ФЕДЕРАЛЬНОЕАГЕНТСТВОСВЯЗИ

ФЕДЕРАЛЬНОЕГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕОБРАЗОВАТЕЛЬНОЕУЧРЕЖДЕНИЕВЫСШЕГООБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙУНИВЕРСИТЕТТЕЛЕКОММУНИКАЦИЙ
им. проф. М. А. БОНЧ-БРУЕВИЧА»
(СПбГУТ)

А. Н. Кривцов

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА C/C++

УЧЕБНОЕ ПОСОБИЕ

СПб ГУТ)))

САНКТ-ПЕТЕРБУРГ

2018

Оглавление

ЧАСТЬ 1. ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ. ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ С/С++

1 ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

1.1 Основы программирования

1.1.1 Общие понятия и определения элементов технологии программирования

В толковом словаре по вычислительным системам [Р1] приводится ряд определений, которыми мы будем руководствоваться при изучении раздела «Организация процесса разработки программных средств». Рассмотрим некоторые из них. Так, например, опираясь на определение информационной технологии [Р1: I.083], можно дать такое определение *технологии программирования*.

Определение. Под ***технологией программирования*** мы будем понимать методы разработки программ в совокупности с аппаратным и программным обеспечением для реализации задачи на персональном компьютере.

Определение *программы* в толковом словаре звучит так:

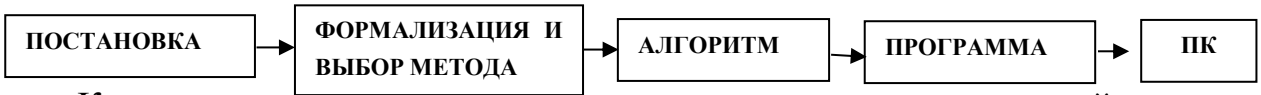
Определение [Р1: Р.256]**.** ***Программа*** – это набор операторов (команд), который может быть представлен как единое целое в некоторой вычислительной системе и который используется для управления поведением этой системы.

В дальнейшем под вычислительной системой мы будем понимать персональный компьютер (ПК). При этом следует различать понятия «Программа» и «Программирование». Суть понятия *программирования* не заключается только в написании программы.

Определение [Р1: Р.277]**.** В широком смысле, ***программирование*** – это все технические операции, необходимые для создания программы, включая анализ требований и все стадии ее разработки и реализации.

По-существу, прежде чем получить решение задачи на ПК, ее необходимо подготовить для этого. Процесс подготовки предусматривает ряд важных этапов. Рассмотрим их в общем виде.

Решение задачи на ПК сводится к выполнению программы, введенной в память ЭВМ. Составлению программы предшествует разработка алгоритма решения задачи. Алгоритм реализует метод решения. Выбору метода решения предшествует анализ и формализация целевой задачи. Таким образом, процесс подготовки задачи к решению на ПК можно представить в виде такой обобщенной схемы:



Кратко рассмотрим содержание каждого представленного на ней этапа.

1) Постановка задачи

Постановка задачи содержит описание целей и условий ее решения.

Например, *цель решения задачи целераспределения* – назначить свои средства поражения по целям противника так, чтобы обеспечивался максимум числа уничтоженных целей, принимающих участие в нанесении удара по объекту. *Цель задачи распределения ресурсов* - минимизировать затраты при их распределении или достичь максимального эффекта от их распределения.

Иногда впервые поставленная задача требует разработки новых концепций или теорий, но гораздо чаще ситуация описывается в терминах уже сложившихся понятий.

2) Формализация задачи и выбор метода решения

Для обеспечения возможности решения задачи на ЭВМ она должна быть выражена так, чтобы было ясно, как ее реализовать на ПК, и каким методом реализации при этом воспользоваться. ***Для формализации задачи указываются исходные данные, начальные условия*** *и необходимая* ***точность вычисления****, а также некоторые* ***ограничения*** (например, на время решения).

Часто понятие формализации связывают с последовательным выражением условий задачи в математических терминах (формулах). Однако не редко приходится решать задачи, которые формально представить невозможно или крайне сложно. К такой области относятся, например, лингвистические задачи. В таких случаях под формализацией задачи можно понимать ее логическую интерпретацию с целью получения возможного конструированного решения на ПК. Возможны случаи, когда задача с самого начала возникала как математическая (например, расчет по известным формулам). Считается, что такие задачи не требуют формализации.

Выбор того или иного метода зависит от типа решаемой задачи.

Так, например, при решении систем линейных алгебраических уравнений различают три типа систем:

– с положительно определенными симметричными матрицами;

– с симметричными, но не положительными матрицами;

– с произвольными матрицами.

Метод решения для последнего типа значительно сложнее двух предыдущих и требует для своей реализации на ПК больших затрат машинного времени и объема оперативной памяти. Однако он является универсальным и может быть использован для любого типа системы, т.е. при выборе математического метода, кроме типа задачи, необходимо также учитывать и такие характеристики ПК, как быстродействие и объем памяти.

Часто задача бывает такова, что ее точное решение получить невозможно. Можно получить только какое-либо приближенное решение с допустимой погрешностью. Например, какая точность попадания ракеты в цель будет достаточной для уничтожения цели? Или, какое количество передающих антенн, их местоположение, высота и др. должно быть установлено, чтобы мобильная связь в данном регионе была устойчивой? Или, с какой точностью (сколько значащих цифр после запятой) нужно получить результат деления числа 1 на число 3? Для решения такого класса задач существуют разнообразные численные методы. Численные методы рассматриваются в вычислительной математи-

ке. Они позволяют свести решение любой задачи к последовательному выполнению четырех арифметических действий и операций. Например, для решения систем обыкновенных дифференциальных уравнений первого порядка существует несколько численных методов решения (Адамса, Хэмминга, Рунге-Кута и др.), которые отличаются временем решения и точностью. Из численных методов выбирается тот, который обеспечивает решение предъявленных к задаче требований при заданных ограничениях на ресурсы.

3) Разработка алгоритма программы.

Формализация задачи позволяет выводить ***точные логические следствия из известных нам данных об исходном и конечном состоянии***. Сравнительно редко путь, который ведет от исходного состояния к конечному состоянию, бывает сразу ясен. Как правило, он намечается в результате логического анализа формальной постановки задачи, выявленных новых свойств и отношений между понятиями и объектами, о которых в ней идет речь. Сама задача разбивается на элементарные шаги и действия, выполняемые в дальнейшем ПК. Этот путь решения задачи – последовательность действий, которые необходимо выполнить, чтобы от исходного состояния прийти к желаемому новому состоянию – называется алгоритмом решения задачи.

Более точное определение *алгоритма* в толковом словаре звучит так.

Определение [Р1: А.083]. ***Алгоритм*** – это заранее заданная последовательность четко определенных правил или команд для получения решения задачи за конечное число шагов.

Алгоритм решения задачи может быть записан и предъявлен различными способами и с различной степенью детализации. Для представления алгоритма в виде, понятном для ПК, служат языки программирования.

4) Написание программы и подготовка ее к вводу в ЭВМ.

Целью данного этапа является ***запись алгоритма на языке программирования*** и перенос текста программы на носитель, с которого она может быть введена в ПК.

Определение [Р1: Р.278]. ***Язык программирования*** (ЯП) – это система обозначений, служащая для точного описания программ или алгоритмов для ПК.

Языки программирования являются искусственными языками, в которых синтаксис и семантика строго определены. Поэтому при применении их по назначению они не допускают свободного толкования выражения, характерного для естественного языка.

Если ЯП выбран, то и запись алгоритма на нем, и перенос текста программы на носитель можно совместить, работая в интегрированной среде программирования данного ЯП. При выборе ЯП необходимо руководствоваться не только критерием простоты написания программы и сокращением сроков ее отладки. В тех случаях, когда создаются программы, предназначенные для многократного их использования, более важно обеспечить их высокую эффективность: точность вычислений, быстродействие, многофункциональность, совместимость и другие критерии.

6) Отладка программы и ее выполнение на ПК.

Основная цель данного этапа – ***выявление и исправление ошибок в программе**, и получение конечного результата решения задачи*.

Определение. Процесс поиска ошибок в программе называется ***тестированием*** программы, процесс устранения ошибок – ***отладкой*** программы.

По существующим оценкам на отладку программы программист затрачивает до 40% времени, отводимого на ее разработку. Кратко, этап выполнения программы на ПК состоит из трех шагов: *трансляции* программы, ее *редактировании* и ее *выполнении*. Эти шаги могут повторяться неоднократно при отладке программы до тех пор, пока получаемые результаты не будут соответствовать предъявляемым к задаче требованиям.

Для создания программы, на каком-либо выбранном ЯП, необходимо иметь интегрированную среду программирования, включающую:

1. *Текстовый редактор*. Хотя можно использовать любой текстовый редактор для написания исходного кода программы, лучше использовать специализированный, встроенный в систему редактор, так как он настроен под синтаксис данного ЯП. *Назначение такого редактора – получение файла с исходным текстом программы*.

2. *Программу – компилятор*. Как было сказано выше, такая программа переводит исходный текст программы, набранной в стиле ЯП, в машинный код. Если, при этом, будут обнаружены синтаксические ошибки, то результирующий код создан не будет. Т. е. *задача компилятора – создание некоего промежуточного кода программы, «понятного» для процессора*. Такой код принято называть объектным кодом. Как правило, в результате процесса такого перевода создается файл с двоичными данными, который имеет расширение **.obj**.

3. *Программу – редактор связей* (сборщик). Исходный текст большой программы можно представлять в виде нескольких модулей. Каждый модуль компилируется в отдельный файл с объектным кодом, эти файлы затем надо объединить в одно целое. К ним добавляется машинный код подпрограмм, реализующих стандартные функции, которые содержаться в различных специальных файлах, например, – в библиотеках, имеющих расширение **.lib**. *Редактор связей «связывает» объектные модули и библиотеки и формирует на выходе «исполняемый» код для конкретной платформы.* Исполняемый код – это законченная программа. Итоговый файл этой программы имеет расширение **.exe** или **.com**.

Таким образом, от написания исходного кода программы до получения исполняемого файла, требуется пройти ряд специфических этапов. Для этого процесса необходима специализированная среда реализации исходной задачи.

Определение. Система, состоящая из:

- специализированного текстового редактора;
- компилятора;
- редактора связей;
- библиотек функций

называется ***интегрированной системой программирования*** (ИСП).

В современных ИСП есть еще отладчик, позволяющий анализировать программу по шагам ее исполнения.

Из сказанного выше следует различать такие понятия, как «исходный текст программы на алгоритмическом ЯП», «объектный код программы» и «программа – исполняемый код».

Как правило, при подготовке задачи к решению на ПК пункты 1)-2) вызывают наибольшие затруднения. И это вполне объективно, т.к. от степени их проработки во многом зависит и качество конечного решения. Поэтому, для более полного уяснения, рассмотрим эти этапы на конкретном элементарном примере.

1. Постановка задачи

Вычислить наименьшее значение функции вида $f(x) = x^2 - x + e^x$ на отрезке [*a, b*] и определить точку, где эта функция достигает минимального значения. Задача должна быть выполнена на ПК под управлением MS Windows. При организации программы вывод наименьшего значения функции и вывод точки минимума оформить в виде подпрограммы.

2. Формализация и выбор метода решения

1) Функция непрерывна (в ней нет точек разрыва), значит отрезок [*a, b*] может быть любым.

2) Т.к. искомое решение требуется найти в точке $x \in [a, b]$, то необходимо определить шаг h и количество точек n_i для сравнения значений функции в этих точках.

3) Начальные данные: отрезок [*a, b*]; количество точек n; шаг поиска решения $h = \frac{b-a}{n}$; начальная точка $x_0 = a$.

4) Решение:

- определить $f(x_0) = x_0^{\,2} - x_0 + e^{x_0}$ – значение функции в начальной точке;
- определить следующую точку x_1 по формуле $x_1 = x_0 + h$;
- определить $f(x_1)$;
- сравнить $f(x_0)$ и $f(x_1)$ и выбрать из них наименьшую $f_{min}(x)$, запомнить значение минимальной точки x_{min} - это или x_0, или x_1;
- определить точку x_2: $x_2 = x_1 + h$;
- сравнить $f(x_2)$ и $f_{min}(x)$ и выбрать наименьшую из них, запомнив значение точки, где $f(x) = f_{min}$;
- продолжить выбор и сравнение, пока $f(x_i) \le f(b)$;
- вывести f_{min} и x_{min}.

