаемый материал. Несмотря на небольшой объём в конспектах изложено много вопросов, которые затронуты менее глубоко во многих других учебных пособиях по языку С++.

Конспект лекций поможет студентам подготовиться к теоретическому этапу экзамена, а в течение семестра готовиться к занятиям по дисциплине «Основы алгоритмизации и программирование» при изучении языка С++.

Необходимость создания таких конспектов связана с тем, что материал рабочей программы разбросан по многим учебным изданиям, и не все учебные издания имеются в библиотеке техникума.

Составитель: Савина О.Н. – преподаватель Асбестовского полтехникума

СОДЕРЖАНИЕ

МОДЕЛИРОВАНИЕ И ФОРМАЛИЗАЦИЯ

Моделирование имеет чрезвычайно важное значение в человеческой деятельности. Без создания модели невозможно решение задачи на компьютере.

Модели позволяют представить в наглядной форме объекты и процессы, недоступные для непосредственного восприятия (очень большие или очень маленькие объекты, очень быстрые или очень медленные процессы и др.). Модели играют чрезвычайно важную роль в проектировании.

Моделирование – это метод познания, состоящий в создании и исследовании моделей.

Каждый объект имеет много различных свойств. В процессе построения модели выделяются наиболее существенные для проводимого исследования свойства.

Модель – это новый объект, который отражает существенные особенности изучаемого объекта, явления или процесса. Любая модель строится и исследуется при определённых допущениях, гипотезах. Модель – результат отображения одной структуры на другую. Отобразив физический объект на математический объект, получим математическую модель физического объекта.

Основные свойства модели:

- ✓ конечность – модель отображает оригинал в конечном числе его отношений;
- ✓ упрощенность – модель отображает только существенные стороны объекта и должна быть проста для исследования;
- ✓ приблизительность – действительность отображается моделью оценочно, или приблизительно;
- ✓ адекватность моделируемой системе – действительность отображается моделью оценочно, или приблизительно;
- ✓ наглядность обозримость основных свойств и отношений;
- ✓ информативность – модель должна содержать достаточную информацию о системе и давать возможность получать новую информацию.

Все модели можно разбить на два класса: предметные (материальные) и информационные.

Предметные модели воспроизводят геометрические, физические и другие свойства объектов в материальной форме (глобус, макеты сооружений и т.д.).

Информационные модели представляют собой объекты и процессы в образной (рисунки, фотографии) или знаковой форме (знаковая система).

Знаковая информационная модель может быть представлена в форме текста, формулы, таблицы. Если знаковая информационная модель строится с помощью формальных языков, то получается формальная информационная модель (математическая, логическая). Язык математики является совокупностью формальных языков: алгебры, геометрии, теории вероятностей, теории множеств и др.

Математическая модель – это система математических соотношений (формул, уравнений, неравенств), отражающих существенные свойства объекта или явления.

Процесс построения информационных моделей с помощью формальных языков называется ***формализацией.***

Типы моделей:

- ✓ модель называется ***статической***, если среди параметров, участвующих в описании модели, нет временного параметра (закон Ньютона F=ma);
- ✓ модель называется ***динамической,*** если среди параметров модели есть временной параметр, т.е. она отображает объект во времени (свободное падение тела $S=gt^2$);
- ✓ модель называется ***дискретной***, если она описывает поведение системы только в дискретные моменты времени ($S=gt^2$, если t=0, 1, 2, 3, . . . 10);
- ✓ модель называется ***непрерывной***, если она описывает поведение системы для всех моментов времени из некоторого промежутка ($S=gt^2$, где $0<t<100$);
- ✓ модель называется ***имитационной***, если она предназначена для испытания или излучения, проигрывания возможных путей развития и поведения объекта с помощью варьирования параметров модели.
- ✓ модель называется ***множественной***, если она представима с помощью некоторых множеств и отношений принадлежности им и между ними;
- ✓ модель называется ***алгоритмической,*** если она описана некоторым алгоритмом, определяющим её функционирование.

В процессе исследования формальных моделей часто производится их визуализация. Для визуализации алгоритмов используются блок-схемы.

Наиболее эффективно математическую модель можно реализовать на компьютере в виде алгоритмической модели. Для этого нужно:

- ✓ выделить предположения, на которых будет основываться математическая модель;
- ✓ определить, что считать исходными данными и результатами;
- ✓ записать математические соотношения, связывающие результаты с исходными данными**.**

ЭТАПЫ РЕШЕНИЯ ЗАДАЧИ

Процесс разработки задачи на компьютере можно разделить на несколько основных этапов.

1 этап - постановка задачи:

- ✓ сбор информации о задаче;
- ✓ формулировка условия задачи;
- ✓ определение конечных целей решения задачи;
- ✓ определение формы выдачи результатов;
- ✓ анализ всех величин, используемых в задаче, т.е. необходимо составить список величин, от которых будет зависеть решение задачи (входные данные) и которые должны получиться (выходные данные), учесть промежуточные данные;
- ✓ описание данных (их типов, диапазонов величин, структуры и т.п.);

На первом этапе разработки задачи строится описательная информационная модель. Такая модель выделяет существенные параметры задачи, а несущественными параметрами пренебрегает.

2 этап - анализ и исследование задачи:

- ✓ анализ существующих аналогов;
- ✓ анализ технических и программных средств;
- ✓ разработка математической модели;
- ✓ разработка структур данных.

На втором этапе создается формализованная модель, т.е. описательная информационная модель записывается с помощью какого-либо формального языка. В такой модели с помощью формул, функций, неравенств фиксируются формальные соотношения между начальными и конечными значениями параметрами задачи, а также накладываются ограничения на допустимые значения этих параметров, т.е создается математическая модель задачи.

Однако далеко не всегда удается найти формулы, явно выражающие искомые величины через исходные данные. В таких случаях используются приближенные математические методы, позволяющие получать результаты с заданной точностью.

3 этап – проектирование.

На третьем этапе необходимо формализованную информационную модель преобразовать в компьютерную модель, т.е. выразить её на понятном для компьютера языке, однако не привязывать к конкретному языку программирования. Необходимо построить алгоритм решения задачи.

Разработка алгоритма:

- ✓ выбор метода проектирования алгоритма;
- ✓ выбор формы записи алгоритма (блок-схема, псевдокод и др.);
- ✓ выбор тестов и метода тестирования;
- ✓ проектирование алгоритма.

Алгоритмическая модель универсальна и не зависит от языка программирования, на котором она в дальнейшем будет реализована.

4 этап - кодирование.

- ✓ выбор языка программирования;
- ✓ уточнение способов организации данных;
- ✓ запись алгоритма на выбранном языке программирования.

5 этап - анализ результатов тестирование и отладка

Выполняется анализ результатов решения задачи и уточнение в случае необходимости математической модели с повтором этанов 2 — 5, перечисленных выше.

Наличие ошибок в только что разработанной программе — это волне нормальное, закономерное явление. Практически невозможно составить реальную (достаточно сложную) программу без ошибок. Нельзя сделать вывод, что программа правильна, лишь на том основании, что она не отвергнута компьютером и выдала результаты. Ведь все, что достигнуто в данном случае, — получение каких-то результатов, необязательно правильных. В программе при этом может оставаться большое число ошибок.

Отладка программы — это процесс поиска и устранения ошибок в программе, производимый по результатам ее прогона на компьютере.

Тестирование программы — это испытание, проверка правильности работы программы в целом либо ее составных частей.

Отладка (от *англ.* debugging — вылавливание «жучков») и тестирование (от *англ.* test — испытание) — это два четко различимых и непохожих друг на друга этапа:

- ✓ при отладке происходит локализация и устранение синтаксических ошибок и явных ошибок кодирования;
- ✓ в процессе тестирования проверяется работоспособность программы, не содержащей явных ошибок.

Тестирование устанавливает факт наличия ошибок, а отладка выясняет причину. Термин «отладка» появился в 1945 г., когда один из первых компьютеров «Марк-1» прекратил работу из-за того, что в его электрические цепи попал мотылек и заблокировал своими останками одно из тысяч реле машины.

Как бы тщательно ни была отлажена программа, решающим этапом, устанавливающим ее пригодность для работы, является контроль программы по результатам ее выполнения на системе тестов. Программу условно можно считать правильной, если её запуск для выбранной системы тестовых исходных данных во всех случаях дает правильные результаты.

Для реализации метода тестов должны быть заранее известны эталонные результаты(ручная трассировка). Вычислять эталонные результаты нужно обязательно до, а не после получения машинных результатов. В противном случае существует опасность невольной подгонки вычисляемых значений под желаемые, полученные ранее на машине.

Тестовые данные должны обеспечить проверку всех возможных условий возникновения ошибок:

- ✓ должна быть испытана каждая ветвь алгоритма;
- ✓ очередной тестовый прогон должен контролировать то, что еще не было проверено на предыдущих прогонах;
- ✓ первый тест должен быть максимально прост, чтобы

проверить, работает ли программа вообще;

- ✓ арифметические операции в тестах должны предельно упрощаться для уменьшения объема вычислений;
- ✓ минимизация вычислений не должна снижать надежности контроля;
- ✓ тестирование должно быть целенаправленным и систематизированным, так как случайный выбор исходных данных приведёт к трудностям в определении ручным способом ожидаемых результатов; кроме того, при случайном выборе тестовых данных могут оказаться непроверенными многие ситуации;
- ✓ усложнение тестовых данных должно происходить постепенно.

6 этап - сопровождение программ

Сопровождение программ — это работы, связанные с обслуживанием программ в процессе их эксплуатации.

Многократное использование разработанной программы для решения задач заданного класса требует проведения дополнительных работ, связанных с доработками программы для решения конкретных задач, проведения дополнительных тестовых просчетов и т.п.

Программа, предназначенная для длительной эксплуатации, должна иметь соответствующую документацию и инструкцию по использованию.

ОСНОВНЫЕ ПОНЯТИЯ СТРКУТУРНОГО ПРОГРАММИРОВАНИЯ. ОСНОВЫ АЛГОРИТМИЗАЦИИ

В процессе разработки программного продукта программист должен пройти определённые этапы решения задачи. На этапе проектирования строится алгоритм будущей программы.

Возникновение термина «алгоритм» связывают с именем великого узбекского математика IX в. Аль Хорезми, который дал определение правил выполнения основных арифметических операций. В европейских странах его имя трансформировалось в слово «алгоритм», а затем уже в «алгоритм».

В дальнейшем этот термин стали использовать для обозначения совокупности правил, определяющих последовательность действий, выполнение которых приведет к достижению поставленной цели.

Алгоритм — это строгая система правил или инструкций для исполнителя, определяющая некоторую последовательность действий, которая после конечного числа шагов приводит к достижению искомого результата.

Исполнитель алгоритма — это абстрактная или реальная (техническая, биологическая или биотехническая) система, способная выполнить действия, предписываемые алгоритмом.

Основные свойства алгоритмов:

- ✓ ***понятность*** — исполнителю алгоритма должна быть известна система команд исполнителя;
- ✓ ***дискретность*** (прерывность, раздельность) — алгоритм должен представлять процесс решения задачи как последовательное выполнение простых (или ранее определенных) команд;
- ✓ ***определенность*** — каждое правило алгоритма должно быть четким и однозначным. Алгоритм должен иметь одно начало и один конец. Благодаря этому свойству выполнение алгоритма носит механический характер и не требует никаких дополнительных указаний или сведений о решаемой задаче;
- ✓ ***результативность*** (или конечность) состоит в том, что алгоритм должен приводить к решению задачи за конечное число шагов;
- ✓ ***массовость*** означает, что алгоритм решения задачи разрабатывается в общем виде, т.е. он должен быть применим для некоторого класса задач, различающихся лишь исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется областью применения алгоритма.

Способы задания (записи) алгоритмов весьма разнообразны:

- ✓ словесно-формульный;
- ✓ графический (блок-схема или структурограмма);
- ✓ псевдокод;
- ✓ на языке программирования высокого уровня.

Рассмотрим способы задания алгоритмов на примере.

Пример: Разработать алгоритм определения номера четверти, в которой лежит точка с координатами X и Y.
I четверть – X≥0 и Y≥0; **II четверть** – X<0 и Y>0**;**
III четверть – X<0 и Y<0**;** **IV четверть** – X>0 и Y<0;

Словесно-формульная запись алгоритма:

1. Начало
2. Ввести координаты точки X и Y.
3. Проверить значение X
   если X≥0, то перейти к п.4, иначе перейти к п.7
4. Проверить значение Y
   если Y≥0, то перейти к п.5, иначе перейти к п.6
5. Присвоить номер четверти N=1, перейти к п.10
6. Присвоить номер четверти N=4, перейти к п.10
7. Проверить значение Y
   если Y≥0, то перейти к п.8, иначе перейти к п.9
8. N=2, перейти к п.10
9. N=4, перейти к п.10
10. Выдать значение N
11. Конец

Графический способ описания алгоритма

Графический способ представления алгоритмов является более компактным и наглядным по сравнению со словесным.

При графическом представлении алгоритм изображается в виде последовательности связанных между собой функциональных блоков. Такое графическое представление называется схемой алгоритма или блок-схемой.

В блок-схеме каждому типу действий (вводу исходных данных, вычислению значений выражений, проверке условий, управлению повторением действий, окончанию обработки и т.п.) соответствует геометрическая фигура, представленная в виде блочного символа. Блочные символы соединяются линиями переходов, определяющими очередность выполнения действий. В таблице приведены обозначения, наиболее часто используемые в блок-схемах.

Блок «процесс» применяется для обозначения действия или последовательности действий, изменяющих значение, форму представления или размещения данных. Для улучшения наглядности схемы несколько отдельных блоков обработки можно объединять в один блок. Представление отдельных операций достаточно свободно.

Блок «решение» используется для обозначения переходов управления по условию. В каждом блоке «решение» должны быть указаны условие (сравнение), которое определяет ход вычислительного процесса.

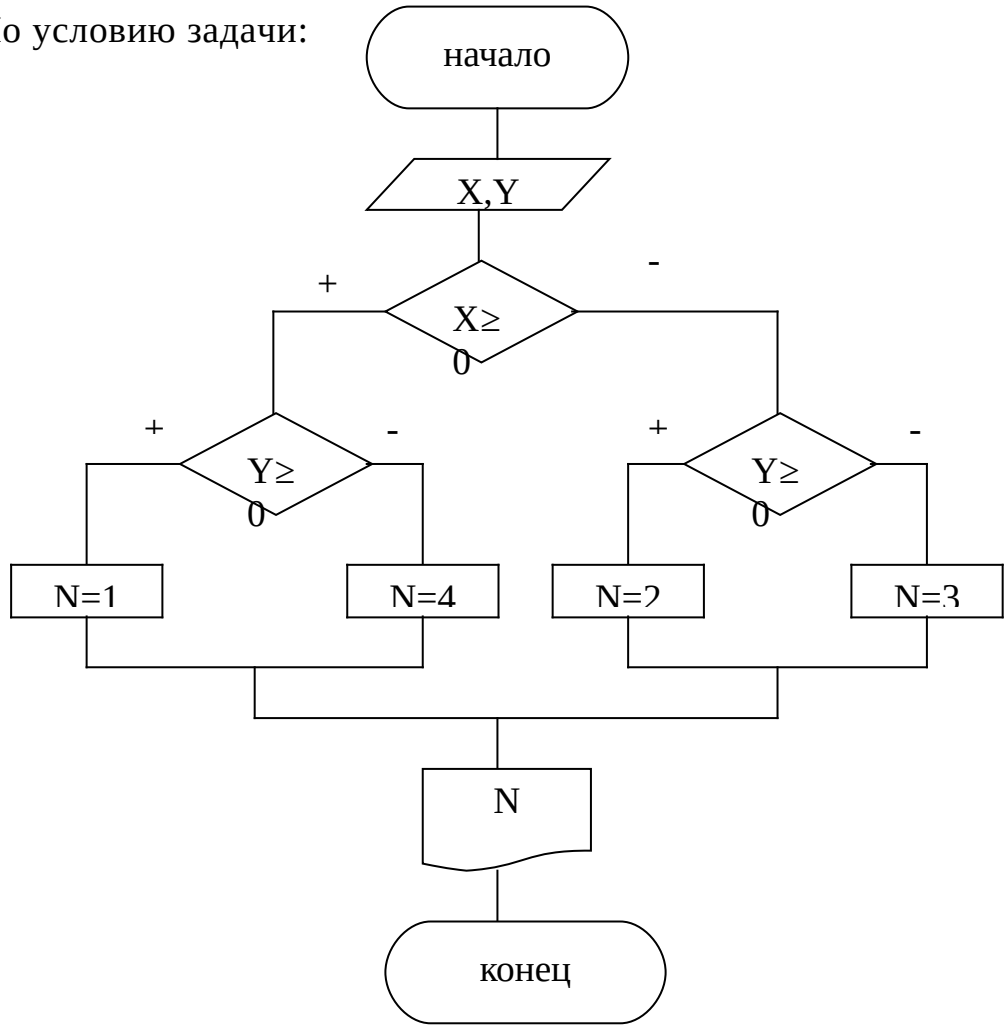
Блок «модификация» (видоизменение, преобразование) используется для организации циклических конструкций. Внутри блока записывается

параметр цикла, для которого указывается его начальное значение, конечное значение и шаг изменения значения параметра цикла для каждого повторения.

Блок «предопределенный процесс» используется для указания обращений к вспомогательным алгоритмам, существующим автономно в виде некоторых самостоятельных модулей, и для обращения к библиотечным подпрограммам.