5) известно из математики, что для решения подобных задач существуют методы типа: «пузырьковый (последовательный перебор)», «деление отрезка пополам», «золотого сечения», «парабол» и др. Так как в постановке задачи не сказано о точности, то выбирается любой, например, «пузырьковый» метод;

6) условия задачи могут быть реализованы на любом алгоритмическом языке программирования, ИСП которого удовлетворяет системным требованиям задачи, например, на Visual C++ любой версии.

Любую программу на АЯ характеризуют различные показатели ее качества. К ним можно отнести, например, наглядность или читаемость исходного текста программы, сложность алгоритма, скорость выполнения задачи, точность полученных результатов и т.д. Некоторые из этих показателей достигаются за счет выбора способа процесса разработки программы – технологического подхода к программированию. Существует несколько подходов. Если для реализации программы выбран какой-либо алгоритмический язык, то для реализации программ на нем наиболее популярны, в частности, такие подходы, как:

- *модульное* программирование;
- *структурное* программирование;
- *объектно-ориентированное* программирование.

Мы не будем рассматривать модульное программирование и программные средства его реализации, так как это не входит в программу нашего курса.

Отметим только его сущность, ссылаясь на информацию, приведенную в **http://ru.wikipedia.org/wiki .**

Модульность в языках программирования принцип, согласно которому программное средство (ПС, программа, библиотека, веб-приложение и др.) разделяется на отдельные именованные сущности, называемые *модулями*. Модульность часто является средством упрощения задачи проектирования ПС и распределения процесса разработки ПС между группами разработчиков. При разбиении ПС на модули для каждого модуля указывается реализуемая им функциональность, а также связи с другими модулями.

Программный код часто разбивается на несколько файлов, каждый из которых компилируется отдельно от остальных. Такая модульность программного кода позволяет значительно уменьшить время перекомпиляции при изменениях, вносимых лишь в небольшое количество исходных файлов, и упрощает групповую разработку.

Более подробно рассмотрим суть второго и третьего подходов: структурное и объектно-ориентированное программирование.

1.1.2 Понятия структурного программирования

В основу структурного программирования положены следующие достаточно простые положения:

1. Программа должна составляться мелкими шагами. Размер шага определяется количеством решений, применяемых программистом на этом шаге.

2. Сложная задача должна разбиваться на достаточно простые, легко воспринимаемые части, каждая из которых имеет только один вход и один выход.

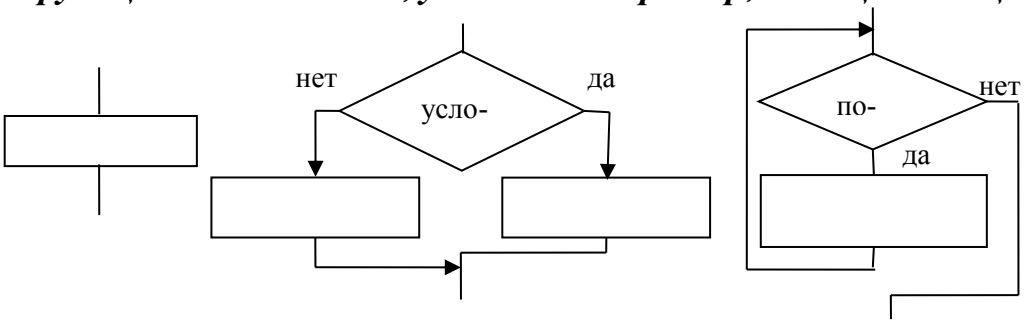
3. Логика программы должна опираться на минимальное количество достаточно простых базовых управляющих структур, подобно тому, как любая функция алгебры логики может быть выражена через функционально полную систему (например, дизъюнкцию, конъюнкцию, отрицание).

Использование этих положений позволяет внести определенную систему в труд программиста и составлять удобочитаемые программы, которые можно изучать и проверять, последовательно читая небольшие однострочные сегменты программного текста.

***Первым**, фундаментальным **принципом** структурного программирования является доказанная теорема о структурировании*. Эта теорема устанавливает следующее.

Теорема. Как бы сложна не была задача, блок-схема соответствующей программы всегда может быть представлена ограниченным числом элементарных ***управляющих структур*.**

Доказано, что такими *элементарными управляющими структурами* являются ***функциональный блок, условный оператор, обобщенный цикл*.**



Функциональный блок можно представитm как отдельный вычислительный оператор или как любую последовательность операторов с единственным входом и единственным выходом.

Условный оператор предполагает выбор одной из альтернатив в зависимости от выполнения или невыполнения заданного условия.

Обобщенный цикл представляет собой выполнения какого-либо действия до тех пор, пока выполняется некоторое логическое условие

Нетрудно видеть, что конструкция ***условного оператора*** и ***обобщенного цикла*** также может быть сведена к блоку с одним входом и одним выходом, т.е. к функциональному блоку. Поэтому любая программа, состоящая из таких конструкций, может быть сведена к программе с последовательной структурой, в которой последовательность передачи управления соответствует расположению функциональных блоков в тексте программы.

Действительно, любую программу можно представить, в общем виде,

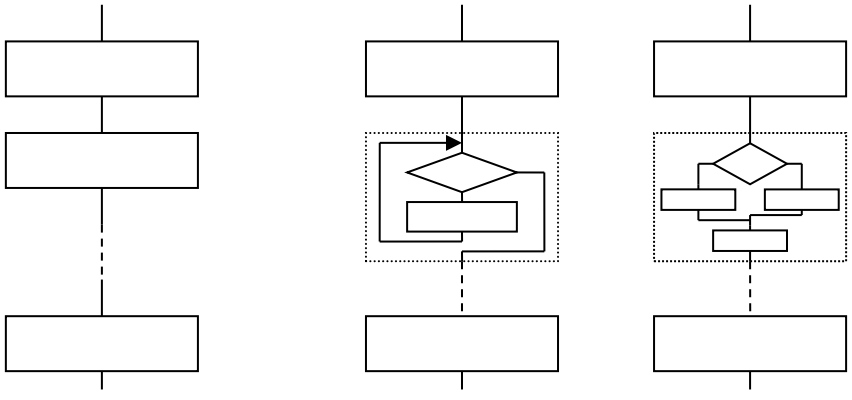


Рис. A Рис. B Рис. C

как последовательность функциональных блоков (рис. А). В то же время, любой функциональный блок можно детализировать, используя вместо него, например, оператор обобщенного цикла (рис. В: заменен блок 2), или условный оператор (рис. С: заменен блок 2).

Реализация этого (первого) принципа при программировании, предполагает использование либо непосредственно исполняемых в линейном порядке выражений (например, выражений над арифметическими или строковыми переменными), либо одной из следующих конструкций:

- вызов процедур (процедур-подпрограмм или процедур-функций) и встроенных функций – любое обращение к конструкции с одним входом и одним выходом;
- вложенные на произвольную глубину условные операторы;
- циклические структуры.

Заметим, что вызовы процедур не ухудшают наглядности и понятности программы, так как это управление передается в одну и ту же область, а после завершения выполнения в точку, следующую непосредственно за оператором вызова. В то же время при использовании оператора безусловного перехода, в общем случае, управление передается каждый раз в новую точку программы, и поэтому трудно контролируются.

Допустимы также следующие расширения указанных выше конструкций:

- использование конструкций типа «Выбор одного из многих», допустимых в некоторых «алголоподобных» языках программирования (в частности, в Паскале или Си);
- использование конструкций цикла, в котором, сначала, выполняется функциональный блок, а затем – проверяется необходимость его повторения;
- использование процедур с несколькими точками входа и несколькими выходами.

Вторым принципом *структурного программирования является исключение, где это возможно, оператора безусловного перехода* (“go to”). Наихудшим применением этого оператора безусловного перехода считается переход на оператор, расположенный ранее в тексте программы.

Третий принцип *структурного программирования относится к стилю оформления программы*, в частности:

- текст вашей программы должен быть напечатан с правильными сдвигами, чтобы разрывы в последовательности выполнения легко прослеживались и легко угадывались операторы начала и конца блоков или готовых модулей.

Обобщая сказанное, можно утверждать, что в структурном программировании важна форма написания программы и дисциплина ее написания при соблюдении определенных правил, а также программист, способный создать ясную и доступную программу из базовых логических конструкций.

1.1.3 Понятия объектно-ориентированного программирования

При структурном подходе к программированию, решаемая задача разбивается на множество простых составляющих, имеющих один вход и один вы-

ход и состоящих из основных (базовых) структурных конструкций.

При этом данные отделены от функций над ними и передаются в отдельные части всей программы (в подпрограммы) в качестве параметров, или подпрограммы используют данные, специфичные для себя и независимые от других подпрограмм.

Естественно, что в процессе развития программирования, закономерно возник вопрос об организации данных в программе таким образом, чтобы *связать данные с обрабатывающими их функциями* в единое целое – ***объект***.

Например, пусть объектом является такое насекомое, как бабочка. Все, что мы знаем об этой бабочке: ее цвет, форму, название, способность летать, пить нектар и т.д., включается в этот объект, как его *свойства*. А все возможные действия над этим объектом реализуются как его методы (функции), например:

объект:	«бабочка»:
свойства:	«бабочка летит», …;
методы:	«летит куда?» – вверх; «летит как?» – быстро.

Иными словами, для объекта существенно и то, что он на самом деле собой представляет, и то, что он «умеет».

Объекты могут быть абстрагированы (обобщены) в более широкие *классы*, например:

- стул, стол, шкаф... – класс «мебель»,

или в подклассы, например:

- табуретка, трон, кресло ...– подкласс «стул».

Каждому классу (подклассу) могут быть приписаны свои свойства (*атрибуты*), например,

- «трон» больше «табуретки»,

и свои способы поведения (*функции*), например,

- «шкаф для хранения одежды», или «письменный стол».

Следовательно, можно дать такое определение.

Определение. ***Объектно-ориентированный подход*** (ООП) к программированию представляет собой технологию программирования, в основе которой лежит подход, позволяющий формировать модели объектов реального мира.

Объектно-ориентированный подход к программированию основывается на понятиях *инкапсуляции, наследовании и полиморфизме*.

Определение. ***Инкапсуляция*** – слияние данных и функций, работающих с этими данными, которое порождает абстрактные типы данных, определенные пользователем.

Эти типы данных и называются *классами*.

Определение. ***Класс*** – это объект, в который заключаются данные и оперирующие с ними функции.

Таким образом, класс состоит из спецификаторов:

- ***членов–данных*** – данные необходимые для представления объектов этого типа;
- ***членов–функций*** – функции представляющие операции для работы с этими объектами.

Данные, принадлежащие объекту некоторого класса, обуславливают состояние данного объекта, а набор членов-функций – поведение объектов класса.

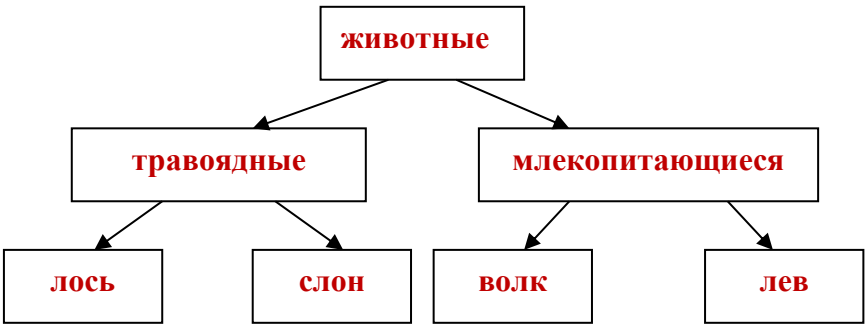
Формальное описание класса, например, на языке СИ, имеет вид:

<**Ключевое слово**> [имя класса]
{
[объявление полей данных;]
[список членов]
[прототипы функций-членов;]
[определение функций-членов;]
}

Определение. ***Наследование*** – свойство, при котором, новый объект приобретает атрибуты и формы поведения ранее определенных объектов.

При помощи наследования создается *иерархия* классов, формирующая основу программы пользователя. Верхние уровни такой иерархии обладают наиболее общими свойствами. Более низкие уровни обладают более специфичными и более детализированными свойствами. В то же время, каждый объект нижнего уровня наследует свойства объектов, стоящих по иерархии выше его.

Например, рассмотрим такой аналог иерархического дерева классов.



Объекты и лось, и слон, и волк, и лев хотя и относятся к разным подклассам (травоядные и млекопитающиеся), но все они принадлежат одному (верхнему) классу – животные.

Определение. ***Полиморфизм*** – способность объекта реагировать на некоторый запрос сообразно своему типу.

Это свойство, при котором одна и та же функция или оператор могут иметь несколько значений, в зависимости от контекста, в котором они используются. Например:

- «бабочка летит» КАК? – быстро, красиво, плавно,…

но одновременно можно и так:

- «бабочка летит» КУДА? – на север, к цветку, вверх,…

В таких общих представлениях, трудно прокомментировать все способы представления на АЯ свойств, характерных для ООП в программировании. Для каждого конкретного ЯП их представление в синтаксической конструкции этого ЯП различно.

Мы не ставим целью изучение каких-то конкретных возможностей конкретного ЯП для апробации его возможностей в технологии ООП. Однако следует подчеркнуть, что данная технология очень эффективна при разработке больших и сложных программных систем, например, приложений пакета MS Office.

1.2 Примеры задач для разработки программ методом структурного программирования

Задача 1.

Условие. Дан треугольник со сторонами, величины которых соответственно равны числам *a, b, c***.** Определить высоты этих сторон. Значения сторон ввести в ПК с клавиатуры, а вывод их высот после вычислений осуществить на экран монитора.

Примерное решение.

1. Формализация и выбор метода.

1) Начальными значениями являются величины сторон *a, b, c***.** В постановке задачи не сказано, какого типа эти числа – целые или вещественные (действительные). Так как целые числа являются подмножеством вещественных чисел, то целесообразно приписать значениям *a, b, c* вещественный тип.

2) Известно из математики, что для нахождения высот треугольника используют формулы:

$$h_a = \frac{2S}{a}; \quad h_b = \frac{2S}{b}; \quad h_c = \frac{2S}{c}.$$

Здесь S – площадь треугольника, которая вычисляется по формуле Герона: $S = \sqrt{p(p-a)(p-b)(p-c)}$, а величина p является полупериметром и вычисляется по формуле: $p = \frac{a+b+c}{2}$.

3) Из математического метода решения следует, что для программной реализации необходимо определить еще пять величин: для идентификации высот – пусть это будут соответственно $h1$, $h2$, $h3$; для идентификации площади – величину S; для идентификации полупериметра – величину p. Так как при их вычислении основу составляют числа *a, b, c*, тип которых вещественный, то и тип всех остальных чисел также должен быть вещественным.

2. Структурная блок-схема алгоритма.

Из формализации следует следующая последовательность действий:

Шаг 1. Определяются величины, участвующие в вычислениях и их тип;

Шаг 2. Водятся числа *a, b, c*

Шаг 3. По формуле $p = \frac{a+b+c}{2}$ вычисляется полупериметр;

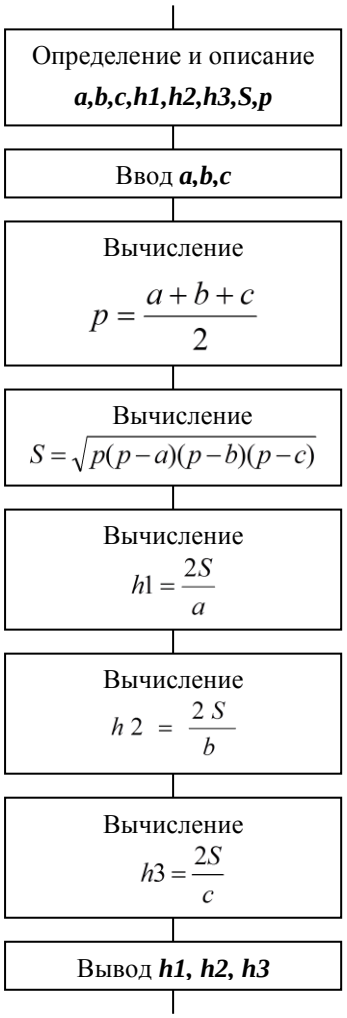
Шаг 4. По формуле Герона вычисляется площадь треугольника: $S = \sqrt{p(p-a)(p-b)(p-c)}$;

Шаг 5. По формуле $h1 = \frac{2S}{a}$ вычисляется высота стороны *a*;

Шаг 6. По формуле $h2 = \frac{2S}{b}$ вычисляется высота стороны *b*;

Шаг 7. По формуле $h3 = \frac{2S}{c}$ вычисляется высота стороны *c*;

Шаг 8. Значения величин *h*1, *h*2, *h*3, согласно условию задачи, выводятся на экран.



Из алгоритма следует, что каждый его последующий шаг – это отдельное вычислительное, зависимое только от значений, полученных на предыдущих шагах. Ни на одном из шагов нет действий выбора по условию или возврата по условию к ранее пройденным шагам. Следовательно, блок-схему реализации данного алгоритма можно составить, используя только базовую управляющую структуру типа «функциональный блок». Такая блок-схема показана на рисунке справа.

Задача 2.

Условие. Определить и вывести на экран дискриминант решения квадратного уравнения вида: $y=ax^2+bx+5$, если его значение положительно. Значения величин *a* и *b* задаются с клавиатуры. Если величина дискриминанта отрицательна, то на экран выводится сообщение об этом.

Примерное решение.

1. Формализация и выбор метода.

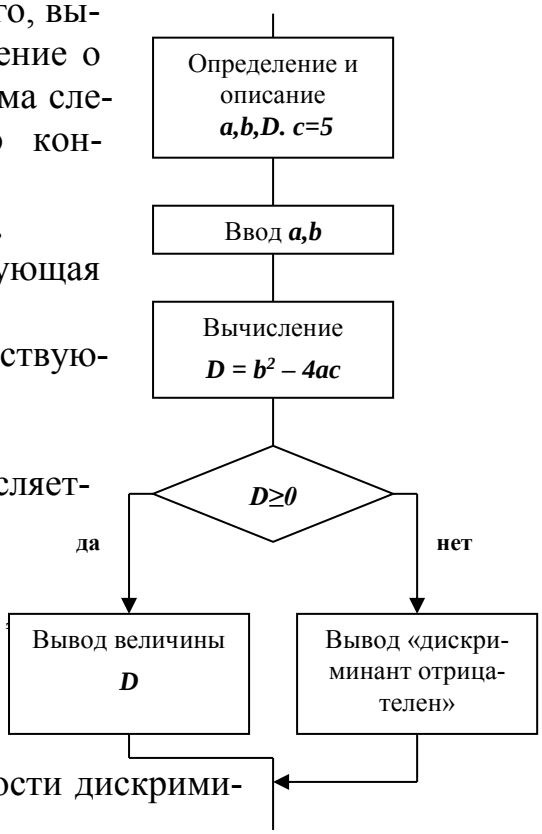
1) Начальными значениями являются величины сторон *a, b***.** В постановке задачи не сказано, какого типа эти числа – целые или вещественные. Так как целые числа являются подмножеством вещественных чисел, то целесообразно приписать значениям *a, b* вещественный тип.

2) Известно из математики, что для нахождения дискриминанта используется формула $D = b^2 - 4ac$.

3) Из математического метода решения следует, что для программной реализации необходимо определить еще одну величину: для идентификации дискриминанта – величину *D*; Так как при ее вычислении основу составляют числа *a, b*, тип которых вещественный, то и тип числа *D* также должен быть веще-

ственным.

4) Так как в процессе вычислений потребуется осуществлять проверку знака дискриминанта, и, в зависимости от него, выводить либо значение числа *D*, либо сообщение о его отрицательности, то в структуре алгоритма следует использовать базовую управляющую конструкцию типа «условный оператор».



2. Структурная блок-схема алгоритма.

Из формализации задачи следует следующая последовательность действий:

Шаг 1. Определяются величины, участвующие в вычислениях и их тип;

Шаг 2. Водятся числа *a, b.*

Шаг 3. По формуле: $D = b^2 - 4ac$ вычисляется дискриминанта;

Шаг 4. Задается условие проверки на неотрицательность величины *D.*

Шаг 5a. Если условие выполняется, то выводится значение *D*

Шаг 5b. Если условие не выполняется, то выводится сообщение об отрицательности дискриминанта.

Блок-схема реализации данного алгоритма с использованием базовых управляющих структур типа «функциональный блок» и типа «условный оператор» показана на рисунке справа.

Задача 3.

Условие. Ввести с клавиатуры последовательность из пяти целых чисел и подсчитать их сумму после ввода всех пяти чисел. Если значение суммы меньше числа 100, то вывести на экран сообщение об этом. Если значение суммы больше или равно числу 100, то вывести полученное значение.

Примерное решение.

1. Формализация и выбор метода.

1) Начальными значениями являются пять произвольных неизвестных чисел. Назовем их, например, *a1,a2,a3,a4,a5*. В постановке задачи определен их тип – это целые числа.

2) Известно из математики, что для нахождения суммы чисел используется элементарная формула последовательного арифметического суммирования: *a1+a2+a3+a4+a5*. Обозначим через *S* суммарное значение чисел. Его тип также целое число, так как все другие числа – целые.

3) Известно из программирования, что для реализации математической формулы суммы чисел: *S = a1+a2+a3+a4+a5* программная формула имеет вид: *S = S+a1+a2+a3+a4+a5*. При этом при определении *начальных значений* и типов чисел следует установить начальное значение *S = 0.*

4) Так как в процессе вычислений потребуется осуществлять процедуру подсчета суммы при проверке условия перебора всех введенных чисел (вначале $S = S+a1 = 0+a1 = a1$, затем $S = S+a2 = a1+a2$ и т.д. пока не выполнится условие $S = S+a5 = a1 +a2+a3+a4+a5$, то необходимо определить счетчик количества уже используемых в вычислении чисел a. Пусть таким счетчиком будет число i**.** Это число целого типа, так как оно соответствует целым номерам-индексам чисел a. И именно его значение мы будем проверять при последовательном вычислении суммы. Введение счетчика позволяет написать формулу подсчета суммы в более формальном виде:

$$S = S+a_i,\ i=1,2,\ldots,5.$$

И проверку индекса i, и подсчет суммы S удобно реализовать в алгоритме, используя базовую управляющую конструкцию типа «обобщенный цикл».

5) Кроме того, в процессе вычислений, на конечном этапе согласно условиям задачи, потребуется выводить значение суммы после проверки явно заданного ограничения: только в том случае если $S\geq100$. Поэтому, в структуре алгоритма следует использовать также и базовую управляющую конструкцию типа «условный оператор».

2. Структурная блок-схема алгоритма

Из формализации задачи следует следующая последовательность действий:

Шаг 1. Определяются величины, участвующие в вычислениях и их тип. При этом задается число $S=0$;

Шаг 2. Последовательно вводятся числа $a1,a2,a3,a4,a5$**.**

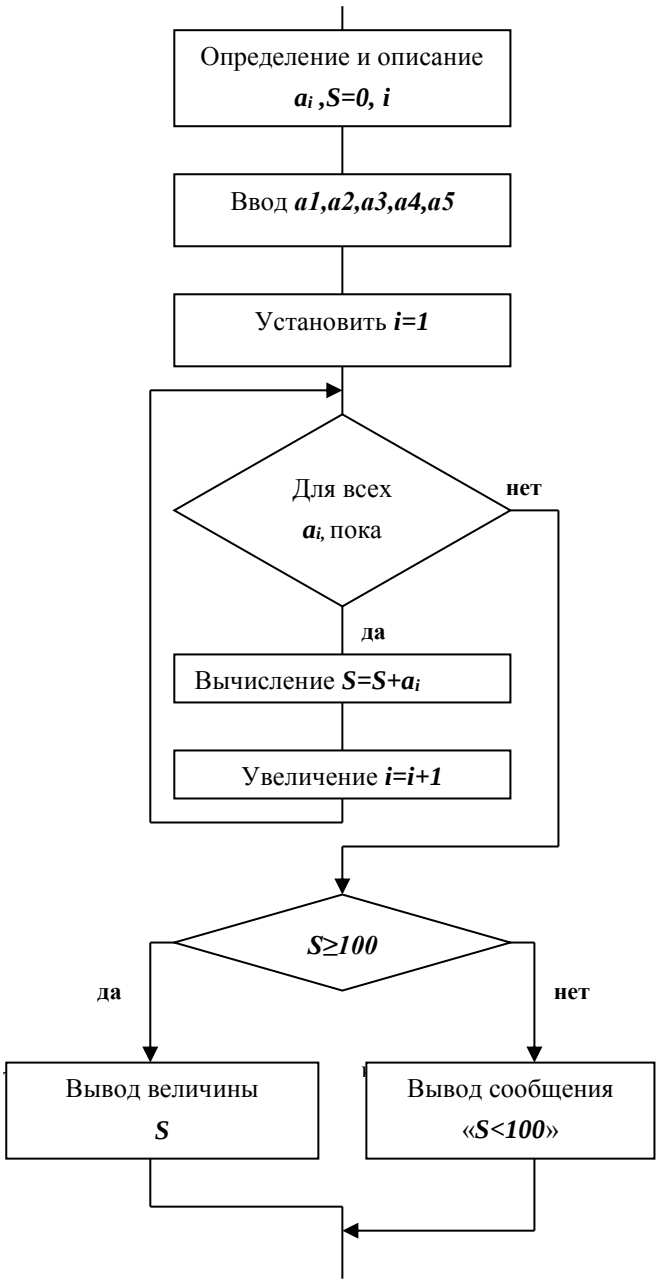
Шаг 3. Устанавливается начальное значение счетчика i**.**

Шаг 4. После ввода всех чисел с помощью счетчика задается условие повторения вычисления величины S**.** Это условие проверяется до тех пор, пока не будут рассмотрены все введенные числа.