Наименование	**Обозначение**	**Пояснение**
Процесс	A= x+y	Действие или последовательность действий
Решение	+ x < y -	Проверка условий
Ввод-вывод	A, B	Ввод-вывод в общем виде
Модификация	I =1(1)10	Начало цикла
Пуск-остановка	начало	Начало, конец алгоритма, вход и выход в подпрограмму
Предопределённый процесс	Расчет суммы	Вычисление по подпрограмме
Документ	A, B, C	Вывод результатов на печать
Переходы	**1** **2**	1 – переходы по блок-схеме внутри страницы; 2 – переход с одной стр. на другую

По условию задачи:



Третий способ записи алгоритмов – псевдокод.

Псевдокод – это искусственный и неформальный язык, который помогает программисту разрабатывать алгоритмы, подобен разговорному языку.

АЛГ определение номера четверти
ДЕЙСТВ. X,Y;
ЦЕЛЫЕ N;
АРГ. X,Y; РЕЗ. N;
НАЧ.
ВВОД X,Y;
ЕСЛИ X≥0 **ТО**
ЕСЛИ Y≥0 **ТО** N=1
ИНАЧЕ N=4
ИНАЧЕ ЕСЛИ Y≥0 **ТО** N=2
ИНАЧЕ N=3;
ВЫВОД N;
КОН.

Особенность: служебные слова и команды записываются в сокращенном виде, выделяются шрифтом или подчёркиваются

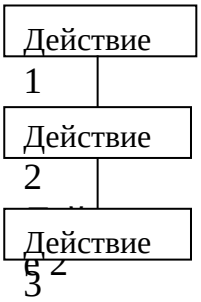
БАЗОВЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ

Структурное программирование до настоящего времени остаётся основой в технологии программирования. Соблюдение его принципов позволяет программисту быстро научиться писать ясные, безошибочные, надёжные программы.

В основе структурного программирования лежит теорема, которая была строго доказана в теории программирования. Суть её в том, что алгоритм для решения любой логической задачи можно составить только из структур ***«следование, ветвление, цикл».*** Их называют базовыми алгоритмическими структурами.

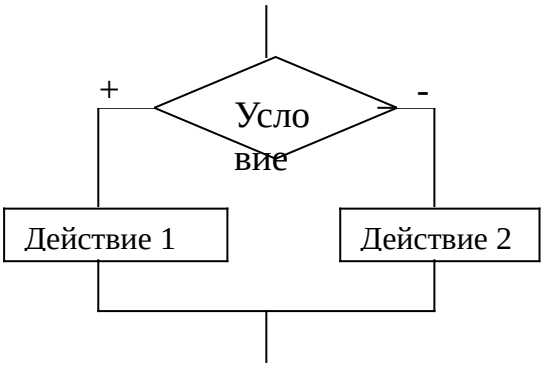
Рассмотрим базовые алгоритмические структуры.

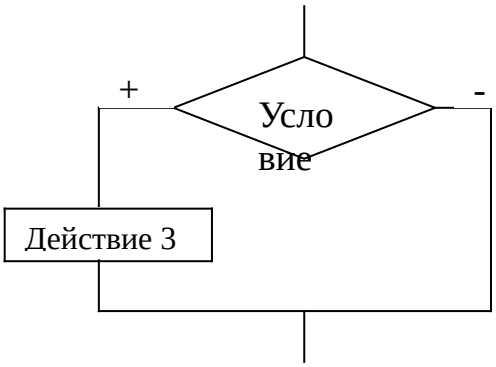
Следование – это линейная последовательность действий. Каждый блок может содержать в себе как простую команду, так и сложную структуру, но обязательно должен иметь один вход и один выход.



Ветвление – алгоритмическая альтернатива. Управление передается одному из двух блоков в зависимости от истинности или ложности условия. Затем происходит выход на общее продолжение (конец ветвления).

Две разновидности ветвления: полная и неполная формы ветвления.

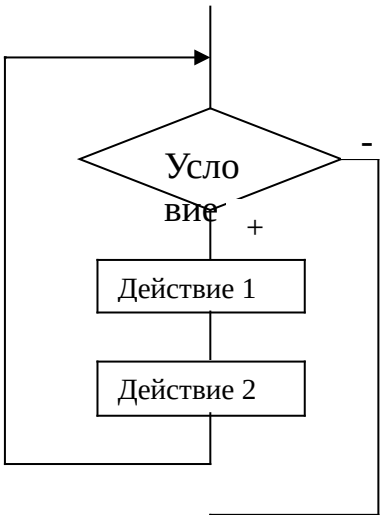


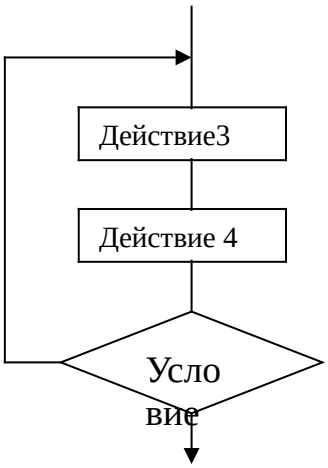


В полной форме ветвления:
если *условие* истинно, **то** выполняется *действие 1*, **иначе** *действие 2 (условие ложно)* конец ветвления (к.в)

В неполной форме ветвления:
если *условие* истинно, **то** выполняется *действие 3* конец ветвления

Цикл – повторение некоторой группы действий по условию (итерационные циклы). Различают два типа цикла: цикл с предусловием (цикл-пока) и цикл с постусловием (цикл-до).





Цикл с предусловием:

пока *условие* истинно, **повторять** *тело цикла* (это *действие 1 и действие 2*).

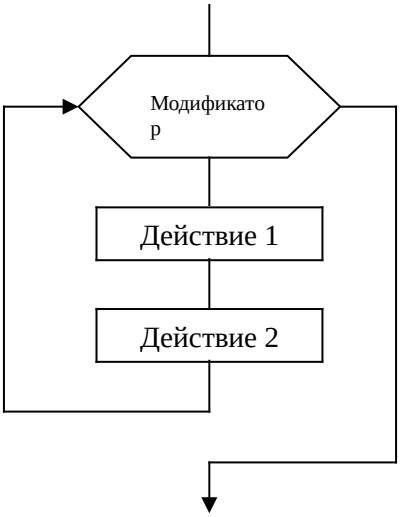
Если условие ложно, то конец цикла или тело цикла не выполнится ни разу.

Цикл с постусловием:

повторять *тело цикла (действие 3 и действие 4*) **до тех пор пока** *условие* ложно (это для Паскаля).

Повторение кончается, когда условие станет истинным. Если условие истинно, то тело цикла выполнится хотя бы один раз.

Третья разновидность циклов – арифметический цикл или цикл с параметром (счётчиком).



Модификатор: параметр цикла изменяется от начального значения до конечного с определённым шагом, записывается следующим образом:
п.ц. = нач.знач.(шаг)конеч.знач

Тело цикла (действие 1 и действие 2) выполняется столько раз, сколько значений принимает параметр цикла в заданных пределах.

Теоретически необходимым и достаточным является лишь первый тип цикла – цикл с предусловием. Любой циклический алгоритм можно построить с его помощью. Это более общий вариант цикла, чем цикл с постусловием.

Иногда в литературе структурное программирование называют программированием без GOTO. Действительно, при таком подходе нет места безусловному переходу. Неоправданное использование в программах оператора GOTO лишает ее структурности, а значит, всех связанных с этим положительных свойств: прозрачности и надежности алгоритма. Хотя во всех процедурных языках программирования этот оператор присутствует, однако, придерживаясь структурного подхода, его употребления следует избегать.

Сложный алгоритм состоит из соединенных между собой базовых структур. Соединяться эти структуры могут двумя способами: *последовательным* и *вложенным.* Если блок, составляющий тело цикла, сам является циклической структурой, то, значит, имеют место вложенные циклы. В свою очередь, внутренний цикл может иметь внутри себя еще один цикл и т.д. В связи с этим вводится представление о глубине вложенности циклов. Точно так же и ветвления могут быть вложенными друг в друга.

Структурный подход требует соблюдения стандарта в изображении блок-схем алгоритмов. Чертить их нужно так, как это делалось во всех приведенных примерах. Каждая базовая структура должна иметь один вход и один выход. Нестандартно изображенная блок-схема плохо читается, теряется наглядность алгоритма.

Языки программирования Паскаль и С++ называют языками структурного программирования. В них есть все необходимые управляющие конструкции для структурного построения программы. Наглядность такому построению придаёт структуризация внешнего вида текста программы. Основной используемый для этого приём – сдвиги строк, которые должны подчиняться следующим правилам:

- ✓ конструкции одного уровня вложенности записываются на одном вертикальном уровне, начинаются с одной позиции в строке;
- ✓ вложенная конструкция записывается смещенной по строке на несколько позиций вправо относительно внешней для неё конструкции.

Структурная методика алгоритмизации – это не только форма описания алгоритма, но это ещё и способ мышления программиста. Создавая алгоритм, нужно стремиться составлять его из стандартных структур. Если использовать строительную аналогию, можно сказать, структурная методика построения алгоритма подобна сборке здания из стандартных секций в отличие от складывания по кирпичику.

Еще одним важнейшим технологическим приемом структурного программирования является *декомпозиция решаемой задачи на подзадачи* — более простые с точки зрения программирования части исходной задачи. Алгоритмы решения таких подзадач называются *вспомогательными алгоритмами.* В связи с этим возможны два пути в построении алгоритма:

- ✓ «сверху вниз»: сначала строится основной алгоритм, затем вспомогательные алгоритмы;
- ✓ «снизу вверх»: сначала составляются вспомогательные алгоритмы, затем основной.

Первый подход еще называют методом ***последовательной детализации***, второй **- *сборочным*** методом.

Сборочный метод предполагает накопление и использование библиотек вспомогательных алгоритмов, реализованных в языках программирования в виде подпрограмм, процедур, функций. При последовательной детализации сначала строится основной алгоритм, а затем в него вносятся обращения к вспомогательным алгоритмам первого уровня. После этого составляются вспомогательные алгоритмы первого уровня, в которых могут присутствовать обращения к вспомогательным алгоритмам второго уровня, и т.д.

Вспомогательные алгоритмы самого нижнего уровня состоят только из простых команд.

Метод последовательной детализации применяется в любом конструировании сложных объектов. Это естественная логическая последовательность мышления конструктора: постепенное углубление в детали. В нашем случае речь идет тоже о конструировании, но только не технических устройств, а алгоритмов. Достаточно сложный алгоритм другим способом построить практически невозможно.

Методика последовательной детализации позволяет организовать работу коллектива программистов над сложным проектом. Например, руководитель группы строит основной алгоритм, а разработку вспомогательных алгоритмов и написание соответствующих подпрограмм поручает своим сотрудникам. Участники группы должны лишь договориться об интерфейсе (т.е. взаимосвязи) между разрабатываемыми программными модулями, а внутренняя организация программы — личное дело программиста.

ЗНАКОМСТВО С ЯЗЫКОМ С++

Язык программирования С был разработан в период с 1969 по 1973 годы американскими программистами Кеном Томпсоном и Денисом Ритчи. Язык С разрабатывался как язык системного программирования и предназначался для кодирования операционной системы Unix, которая была бы переносима на различные аппаратные платформы.

Бьерн Страуструп начал работать над разработкой С++ в 1979 году. Идея создания нового языка берёт начало от опыта программирования Страуструпа для диссертации. Он обнаружил, что язык моделирования Симула имеет такие возможности, которые были бы очень полезны для разработки большого программного обеспечения, но работает слишком медленно. В то же время язык BCPL достаточно быстр, но слишком близок к языкам низкого уровня и не подходит для разработки большого программного обеспечения.

Страуструп решил дополнить язык С возможностями, имеющимися в языке Симула. Язык Си, будучи базовым языком системы UNIX, является быстрым, многофункциональным и переносимым. Страуструп добавил к нему возможность работы с классами и объектами. В результате, практические задачи моделирования оказались доступными для решения как с точки зрения времени разработки так и с точки зрения времени вычислений.

В 1983 г. в С++ были добавлены новые возможности, такие как виртуальные функции, перегрузка функций и операторов, константы, новый стиль комментариев (//). В 1985 г. вышло первое издание «Языка программирования Си++», обеспечивающее первое описание этого языка, что было чрезвычайно важно из-за отсутствия официального стандарта. В 1989 г. состоялся выход Си++ версии 2.0. Его новые возможности включали множественное наследование, абстрактные классы, статические функции-члены. Последние обновления включали шаблоны исключения пространства имён, булевский тип.

Новые возможности языка С++ по сравнению с языком С

- поддержка объектно-ориентированного программирования через классы;
- поддержка обобщённого программирования через шаблоны;
- дополнения к стандартной библиотеке;
- дополнительные типы данных;
- исключения;
- встраиваемые функции;
- перегрузка операторов;

Достоинства языка С++

- Масштабируемость. На языке С++ разрабатывают программы для самых различных платформ и систем.
- Возможность работы на низком уровне с памятью, адресами, портами.

Недостатки языка С++

- Плохая поддержка модульности. Подключение интерфейса внешнего модуля через препроцессорную вставку заголовочного файла (#include) серьезно замедляет компиляцию, при подключении большого количества модулей.

- Язык C++ является сложным для изучения.
- Препроцессор C++ (унаследованный от C) очень примитивен. Это приводит к тому, что с его помощью нельзя осуществлять некоторые задачи метапрограммирования

Критика языка C++

- C++ унаследовал многие проблемы языка C:
  - Операция присваивания обозначается как =, а операция сравнения как ==. Их легко спутать, и такая конструкция будет синтаксически правильной, но приведёт к труднонаходимому багу. Особенно часто это происходит в операторах **if** и **while**.
  - Операции присваивания (=), инкрементации (++), декрементации (--) и другие возвращают значение. В сочетании с обилием операций это позволяет, но не обязывает, программиста создавать трудночитаемые выражения.

Дизайн C++

В книге «Дизайн и развитие C++» Бьярне Страуструп описывает некоторые правила, которые он использовал при проектировании C++. Вот некоторые из этих правил C++:

- разработан как универсальный язык со статическими типами данных, эффективностью и переносимостью языка C.
- разработан так, чтобы непосредственно и всесторонне поддерживать множество стилей программирования (процедурное программирование объектно-ориентированное программирование и обобщённое программирование).
- разработан так, чтобы давать программисту свободу выбора, даже если это даёт ему возможность выбирать неправильно.
- разработан так, чтобы максимально сохранить совместимость с C, тем самым делая возможным лёгкий переход от программирования с C++ на C

Будущее развитие

C++ продолжает развиваться, чтобы отвечать современным требованиям. Одна из групп, занимающихся языком C++ в его современном виде совершенствует возможности языка путём добавления в него особенностей метапрограммирования.

СТРУКТУРА ЯЗЫКА С++

ЭЛЕМЕНТЫ ЯЗЫКА СИ++

Алфавит языка

В алфавит языка С++ входят:

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

К специальным символам относится также пробел.

В языке С++ используются ***лексемы***, которые формируются из символов алфавита. К лексемам относятся:

- идентификаторы;
- ключевые слова или служебные слова;
- константы;
- знаки операций;
- разделители;
- комментарии.

Идентификаторы – это имена переменных, функций, меток, используемых в программе.

Правила пользования идентификаторов:

- идентификатором может быть любая последовательность строчных и прописных латинских букв, цифр и знака подчеркивания, начинающаяся не с цифры;
- максимальная длина идентификатора 32 символа (ограничения на длину идентификатора могут различаться в разных реализациях языка; компиляторы фирмы Borland позволяют использовать до 32 первых символов имени; в некоторых других реализациях допускаются идентификаторы длиной не более 8 символов);
- идентификаторы в С++ чувствительны к регистру (прописные/строчные буквы). Это значит, что, например, `flag, FLAG, Flag, FlAg` – разные идентификаторы;
- нельзя использовать 2 символа подчеркивания подряд;
- нельзя использовать ключевые слова в качестве идентификатора.

Ключевые (служебные) слова – это идентификаторы, зарезервированные в языке для специального использования: обозначение типов данных, операторов языка, классов памяти. Они не могут быть использованы как свободно выбираемые имена. Полный список служебных слов зависит от реализации языка, т. е. различается для разных компиляторов. Однако существует неименное ядро, которое определено стандартом С++. Вот его список:

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

Ограничение - ключевые слова распознаются только в том случае, когда они написаны **строчными буквами**.

Константы – это лексемы, предоставляющие изображения фиксированного числового, строкового и символьного значения.
Константы бывают:

- целые числа;
- вещественные числа;
- символьные;
- строковые;
- перечисляемые;
- типизированные константы;
- неарифметическая константа.

Целые константы могут быть десятичными, восьмеричными и шестнадцатеричными.

Десятичная целая константа определена как последовательность десятичных цифр, начинающаяся не с нуля, если это не число нуль.

Восьмеричные целые константы начинаются всегда с нуля: 016 имеет десятичное значение 14. Если в записи восьмеричной константы встретится недопустимая цифра 8 или 9, то это воспринимается как ошибка.

Последовательность шестнадцатеричных цифр, которой предшествует 0x, считается шестнадцатеричной константой. В шестнадцатеричные цифры кроме десятичных входят латинские буквы от a (или A) до f (или F). Таким образом, 0x16 имеет десятичное значение 22, а OxF - десятичное значение 15.