Шаг 5. Если условие выполняется, то вычисляется значение S.

Шаг 6. Происходит увеличение счетчика и вновь осуществляется его проверка

Шаг 7. Если условие не выполняется (все числа исчерпаны), вычис-



ления *S* прекращаются и осуществляется проверка ее величины по новому условию *S≥100*.

Шаг 8a. Если это условие выполняется, то выводится значение *S*.

Шаг 8b. Если условие не выполняется, то выводится сообщение о том, что сумма меньше числа 100.

Блок-схема реализации данного алгоритма с использованием базовых управляющих структур типа «функциональный блок», «обобщенный цикл» и «условный оператор» показана на рисунке справа.

1.3 Представление алгоритмов

Совершенно ясно, что для описания любого действия (будь то - поведение человека или схема маршрута движения), решения любой задачи (математической, бытовой, прикладной (например, военной)) необходимо разработать алгоритм такого действия или решения.

Понятие алгоритма является центральным понятием информатики и программирования. Термин «алгоритм» своим происхождением обязан имени арабского ученого Аль-Джафара Магомеда Бен Муссы Хорезми, родом из Хивы. Он еще в IX в. написал учебник по математике для купцов и стряпчих с арабскими цифрами, сформулировав правила выполнения четырех арифметических действий, их приоритет при вычислении математического выражения. Сокращенно Аль-Хорезми в латинском написании «Алгоритми». Отсюда и название.

По определению (см. п.1.1.1), **алгоритм** – это заранее заданная последовательность четко определенных правил или команд для получения решения задачи за конечное число шагов.

Алгоритмы в программировании играют далеко не второстепенную роль. От того, насколько продуман алгоритм задачи, зависит и правильность, и эффективность работы программы.

Алгоритм предназначен для решения двух основных задач программирования: а) получить точное решение через программу; б) получить эффективную для решения программу. Это не только прикладные задачи, но и задачи научные. Поэтому в теории и практике алгоритмизации и программирования существует большой выбор методов и инструментов для их решения, которые, в условиях постоянного совершенствования информационных технологий (ИТ), также динамично совершенствуются и изменяются.

Для реализации алгоритма на каком-либо языке программирования ег синтаксис предоставляет разнообразные средства. Поэтому, так же как и мысль, выражаемая человеком различными словами и различными предложениями, построенными из этих слов, становится понятной и доступной, так и программа в терминах языка программирования может быть составлена разнообразными способами. И так же, как в естественном языке, главное в ней – получить адекватный образ, выраженный результатом ее работы.

Пока мы не будем акцентировать внимание на понятие эффективности ал-

горитмов в смысле их вычислительной сложности. Некоторые из них мы рассмотрим в последних главах на примерах наиболее распространенных задач программирования. В этой главе мы акцентируем вопрос на том, как составить алгоритм, способный решить главную задачу – получить точный результат.

1.3.1 Алгоритм и его свойства

Алгоритм характеризуют его *свойства* и *способы описания*.

Основными **свойствами** алгоритма являются:

Дискретность. Процесс получения конечного результата разбивается на отдельные этапы (дискретные шаги), так, что значения величин на каждом следующем шаге определяются значениями величин, полученных на шаге предыдущем.

Пример. За какое время (*T*) можно попасть из пункта *A* в пункт *D* через пункты *B* и *C*, если известна таблица растояний между этими пунктами. Дискретные щаги: 1) вычислить время (t_1) маршрута из *A* в *B;* 2) вычислить время (t_2) маршрута из *B* в *C*; 2) вычислить время (t_3) маршрута из *C* в *D*. Тогда: $T = \{[(t_1)+t_2]+t_3\}$.

Определенность. Каждое правило алгоритма должно быть четким и однозначным в его толковании и исполнении.

Пример. Чтобы достичь из пункта *A* пункт *D* за определенное время (*T*) через пункты *B* и *C* необходимо: 1) из *A* в *B* двигаться со скоростью V_1*;* 2) из *B* в *C* двигаться со скоростью V_2; 2) из *C* в *D* двигаться со скоростью V_3.

Результативность. За конечное число шагов алгоритма решения задачи должен быть получен требуемый результат.

Пример. Решение в указанном примере может получено за тоже количество шагов, но при изменении условий по скорости движения между пунктами.

Массовость. Алгоритм, в общем виде, разрабатывается так, чтобы его можно было применить для класса задач, отличающихся друг от друга только исходными данными.

Пример. Указанная в примере задача может быть решена для любых транспортных средств, способных выполнить указанные условия.

В алгоритме отражаются логика и способ формирования результатов решения с указанием необходимых расчетных формул, логических условий, соотношения для контроля достоверности выходных результатов. В алгоритме обязательно должны быть предусмотрены все ситуации, которые могут возникнуть в процессе решения задачи.

Алгоритм решения задачи и его программная реализация тесно взаимосвязаны. Специфика применяемых методов проектирования алгоритмов и используемых при этом инструментальных средств разработки программ может повлиять на форму представления и содержания алгоритма обработки данных. *Однако его логическая структура при этом не меняется.*

1.3.2 Способы описания алгоритмов

Различают ***четыре способа*** описания алгоритма: *словесный, формальный, графический, символьный*.

Словесный. Запись алгоритма на естественном языке. Иначе говоря, вся последовательность шагов решения задачи представляется в словесной форме. Например, для задачи «выбрать из трех чисел (*A, B, C*) наибольшее число» словесный алгоритм выглядит так:

1) Сравнить число *A* с числом *B*. Если *A* больше *B*, то перейти к п.2), иначе – к п.3).
2) Сравнить число *A* с числом *C*. Если *A* больше *C*, то *A* – наибольшее. Конец. Иначе – перейти к п.3).
3) Сравнить число *B* с числом *C*. Если *B* больше *C*, то *B* – наибольшее. Конец. Иначе – перейти к п.4).
4) Наибольшее число – *C*. Конец.

Достоинства.

Не требует никаких других специальных знаний представления, кроме естественного языка.

Недостатки:

а) не пригоден для машинной (автоматической) реализации;

б) не нагляден.

Формальный. Как постановка задачи, так и ее алгоритм могут быть выражены в формальной (математической или логической) форме. Например, алгоритм задачи «вычисления площади треугольника по формуле Герона» выглядит так:

1) Дано: *a,b,c* – стороны треугольника.
2) Решение:

$p = \frac{a+b+c}{2}$ – полупериметр.

$S = \sqrt{p(p-a)(p-b)(p-c)}$ – площадь.

Достоинства:

а) практически все формулы и математические термины поддаются для программирования в явном математическом виде;

б) последовательность и логика выполнения задачи кратки, более ясны и наглядны.

Недостатки:

а) не всякую задачу можно представить в формализованном виде;

б) не всякие математические операции можно реализовать в явном виде на ЭВМ (например, отыскание производных), требуется дополнительная их алгоритмизация.

Символьный. Представление алгоритма решения задачи на одном из языков программирования. Хотя алгоритмический язык и имеет сходство с естественным языком, между ними есть существенное различие. Естественный язык ориентирован на человека и может использоваться как промежуточный этап в разработке программ (особенно на начальной стадии). Алгоритмический язык ориентирован на ЭВМ. Вычислительная машина интерпретирует естественный язык в исполняемые команды языка алгоритмического. Отсюда сле-

дует, что в программе на алгоритмическом языке обязательно присутствуют операции ввода – вывода.

Достоинства:

а) обеспечивает автоматическое выполнение алгоритма на ЭВМ;

б) наиболее точное представление алгоритма для автоматической реализации.

Недостатки:

а) недостаточно нагляден;

б) требует знания хотя бы одного языка программирования.

Графический. Представление алгоритма с использованием специальных графических символов. Полное представление алгоритма, описанное таким образом, образует, так называемую, *блок – схему* ЭВМ-алгоритма.

Достоинства:

а) нагляден, ясны структура и логика программы;

б) независим от языка программирования: блок-схемы стандартны для переноса на любой язык программирования.

Недостатки:

а) громоздок при описании программ с большим кодом.

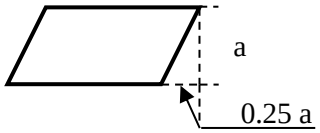
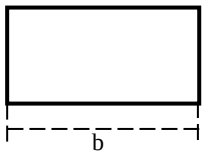
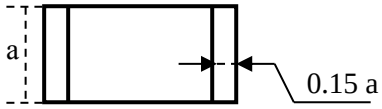
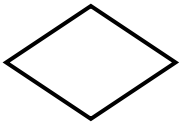
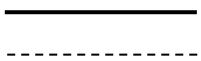
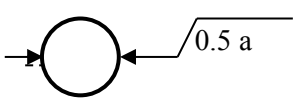
Но именно привлекательные достоинства этого способа привели к тому, что сегодня для любой программной документации какого-либо изделия или системы, графический способ является обязательным.

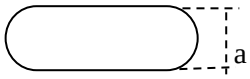
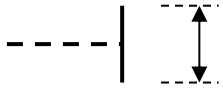
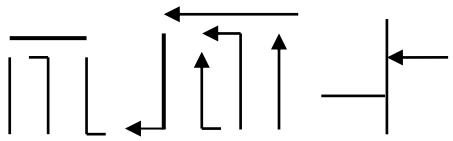
Для этого существующим законодательством по разработке алгоритмов и программ определены специальные графические символы, предназначенные для графического описания в блок-схеме программы отдельных этапов программной реализации задачи.

1.3.3 Символы, используемые в схемах программ и правила их применения

Согласно единой системе программной документации (ЕСПД), существуют специальные ГОСТы для правильного изображения элементов блок-схемы. Полный перечень схем алгоритмов, программ, данных и систем, приведен в ГОСТ 19.701-90. Рассмотрим некоторые из них, наиболее часто применяемые при алгоритмизации задач.

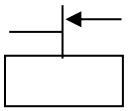
Схемы алгоритмов состоят из символов, краткого пояснительного текста и соединяющих линий. Существуют специальные символы для описания схем данных, схем программ, схем описания работы системы, схем взаимодействия программ, схем описания ресурсов системы. Мы рассмотрим только символы для описания блок-схем программ, и только те из них, которые мы чаще всего будем использовать при описании наших программ. Эти символы приведенны в следующей таблице.

Вид и характеристики символа	Наименование и назначение
	Символ отображает **данные**. Обычно применяется для изображения ввода – вывода данных. Как правило, эти данные отображаются в математическом виде внутри символа.
	**Процесс.** Символ отображает функцию обработки данных любого вида. Обычно внутри этого символа отображаются математические выражения и операции.
	**Предопределенный процесс.** Процесс, состоящий из одной или нескольких операций или шагов программы, определенных в другом месте. Обычно применяется для отображения имени подпрограммы внутри символа.
	**Подготовка.** Символ отображает модификацию группы команд с целью воздействия на некоторую последующую функцию (переключатель – выбор варианта из многих). Обычно внутри символа отображается условие выбора типа «для».
	**Решение.** Отображает функцию переключательного типа, имеющую один вход и ряд альтернативных выходов, только один из которых активизируется после условия, определенного внутри символа
 Толщина линий пропорциональна толщине символова	**Линия.** Сплошная линия предназначена для отображения потока данных или управления. Пунктирная линия, в основном, применяется для обведения аннотированного участка блок-схемы.
	**Соединитель.** Обрыв/соединитель между частями схемы по линиям потока данных и управления. Парный символ. Один устанавливается в точке разрыва, другой в точке продолжения линии. Парность определяется идентификацией одного и того же «имени» внутри символа.

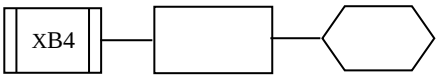
Вид и характеристики символа	Наименование и назначение
 радиус закругления углов 0.25 a	**Терминатор.** Символ, обычно применяемый для изображения начало и конца программы, точки входа/выхода в подпрограмму. Как правило, внутри символа пишутся слова «начало(begin)» или «конец(end)»
 Высота соответствует высоте соответствующего символа	**Комментарий.** Символ используется для добавления комментариев к соответствующему символу схемы. Текст комментария помещается около ограничивающей фигуры символа.
а) б) в) 	**Направление потоков.** Направление потока слева направо и сверху вниз считается стандартным (рис. а)), для него стрелки необязательны. Стрелки желательны в исходящих линиях по направлениям (рис. б)). При слиянии нескольких линий, они соединяются так, как показано на рис. в) – перекрестие в одной точке нежелательно.

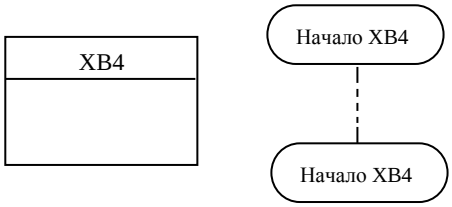
Правила выполнения соединений.

- Все символы, по возможности, должны быть одного размера. Размер параметра **a** (высота) выбирается из ряда {10, 15, 20,…}мм, т.е. кратным пяти. Размер параметра **b** (ширина) может быть 1,5**a** или 2**a.**
- Все символы в схемах могут вычерчиваться в любой ориентации, но предпочтительней является горизонтальная ориентация.
- В схемах следует избегать пересечения линий. Если две и более линий объединяются в одну, то место объединения должно быть смещено.

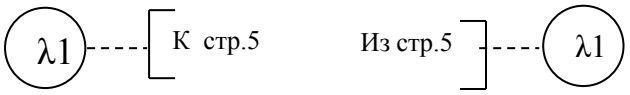


- В схемах может быть использован идентификатор символа (например, номер). Идентификатор символа указывается над символом слева.
- В символах может использоваться подробное представление, которое обозначается с помощью символа с полосой для процесса и данных. В верхней части символа (над полосой) или в средней его части (между двумя полосами) указывается любой идентификатор. Символ означает, что в этом же комплекте документации, но где-то в другом месте, представлено более детальное описание процесса. Детальное описание должно начинаться и заканчиваться символами терминатора.



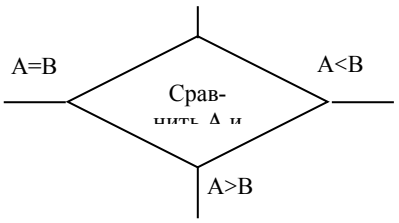


- При необходимости линии в схемах следует разрывать для избежания излишних пересечений или слишком длинных линий, а также, если схема не помещается на одной странице.

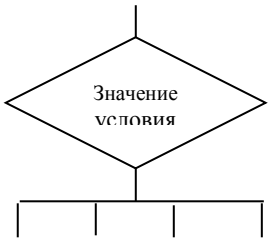


- Несколько выходов из символов следует показывать:

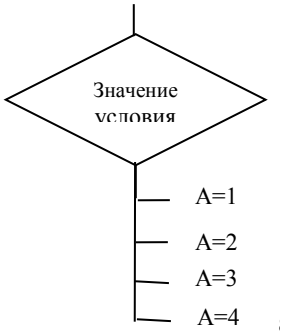
➢ Несколькими линиями от данного символа к другим;



➢ Одной линией от данного символа, которая затем разветвляется в соответствующее число линий;



➢ каждый выход из символа должен сопровождаться соответствующими значениями условий для определения логического пути при выполнении этих условий.



Напомним **основную теорему структурного программирования**, касаемую алгоритмов (см. п. 1.1.2). Звучит она так.

Как бы сложна не была задача, блок-схема соответствующей программы всегда может быть представлена с использованием весьма ограниченного числа элементарных управляющих структур.

Теперь, зная символы для описания блок-схем программ и руководствуясь указанной теоремой, можно приступать к разработке алгоритмов программ любой логической сложности.

В теории алгоритмов различают алгоритмы *линейной*, *разветвляющейся* и *циклической* структуры, а также алгоритмы со структурой *вложенных циклов*. Алгоритмы решения сложных задач могут включать все перечисленные структуры, которые используются для реализации отдельных участков общего алгоритма.

1.4 Примеры алгоритмов линейной и разветвляющейся структур

Рассмотрим наиболее простые алгоритмы: алгоритмы линейной и разветвляющейся структур.

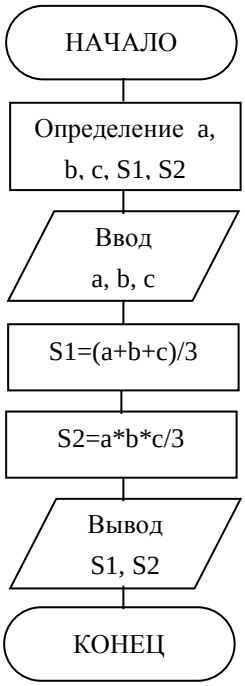
1.4.1 Линейный алгоритм

Определение. *Линейный алгоритм* – это алгоритм, символы которого на схеме изображены в той последовательности, в какой должны выполняться предписываемые действия.

Иначе говоря, все блоки этого алгоритма выполняются последовательно, один за другим, в порядке, заданном схемой. Такой порядок выполнения называется естественным порядком.

ПРИМЕР. Даны три числа *a, b, c*. Разработать алгоритм программы для нахождения среднего арифметического и среднего геометрического этих чисел.

РЕШЕНИЕ. Определим общую структуру данной задачи:



1) Используемые переменные: *a*, *b*, *c*, *S1*, *S2*.

2) Входные данные: числа *a, b, c*;

3) Вычислительный блок: определим для вычисления среднего арифметического чисел переменную *S1*, а для вычисления среднего геометрического – переменную *S2*. Вычисление значений этих переменных произведем по формулам:

$$S1 = \frac{a+b+c}{3}, \qquad S2 = \frac{a \cdot b \cdot c}{3}$$

4) Выходные данные: результаты вычислений – значения переменных *S*1 и *S*2.

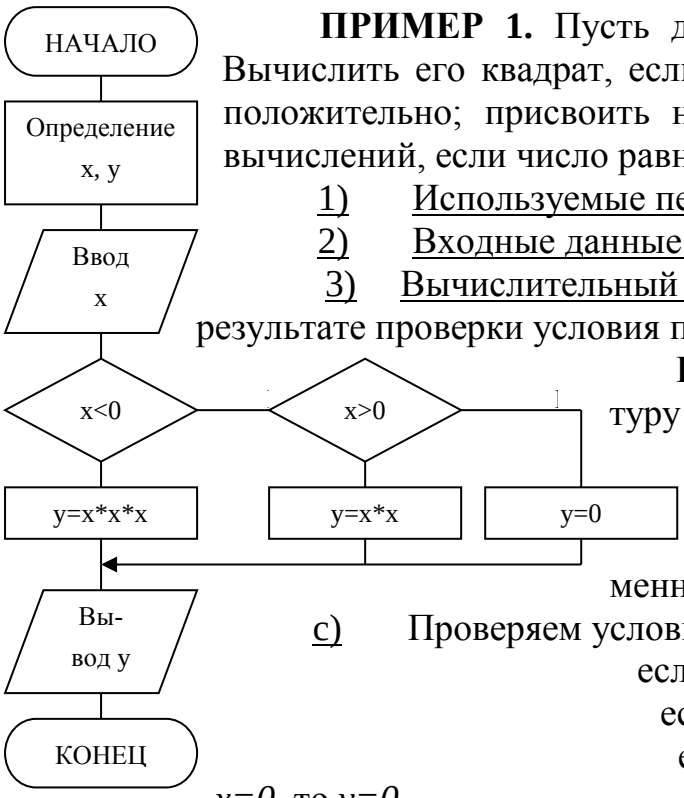
Блок-схема решения данной задачи (линейный алгоритм) имеет вид, представленный на рисунке слева.

1.4.2 Алгоритм разветвляющейся структуры

На практике редко удается представить решение задачи в виде алгоритма линейной структуры. Часто в зависимости от каких-либо промежуточных результатов последующие вычисления осуществляются либо по одним, либо по другим формулам. То есть, в зависимости от выполнения некоторого логического условия процесс вычисления осуществляется по одной из ветвей.

Определение. Алгоритм, в котором предусмотрено разветвление выполняемой последовательности действий в зависимости от результатов проверки некоторого условия называется *алгоритмом с разветвляющейся структурой*.

В общем случае число ветвей в таком алгоритме не обязательно равно двум.



ПРИМЕР 1. Пусть дано некоторое действительное число. Вычислить его квадрат, если оно отрицательно; его куб, если оно положительно; присвоить нулевое значение значение результату вычислений, если число равно нулю.

1) Используемые переменные: *x* и *y*.
2) Входные данные: число *x*.
3) Вычислительный блок: вычисление значения числа *y* в результате проверки условия по значению числа *x*

РЕШЕНИЕ. Определим общую структуру данной задачи:

a) Определяем переменные *x* и *y*;

b) Вводим значение переменной *x*;

c) Проверяем условия:

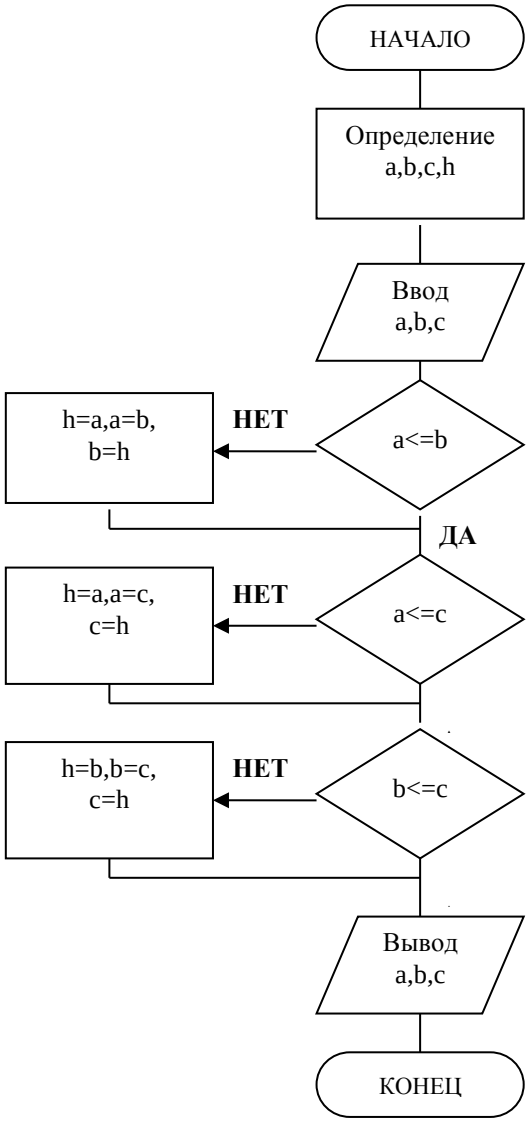
если $x<0$, то $y=x^2$;

если $x>0$, то $y=x^3$;

если $x=0$, то $y=0$.

4) Выходные данные: искомые результаты вычислений – значение переменной *y*.

Блок-схема решения данной задачи имеет вид, показанный на рисунке **слева**



ПРИМЕР 2. Упорядочить три числа *a, b, c* по возрастанию так, чтобы переменной *a* соответствовало наименьшее из чисел, переменной *b* - среднее, переменной *c* - самое большое число.

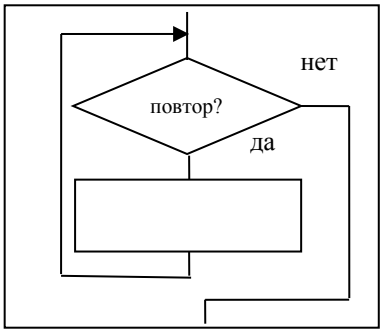
РЕШЕНИЕ. Для решения данной задачи следует последовательно сравнить между собой значения этих переменных. Сначала на условие $a \le b$, затем на условие $a \le c$, далее на условие $b \le c$. При выполнении одного из условий осуществляется очередное сравнение. В противном случае, сначала осуществляется перестановка значений соответствующих переменных, а лишь затем переход к следующему сравнению. Для перестановки значений двух переменных используется вспомогательная переменная *h*.