Вещественные константы, т.е. константы с плавающей точкой, даже не отличаясь от целых констант по значению, имеют другую форму внутреннего представления в ЭВМ. Эта форма требует использования арифметики с плавающей точкой при операциях с такими константами. Поэтому компилятор должен уметь распознавать вещественные константы. Распознает он их по внешним признакам. Константа с плавающей точкой может включать следующие шесть частей: целая

часть (десятичная целая константа); десятичная точка; дробная часть (десятичная целая константа); признак (символ) экспоненты e или E; показатель десятичной степени (десятичная целая константа, возможно со знаком); суффикс F (или f) либо L (или l).

В записях вещественных констант могут опускаться: целая или дробная часть (но не одновременно); десятичная точка или признак экспоненты с показателем степени (но не одновременно); суффикс. Примеры: **66.0; .0; .12; 3.14159F; 1.12e-2; 2E+6L; 2.7l.**

При отсутствии суффиксов F(f) или L(l) вещественные константы [имеют форму внутреннего представления, которой в языке C++ соответствует тип данных double. Добавив суффикс f или F, константе придают тип float. Константа имеет тип long double, если в ее представлении используется суффикс L или 1.

Символьные константы – это один или два символа, заключенные в апострофы. Односимвольная константа имеет стандартный тип **char.** Примеры констант: 'z', ‘+’, ‘A’.

В C++ есть специальные символы, начинающиеся со знака '\'.

‘\n’ – перевод строки;
‘\t’ – табуляция горизонтальная;
‘\v’ – табуляция вертикальная;
‘\r’ – возврат каретки;
‘\0’ – нулевой символ (конец строки).

Строка или строковая константа, иногда называемая литерной строкой, определяется как последовательность символов, заключенная в кавычки (не в апострофы): ”Изучаем новый язык программирования”. Все символы строки размещаются подряд. Каждый символ занимает ровно 1 байт. В конце записи строковой константы компилятор помещает символ ‘\0’ – конец строки. Поэтому запись 'F' содержит 1 байт, а запись ”F” – 2 байта.

Перечислимые константы (или константы перечисления, иначе константы перечислимого типа) вводятся с помощью служебного слова **enum**. По существу это обычные целочисленные константы (типа **int**), которым приписаны уникальные и удобные для использования обозначения. В качестве обозначений выбираются произвольные идентификаторы, не совпадающие со служебными словами и именами других объектов программы. Пример:

enum color{black, red, blue,gren};

Типизированные константы. Используются как переменные, значения которых не может быть изменено после инициализации. Объявляются с помощью ключевого слова **const. Пример: const int a = 50;**

Неарифметическая константа **-** нулевой указатель **NULL**

Знаки операций обеспечивают формирование и последующее вычисление выражений. Выражение есть правило для получения значения. Один и тот же знак операции может употребляться в различных выражениях и по-разному интерпретироваться в зависимости от контекста. Для изображения операций в большинстве случаев используется несколько символов.

Операции бывают унарные (для оного операнда) и бинарные (для двух операндов).

Унарные операции

& операция получения адреса операнда;

* операция обращения по адресу, т.е. раскрытия ссылки, иначе операция разыменования (доступа по адресу к значению того объекта, на который указывает операнд). Операндом должен быть адрес;

! логическое отрицание (НЕ) значения операнда; применяется к скалярным операндам; целочисленный результат 0 (если операнд ненулевой, т.е. истинный) или 1 (если операнд нулевой, т.е. ложный). В качестве логических значений в языке С++ используют целые числа: 0 - ложь и не нуль (! 0) - истина. Отрицанием любого ненулевого числа будет 0, а отрицанием нуля будет 1. Таким образом: ! 1 равно 0; ! 2 равно 0; ! (-5) равно 0; !0 равно 1;

++ увеличение на единицу (инкремент или автоувеличение);

префиксная операция - увеличение значения операнда на 1 до его использования;

постфиксная операция - увеличение значения операнда на 1 после его использования.

-- уменьшение на единицу (декремент или автоуменьшение);

префиксная операция - уменьшение значения операнда на 1 до его использования;

постфиксная операция - уменьшение значения операнда на 1 после его использования;

sizeof операция вычисления размера (в байтах) для объекта того типа, который имеет операнд **sizeof (тип)**.

Бинарные операции. Эти операции делятся на следующие группы:

- аддитивные;
- мультипликативные;
- операции отношений;
- логические;
- присваивания;

Аддитивные операции: сложение арифметических операндов и вычитание

арифметических операнде или указателей.

Мультипликативные операции:

* умножение операндов арифметического типа;

/ деление операндов арифметического типа. Операция стандартна. При целочисленных операндах абсолютное значение результата округляется до целого. Например, 20/3 равно 6, -20/3 равняется -6;

% получение остатка от деления целочисленных операндов (целение по модулю). При неотрицательных операндах остаток положительный 13%4 = 1, (-13) % 4 = -1, 13% (-4) = +1, а (-13)%(-4) =-1.

Операции отношения (сравнения):

< меньше, чем;

> больше, чем;
<= меньше или равно;
>= больше или равно;
== равно;
!= не равно;

Логические операции

&& конъюнкция (И) арифметических операндов или отношений. Целочисленный результат 0(ложь) или 1(истина);

|| дизъюнкция (ИЛИ) арифметических операндов или отношений. Целочисленный результат 0(ложь) или 1(истина);

! логическое отрицание (НЕ) значения операнда; применяется к скалярным операндам; целочисленный результат 0 (если операнд ненулевой, т.е. истинный) или 1 (если операнд нулевой, т.е. ложный).

Операции присваивания

= присвоить значение выражения-операнда из правой части операнду левой части: P = 10.3 – 2*x;

*= присвоить операнду левой части произведение значений обоих операндов: P *= 2 эквивалентно P = P * 2;

/= присвоить операнду левой части частное от деления значения левого операнда на значение правого:

P /= 2.2 – d эквивалентно P = P / (2.2 - d);

%= присвоить операнду левой части остаток от деления целочисленного значения левого операнда на целочисленное значение правого операнда: N % = 3 эквивалентно N = N % 3;

+= присвоить операнду левой части сумму значений обоих операндов: A += в эквивалентно A = A + B;

-= присвоить операнду левой части разность значений левого и правого операндов: x -= 4.3 - z эквивалентно x = x - (4.3 - z);

Разделители

Квадратные скобки ` [ ] ` ограничивают индексы одно- и многомерных массивов и индексированных элементов.

Круглые скобки ` ( ) `:

- выделяют условные выражения (в операторе ″если″);
- входят как обязательные элементы в определение, описание (в прототип) и вызов любой функции, где выделяют соответственно список формальных параметров и список фактических параметров (аргументов);
- группируют выражения, изменяя естественную последовательность выполнения операций;
- входят как обязательные элементы в операторы циклов;
- необходимы при явном преобразовании типа.

Фигурные скобки ` { } ` обозначают соответственно начало и конец составного оператора в условном операторе; используются для выделения списка

компонентов в определениях типов структур, объединений, классов; используются при инициализации массивов и структур при их определении.

Запятая `,` разделяет элементы списков.

- Во-первых, это списки начальных значений, присваиваемых индексированным элементам массивов и компонентам структур при их инициализации.
- Другой пример списков – списки формальных и фактических параметров и их спецификаций в функциях.
- Третье использование запятой как разделителя – в заголовке оператора цикла **for.**
- Запятая как разделитель используется также в описаниях и определениях объектов (переменных) одного типа.

Точка с запятой `;` завершает каждый оператор, каждое определение (кроме определения функции) и каждое описание. Любое допустимое выражение, за которым следует ' ; ', воспринимается как оператор. Это справедливо и для пустого выражения, т.е. отдельный символ "точка с запятой" считается пустым оператором.

Двоеточие `:` служит для отделения (соединения) метки и помечаемого ею оператора:

Многоточие - это три точки '...' без пробелов между ними. Оно используется для обозначения переменного числа параметров у функции при ее определении и описании (при задании ее прототипа).

Звездочка `*`, как уже упоминалось, используется в качестве знака операции умножения и знака операции разыменования (получения значения через указатель). В описаниях и определениях звездочка означает, что описывается указатель на значение использованного в объявлении типа.

Знак `=`, как уже упоминалось, является обозначением операции присваивания. Кроме того, в определении он отделяет описание объекта от списка его инициализации:

Символ `#` (знак номера или диеза в музыке) используется для обозначения директив (команд) препроцессора. Если этот символ является первым отличным от пробела символом в строке программы, то строка воспринимается как директива препроцессора.

Символ `&` играет роль разделителя при определении переменных типа ссылки.

Операции выбора компонентов структурированного объекта:

(точка) прямой выбор (выделение) компонента структурированного объекта, например объединения. Формат применения операции:

имя структурируемого объекта . имя компонента

-> косвенный выбор (выделение) компонента структурированного объекта, адресуемого указателем.

Комментарии. В С++ в качестве ограничителей комментариев могут использоваться как пары символов /* и */, принятые в языке, так и символы //, используемые только в С++. Признаком конца такого комментария является невидимый символ перехода на новую строку. Примеры:

```
/* Это комментарий, допустимый в Си и C++ */
// Это строчный комментарий, используемый только в C++.
```

ТИПЫ ДАННЫХ С++

Переменная – ячейка памяти. Назначение типа определяет размер этой ячейки, какие операции можно выполнять с данной переменной, допустимые значения, которые может принимать переменная.

Классификация типов данных:

- Простые (скалярные) – целые, вещественные, логические (булевские).
- Структурированные – массивы, структуры, объединения, классы.
- Адресные – указатели, ссылки.

ПРОСТЫЕ ТИПЫ ДАННЫХ

Целый тип

char – целый символьный тип;
short int – короткий целый тип;
int – целый тип;
long int – длинный целый тип.

Каждый из целочисленных типов может быть определён как знаковый **(signed)** или беззнаковый **(unsigned**) тип. При отсутствие ключевого слова, по умолчанию принимается знаковый тип.

Вещественный тип

float – вещественный тип одинарной точности;
doudle – вещественный тип двойной точности;
long double – вещественный тип расширенной точности;

Логический тип

bool – логический тип данных, принимает значение «истина» (true или 1), или «ложь» (false или 0).

Тип данных	**Размер, байт**	**Диапазон значений**
char	1	-128… +128
int	2/4	зависит от системы
unsigned char	1	0… 255
unsigned int	2/4	зависит от системы
short int	2	-32768… 32767
unsignets hort	2	0… 65535
long int	4	-2147483648… 2147483648
unsigned long int	4	0… 4294967295
float	4	±(3.4E-38… 3.4E+38)
double	8	±(1.7E-308… 1.7E+308)
long double	10	±(3.4E-4932… 1.1E+4932)

Анализируя данные таблицы, можно сделать следующие выводы:

- если не указан базовый тип, то по умолчанию подразумевается `int`;
- если не указан модификатор знаков, то по умолчанию подразумевается `signed`;

- с базовым типом `float` модификаторы не употребляются;
- модификатор `short` применим только к базовому типу `int`.

ПРИВИДЕНИЕ ТИПОВ (ЯВНОЕ И НЕЯВНОЕ)

Приведением типов называется преобразование значения одного типа к значению другого типа, при этом возможно как сохранение величины этого значения, так и изменение этой величины. Приведение типов также называют преобразованием типов.

При деление двух переменных целого типа результат получится целый. Для получения вещественного результата, необходимо явное преобразование типа.

int a=5; int b=2; float t;
t=a/b (при этом t будет равняться 2)
t= (float)a/b (при этом t будет ровняться 2,5).

На момент вычисления изменили тип переменной **a** – это и есть ***явное приведение*** типов. Приведённый тип указывается в круглых скобках перед переменной.

Неявное преобразование типов используется когда в бинарной операции операнды имеют различные типы. В этом случае компилятор выполняет преобразование типов автоматически, т.е все они приводятся к типу с наибольшим диапазоном значений.

int a=5; float b=2; float t;
t=a/b (результат = 2,5)

При неявном преобразование типов выполняется правило переводов низших типов в высшие для точности представления данных и их непротиворечивости (низший тип – **char,** высший **- long double;** см.таблицу).

СИНТАКСИС ОПИСАНИЯ ПЕРЕМЕННЫХ В ПРОГРАММАХ

Переменная или объект – это именованная область памяти, в которой хранятся данные. Для именования переменных используются идентификаторы. Прежде чем переменная будет использована в программе, она должна быть объявлена. Объявление переменной выглядит следующим образом:

```
имя_типа список_переменных;
```

Примеры описаний:

```
char symbol, cc;
int nomber, row;
float x, X, cc3 ;
double e, b4;
long double max_num;
```

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

```
тип имя_переменной = начальное_значение
```

Например:

```
float pi=3.14159, c=1.23;
unsigned int year=2000;
```

СИНТАКСИС ОПИСАНИЯ КОНСТАНТ В ПРОГРАММАХ

Именованные константы (константные переменные) - это фиксированные значения данных, которые не могут изменяться.

Употребляемое для их определения служебное слово **const** принято называть ***квалификатором доступа.*** Квалификатор **const** указывает на то, что данная величина не может изменяться в течение всего времени работы программы. В частности, она не может располагаться в левой части оператора присваивания. Примеры описания константных переменных:

```
const float pi=3.14159;
const int iMIN=1, iMAX=1000;
```

Еще одной возможностью ввести именованную константу является использование препроцессорной директивы **#define** в следующем формате:

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

Например:

```
#define iMIN 1
#define iMAX 1000
```

Тип констант явно не указывается и определяется по форме записи. В конце директивы не ставится точка с запятой. На стадии препроцессорной обработки указанные имена заменяются на соответствующие значения. Например, если в программе присутствует оператор

```
X=iMAX-iMIN
```

то в результате препроцессорной обработки он примет вид: `X=1000-1`

При этом идентификаторы `iMAX` и `iMIN` не требуют описания внутри программы.

СТРУКТУРА ПРОГРАММЫ НА ЯЗЫКЕ С++

Состоит из следующих частей:

1. В первой строке записывает комментарий, поясняющий условие решаемой задачи.
2. Во второй строке помещается команда (директива) препроцессора, обеспечивающая включение в программу средств связи со стандартными потоками ввода и вывода данных. Указанные средства находятся в файле с именем ***iostream.h –*** заголовочный файл.

   Стандартный поток ***cin*** обеспечивает считывание символов с клавиатуры и преобразование их в соответствующие числовые значения переменных.

   Стандартный поток вывод ***cout*** обеспечивает вывод данных на экран дисплея.

   #include< iostream.h>

   # - директива препроцессора.

   Препроцессор – это программа, действующая как фильтр на этапе компиляции. Перед тем как попасть на вход компилятора, исходная программа проходит через препроцессор.

   # include <заголовочный файл> (файлы заголовков)

Файлы заголовков в директиве include заканчиваются на .h. Угловые скобки <> указывают препроцессору, что этот файл ищется в стандартном каталоге подключаемых файлов.

Заголовочные файлы:

math.h – подключение математических функций;

string.h - подключение строковых функций.

stdio.h – функции форматированного ввода-вывода.

conio.h – работа с консолью (экраном и клавиатурой).

ctype.h – функции преобразования символов.

Примечание: в конце директивы ; не ставится.

3. Объявление пользовательских констант и глобальных переменных
   # define PI 3.14

4. Объявление функции и/или их прототипов.

5. Каждая программа на языке C++ должна иметь в своем составе функцию с именем main (), с которой начинается выполнение программы. Поэтому эта функция называется главной и имеет соответствующее фиксированное имя. По умолчанию функции main имеет тип int. Это означает, что данная функция возвращает в место вызова целочисленное значение. Местом вызова главной функции является среда выполнения или среда операционной системы. Принято соглашение, что любая программа при аварийном завершении должна возвращать в операционную систему ненулевое значение. При правильном выполнении программы передаётся нулевой результат. В конце функции main() всегда стоит оператор return 0.

```
main ()
{тело функции (основная программа);
return 0;
}
```

Правила объявления функции main().

а) После заголовка функции никаких знаков препинания не ставятся.

в) Тело функции включает

- объявление локальных переменных и констант и их инициализацию (присвоение начальных значений);
- ввод исходных данных (диалог с пользователями);
- обработку – обращение к функциям вычисления выражений;
- выполнение операторов;
- вывод результата;
- возвращение кода завершения главной функции (return 0;)

Общие принципы, позволяющие написать синтаксически правильную программу, таковы:

- прежде чем использовать функции, переменные, типы данных, следует объявить их или подключить файлы с их объявлениями (сделать известными компилятору);
- чтобы вызов функции мог быть выполнен, функция должна быть определена, т.е. описаны действия, которые она осуществляет;

- в любой последовательности действий нужно стремиться к триаде: инициализация (ввод), обработка, возвращение значения (вывод).

Правила оформления текста программы, направленные на облегчение понимания смысла и повышение наглядности, таковы:

- разделять логические части программы пустыми строками;
- разделять операнды и операции пробелами;
- для каждой фигурной скобки отводить отдельную строку;
- в каждой строке должно быть, как правило, не более одного оператора;
- ограничивать длину строки 60-70 символами;
- отступами слева отражать вложенность операторов и блоков;
- длинные операторы располагать в нескольких строках;
- проводить алгоритмизацию так, чтобы определение одной функции занимало, как правило, не более одного экрана текста;
- стремиться использовать типовые заготовки фрагментов программ, включая и типовую структуру блока и программы в целом.