Определение входных/выходных данных и блока вычислений определяется по аналогии с задачами, представленными выше. Блок-схема ЭВМ-алгоритма приведена справа.

1.5 Примеры алгоритмов циклической структуры

Часто при решении задач приходится многократно вычислять значения по одним и тем же математическим или логическим зависимостям для различных значений входящих в них величин. Такие многократно повторяемые участки вычислительного процесса называют *циклами*. Их использование позволяет существенно сократить объем схемы алгоритма и длину соответствующей ему программы. Различают *циклы с заданным* и *неизвестным числом повторений*. К последним циклам относятся итерационные (пошаговые) циклы, характеризующиеся последовательным приближением к искомому значению с заданной точностью.

Определение. Алгоритм, в котором повторяется последовательность одних и тех же действий заданное число раз называется *алгоритмом циклической структуры*.



Согласно основной теореме структурного программирования в блок-схемах алгоритмов для таких вычислительных процессов используется элементарная конструкция, называемая «обобщенный цикл» (рисунок слева).

Для организации цикла следует выполнить следующие действия:

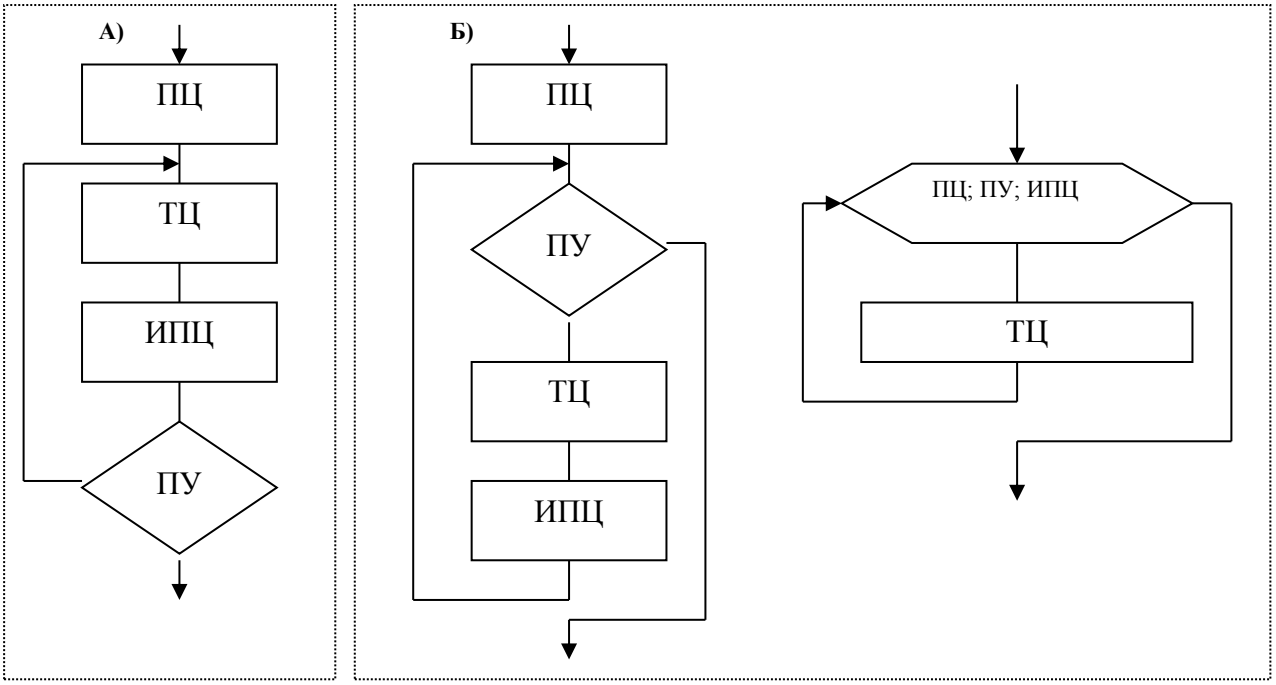
- Осуществить подготовку цикла (ПЦ): задать перед выполнением цикла начальное значение переменной, изменяющейся в цикле. Такая переменная называется *параметром цикла;*
- Выполнить тело цикла (ТЦ): осуществить последовательность действий заданное число раз;
- Осуществлять изменение параметра цикла (ИПЦ): увеличивать/уменьшать значение параметра цикла на какую-то величину для определения возможности перехода к следующей итерации. Такая величина называется *шагом цикла*;
- Осуществлять проверку условия окончания цикла (ПУ): на каждом шаге проверяется значение параметра цикла, которое сравнивается с заданной величиной (признаком окончания цикла) для определения выхода из циклического процесса.

На практике циклы различают по структуре алгоритма.

По структуре циклические алгоритмы различают на циклы с *предусловием* и с *постусловием*.

Это зависит от того, как организован процесс проверки условия окончания цикла. Если условие проверяется перед циклом, то имеет место *алгоритм с предусловием*. Если проверка осуществляется после последовательности циклических действий, то имеет место *алгоритм с постусловием*.

Схематично такие алгоритмы их можно изобразить так:



На схеме А) алгоритм с постусловием. На схеме Б) – выриванты представления алгоритмов с предусловием.

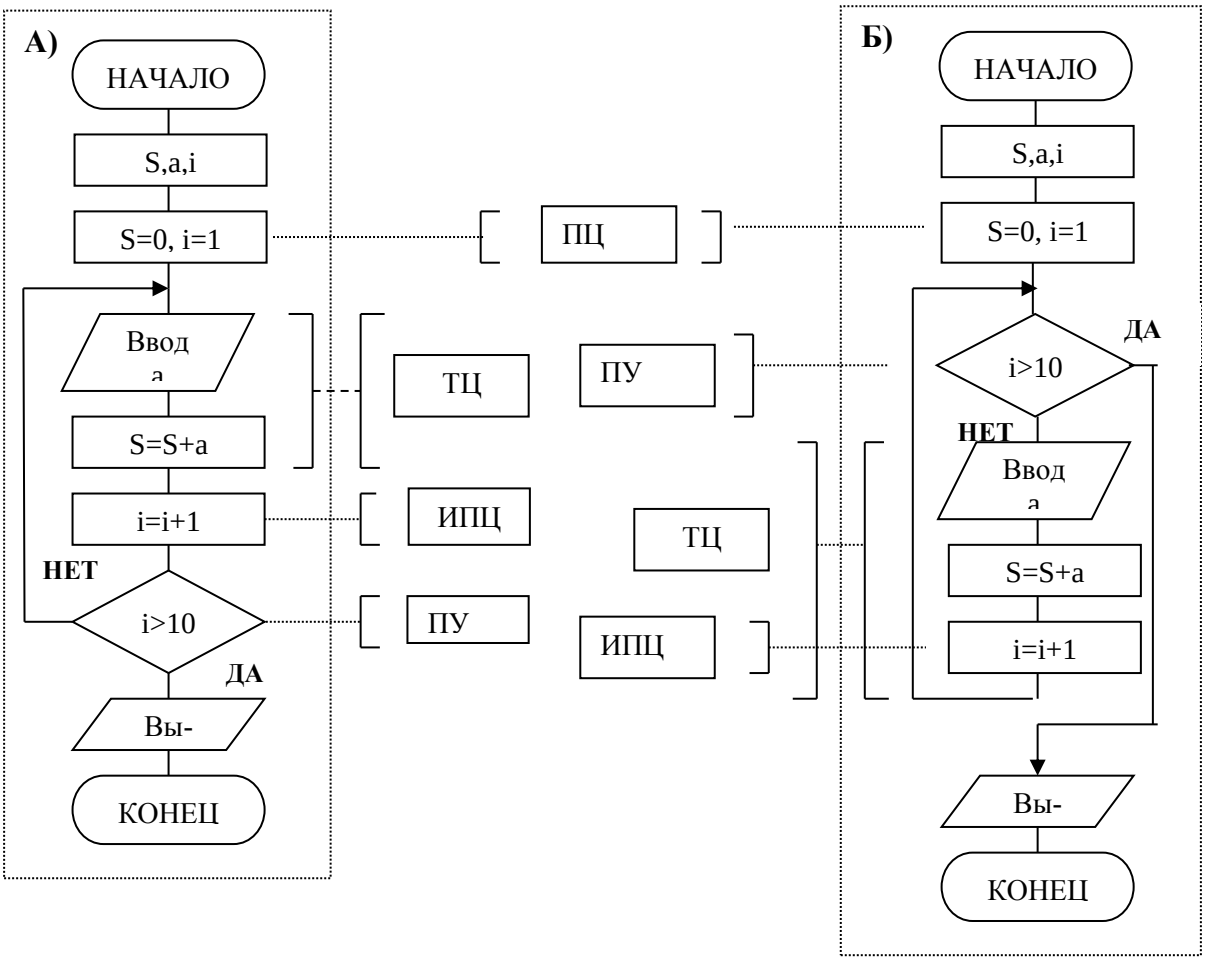
Важной и ***отличительной особенностью*** является то, что для алгоритма Б) тело цикла может быть *не выполнено совсем*, даже один раз. И напротив, независимо от условия, в алгоритме А) тело цикло выполнится *хотя бы один* раз.

ПРИМЕР 1.

Разработать алгоритм программы для подсчета и вывода суммы из 5-ти последовательно вводимых целых чисел.

РЕШЕНИЕ. Из условия данной задачи, очевидно, что математически такую постановку можно выразить формулой: $S = \sum_{i=1}^{5} a_i$. Это означает, что необходимо определить три переменные *S, a, i.* Причем здесь в качестве параметра цикла будет выступать переменная *i*, имеющая начальное значение, равное единице и конечное значение, равное пяти. Следовательно, известны такие компоненты, как **шаг цикла** (очевидно, что его начальное значение равно единице и направлен в сторону увеличения) и **условие окончания** цикла (достигается, когда параметр *i* станет больше пяти). **Телом цикла** являются процедуры последовательного ввода чисел и вычисление их суммы.

Ниже приведены блок-схемы ЭВМ-алгоритмов с пост- (А)) и предусловием (Б)), иллюстрирующие программное решение данной задачи.



К разновидностям алгоритмов, имеющих циклическую структуру, относятся и алгоритмы, в которых могут одновременно изменяться *несколько параметров*, или алгоритмы, имеющие *структуру вложенных циклов*. Вначале рассмотрим алгоритмы с изменяющимися одновременно несколькими параметрами.

Примером таких задач служат, например задачи, в которых одновременно с определением значения какого-либо элемента в массиве элементов (последовательности элементов одинакового типа), определяют и порядковый номер данного элемента в этом массиве. Например, в математике такими элементами могут быть элементы вектор-столбцов, вектор-строк или матриц.

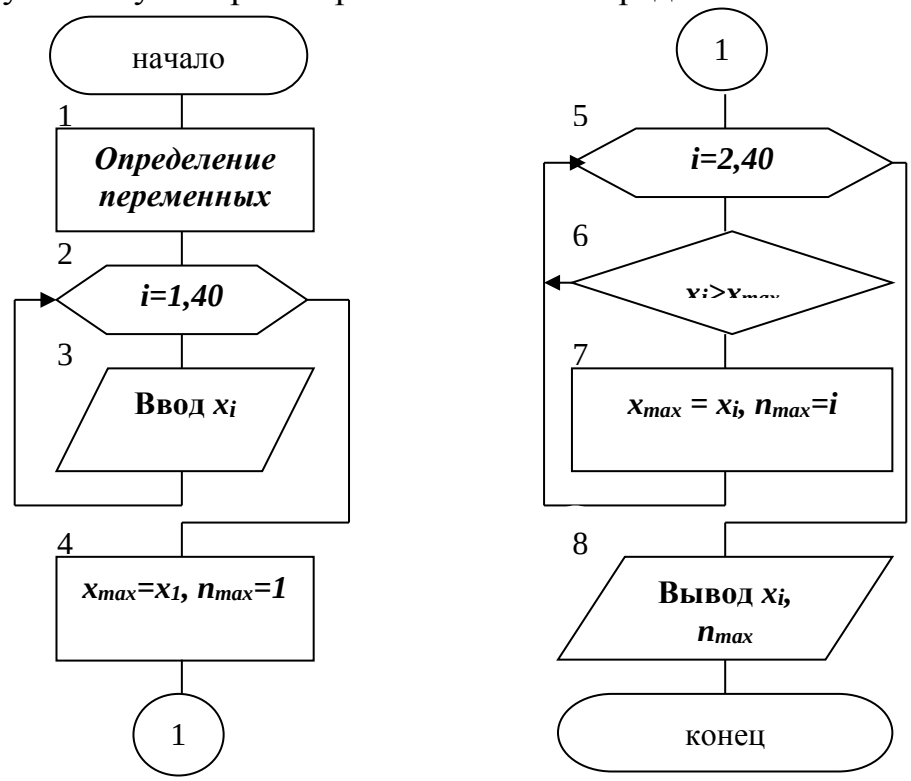
ПРИМЕР 2. Найти наибольший элемент массива $(x_1, x_2, \ldots, x_{40})$ и его порядковый номер. Элементы массива задаются с клавиатуры. Значение максимального элемента и его порядковый номер (индекс) выводится на экран.

РЕШЕНИЕ.

Не касаясь подробно аспектов формализации задачи, отметим, что по ее условию нет необходимости сравнивать значения на этапе ввода элементов. Предполагается, что массив уже существует – это означает, что его элементы введены. Этот участок алгоритма программы можно реализовать с помощью организации обычного цикла с условием проверки параметра цикла.

Далее. Учитывая, что массив чисел существует, в качестве начального значения для сравнения всех элементов массива на максимальное значение примем первый элемент, присвоив ему «условное максимальное» значение. И так как сравнение этого элемента с самим собой не имеет смысла, то выполнение цикла проверки можно начинать со второго элемента. При этом начальный номер максимального элемента будет соответствовать значению начального элемента массива, то есть, будет равен единице. Таким образом, цикл будет выполняться, начиная со второго элемента.

Итак, нам необходимо вывести на экран значение того элемента в массиве, значение которого окажется больше или равно значению начального элемента и больше или равно значений всех остальных элементов этого массива. И, кроме того, нам нужно также вывести значение индекса этого элемента. Полную схему алгоритма решения можно представить так:



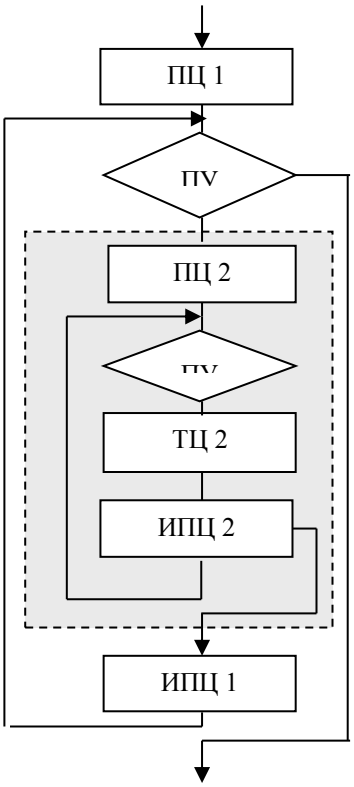
В этом алгоритме, в блоке ***6*** одновременно проверяются два изменяющихся параметра: индекс номера элемента массива ***i*** и значение, соответствующего этому индексу, элемента массива x_i. В случае выполнения условия переопредяляется значение максимального элемента и его индекса. В противном случае – рассматривается следующий элемент.

1.6 Примеры алгоритмов сложных вычислительных процессов

Сложные вычислительные процессы, как правило, организованы так, что в их логической структуре взаимодействуют различные циклические процессы. В большинстве случаев, при организации достаточно серьезной задачи, невозможно обойтись без такой ее организации, при которой циклы не находились бы один внутри другого.

Определение. Алгоритм, в котором один цикл находится внутри другого цикла и представляет его тело, называется, *алгоритмом со структурой вложенного цикла.*

В таком алгоритме различают термины «внешний» и «внутренний» цикл.



В цикл, называемым *внешним*, могут входить один или несколько циклов, называемых *внутренними*. Организация как внешнего, так и внутреннего циклов осуществляется по тем же правилам, что и организация простого цикла. Параметры внешнего и внутреннего циклов разные и изменяются не одновременно. То есть, при одном значении параметра внешнего цикла, параметр внутреннего цикла поочередно принимает все значения. Схематично такие алгоритмы можно представить так, как показано на рисунке **справа.**

Как правило, такие алгоритмы применяются, когда существует информация, данные которой изменяются не по одному параметру и, при этом, изменение параметров производится не синхронно. Другими словами, при единичном изменении одного параметра, другой параметр может изменяться много раз.

Примером такой задачи может служить задача, например, нахождения какого-либо элемента прямоугольной матрицы.

Как известно, матрица имеет два определяющих ее элемента-параметра: номер строки и номер столбца.

Если обозначить через i номер строки, через j номер столбца, через a_{ij} элемент, находящийся на пересечении строки и столбца, то матрицу A можно представить так:

$$A = \begin{vmatrix} a_{11} & a_{12} & . & a_{1j} \\ a_{21} & a_{22} & . & a_{2j} \\ . & . & . & . \\ a_{i1} & a_{i2} & . & a_{ij} \end{vmatrix}$$

Теперь, если, например, потребуется найти максимальный элемент среди всех элементов матрицы, то сделать это можно будет следующим образом. Вначале на максимальное значение проверяются все элементы первой строки, затем все элементы второй строки и т.д., пока не будут проверены все j элемен-

тов последней ***i-той*** строки. То есть при такой организации вычислительного процесса внешний цикл будет построен по параметру ***i***, а внутренний цикл по параметру ***j***.

Сказанное рассмотрим на примере.

ПРИМЕР 1. Разработать алгоритм программы для определения потенциальной возможности РЛС по дальности обнаружения целей на малых высотах при разной высоте антенны по формуле $D_{ij} = 4.12(\sqrt{H_j} + \sqrt{h_i})$, где высота антенны может быть $h_i \in \{6,9,12,15\}$*м.*, а высоты целей лежат в диапазоне $H_j \in \{100,200,300,400,500\}$*м.*

РЕШЕНИЕ. Проведем предварительные исследования и формализуем данную задачу.

1. Заметим, что величины, определяющие как высоты антенны, так и высоты цели, могут принимать различные значения. Поэтому целесообразно ввести индексацию для этих величин (т.е. каждой из них присвоить какой-либо номер). Примем в качестве индекса для высот антенны параметр ***i={1, 2, 3, 4}***, а для высот цели параметр ***j={1, 2, 3, 4, 5}***. Тогда ***h_1=6, h_2=9, h_3=12, h_4=15***, а ***H_1=100, H_2=200, H_3=300, H_4=400, H_5=500***.

2. Так как величина дальности – функция, зависящая от двух аргументов (от ***h_i*** и ***H_j***), то величина ***D*** имеет два индекса ***$D_{i,j}$***. Ее значения удобно представить в виде матрицы размером ***D(4×5)*** в следующем виде:

$$D_{ij} = \begin{pmatrix} D_{11} & D_{12} & D_{13} & D_{14} & D_{15} \\ D_{21} & D_{22} & D_{23} & D_{24} & D_{25} \\ D_{31} & D_{32} & D_{33} & D_{34} & D_{35} \\ D_{41} & D_{42} & D_{43} & D_{44} & D_{45} \end{pmatrix}.$$

В таком представлении элемент матрицы, например, ***D_{23}*** имеет вид

D_{23}=D(2, 3)=D(h_2=9, h_3=300),

т.е. его значение (дальность обнаружения цели) зависит от высоты антенны (***h_2=9*** м.) и высоты полета цели (***H_3=300*** м.) и вычисляется по приведенной выше формуле.

Таким образом, очевидно, что для вычислений значений ***D_{ij}*** следует вначале, установив одну из высот антенны, например ***h_1***, получить все значения ***D_{1j}*** для нее, изменяя последовательно значения высот цели ***H_j***. Так будет заполнена первая строка матрицы. Как только это произойдет, следует изменить индекс ***i*** и повторить процедуру заполнения элементов матрицы теперь уже для этой строки. И так продолжать до тех пор, пока все номера индекса ***i*** не будут исчерпаны.

Из приведенных замечаний следует, что алгоритм должен иметь два цикла: внешний по параметру ***i*** и внутренний по параметру ***j*** для каждого ***i***.

3. По условиям постановки задачи легко выявить некоторые математические закономерности для высот антенны и цели. Заметим, что величина каждой следующей высоты антенны отличается от предыдущей на ***3*** единицы, а высота полета цели на ***100*** единиц. И так как нам не требуется непосредственно хра-

нить значения величин h_i и H_j в памяти ЭВМ, то их имена (***h*** и ***H***) можно использовать в качестве изменяемых параметров, имеющих начальное и конечное значение и явно выраженный шаг изменения. При этом ***h***–параметр внешнего цикла, ***H***–параметр внутреннего цикла. Таким образом

$h_{нач} = 6, \text{шаг } \Delta h = 3, h_{кон} = 15;$

$H_{нач} = 100, \text{шаг } \Delta H = 100, h_{кон} = 500$

4. Теперь все готово для определения всех компонент алгоритма.

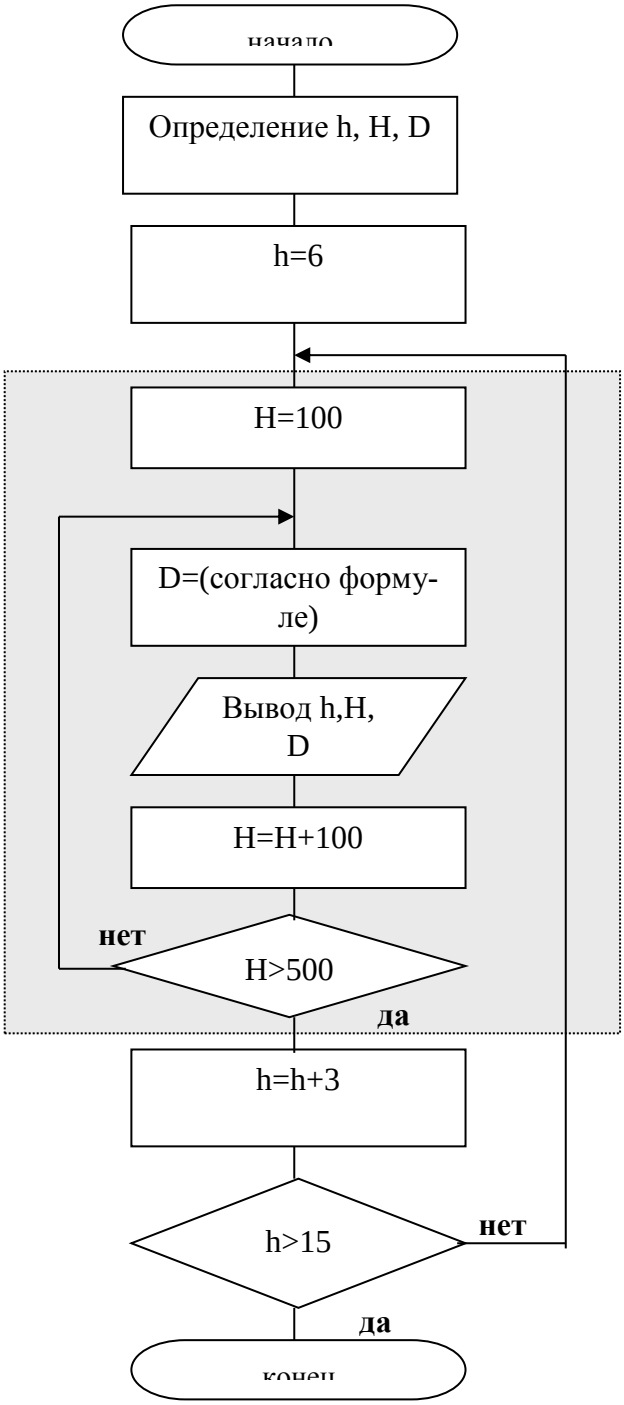
А) **используемыми** в нем **переменными** будут переменные ***h***, ***H*** и ***D***. Причем, первые две – это целые числа, а последняя – вещественное число;

Б) **входными данными** являются начальные значения для переменных ***h*** и ***H***, причем, эти значения не требуют ввода в память ЭВМ (например, с клавиатуры), а могут быть заданны непосредственным описанием в программе;

В) **вычислительный блок** (тело цикла) предполагает вычисление величины ***D*** и изменение параметров циклов;

С) **выходными данными** являются значения величин ***h***, ***H***, и ***D*** на каждом шаге внутреннего цикла для каждого шага цикла внешнего. Процесс их вывода также может быть организован в теле цикла.

На основании детального анализа задачи можно приступать к разработке блок-схемы ЭВМ-алгоритма. Вариант алгоритма с постусловием представлен ниже.