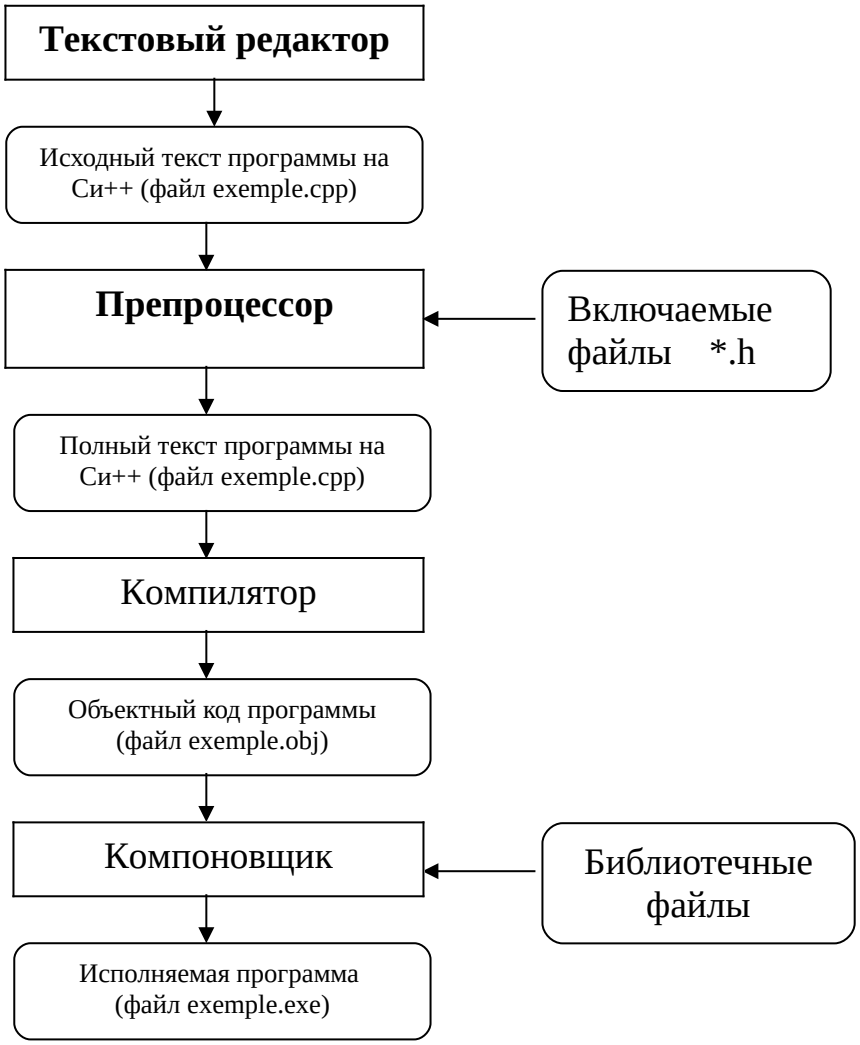
Пример1: вычисление площади круга

```
// вычисление площади круга
#include<iostream.h>
#define PI 3.14
main()
{  float r, pl;
   cout<<”Ввести радиус”<<’\n’;
   cin>>r;
   pl=r*r*PI;  // вычисление площади по формуле
   cout<<”Площадь круга = ”<<pl<<’\n’;
   return 0;
 }
```

Пример 2: нахождение суммы цифр трёхзначного числа

```
// вычисление суммы
#include<iostream.h>
main()
{  int n, a,b, sum=0;
   cout<<”Ввести число”<<’\n’;
   cin>>n;
   a = n % 10;   //  выделяем последнюю цифру числа
   n = n / 10;    //  отбрасываем последнюю цифру числа
   b = n % 10;   //  выделяем вторую с конца цифру числа
   n = n / 10;    //  отбрасываем вторую цифру числа;
                  //    n – первая цифра числа
   sum = a + b + n;
   cout<<”Сумма = ”<<sum<<’\n’;
   return 0;
 }
```

ЭТАПЫ РАБОТЫ С ПРОГРАММОЙ НА С++ в системе программирования (рисунок 1 – прямоугольниками отображены системные программы, а блоки с овальной формой обозначают файлы на входе и на выходе этих программ).

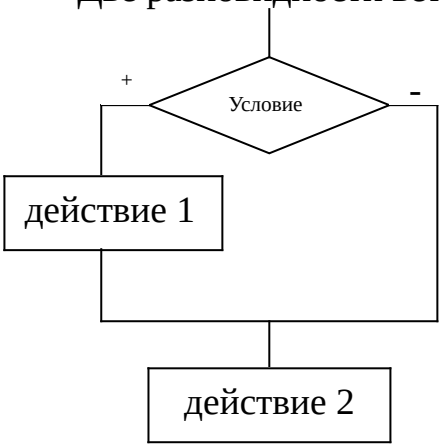


1. С помощью *текстового редактора* формируется текст программы и сохраняется в файле с расширение `.cpp`. Пусть, например, это будет файл с именем `exemple.cpp`.
2. Осуществляется этап препроцессорной обработки, содержание которого определяется директивами препроцессора, расположенными перед заголовком программы (функции). В частности. по директиве `#include` препроцессор подключает к тексту программы заголовочные файлы (*.h) стандартных библиотек.
3. Происходит *компиляция текста* программы на С++. В ходе компиляции могут быть обнаружены синтаксические ошибки, которые должен исправить программист. В результате успешной компиляции получается *объектный код программы* в файле с расширением `.obj`. Например, `exemple.obj`.
4. Выполняется этап *компоновки* с помощью системной программы Компоновщик (`Linker`). Этот этап еще называют *редактированием связей*. На данном этапе к программе подключаются библиотечные функции. В результате компоновки создается исполняемая программа в файле с расширением `.exe`. Например, `exemple.exe`.

ОПЕРАТОРЫ ЯЗЫКА С++

ОПЕРАТОРЫ ВЫБОРА IF И IF…ELSE

Язык С++ обладает исчерпывающим набором конструкций, позволяющим управлять порядком выполнения отдельно взятых ветвей программы. Для осуществления ветвления используются так называемые условные операторы. Две разновидности ветвления: неполная и полная.

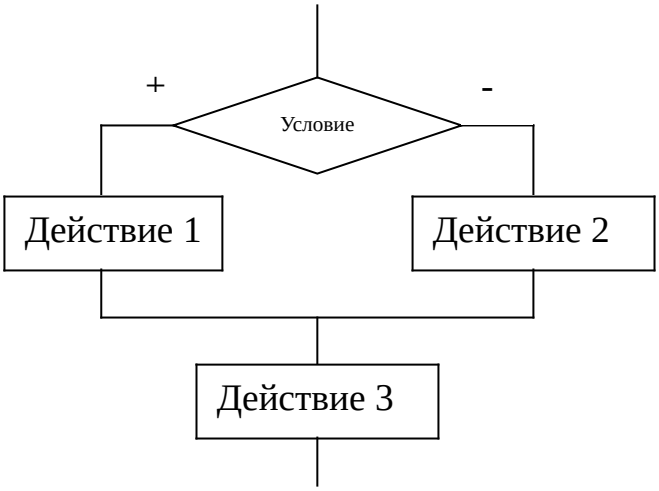


Синтаксис: if (условие) действие 1;

Оператор **if** производит ветвление программы в зависимости от результата проверки некоторого условия на истинность. Условие содержит операторы сравнения: >, <, >=, <=, = =, ! = . Если ***условие*** принимает истинное значение, то выполняется ***действие 1***. В противном случае выполнение программы переходит к ***действию 2***.

В конструкциях языка С++ операторы могут быть составными. Это означает, что в зависимости от принятого решения выполняется не один, а целый блок операторов. Составной оператор начинается с открывающейся фигурной скобки и заканчивается закрывающейся фигурной скобкой. Все содержимое составного оператора рассматривается компилятором как единый оператор. ***Синтаксис с составным оператором***

```
if (условие) { действие 1;
               действие 2;
               действие 3;
             }
```



Полная форма ветвления имеет синтаксис:

if (условие) действие 1; else действие 2;

Если ***условие*** истинно, то выполняется ***действие 1*** с последующим переходом к ***действию 3***.

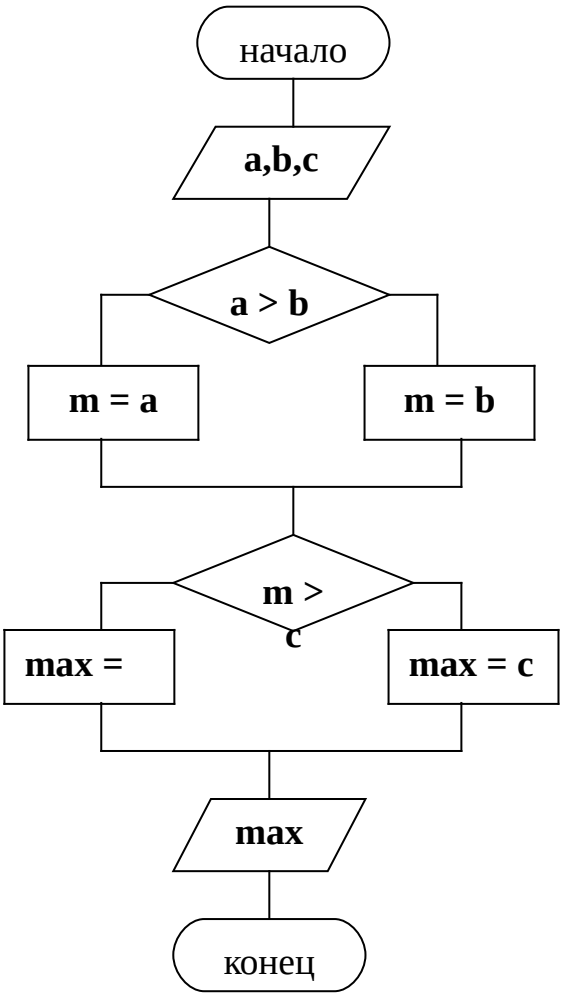
Если ***условие*** ложно, то выполняется ***действие 2*** с последующим переходом к ***действию 3***.

При использовании составного оператора в конструкции **if…else** точка с запятой перед **else** не ставится.

```
if (условие)  {действие 1;
               действие 2;
              }
else    {действие 3;
         действие 4;
        };
```

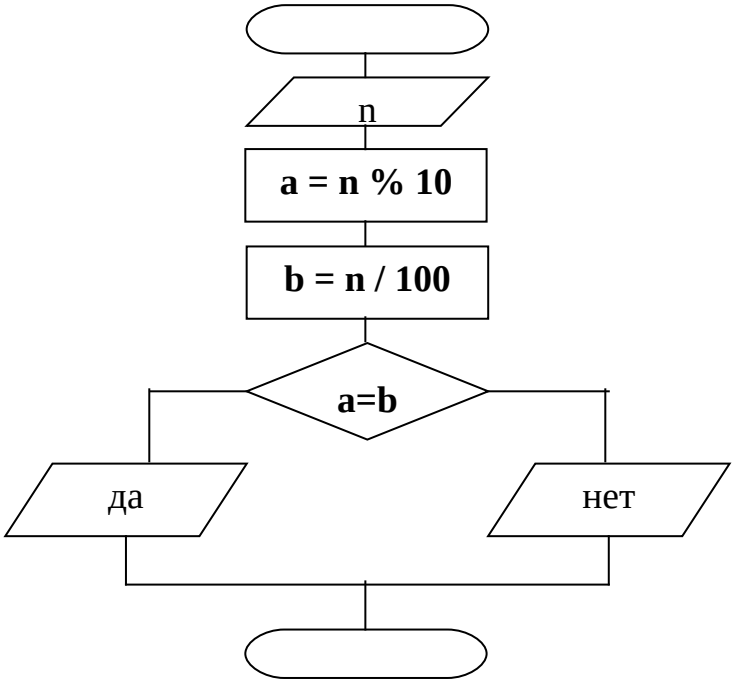
Условие может быть ***простым*** и ***сложным***. Для формирования сложного условия используются ***логические операции*** **(&&, | |, !).**Типичной ошибкой студентов является использование в условных конструкциях оператора присваивания (=) вместо оператора сравнения (= =).

Пример: найти максимальное число из трёх чисел.



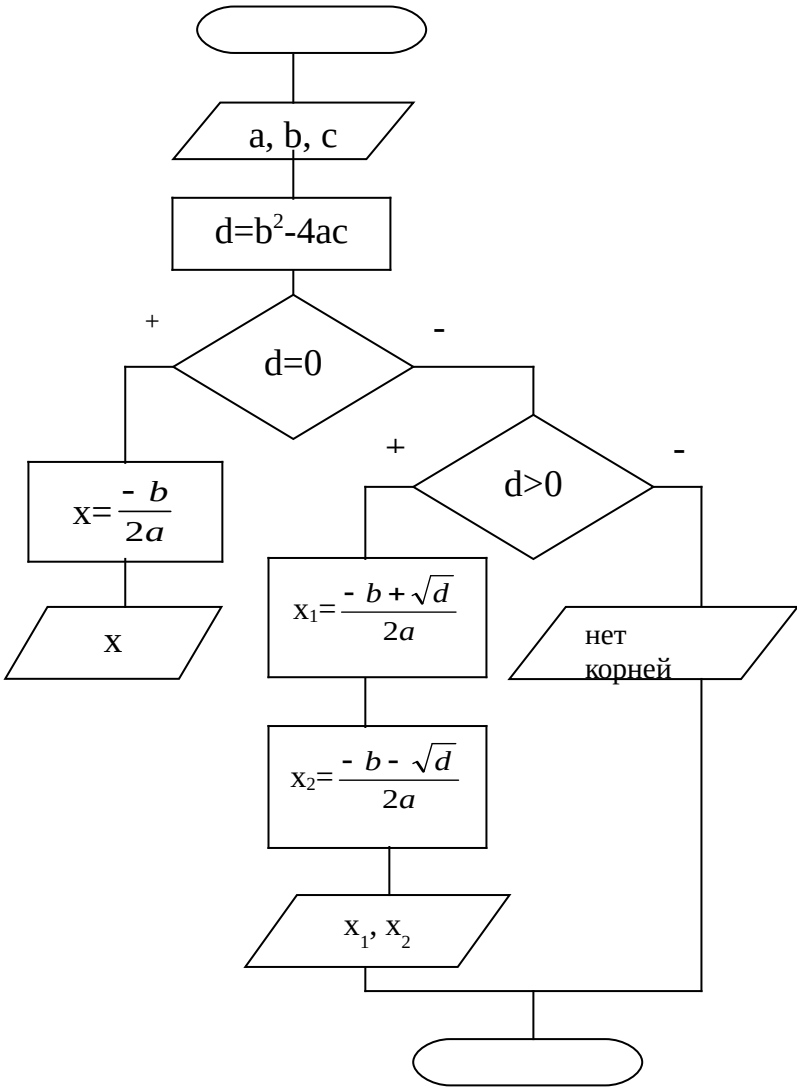

```
// нахождения max
# include <iostream.h>
main()
{ int a, b, c, m, max;
  cout <<” Ввести три числа”<<’\n’;
  cin >>a>>b>>c;
  if (a>b)  m = a; else m = b;
  if (m>c) max = m;  else  max = c;
  cout<<”max=”<<max<<’\n’;
  return 0;
}
```

Пример: Дано трехзначное число. Определить, является ли оно «палиндромом».




```
// палиндром
# include <iostream.h>
main()
{ int n, a,b;
 cout <<”Ввести число n ”<<’\n’;
 cin>>n;
// находим последнюю цифру
   числа
  a=n % 10;
// находим первую цифру числа
   b=n / 100;
   if (a= = b)
   cout << n<<” палиндром”<<’\n’;
   else
   cout <<n<<”не палиндром”;
   return 0;
}
```

Пример: Вычислить корни квадратного уравнения $ax^2 + bx + c = 0$



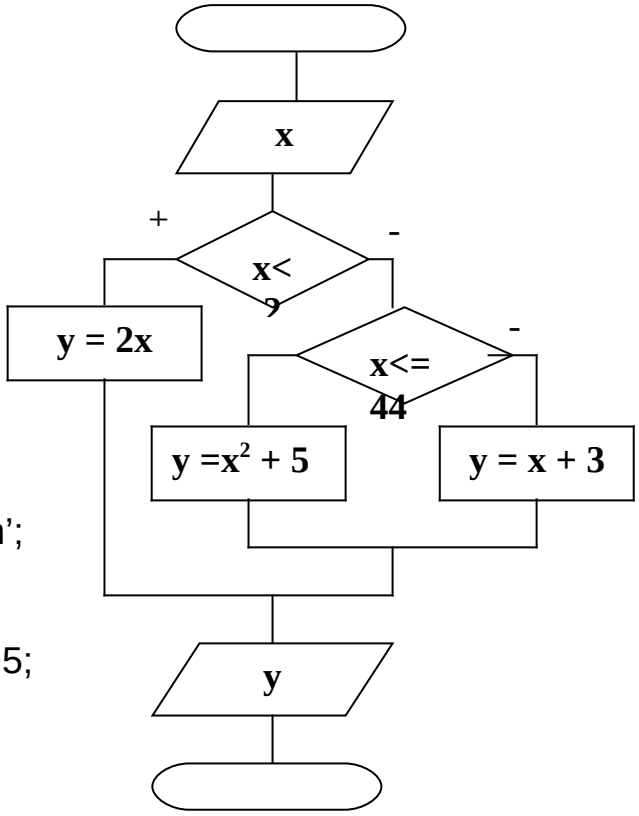

```
// корни квадратного уравнения
#include <iostream.h>
main()
{ float a, b, c, d, x, x1, x2;
  cout <<"ввести коэффициенты
            уравнения"<<'\n';
   cin >>a>>b>>c;
   d=b*b-4*a*c;
   if (d= =0) {  x= -b/(2*a);
                 cout<<"x="<<x<<'\n';
              }
   else
   if (d>0) {x1=(-b+sqrt(d))/(2*a);
             x2=(-b-sqrt(d))/(2*a);

             cout<<"x1="<<x1<<'\n';

             cout<<"x2="<<x2<<'\n';
            }
   else cout <<"Корней нет"<<'\n';
```

Пример: Вычислить функцию:

$$Y=\begin{cases} 2x, & \text{если } x<2 \\ x^2+5, & \text{если } 2\le x\le 4 \\ x+3, & \text{если } x>4 \end{cases}$$




```
// вычисление функции
# include <iostream.h>
main()
{ int x, y;
   cout <<"Ввести значение x "<<\'n';
   cin >>x;
   if (x<2) y = 2*x; else
                        if (x<=4) y = x*x+5;
                        else y = x+3;
   cout <<"y=",<<y<<'\n';
   return 0;
}
```

ОПЕРАТОР МНОЖЕСТВЕННОГО ВЫБОРА - SWITCH

Синтаксис: switch (ключ)

```
{ case константное_выражение_1: оператор_1; break;
  case константное_выражение_2: оператор_2; break;
  case константное_выражение_n: оператор_n; break;
  default: оператор_n+1;
};
```

Оператор switch представляет собой своеобразный «переключатель». Работает следующим образом. На первом этапе анализируется ***ключ*** и осуществляется переход к той ветви программы, для которой его значение совпадает с указанным ***константным_выражением.*** Далее следует выполнение оператора или группы операторов, пока не встретиться ключевое слово break**.** Происходит выход из тела оператора switch или не бует достигнут конец блока – закрывающаяся фигурная скобка. Если ключ не совпадет ни с одним ***константным_выражением*** , выполняется ветвь, определенная с помощью ключевого слова default.

Правила использования оператора **switch*:***

1. В качестве ключа может использоваться арифметическое выражение, переменная, имеющие целый тип (вещественный тип не используется).
2. Константное_выражение – это конкретное число целого типа.
3. Каждая ветвь case должна заканчивается оператором break***.***
4. Фигурные скобки для заключения группы операторов не используются.
5. Ветвь default выполняется тогда, когда ни одна из ветвей case не подходит и всегда должна быть последней.
6. Ветвь default не обязательна.

Пример: произвести арифметическую операцию с двумя операндами в соответствии с введенным знаком.

```
# include <iostream.h)>
 main()
{ double x, y, z;
   char op;      // арифметическая операция
   cout<<”Введите два операнда”<<’\n’;
   cin>>x>>y;
   cout<<”Введите знак арифметической операции”<<’\n’;
   cin>>op;
   switch(op)
       {case ‘+’:  z=x+y;  cout<<”z=”<<z<<’\n’;  break;
        case ‘-’:   z=x-y;   cout<<”z=”<<z<<’\n’;   break;
        case ‘*’:   z=x*y;   cout<<”z=”<<z<<’\n’;   break;
        case ‘/’:   if (y != 0) {z=x/y;  cout<<”z=”<<z<<’\n’;  break;
        default:   cout<<”Неверный ввод”;
        }
    return 0;
 }
```

ОПЕРАТОР БЕЗУСЛОВНОГО ПЕРЕХОДА – GOTO И МЕТКИ

Метка представляет собой идентификатор с расположенным за ним символом двоеточия (:). Метками помечают какой-либо оператор, на который в дальнейшем должен быть осуществлён безусловный переход.

Безусловная передача управления на метку производится при помощи оператора goto. Оператор goto может осуществлять переход (адресоваться) к меткам, обязательно расположенным в одном с ним теле функции.

Синтаксис: **goto метка;**

Данный оператор – очень мощное и небезопасное средство управления поведением программы. Использовать его нужно крайне осторожно, так как, например переход внутрь цикла (обход кодов инициализации) может привести к трудно локализуемым ошибкам.

ОПЕРАТОРЫ ЦИКЛА

Мощным механизмом управления ходом последовательности выполнения программы является использование циклов. Цикл является конструкцией языка, которая улучшает обработку повторяющихся действий.

Разновидности циклов: for, while, do…while.

ОПЕРАТОР ЦИКЛА FOR

Цикл for используется в тех случаях, когда заранее известно сколько раз должна повторяется циклическая часть программы.

Синтаксис: for (выражение_1; выражение_2; выражение_3) {тело цикла};

выражение_1- инициализация переменной цикла; все выражения, входящие в инициализацию цикла, определяются только один раз при входе в цикл;

выражение_2 - проверка условия на продолжение цикла; если условие истинно, то выполняется тело цикла, в противном случае цикл заканчивается.

выражение_3 - изменение переменной цикла происходит после каждой итерации (повтора) цикла.

Пример:
```
for ( int i =1; i < 10; i ++)
cout<<” k = ” << i * I <<’\n’;
```

Трассировка:

i = 1	k = 1	i = 3	k = 9	i = 5	k = 25	……	
i = 2	k = 4	i = 4	k = 16	i = 6	k = 36	i = 9	k = 81

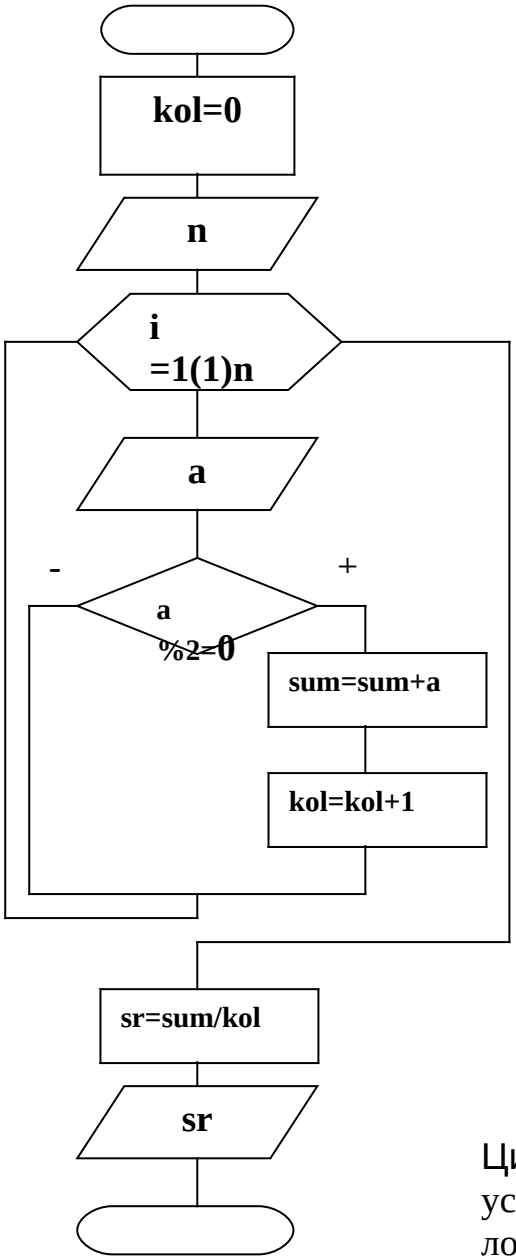
Все три компонента цикла for являются необязательными, т.е. могут отсутствовать:

1. int i=0;
   for (; I < n; i ++)
   {тело цикла}

2. int i = 0;
   for (; i < n;)
   { …; i ++; }

3. for (; ;) { тело цикла };
   выражение_2 отсутствует, значит оно постоянно истинно – бесконечный цикл.

Пример: Дана последовательность натуральных чисел из n элементов. Найти среднее арифметическое четных элементов последовательности.




```
// среднее значение элементов
#include<iostream.h>
#define n 10
main()
{ int a, kol = 0, i;
  float sum = 0, sr;
  cout<<"Ввести последовательность"<<'\n';
  for ( i = 0; i < n; i + +)
       {cout<<"Введите элемент послед-ти";
        cin>>a;
        if (a % 2 = = 0) { sum = sum + a;
                           kol = kol + 1;
                          };
        };
   sr = sum / kol;
   cout<<"Среднее значение ="<<sr<<'\n';
   return 0;
  }
```

ОПЕРАТОР ЦИКЛА WHILE

(цикл с предусловием)

Цикл **while** выполняет тело цикла до тех пор, пока условие остаётся истинным. Если условие сразу ложно, то тело цикла не выполнится ни разу.

Синтаксис: **while (условие) {тело цикла};**

ОПЕРАТОР ЦИКЛА DO…WHILE (*цикл с постусловием)*

В отличие от оператора **while,** цикл **do…while** сначала выполняет тело цикла, а затем уже осуществляет проверку условия на истинность. Такая конструкция гарантирует, что тело цикла будет обязательно выполнено хотя бы один раз.

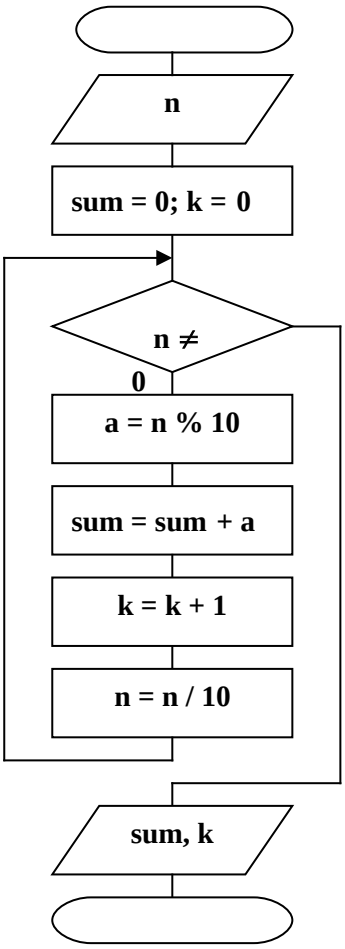
Синтаксис: **do { тело цикла }**

while (условие);

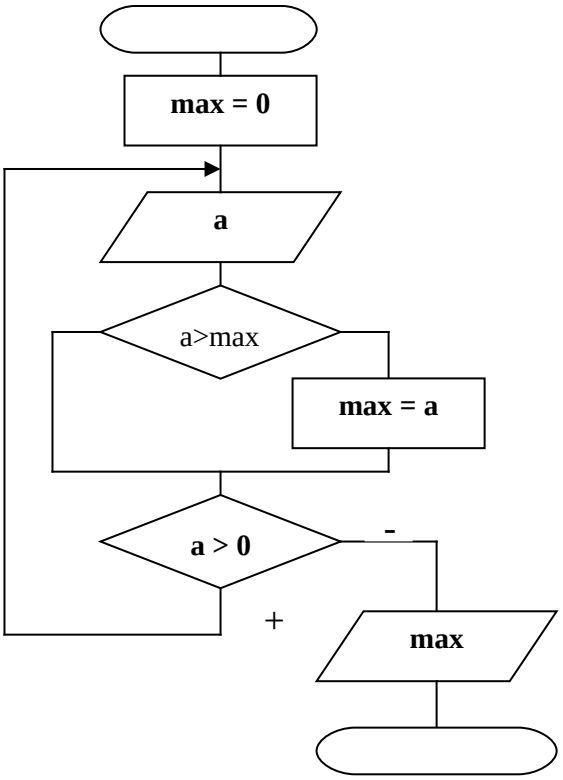
Особенности циклов while, do…while:

1. Тело циклов выполняется тогда, когда условие принимает истинное ***ЗНАЧЕНИЕ.***
2. Чтобы не было зацикливания в теле цикла должен быть оператор, воздействующий на условие.

Пример: Дано целое неотрицательное число. Определить количество цифр и сумму цифр числа в его десятичной записи. Например, для n = 113562 сумма равна 18, количество цифр 6.



Пример: Написать программу, которая определяет максимальное число из введенной с клавиатуры последовательности положительных чисел (последовательность неограниченна, ноль завершает ввод данных).




```
// сумма и количество цифр числа
#include<iostream.h>
main()
{ long int n;
  int a, sum = 0, k = 0;
  cout<<”Ввести число”<<’\n’;
  cin>>n;
  while (n != 0)
      { a = n % 10;
        sum = sum + a;
        k = k + 1;
        n = n / 10;
       };
  cout<<”sum =”<< sum<<’\n’;
  cout<<” k = “<< k<< ‘\n’;
  return 0;
}
```

```
// максимальное число
#include<iostream.h>
main()
{ int a, max = 0;
  do
     { cout<<”Введите число”<< ‘\n’;
        cin>>a;
        if (a > max) max = a;
      }
   while (a > 0);
 cout<<”Максимальное число =”<<max;
 cout<< ‘\n’;
 return 0;
}
```

ВЛОЖЕННЫЕ ОПЕРАТОРЫ ЦИКЛА

Если телом цикла является циклическая структура, то такие циклы называется ***вложенными.***

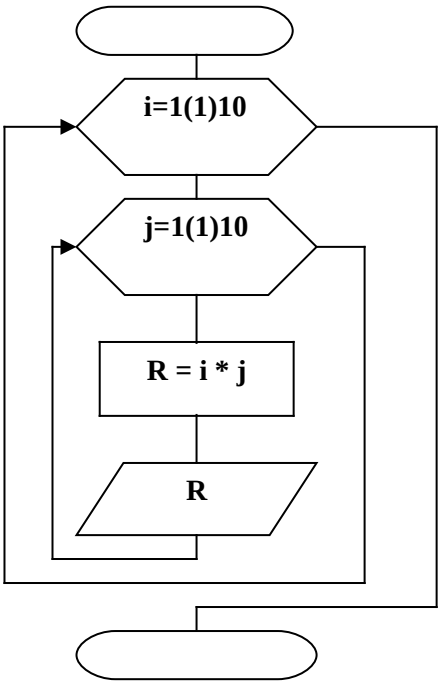
Цикл, содержащий в себе другой цикл называется ***внешним.***

Цикл, содержащийся в течение другого цикла называют ***внутренним.***

Внешний и внутренние циклы могут быть 3-х видов: while, for, do while.

Пример – таблица умножения.

```
#include<iostream.h>
#include<conio.h>
main()
{clrscr();
  int i,j;
  for ( i = 1; I < 10; i + +)
     { for ( j = 1; j < 10; j + +)
        cout<<i * j<< ‘ \ t ‘;
        cout<<’ \ n ‘;
      }
  return 0;
}
```



ОПЕРАТОРЫ BREAK И CONTINUE

Оператор break используется в операторе switch и циклах. Служит для принудительного выхода из цикла или оператора switch. Оператор break прекращает выполнение цикла и осуществляет передачу к следующему за циклом оператору.

Оператор continue используется только оператором цикла, с его помощью завершается текущая итерация (повтор) и начинается проверка условия дальнейшего продолжения цикла, т.е условие начала следующей итерации.

Пример:

```
for (int i = 0; i < 10; i + +)
 { if (i = = 5) break;
   cout<< i <<’\t’;
  }                          результат: 0  1  2  3  4

for (int i = 0; i < 10; i + +)
 { if (i = = 5) continue;
   cout<< i <<’\t’;
  }                          результат: 0  1  2  3  4  6  7  8  9
```

МАССИВЫ

ОБЩИЕ ПОНЯТИЯ. ОДНОМЕРНЫЕ МАССИВЫ

Массив – это множество переменных одного и того же типа, имеющих одно общее имя (имя массива). Отдельная единица таких данных, входящих в массив, называется элементом массива. В качестве элементов массива или переменной с индексом могут быть данные любого типа (один тип данных для каждого массива).

Доступ к отдельной переменной осуществляется по индексу.

Индекс определяет положение элемента в массиве. Индекс элемента массива и его содержимое – это разные понятия.

Объявление одномерного массива, синтаксис:

тип_ массива имя_массива [число элементов массива]

int mas [10] целочисленных

↓ ↓ ↓

тип имя кол-во

} резервирует в памяти место для размещения 10 элементов (в памяти элементы массива располагаются непосредственно один за другим)

Правила объявления массива:

1) Индекс первого элемента массива в С++ равен 0 (нулю).
2) Объявление массива в С++ влечет за собой точное определение числа его элементов (количество элементов массива должно быть всегда целым числом).
3) Доступ к отдельным элементам массива осуществляется по имени массива и индексу элемента массива: a[0] = 1; a[1] = 2.
4) В инструкции объявления массива удобно использовать константу, объявленную в директиве # define N 5

Правила использования элементов массива:

1) Необходимо присваивать значение элементу массива до того, как он будет считываться
2) При решении задач использовать допустимый индекс
   int mas[10]; // объявленный массив
   x = mas[11]; // ошибка, элементы с таким индексом не существуют
3) В качестве индекса элемента массива можно использовать выражение целого типа – константу или переменную
   x[2*i - 1]; mas[i + 1]; a[2*i]

Инициализация массивов

Инициализацию массивов можно проводить при их объявлении.

```
int mas[12] = {2, 4, 7, 11, 12, 13, 8, 5, 1, 2, 6, 9};
mas[0] = 2;  mas[4] = 12;    mas[8] = 1;    mas[11] = 9;
```

Если в списке инициализации значений указано меньше, чем объявлено в размере массива, то имеет место частичная инициализация. В этом случае иногда после последнего значения в инициализирующем выражении для наглядности ставят запятую:

int mas[12] = {2, 4, 7, }; или int mas[12] = {2, 4, 7};

остальные элементы принимают значение равные нулю.

int mas[10] = {0} – массив mas [10] инициализирован нулевыми элементами.

При объявлении массива с одновременной его инициализацией, разрешается опускать значение размера в квадратных скобках. При этом компилятор самостоятельно подсчитает количество элементов в списке инициализации.

int mas[ ] = {0, 2, 4, 6, 8, 10};

Если в программе потребуется определить, сколько элементов в массиве, то можно воспользоваться формулой

int size = sizeof (mas) / sizeof (mas[0]); где

sizeof (mas) определяет возвращает общий размер, занимаемый массивом mas[ ] в памяти (в байтах);

sizeof (mas[0]) - размер одного элемента массива/

sizeof – оператор для определения числа байтов, необходимых для хранения объекта

РАБОТА С ЭЛЕМЕНТАМИ МАССИВА

Ввод элементов массива

а) инициализация при объявлении массива

```
float a[5] = {1.3, 2.5, 4.4, 7.9}
```

б)
```
float a[10];      // объявление массива
cout << “Ввод элементов массивап“<<’ \ n’;
for (int i=0; i<10; i++)    cin >> a[i];
```

в) ввод по формуле:

```
double a[7];  int i;  // объявление
for (i=0; i<=6; i++)     a[i] = cos(i*i);
```

г) использование генератора случайных чисел

```
# include < stdlib.h >
main ( )
{ randomize( );
   int a[10];
   for (int i = 0; i < 10; i++)   a[i] = random(30);      [в интервале 0÷30]
  ( или  a[i] = 10 + random[41]).
```

Вывод элементов массива

а) for (i=0; i<=9; i++)
cout << a[i] << ‘|n’; //вывод в столбик

б) for (i=0; i<=9; i++)
cout << a[i] << ‘\t’; // вывод в строку
cout << ‘ \ n’;

Пример: Дан одномерный массив размером n. Найти индексы первого и последнего отрицательного элемента. Если отрицательных элементов в массиве нет или имеется только один элемент, то выдать соответствующее сообщение.

```
#include<iostream.h>
# define n 10
main()
{ int mas[n], i, p1 = -1, p2 = -1;   // p1 и p2 местоположение отрицательных
                                     //  элеметов
  cout<<”Ввести массив”<<’ \ n ‘;
  for (i = 0; i < n; i + +)
     cin>>mas[ i ];
  // обработка элементов массива
  for (i = 0; i < n; i + +)
      if (mas[ i ] < 0) { p1= i; break;};
  for (i = 0; i < n; i + +)
      if (mas[ i ] < 0)  p2 = i;
  if ((p1 != -1) && (p2 != -1))  cout<<”p1 = ”<<p1<<” p2 = ”<<p2<<’ \ n ‘;
                          else cout<<”Отрицательных элементов нет”<<’ \ n ‘;
  if ((p1 != -1) && (p2 = -1))  cout<<”Один отрицательный элемент”<<’ \ n ‘;
```

ДВУМЕРНЫЕ МАССИВЫ

Двумерный массив - набор элементов одного и того же типа, имеющих одно общее имя и два индекса для указания отдельного элемента. Первый индекс – номер строки, второй – номер столбца.

Синтаксис объявления массива:

тип_ массива имя_массива [число строк] [число столбцов]

Общее число элементов массива равно число строк * число столбцов

int mas[5][4], a[3][4];

	0 столбец	1 столбец	2 столбец	3 столбец
0 строка	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1 строка	a[1][0]	a[1][1]	a[1][2]	a[1][3]
	a[2][0]	a[2][1]	a[2][2]	a[2][3]

2 строка

В памяти двумерный (или многомерный массив) храниться как одномерный, строки массива расположены последовательно одна за другой.

Ввод элементов массива

По аналогии с одномерным массивом (с клавиатуры, через генератор случайных чисел, с использованием формул).

```
for (i=0;i<3;i++)
   for(j=0;j<4;j++)
   cin>>a[i][j]
```

Вывод элементов массива

```
for (i=0;i<3;i++)
   {for(j=0;j<4;j++)
    cout<<a[i][j]<<' \ t ';
    cout<<' \ n ';
   }
```

Для работы с элементами двумерного массива используется два цикла – внешний и внутренний (структура вложенных циклов). Начальные значения определяются во внешнем цикле, а работа начинается с внутреннего цикла.

Пример: Дан двумерный массив, размером n x m. Найти нечетное количество элементов каждой строки массива. Полученный результат записать в одномерный массив и вывести его на экран.

```
#include <iostream.h>
#define n 4
#define m 5
void main()
{ int mas[n][m], i, j, kol ;
  int a[n];      //одномерный массив, кол-во элементов = кол-ву строк
  cout<<" Ввести массив"<<endl;
  for (i=0; i < n; i++)
     for(j=0; j < m; j++)
     cin>>mas[i][j];
  cout<<"Вывод исходного массива"<<endl;
  for (i=0; i < n; i++)
    {for(j=0; j<m; j++)
      cout<<mas[i][j]<<' \ t ';
      cout<<' \ n ';
     }
  // обработка элементов массива по строкам
  for (i=0; i < n; i++)
    { kol = 0;
      for (j=0; j < m; j++)
      if (mas[i][j] % 2 != 0) kol ++;
      a[i] = kol; }
    cout<<"Вывод одномерного массива"<<endl;
```

```
    for (i=0; i<n; i++)  cout<<a[i]<< ' \ t ';
    cout<<' \ n ';
}
```

МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

ОПРЕДЕЛЕНИЯ, ОПИСАНИЯ И ВЫЗОВЫ ФУНКЦИЙ

Разработка программы – это довольно непростой процесс. При решении любой задачи мы используем структурное программирование, т.е. нисходящую разработку алгоритма с пошаговой детализацией. Поставленная перед разработчиком (программистом) задача разбивается на менее крупные задачи, которые в свою очередь делятся еще на несколько менее сложных задач, и так до тех пор, пока самые мелкие задачи не будут решены с помощью базовых конструкций языка (следование, выбор, повтор).

При решении сложной задачи также используется **модульное программирование** – это разновидность структурного программирования. Суть: конструирование программы из небольших фрагментов или модулей, каждый из которых более управляем, чем сложная (большая) программа.

Модули в С++ - это функции и классы.

Функции

математические функции (из стандартной библиотеки)

функции, определенные пользователем

Математические библиотечные функции

sqrt(x) – корень квадратный;
pow(x,y) – возведение x в степень y;
abs(x)
fabs(x)
dabs(x)
labs(x)
} абсолютное значение целого числа, вещественного, длинного числа
exp(x) – экспоненциальная функция e^x;
log(x) – натуральный логарифм;
log10(x) – десятичный логарифм;
sin(x), cos(x), tan(x) – тригонометрические функции.

ФУНКЦИИ, ОПРЕДЕЛЕННЫЕ ПОЛЬЗОВАТЕЛЕМ

Функция – это независимая совокупность объявлений и операторов, обычно предназначенная для выполнения определенной задачи.
При программировании на С++ функция – это основное понятие, без которого невозможно обойтись.

Во-первых, каждая программа имеет главную функцию (единственную) с именем main(). Именно функция main() обеспечивает создание точки входа в откомпилированную программу.

Во-вторых, кроме функции main(), в программу может входить произвольное количество неглавных функций, выполнение которых инициируется вызовами из функции main().

Принципы модульного программирования.

1. ***Логическая завершенность.***
   Функция (модуль) должна реализовывать логически законченный, целостный алгоритм.
2. ***Ограниченность.***
   Функция должна быть ограничена в размерах, в противном случае ее необходимо разбить на логически завершенные части – модули, вызывающие друг друга.
3. ***Замкнутость.***
   Функция должна использовать глобальные данные, иметь «связь с внешним миром» помимо программного интерфейса, не должна содержать ввода и вывода во внешние потоки.
4. ***Универсальность.***
   Функция должна быть универсальна, параметры процесса обработки и сами данные должны передаваться извне, а не подразумеваться и устанавливаться постоянными.
5. ***Принцип «черного ящика».***
   Функция должна иметь продуманный «программный интерфейс», т.е. набор фактических параметров и результат функции, через который она «подключается» к другим частям программы (вызывается).

ОПРЕДЕЛЕНИЕ ФУНКЦИИ

Синтаксис:

```
тип_функции   имя_функции (список формальных параметров)
  {
        тело функции;
  }
```

Правила объявления функции

1. ***Тип функции*** - это тип возвращаемого функцией результата.
   Если функция не возвращает никакого результата, то для нее указывается тип void.
2. ***Имя функции*** – идентификатор, задаваемый программистом.
3. ***Список формальных параметров*** определяет имена параметров и порядок, в котором они принимают значения при вызове функции.
   Список всегда ограничен круглыми скобками даже в случае, когда он пуст (main(); randomize() и т.д.)
   Список формальных параметров имеет следующий синтаксис:

   (тип_перем_1 перем_1, тип_перем_2 перем_2)

   Например: float fun (int a, int b, float d) {тело функции}

Ошибка, если при объявлении формальных переменных не указывается тип для каждой переменной (int a, b) – нельзя!

4. ***Тело функции*** – это либо составной оператор, либо блок, заключается в фигурные скобки, после закрывающейся скобки **«;»** не ставится.
   Составной оператор – содержит операторы, которые определяют действие функции. ***Блок*** – содержит объявление локальных переменных и операторы. ***Локальные переменные*** – это переменные объявленные в функции, действие которых заканчивается после завершения работы функции.

Замечание! Тело функции не может содержать в себе определения других функций. Состав тела функции - переменные, константы, операторы, оператор возврата return.

5. ***Оператором возврата*** из функции в точку ее вызова является оператор return. Он может использоваться в функциях в двух формах:
   1) return переменная;
   2) return выражение;

Примеры:

а)
```
int maxi (int x, int y)
{ int max;
  if (x>y) max = x;
  else max = y;
  return max;
}
```

в)
```
double cub( float y) { return y*y*y;}
```

Если функция имеет тип void, то функция не возвращает никакого значения

в качестве своего результата. Например, функция обмена значениями двух переменных

```
void swap (inta, int b)
{  int t;
   t=a;
   a=b;
   b=t;
}
```

6. ***Вызов функции***

   Обращение к функции – это и есть вызов функции, которое записывается через выражение:

   имя_функции (список фактических параметров)

   При обращении к функции (вызове функции) формальные параметры заменяются фактическими, при этом должны соблюдаться правила соответствия по последовательности, по типам и количеству.
   Фактический параметр – это выражение того же типа, что и соответствующего ему формального параметра.

ОБЪЯВЛЕНИЕ ФУНКЦИИ (ПРОТОТИП ФУНКЦИИ)

Прототипом называется предварительное описание функции, в котором содержатся все необходимые сведения для правильного обращения к ней: тип и

имя функции, типы формальных параметров и их порядок следования. Прототип функции позволяет компилятору выполнить контроль соответствия типов параметров и правильность вызова функции.

Синтаксис:

```
тип_функции_имя_функции (список форм параметров);
main()
{
  тело главной функции
}

//определение функции
тип_функции_имя_функции (список форм параметров)
{
  тело функции
}
```

Отличие описания функции от определения

1) Основное различие – точка с запятой в конце объявления прототипа.
2) Необязательность имен формальных параметров в прототипе, например int maxi (int, int).

```
#include <iostream.h>
int kvadrat(int);  //прототип функции нахождения квадрата числа
main()
{ for (int x=1; x<=10; x++)
  cout<<kvadrat(x)<<" ";
  cout<<'n';
  return 0;
}
 // определение функции
 int kvadrat (int y);
 { return y*y; }
```

Пример: Вычислить площадь круга при различных значениях радиуса.
Написать функцию для вычисления площади круга.

```
#include <iostream.h>
# define PI  3.14
float pl (float rad)
{ float skr;
  skr = rad*rad*PI;   // функция нахождения площади круга
  return skr;
}
 void main()
{ float r1, r2, s1, s2;
  cout<<"Введите два радиуса"<<endl;
  cin>>r1>>r2;
  s1= pl(r1);        // вызов функции для фактического параметра r1
```

```
cout<<"Площадь круга при r1 ="<<s1<<endl;
s2 = pl(r2);       // вызов функции для фактического параметра r2
cout<<"Площадь круга при r2 ="<<s2<<endl; }
```

МАССИВЫ В КАЧЕСТВЕ ПАРАМЕТРОВ ФУНКЦИИ

Два способа использования одномерных массивов в качестве параметров функции.

I СПОСОБ. Синтаксис:

тип_функции имя_функции (тип_массива имя_массива [n])
{ тело функции}

n – количество элементов в массиве

Вызов функции: имя_функции (имя_массива);

Используется для массивов одинаковой длины.

Пример: Написать функцию нахождения минимума в одномерном массиве.

```
int mini (int a[n])
 { int min = 999;
   for(int i = 0; i < n; i ++)
    if (a[i] < min) min = a[i];
    return min;
 }
```

Вызов функции в главном модуле:

```
main()
{ int mas[m], tmin;
   ……................
   tmin = mini (mas);
   ……..................
 }
```

II СПОСОБ. Синтаксис:

тип_функции имя_функции (тип_массива имя_массива [ ], int n)
{ тело функции}

n – количество элементов в массиве

Вызов функции: имя_функции (имя_массива, кол-во элементов мас-ва);

В определении функции используется два формальных параметра, значит и при вызове тоже используется два фактических параметра (имя массива и количество элементов). Второй способ используется, когда функция с таким описанием формальных параметров, используется для массивов разной длины.

Пример: Напишите функцию, которая подсчитывает среднее арифметическое элементов массива, кратных 3. В програм .

```
 #include <iostream.h>
 # define n 5
 # define m 7
// функция
 float sredn (int a [ ], int t )
 { float sr, sum = 0;  int  kol = 0;
   for(int i = 0; i < t; i ++)
      if (a[i] % 3 = = 0) { sum = sum + a[i];
                            kol = kol + 1;
```

```
void main()
{int b[n], c[m], i;
 float sr1, sr2;
 cout<<"Ввести 1 массив"<<endl;
for (i= 0; i < n; i ++)
    cin>>b[i];
 cout<<"Ввести 2 массив"<<endl;
 for (i= 0; i < m; i ++)
    cin>>c[i];
  sr1 = sredn (b, n);
  sr2 = sredn (c, m);
  cout<<"Среднее 1 м-ва ="<<sr1;
  cout<<"Среднее 2 м-ва ="<<sr2;
  }
```

```
                    };
    sr = s / kol;
    return sr;
}
```

МЕТОДЫ СОРТИРОВКИ И ПОИСКА ДАННЫХ

АЛГОРИТМЫ СОРТИРОВКИ

Для решения многих задач необходимо упорядочить данные по определенному признаку. Этот процесс называется сортировкой. Сортировка используется в массивах.

Элементы массива можно сортировать:

- по возрастанию (каждый следующий элемент больше предыдущего *a* [1] < *a* [2] < *a* [3] и т.д.);
- по неубыванию (каждый следующий элемент не меньше предыдущего *a* [1] <= *a* [2] <= *a* [3] и т.д.);
- по убыванию (каждый следующий элемент меньше предыдущего *a* [1] > *a* [2] > *a* [3] и т.д.);
- по невозрастанию (каждый следующий элемент не больше предыдущего *a* [1] >= *a* [2] >= *a* [3] и т.д.).

Существующие методы сортировки обычно разбивают на три класса, в зависимости от лежащего в их основе приема:

- сортировка выбором;
- сортировка обменами;
- сортировка включениями (вставками).

Алгоритмы сортировки отличаются друг от друга степенью эффективности, под которой понимается количество сравнений и количество обменов, произведенных в процессе сортировки.

СОРТИРОВКА ПРОСТЫМ ВЫБОРОМ

Идея – находится элемент с наименьшим значением (наибольшим) и меняется местами с первым элементом (с последующим элементом). Среди оставшихся элементов опять ищется наименьший (наибольший), который меняется со вторым (с предпоследним) и т.д.

```
void select (int x[ ], int n)
{ int i, j, k;
   int q, tmp;
   for (i=0; i<n-1; i++)
      { q=0; k=I; tmp=x[i]
         for (j=i+s; j<n; j++)
            { if (x[j]<tmp)   { k=j;
                                tmp=x[j]
                                q=1;
            }                 }
      if (q´=0)
```

```
            { x[k]=x[i]; x[i]=tmp; }
        }
    }
```

СОРТИРОВКА ОБМЕНАМИ (ПУЗЫРЬКОВАЯ СОРТИРОВКА)

Идея – сравниваются два соседних элемента массива, в результате которого меньшее число (более легкий пузырек) перемещается на одну позицию влево. Обычно такой просмотр организуется с конца массива и после первого прохода самое маленькое число перемещается на первое место. Затем все повторяется от конца массива до второго элемента и т.д.

Первый вариант данной сортировки

```
void buble (int x[ ], int n)
 { int i, j, tmp;
      for (i=1; i<n; i++)
          for (j=n-1; j>=i; j--)
            if (x[j-1]>x[j])   { tmp=x[j-1];
                                 x[j-1]=x[j]
                                 x[j]=tmp;
                               }
          }
```

Второй вариант сортировки

```
 void buble1 (int x[ ], int n)
     { int i, j, tmp, q;
       m: q=0;
          for (i=0; i<n-1; i++)
               if (x[i]>x[i+1])
                    { tmp=x[i];
                      x[i]=x[i+1];
                      x[i+1]=tmp;
                      q=1;
                    }
          if (q´= 0) goto m;
     }
```

СОРТИРОВКА МЕТОДОМ ВСТАВКИ

Идея этого метода базируется на последовательном пополнении ранее упорядоченных элементов. На первом шаге сортируется первые два элемента. Далее берется третий элемент и для него подбирается такое место в отсортированной части массива, что упорядоченность не нарушается. К трем упорядоченным добавляется четвертый. И так продолжается до тех пор, пока к n-1 ранее упорядоченным элементам не присоединяется последний.

```
void insert (int x[ ], int n)
  { int i, j, tmp;
     for (i=1; i<n; i++)
       { tmp=x[i]
         for (j=i-1; j>=0; && tmp<x[j]; j--)
              x[j+1]=x[j];
         x[j+1]=tmp;
        }   }
```

Существуют и другие разновидности сортировок:

– сортировка методом Шелла;

- сортировка методом Хоара;
- сортировка слияниями.

АЛГОРИТМЫ ПОИСКА

Алгоритмы поиска, как и алгоритмы сортировки, являются основными алгоритмами обработки данных прикладных задач.

Алгоритмы поиска:

- линейный (последовательный) поиск;
- бинарный поиск;
- интерполяционный поиск.

ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК

Если исходный массив не упорядочен, то единственно разумным способом является последовательный перебор всех элементов массива и сравнение их с заданным значением.

Классический алгоритм поиска элемента q в массиве a[n]:

1 шаг: установить начальный индекс равный 1 (j=1)

2 шаг: проверить условие q=*a*[j], если оно выполняется, то сообщить, что искомое значение находится в массиве на j-ом месте и прервать работу. В противном случае продолжить работу;

3 шаг: увеличить индекс на 1;

4 шаг: проверить условие j<n+1, если выполняется, то вернуться к шагу 2, в противном случае выдать сообщение, что данное значение *q* в массиве не содержится.

```
int ssearch (int q, int a[ ], int n)
  { int j;
     for (j=0; j<n; j++)
        if (q= =a[j])  return j;
        return -1;
  }
```

БИНАРНЫЙ ПОИСК (ДВОИЧНЫЙ)

Метод половинного деления. Применяется только для **предварительно упорядоченных массивов.**

Даны целое число *x* и массив *a*[n], отсортированный в порядке неубывания. Идея бинарного метода состоит в том, чтобы проверить, является ли *x* средним элементом массива. Если да, то ответ получен, если нет, то возможны два случая:

а) *x* < среднего значения, тогда из рассмотрения исключаются все элементы массива, расположенных в нем правее среднего;

б) *x* > среднего значения, тогда из рассмотрения исключается левая половина массива.

Средний элемент в том и другом случае в дальнейшем не рассматривается. На каждом шаге отсекается та часть массива, где заведомо не может быть обнаружен элемент *x*.

```
int binar (int q, int a[ ], int n)
  { int l, r, m;
     l=0; r=n-1;
     for (; l<=r;)
          { m=(l+r)/2;
             if (q<a[m]) r=m-1;
             else if (q>a[m]) l=m+1;
                   else return m;
          }
     return -1;
  }
```

ИНТЕРПОЛЯЦИОННЫЙ ПОИСК

Если k находится между k_e и k_r, то номер очередного элемента для сравнения определяется формулой: $m = l + (r - l) * (k - k_e) / (k_r - k_e)$

Пример: Два упорядоченных массива объединить в один, тоже упорядоченный.

```
#include<iostream.h>
#define n 5
void main()
{ int a[n], b[n], c[2*n], I, j, k;
   for (i=0;i<n; i++)
      cin>>a[i];
   for(j=0; j<n; j++)
      cin>>b[j];
   i=0; j=0; k=0;
       do { if (a[i]<b[j]) c[k++]=a[i++];
             else if (a[i]>b[j]) c[k++]=b[j++];
                   else { c[k++]=a[i++];
                          c[k++]=b[j++]
                         }
            }
        while ((i<n) && (i<j));
   while (i<n)  c[k++]=a[i++];
   while(i<n)   c[k++]=b[j++];
   for(i=0; i<2*n; i++)
   cout<<c[i]<<'\t';
   cout<<'\n';
}
```

РЕКУРСИВНЫЕ ФУНКЦИИ

Рекурсивная функция – это функция , которая вызывает сама себя либо непосредственно, либо с помощью другой функции.

Прямая рекурсия – когда функция вызывает сама себя. ***Косвенная рекурсия*** – если функция содержит обращение к другой функции, через косвенный вызов определенной функции.

Классический пример рекурсивной функции – вычисление факториала.

5!=5*5*3*2*1=120 → n! = n*(n-1)!

5!=5*(5-1)!=5*4!

↓

4*(4-1)!=4*3!

↓

3*2!

↓

2*1!

↓

1

Ограничения: для отрицательного аргумента факториал не существует, результат принимает нулевое значение. Для нулевого значения результат равен единице (**0! = 1).**

Функция вычисления факториала:

```
long fact(int k)
{ if (k<0) return 0;
   if (k==0) return 1;
   return k*fact(k-1);  //рекурсивный вызов функции
}
```

При вычислении функции с k=3 произойдет повторное обращение к функции fact(2). Это обращение потребует вычисления fact(1). При вычислений fact(0) будет получен результат = 1. Затем цепочка вычислений раскрутится в обратном порядке:

fact(1) = 1*fact(0) = 1;

fact(2) = 2*fact(1) = 2;

fact(3) = 3*fact(2) = 6;

Выполнение рекурсивной функции происходит в два этапа:

I этап: прямой (рассматривается тело функции), при этом весь маршрут последовательных рекурсивных вызовов компьютер запоминает в специальной области памяти, называемой стеком.

II этап: обратный ход – восстанавливается цепочка вычислений, сохраненных в стеке, и выдается результат рекурсивных вызовов.

ПРИНЦИПЫ РЕКУРСИВНЫХ ФУНКЦИЙ

1. Рекурсивная функция разрабатывается как обобщенный шаг процесса, который вызывается в произвольных начальных условиях и приводит к следующему шагу в некоторых новых условиях.
2. Начальные условия очередного шага должны быть формальными параметрами функции.
3. Начальные условия следующего шага должны быть сформированы в виде фактических параметров рекурсивного вызова.
4. Локальными переменными функции должны быть объявлены все переменные, которые имеют отношение к протеканию текущего шага процесса и к его состоянию.
5. В рекурсивной функции обязательна проверка условий завершения рекурсии, при которых следующий шаг процесса не выполняется.

Использование рекурсии не всегда желательно, так как она может вызвать переполнение стека. Рекурсию удобно использовать для вычисления рекуррентных последовательностей. Примерами рекуррентных последовательностей являются арифметическая и геометрическая прогрессии. Рекуррентные формулы имеют вид: $a_i = a_{i-1} + 2$;

$$a_i = 2a_{i-1}.$$

Пример: Вычислить сумму элементов арифметической прогрессии

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
// функция арифметической прогрессии
  long int arif(long int n)
  {if ((n==0)||(n==1)) return 1;
  else return arif(n-1)+2;
  }
main()
  {int k; int s=0, t;
   cout<<"Введите количество элементов прогрессии" <<'\n';
   cin>>k;
   for(int i=1;i<=k;i++)
         { t=arif(i);
           cout<<"a("<<i<<")="<<t<<'\t';
           s+=t;
          }
 cout<<'\n';
 cout<<"s="<<s<<'\n';
 return 0;
 }
```

При написании рекурсивных функций необходимо запомнить одно очень важное правило. В первую очередь **следует оформлять выход из рекурсии!**

Использование рекурсивных функций – красивый прием с точки зрения программистской эстетики. Однако этот путь не всегда рациональный. Как правило, рекуррентный алгоритм с большими или меньшими усилиями можно превратить в обычный циклический процесс. Например, фрагмент программы, вычисляющей факториал, может выглядеть следующим образом:

```
int factor(int x)
{ int f=1;
  for (int i=2; i<=x; i++)
  f*=i;
  return f;
}
```

РЕКУРСИЯ ИЛИ ИТЕРАЦИЯ ?

Сравним эти два подхода в программировании.

1. Рекурсия и итерация основаны на управляющих структурах:
   рекурсия использует структуру выбора, оператор if;
   итерация – структуру повтора.
2. Используют повторение операций.
   Рекурсия реализует повторение посредством повторных вызовов функции; итерация использует повторение явным образом, через оператор for.
3. Рекурсия и итерация включают проверку условия окончания повторений.
   В рекурсиях проверка условия до тех пор, пока не будет достигнут конечный результат; в итерациях выполняется изменение счетчика до тех пор, пока он не примет значение, при котором перестанет выполняться условие продолжение цикла.

Недостатки рекурсии:

1. Рекурсивный механизм вызовов функции приводит к затратам процессорного времени.
2. Каждый рекурсивный вызов приводит к созданию новой копии функции – для этого требуется дополнительная память.

Замечания:

1. Рекурсию предпочитают итерации, если рекурсия естественно отражает задачу и её результаты, т.е когда рекурсивный подход нагляден.
2. Если требуется повысить эффективность программы, то следует избегать использования рекурсий.

ТИПЫ ДАННЫХ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ (ПЕРЕЧИСЛЕНИЯ, СТРУКТУРЫ, ОБЪЕДИНЕНИЯ)

При разработке программ могут потребоваться типы данных, отличные от типов данных, встроенных в язык программирования С++. Введение новых типов данных преследует три цели:

- ограничить множество значений существующего интегрального типа данных (используются перечисления) ;
- определить тип, который описывает данные, имеющие сложную структуру (структуры и объединения);
- переименовать тип данных, чтобы придать ему более осмысленное имя (используется определение typedef).

Замечание: В языке программирования С++ типы данных могут объявляется в любом месте программы.

ПЕРЕЧИСЛЕНИЯ

Перечислением называется тип данных, который включает набор именованных целочисленных констант. Элементы перечисления называются перечислимыми константами.

Синтаксис:

enum имя_перечисления {список значений};

enum – ключевое слово;
имя_перечисления служит для именования типа перечисления;
список значений перечисляет именованные целочисленные константы, принадлежащие перечислению.

Примеры:

enum boolean {false, true};
enum otvet {no,yes} ;
enum color {black, red, blue, green, white};

При объявлении перечисления его элементы могут инициализироваться.

enum color {black=2, red=3, blue=5, green=6, white=8};

По умолчанию перечислимым константам присваиваются последовательные значения, начиная с нуля. Например,

enum color {black, red, blue, green, white};

Это значит, что black = 0, red = 1, blue = 2, green = 3, white = 3.
Если начиная с некоторой перечислимой константы, инициализация прекращается, то следующей перечислимой константе присваивается значение на единицу большее, чем значение предыдущей перечислимой константы. Например, enum color {black, red=3, blue, green=7, white}; Перечислимые константы имеют значения: black = 0, blue = 4, white = 8.

Переменную типа «перечисление» можно объявить следующим образом:

color d; //d – переменная типа color

Объявление типа перечисления и переменной, которая имеет этот тип, может быть объединено в одну инструкцию. Например,

enum color {black, red, blue, green, white} d;

Переменной типа перечисления можно присваивать только значения, принадлежащие её типу. Эти значения могут присваиваться переменной, используя только имена перечислимых констант. Например,

color d; d = black; или d =red;

Целочисленным переменным можно присваивать значения перечислимых переменных. Например,

int t = red; (red = 0).

Особенность: перечисляемые идентификаторы должны отличатся друг от друга, а присваиваемые им значения могут совпадать.

enum color {black=2, red=3, blue=3, green=6, white=6};

Ограничения:

- данные перечислимого типа нельзя использовать с командами ввода-вывода;
- с данными перечислимого типа нельзя выполнять арифметические операции, но можно сравнивать и присваивать.

СТРУКТУРЫ

Структура – это объединено в единое целое множество поименованных элементов разных типов под одним именем. Перечисляемые в структуре переменные называются полями.

Синтаксис:

struct имя_структуры {список полей };

struct – ключевое слово;

имя_структуры для именования структурного типа;

{список_полей}={тип1 имя_поля1; тип2 имя_поля2;…; типn имя_полян;}

Примеры:

```
struct employee {char fio[20];
                 char dol[15];
                 long zpl;
                 };
```

Описание переменных структурного типа:

- Определив структурный тип в программе можно его использовать для объявления переменных данного типа. Например,

  struct point {double x; double v;};

```
point p1, p2, p3;
```

- Переменные структурного типа могут объявляется и при определении самой структуры. Например,

```
struct point {double x; double v;} p1, p2;
```

- Структурную переменную при ее определении можно инициализировать. Значения переменных структуры перечисляют в {} в порядке описания. Например,

```
point p1={1.0, -8.3};
employee={“Иванов С.Ю.”,”инженер”, 249, 3000};
```

- Доступ к полям структуры выполняется с помощью операции точка (.)

переменная.элемент(поле)

Например,

```
p1.x=1.0;  p1.y=-8.3;
employee d1;
d1.fio=”Иванов С.Ю.”;
d1.dol=”инженер”;
d1.tab=249;
d1.zpl=3000;
```

- Для переменных одного и того же структурного типа определена операция присваивания. При этом происходит поэлементное копирование. Например,

```
p1.x = 5.6
p2.x = p1.x;  p2.x = 5.6;
```

- Допускается использование массивов – структур.
- Поля структуры располагаются в памяти последовательно друг за другом.

Пример: Дана структура точка, которая содержит в качестве элементов координаты x и y. Необходимо вычислить расстояние между точками A и B.

```
#include<iostream.h>
#include<math.h>
struct  tochka{double x; double y;};
main()
{ tochka a,b;  double dl;
  cout<<”Введите координаты первой точки”<<endl;
  cin>>a.x>>a.y;
  cout<<”Введите координаты второй точки”<<endl;
  cin>>b.x>>b.y;
  dl = sqrt((a.x – b.x)* (a.x – b.x) + (a.y – b.y)*(a.x – b.y));
  cout<<”Расстояние между точками =”<<dl<<endl;
   return 0;
 }
```

Пример: Дана структура анкета (ФИО ученика, возраст, номер школы, класс, пять оценок по различным предметам). Написать программу, которая вводит данные по структуре и вычисляет среднюю оценку по каждому ученику.

```
#include<iostream.h>
#define n 5
struct anketa{char fio[20]; int vozr,schkola, clas, otm[5];};
main()
 {anketa ank[n];
  int i, j; float sum=0, sr;
  for(i=0; i<n; i++)
       { cout<<"fio:"; cin>>an[i].fio;
         cout<<"vozr:"; cin>>an[i].vozr;
         cout<<"schkola:"; cin>>an[i].schkola;
         cout<<"clas:"; cin>>an[i].clas;
         for(j=0; j<5; j++)
              { cout<<j+1<<" ooo=";
                 cin>>an[i].otm[j];
                 sum=sum+ an[i].otm[j];
               }
        sr=sum/5;
        cout<<"sr="<<sr<<endl;
      }
 return 0;
 }
```

ОБЪЕДИНЕНИЕ

Объединением называется область памяти, используемая для хранения данных разных типов.

Синтаксис:

union имя_объединения {список полей;};

union – ключевое слово;

имя_объединения для именования типа объединения**;**

{список_полей}={тип1 имя_поля1; тип2 имя_поля2;…; типn имя_полян;}.

Пример:

union num {int i; char ch; long h;};

Описание переменных типа объединение:

- Переменная типа объединения объявляется следующим образом:

  num d;
- Либо таким образом: union num {int i; char ch; long h;} d;
- Для доступа к полям объединения используется оператор точка. Например, d.i = 12, d.h = 35000;

- Длина памяти, распределяемой компилятором под объединения, равна наибольшей из длин полей объединения. Адрес этой памяти выравнивается на границу, кратную наибольшей из длин полей объединения. Все элементы объединения имеют один и тот же начальный адрес, то есть размещаются в одном и том же участке памяти.

ПЕРЕИМЕНОВАНИЕ ТИПА

Одна из возможностей создания пользовательских типов – это переименование типа.

Синтаксис:

typedef имя_старого_типа имя_нового_типа

Примеры: typedef int integer
typedef float real

Переименование типов используется для:
- создания более коротких альтернативных имен;
- определения имен типов, которые более привычные пользователю;
- определения нового типа, который лучше описывает, как используется этот тип;
- облегчения переносимости программ.

УКАЗАТЕЛИ И ССЫЛКИ В С++

УКАЗАТЕЛИ

Мощным средством С++ является возможность осуществления непомредственного доступа к памяти. Для этой цели предусматривается специальный тип переменных – указатели. Понятие указатель связано с понятиями компьютерной архитектуры – это адрес, адресация, организация внутренней памяти.

Внутренняя память – упорядоченная последовательность байтов или машинных слов. Номер слова в памяти называется адресом.

Прямая адресация – машинное слово, которое содержит непосредственно адрес памяти.

Косвенная адресация – машинное слово, которое содержит адрес другого машинного слова.

Указатель – переменная, содержимым которой является адрес другой переменной. Указатели используют чаще всего для манипуляций с динамически размещенными объектами.

Другой подход к указателям.

Переменная – это объект, имеющий тип, имя и значение. **Имя переменной** – это адрес того участка памяти, который выделен для неё. **Значение переменной** – это содержимое этого участка памяти. Для работы с адресами переменных используются указатели.

Указатель – это переменная, значением которой служит адрес участка памяти, выделенной для объекта конкретного типа.

Основная операция для указателя – косвенное обращение к переменной (*). Переменная, адрес которой содержится в указателе называется указуемой переменной. Последовательность действий с указателем включает три шага.

Первый шаг - определение указуемых переменных и переменной указателя.

Синтаксис указателя: тип_указателя *имя указателя

Пример: int a, x; // обычные переменные (указуемые)

int *pa; // переменная – указатель на переменную a

Указатели могут ссылаться не на любые переменные, а только на переменные заданного типа. В данном примере только на переменную целого типа.

Второй шаг - связывание указателя с указуемой переменной (инициализация указателя).. Значением указателя является адрес другой переменной, т.е указатель должен быть настроен на переменную, на которую он будет ссылаться.

pa=&a; // указатель содержит адрес переменной a

Объединив два шага можно записать int *pa=&a;

Третий шаг - выполнение операций с указателями.

В любом выражении косвенное обращение по указателю интерпретируется кака переход от него к указуемой переменной с выполнением над ней всех далее перечисленных в выражении операций.

*pa = 100; эта запись эквивалентна записи a=100

x = x + *pa эта запись эквивалентна записи x = x + a

(*pa)++ эта запись эквивалентна записи a++

АДРЕСНАЯ АРИФМЕТИКА УКАЗАТЕЛЯ

Указатель даёт «степень свободы» любому алгоритму обработки данных. Если некоторый фрагмент программы получает данные непосредственно в переменной, то он может обрабатывать её и только её.

Если же данные программа получает через указатель, то обработка данных (указуемых переменных) может производиться в любой области памяти компьютера (или программы). Любой указатель в С++ потенциально ссылается на неограниченную в обе стороны область памяти (массив), заполненную переменными указуемого типа с индексацией элементов массива относительно текущего положения указателя.

Особенности:

- любой указатель потенциально ссылается на неограниченную в обе стороны область памяти, заполненную переменными указуемого типа;
- переменные в области нумеруются от текущей указуемой переменной, которая получает относительный индекс нуль. Переменные в направлении возрастания адресов памяти нумеруются положительными значениями 1, 2, 3, и т.д., а в направлении убывания – отрицательными.

p+i переместить указатель на i переменную после указуемой.

p-i переместить указатель на i переменную перед указуемой.
p++ переместить указатель на следующую переменную.
p-- переместить указатель на предыдущую переменную.

В операциях адресной арифметики транслятором автоматически учитывается размер указуемых переменных, т.е. +i понимается не как смещение на i байтов, слов и прочее, а как смещение на i указуемых переменных. При объявлении указателей необходимо указывать тип объекта, чтобы компилятору было известно, сколько байтов добавлять к адресу при увеличении указателя на единицу.

Достоинство указателя – если программа получает данные через указатель, то обработка этих данных производится в любой свободной части памяти.

УКАЗАТЕЛИ И МАССИВЫ

При объявлении массива ему выделяется память. Как только память для массива выделена, имя массива воспринимается, как указатель того типа к которому отнесены элементы массива, другими словами имя массива без индексов является адресом его первого элемента (с нулевым индексом).

Синтаксис: тип_указателя имя_указателя = &имя_массива или
тип_указателя имя_указателя = &имя_массива[0]

```
int x[4]; // объявлен массив
int *px;  // объявлен указатель
px=&x или px=x  или  px= x[0] // инициализация указателя
```

Для организации доступа к элементам массива существует два доступа. Прямой доступ – это доступ через базовые переменные и доступ через указатели: *(x + i) или *(px + i). Прибавление к указателю единицы обеспечивает переход к следующему элементу массива, независимо от типа его элементов.

При доступе к элементам массива через указатели выполняется два действия:

- фиксируется базовый адрес массива, то есть первый элемент px=&x;
- индекс i используется для вычисления смещения относительно базового адреса массива.

```
int a[4], *ptra, i;
….
ptra=a
for(i=0;i<4;i++)
cout<<"указатель ="<<i<<":"<<(ptra + i)<<"\n";
```

Двумерные массивы располагаются в памяти подобно одномерным массивам, занимая последовательные ячейки памяти.

ССЫЛКИ

Ссылка – особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается, т.е он может

использоваться просто как другое имя или псевдоним объекта. При объявлении ссылки перед её именем ставится знак амперсанд &, а сама она должна быть проинициализирована именем того объекта, на который ссылается.

Ссылка – представляют собой видоизменённую форму указателя, которая используется в качестве псевдонима (другого имени) переменной.

Ссылки могут использоваться в следующих целях:

- вместо передачи в функцию объекта по назначению;
- для определения конструктора копий;
- для перезагрузки унарных операций;
- для предотвращения неявного вызова конструктора копий при передаче в функцию по значению объекта определённого пользователем класса.

Синтаксис: тип &имя_ссылки = имя_переменной

Тип объекта, на который указывается ссылка может быть любым.

Объявление неициализированной ссылки вызовет сообщение об ошибке.

```
char let = 'a';   // символьная переменная
char &ref = let;  // ссылка на символьную переменную
```

Любое изменение ссылки повлечет за собой изменение того объекта, на который данная ссылка указывает

```
int I = 0;
int &rez = i;
rez +=10;   //  то же, что i +=10
```

Замечание:

- использование ссылок не связано дополнительными затратами памяти;
- ссылки нельзя переназначить, т.к. ссылка инициализирована адресом некоторой переменной и любое действие со ссылкой сказывается на сам объект;
- ссылаться можно только на сам объект. Нельзя объявить ссылку на тип объекта.

Можно указывать операции над ссылкой, ни одна из них на саму ссылку не действует.

```
int i = 0;
int& rez = i;
rez++;        // i увеличивается на 1
```

Здесь операция ++ допустима, но rez++ не увеличивает саму ссылку rez, вместо этого ++ применяется к целому, т.е. к переменной i.

Следовательно, после инициализации значение ссылки не может быть изменено: она всегда указывает на тот объект, к которому была привязана при ее инициализации.

```
#include<iostream.h>
int main()
```

```
{ int t = 13,
  &r = t; //инициализация ссылки на t
  //теперь r синоним имени t
   cout<<"Начальное значение t:"<< t;      //выводит 13
   r + = 10;                              //изменение значения t через ссылку
  cout<<"Конечное значение t: "<<t;       // выводит 23
  return 0;
}
```

В данном случае использовали ссылку в качестве псевдонима переменной. В этой ситуации она называется ***независимой ссылкой*** и должна быть инициализирована при объявлении. Такой способ использования ссылок может привести к фатальным и трудно обнаруживаемым ошибкам по причине возникновения путаницы в использовании переменных.

Другое применение ссылок – возможность создания функции с параметрами, передаваемыми по ссылке.

```
#include<iostream.h>
void fun_sqr(int &x); // прототип функции
int main()
{  int t = 3;
   cout<<"Начальное значение t:"<< t;       //выводит 3
   fun_sqr(t);
   cout<<"Конечное значение t: "<<t;       // выводит 9
   return 0;
}
void fun_sqr(int &x)
{x * = x;}
```

В приведённой программе при объявлении параметра – ссылки указывается, что в функцию будет передаваться адрес переменной. Это даёт возможность модифицировать аргумент (фактический параметр), не указывая в явном виде при вызове функции передачу в неё адреса. Другой особенностью является то, что внутри функции отпадает необходимость в использовании операции разыменования * при обращении к значению.

Нельзя возвращать ссылку (указатель) на локальную переменную, так как эта переменная перестаёт существовать в момент возврата из функции.

При применении переменных ссылочного типа накладывается ряд ограничителей:

- нельзя взять адрес переменной ссылочного типа;
- запрещается использовать массивы ссылок;
- не допускается использовать ссылки на битовые поля;
- нельзя создавать указатель на ссылку.

СИМВОЛЫ В ЯЗЫКЕ С+ +

Для представления текстовой информации в языке С+ + используются символы (константы), символьные переменные и строки (строковые константы) для которых в языке С+ + не введено отдельного типа в отличии от некоторых других языков программирования.

Базовый тип для символов и строк ***char***

char a, z;
a = ‘b'; } символьные
z = 'd'; } константы

Строки в языке С+ + - это последовательность символов, заключенная в кавычки. Строка - это массив символов, т.е. она всегда имеет тип char.

char str [0] = «среда»;
char str [10] = {‘c', 'р', 'е', 'д', 'а', '\0'}

Особенность строк:

1. Транслятор, размещая строку в памяти, автоматически добавляет в нее символ конец строки '\0', т.е. нулевой байт. Количество элементов в таком массиве на единицу больше, чем в изображении соответствующей строковой константы.

   Для строки «А» длина = 2 байта

   ля символа 'А' длина = 1 байт

2. При инициализации массива отдельными символами обязательно в конце массива указывается символ '\0'; char str [10] = {‘c', 'р', 'е', 'д', 'а', '\0'}
3. char str [80] = {' '}; инициализация символьного массива пробелами

   по аналогии mas [10] = {0};

ВВОД И ВЫВОД ТЕКСТОВОЙ ИНФОРМАЦИИ

Процедуры ввода - вывода символа

int ch;
ch = getch ( );- ввод кода нажатой клавиши без отображения соответствующего символа на экране

ch = getche ( );- ввод нажатой клавиши с соответствующего символа на экране

ch = getcrar ( );- ввод кода нажатой клавиши вслед за нажатием клавиши enter

Вывод символа : putchar (C1)

Процедуры ввода - вывода строки: gets (str) , puts (str)

```
char str [80];  // объявление строки
cout << «введите строку»;
gets (str);  // ввод строки
puts (str);  // вывод строки
```

СТАНДАРТНЫЕ ПРОГРАММНЫЕ РЕШЕНИЯ

1. Получить символ десятичной цифры из значения переменной, лежащей в диапазоне 0…9

```
int n;
char c;
c = n + ‘0';
```

2. Получить значение целой переменной из символа десятичной цифры

```
if ( c >= '0' && c <= '9');
n = c - '0';
```

3. Получить символ шеснадцатиричной цифры из значения целой переменной, лежащей в диапазоне 0…15

```
if (n <= 9)  c = n + '0';
else  c = n- 10 + 'A';
```

4. Преобразовать строчную латинскую букву в прописную:

```
if (c >= 'a' && c <= 'z')  c = c- 'a' + 'A';
```

СТРОКИ <string.h>

Для работы со строками необходимо использовать заголовочный файл s**tring.h**

Функции для работы со строками

1. **Определение длины строки strlen ( )**

```
char str [ ] = «0123456789»
int l = strlen (str);
cout << «l = » << l;        l =11
```

Завершающий нулевой символ в длину строки входит.

2. **Копирование строк strcpy ( ), strncpy ( )**

Синтаксис: char strcpy (str1, str2)

Выполняется побайтное копирование символов из строки str2 в строку str1. Копирование прекращается только в случае достижения символа '\0' (нуль-терминатор). Перед копированием необходимо проверить выполнение условия, что длина str2 меньше или равна длине str1. В противном случае возможно возникновение ошибок, связанных с наложением данных.

Пример: char str [20];
strcpy (str, «Проверка копирования»);

Копирование через указатели и копирование не всей строки

```
char str1[20] = «Проверка копирования»;
char str2[20];
char *ptr = str1;
ptr + = 9, // ptr указывает на слово «копирование»;
strcpy (str2, ptr);
cout << str2<< '\n';
```

Функция **strncpy ( )** копирует n символов из строки S2 в строку S1.

Пример: char str [40]
strncpy (str, “Borland C+ +”,7)
puts(str); // Borland

3. **Конкатенация (или присоединение) строк strcat ( )**
strncat ( ) – присоединение n символов из другой строки

Синтаксис: strcat (str1, str2)

Строка str2 присоединяется к строке str1. Величина str1 должна быть достаточной для хранения объединенной строки.

Пример: char str [80];
strcpy (str, «Для продолжения»);
strcat (str, «Нажмите клавишу»);

Синтаксис: strncat (str1, str2, n)

Пример: char str [80]; = “Языки программирования”;
char str [30] = “ C+ +, Паскаль, Бейсик”;
strncat (str1, str2, 13);
puts(str1); // Языки программирования: C+ +, Паскаль

4. **Сравнение строк strcmp ( )**

Функция имеет тип int, так как данная функция выполняет сравнение двух строк str1 и str2, различая прописные и строчные буквы, в результате сравнения возвращает одно из следующих значений:

Если результат функции < 0, то str1 < str2.

Если результат функции = 0, то строки эквивалентны

Если результат функции > 0, то str1 > str2

Пример: str1 [ ] = «Borland»
str2 [ ] = «BORLAND»
int i;
i = strcmp (str1, str2); // i = 32

Распишем АСКИ-коды заданных строки и сравним их

Borland	66	111	114	108	97	110	100
BORLAND	66	79	82	76	65	78	68

Разница между кодами символов строк = 32, результат больше 0, значит строка str1> str2

Функция strncmp( ) проводит сравнение определенного числа первых символов двух строк. Регистр символов при этом учитывается.

char str1 [ ] = «Ошибка открытия базы»;
char str2 [ ] = «Ошибка открытия базы»;
int i
i =strncmp (str1, str2, 12)
строки отличаются одним символом
i > 0; str1 > str2;

Функция **stricmp()** сравнивает строки, не различая регистра символов. Возвращается одно из следующих значений: i > 0, i = 0, i < 0;

char str1 [ ] = «Moon»; char str2 [ ] = «MOON»;
int i = strincmp (str1, str2);
i=0, строки эквивалентны

5. **Преобразование строк: strlwr ( ), strupr ( ), strrev ( )**

strlwr ()- преобразует прописные буквы в строчные

Пример: char str [ ] = «HELLO»;
strlwr (str);
cout << str // hello;

strupr ()- преобразует строчные буквы в прописные

strrev ()- реверсирование строки, т.е. изменяет порядок следования символов на обратный.

Пример: char str [ ] = “сон”;
strrev (str); // нос

6. **Поиск символов**

strchr() - производит поиск символа в строке, указывает место первого вхождения символа в строку. Если символ не найден, функция возвращает NULL.

Пример: char str [ ] = «абвгдеёжзийк»;
char *pstr;
pstr = strchr (str, 'ж');

В результате работы программы указатель pstr будет указывать на подстроку «жзийк» в строке str.

strrchr() - возвращает указатель на последний, совпадающий с заданным символом, символ в строке. Если символ не найден, возвращается значение NULL.

Пример: char str [ ] = «абвгджизийк»;
char *pstr;
pstr = strrchr (str, 'и'); // ийк

strspn() - проводит сравнение символов одной строки с символами другой и возвращает позицию (начиная с нуля), в которой строки перестают совпадать.

Функция проверяет каждый символ строки str на соответствие каждому из символов строки group. В результате работы, функция возвращает число совпавших символов.

Пример: char str [ ] = «Загрузка параметров БД»;
char substr [ ] = «Загрузка параметрррр»;
int i = strspn (str, substr);
cout << i; // i=17

символы совпадают до 17 позиции.

Приведенная функция различает регистр символов.

<u>strcspn ( )-</u>

<u>strcspn ( )</u>- сопоставляет символы строки str1 и str2 и возвращает длину строки str1, не входящей в str2. С помощью этой функции можно определить, в какой позиции происходит перекрещение двух символьных массивов.

Пример: char str [ ] = «abcdefghijk»;
int k;
k = strcspn (str, «elf»);

k = 4 - в этой позиции строки имеют первый общий элемент.

strpbrk() - отыскивает место вхождения в строку str1 любого из символов строки str2. Если символы найдены, возвращается место первого вхождения любого символа из str2 в строку str1. В противном случае функция возвращает NULL.

Пример:
```
char str1 [ ] = «abcdefghjk»;
char str2 [ ] = «esb»;
char *ptr;
ptr = strpbrk (str1, str2);
cout << ptr;
```
bcdefghjk, т.к. символ 'b' из строки str2 встречается в строке str1 раньше других.

7. **Поиск подстрок**

strstr ()- данная функция осуществляет сканирование строки str1 и находит место первого вхождения подстроки str2 в строку str1. В случае успешного поиска функция strstr() возвращает указатель на первый символ строки str, начиная с которого следует точное совпадение части str1 обязательно со всей лексемой str2. Если строка str2 не найдена в str1, возвращается NULL.

Пример:
```
char str1 [80] = «Производится поиск элемента»;
char str2 [80] = «Поиск»;
char *ptr;
ptr = strstr(str1, str2);
cout << ptr ;  // поиск элемента.
```

strtok() - выполняет поиск в строке str подстроки, обрамленной с обеих сторон любым символом - разделителем из строки str1.

strtok (str, str1);

В случае успешного поиска данная функция обрезает строку str, помещая символ ‘\0’ в месте, где заканчивается найденная лексема. При повторном поиске лексемы в указанной строке str первым параметром следует указывать NULL.

Пример:Разбить предложение на лексемы, которые выводить с новой строки.
```
# include <iostream.h>
# include <string.h>
void main ( )
  { char str [80] = «Компилятор; компилирует: программу, в машинных кодах»
    char str1 [10] = “  ; : ,”
    char *ptr;
    ptr = strtok (str, str1);
        // первый вызов функции strtok() выделяет первую лексему (слово) и
    // ставит ‘\0’  в конце лекскмы
   while (ptr! = NULL) // пока не конец строки (в конце строки ‘\0’  )
   { cout << ptr << ‘\n’;
     ptr = strtok (NULL, str2);
    // второй вызов функции strtok() - последующий поиск лексем,
    //  которые начинается с нулевого символа.
    }
   }
```

ЛИТЕРАТУРА

1. Подбельский В.В. Язык С++:Учебное пособие. – М.:Финансы и статистика, 1999. -560 с.
2. Побегайло А.П. С/С++ для студента. – СПб.:БХВ-Петербург, 2006. – 528 с.
3. Глушков С.В., Коваль А.В., Смирнов С.А. Язык программирования С++: Учебный курс. – Харьков:Фолио: м.: ООО «Издательство АСТ»,2001. -500 с.