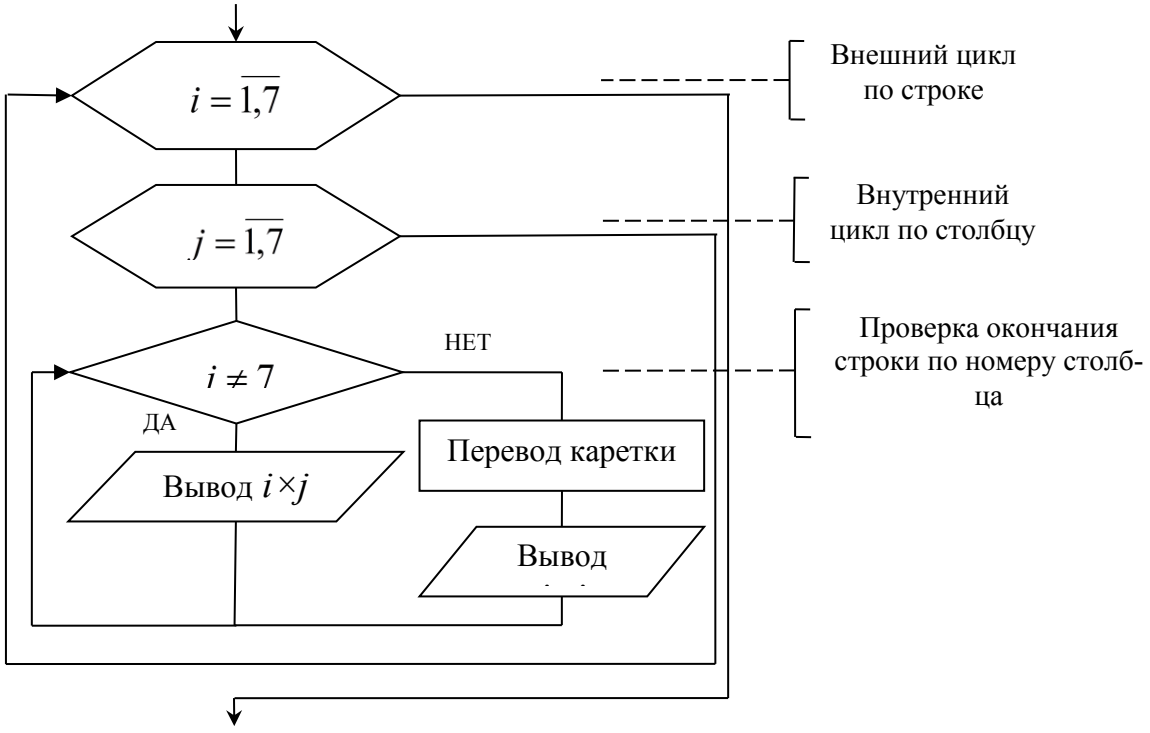


ПРИМЕР 2. Составить алгоритм получения таблицы Пифагора (таблицы умножения) размерности ***i***=7 на ***j***=7 элементов.

Особенностью этого алгоритма является то, что вывод значений должен осуществляться в виде квадрата. То есть, помимо организации вывода значений с помощью двух циклов (внешнего – по строкам и внутреннего – по столбцам), необходимо также проверить условие организации вы-

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	8	10	12	14
3	3	6	9	12	15	18	21
4	4	8	12	16	20	24	28
5	5	10	15	20	25	30	35
6	6	12	18	24	30	36	42
7	7	14	21	28	35	42	49

вода значений столбцов с новой строки.



1.7 Вопросы и задачи для самопроверки по главе 1

1.7.1 Раскрыть понятия и дать определения

1. Что понимать под термином «технология программирования»?
2. Что такое «Программа»?
3. Раскройте основные этапы процесса подготовки задачи к решению на ЭВМ.
4. Что понимать под термином «Язык программирования» и в чем его принципиальное отличие от Естественного языка?
5. Что называется «Интегрированной системой программирования», какие основные компоненты должны входить в ее состав?
6. Сформулируйте основную теорему и раскройте основополагающие принципы структурного программирования.
7. На каких понятиях основывается объектно-ориентированный подход к программированию. Раскройте эти понятия.
8. Дайте определение термину «Алгоритм» и раскройте его основные свойства.
9. Раскройте на примерах способы описания алгоритмов.
10. Приведите символы, наиболее часто применяемые при построении блок-схем алгоритмов и правила соединения этих символов.
11. Какой алгоритм называется «линейным»? Приведите пример и соответствующую блок-схему.

12. Какой алгоритм называется «алгоритмом с разветвляющейся структурой»? Приведите пример и соответствующую блок-схему.
13. Какой алгоритм называется «алгоритмом с циклической структурой»? Приведите пример и соответствующую блок-схему.
14. Как различают циклические алгоритмы по структуре? В чем принципиальное различие этих структур? Приведите примеры.
15. Приведите примеры алгоритмов смешанной структуры а алгоритмов со структурой вложенных циклов.

1.7.2 Разработать задачу методом структурного программирования и построить блок-схему алгоритма

1. Дан треугольник со сторонами, величины которых соответственно равны числам **a, b, c.** Определить высоты этих сторон. Значения сторон ввести в ПК с клавиатуры, а вывод их высот после вычислений осуществить на экран монитора.

2. Составить алгоритм задачи нахождения и вывода на экран минимального элемента среди элементов последовательно вводимых чисел ***x1, x2, …, x21***.

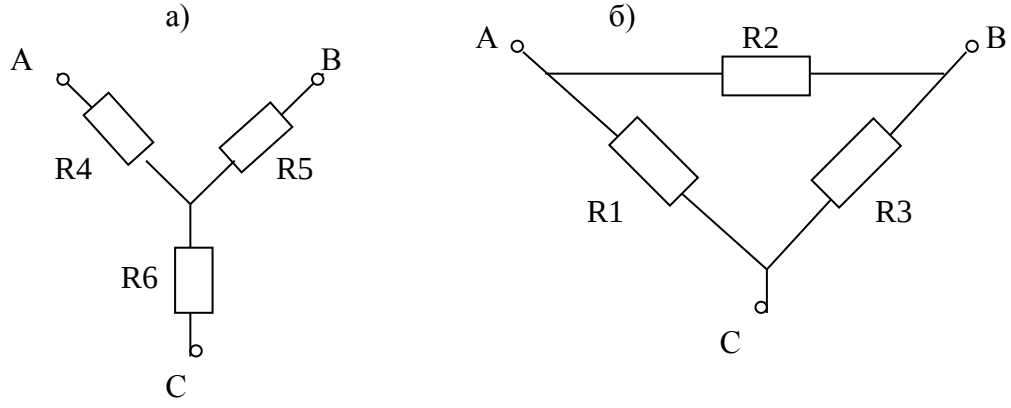
3. Формализовать задачу и составить алгоритм для нахождения и вывода на экран произведения ***20*** чисел, вводимых с клавиатуры.

4. Определить и вывести на экран дискриминант решения квадратного уравнения вида: $\mathbf{y=ax^2+bx+5}$, если его значение положительно. Значения величин **a** и **b** задаются с клавиатуры. Если величина дискриминанта отрицательна, то на экран выводится сообщение об этом.

5. Ввести с клавиатуры последовательность из пяти целых чисел и подсчитать их сумму после ввода всех пяти чисел. Если значение суммы меньше числа 100, то вывести на экран сообщение об этом. Если значение суммы больше или равно числу 100, то вывести полученное значение.

6. Даны три числа ***a, b, c.*** Найти и вывести на экран их сумму и их разность.

7. Определить и вывести на экран сопротивление в электрической цепи (рис. а)–звезда, рис. б)–треугольник). Рассчитать и вывести на экран величины сопротивлений при преобразовании звезды в треугольник и наоборот.



Примечание 1: при преобразовании звезды в треугольник расчетные соотношения:

$$R_1 = \frac{R}{R_5}; R_2 = \frac{R}{R_6}; R_3 = \frac{R}{R_4}; R = R_4R_5 + R_5R_6 + R_6R_4$$

Примечание 2: при преобразования треугольника в звезду расчетные соотношения:

$$R_4 = \frac{R_1R_2}{R}; R_5 = \frac{R_2R_3}{R}; R_6 = \frac{R_3R_1}{R}; R = R_1 + R_2 + R_3$$

8. Вычислить и вывести на экран координаты точки, делящей отрезок ***[a1, a2]*** в отношении ***n1:n2*** по формулам

$$x = \frac{x1 + \frac{n1}{n2}x2}{1 + \frac{n1}{n2}}; \quad y = \frac{y1 + \frac{n1}{n2}y2}{1 + \frac{n1}{n2}},$$

где ***x1, y1*** – координаты точки ***a1***, ***x2, y2*** – координаты точки ***a2***.

9. Вычислить значение функции $Z = \frac{x^3}{y}$, где $y = \sin(nx) + 0.5$ (провести проверку по $y \neq 0$)

10. Вычислить $Z = \begin{cases} Sin(x), x \le a; \\ Cos(x), a < x < b \\ Tg(x), x \ge b \end{cases}$

11. Вычислить корни квадратного уравнения вида: $y=ax^2+bx+5$. Если значение $d=b^2-4ac \ge 0$, то корни действительные и, следовательно, необходимо вычислить $x_{1,2} = \frac{-b \pm \sqrt{d}}{2a}$. Если значение $d<0$, то корни мнимые и, следовательно, необходимо вычислить величины e и f по формулам $e = \frac{-b}{2a}$, $f = \frac{\sqrt{|d|}}{2a}$ Значения величин ***a*** и ***b*** задаются с клавиатуры. Если величина дискриминанта отрицательна, то на экран выводится сообщение об этом.

12. Определить, попадает ли точка с координатами *(x1.y1)* в круг радиуса r. Уравнение окружности $r^2=x^2+y^2 \Rightarrow \mathbf{r = \sqrt{x^2 + y^2}}$ (*Здесь следует вначале вычислить радиус заданного круга по начальным координатам x и y, затем вычислить расстояние до новой точки* $\mathbf{d = \sqrt{(x - x1)^2 + (y - y1)^2}}$ *с координатами x1 и y1. Сравнивая* ***r*** *и* ***d*** *– найти искомое решение)*. Вывести признак N=1, если точка находится внутри круга, и признак N=0, если точка находится вне круга.

13. Используя графический символ «Подготовка» составить алгоритм программы, которая выводит таблицу квадратов первых десяти положительных чисел.

14. Используя графический символ «Подготовка» составить алгоритм программы, которая выводит таблицу степеней двойки от нулевой до десятой.

15. Используя графический символ «Подготовка» составить алгоритм программы, которая выводит таблицу значений функции $y=-2.4x^2+5x-3$ в диапазоне от –2 до 2, с шагом 0.5.

16. Используя конструкцию цикла с предусловием составить алгоритм программы, которая выводит сумму и среднее арифметическое первых *n* положительных целых чисел.

17. Используя конструкцию цикла с предусловием составить алгоритм программы, которая выводит сумму первых *n* членов ряда: 1, 3, 5, 7, …Количество суммируемых членов ряда вводится с клавиатуры во время работы программы.

18. Используя конструкцию цикла с постусловием составить алгоритм программы, которая выводит на экран таблицу умножения на число 7 чисел от 1 до 9.

19. Используя конструкцию цикла с постусловием составить алгоритм программы, которая выводит на экран сумму первых *n* членов ряда: $1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+...$ Количество суммируемых членов ряда вводится с клавиатуры во время работы программы.

20. Вычислить сумму положительных элементов каждой строки матрицы A(10×8). Считается, что элементы матрицы a_{ij} уже заданы, т.е. их не нужно вводить с клавиатуры.

21. Упорядочить элементы последовательности вводимых чисел ***x1, x2, …, x10***, расположив их по возрастанию после окончанию ввода и вывести упорядоченные значения на экран.

22. Предположим, что существует база данных контролирующей программы, состоящая из 10 ($i=1,\ldots,10$) вопросов (v_i), каждому из которых приписан правильный ответ (x_i). Пусть эти вопросы последовательно выводятся на экран. Также на экран выводится 5 вариантов ответов к нему (o_j, $j=1,\ldots,5$). Необходимо составить алгоритм, по которому на каждый выводимый вопрос, обучаемый должен дать свой ответ y_i из предлагаемого набора ответов o_j. Если ответ правильный (совпадает с ответом, приписанным к вопросу в базе контролирующей программы ($y_i = x_i$)), то обучаемому добавляется один балл. Все полученные баллы k суммируются. Если $k \geq 9$, то выводится выставляемая оценка, равная ***5***, если $6 \leq k \leq 8$, то выводится оценка ***4***, если $3 \leq k \leq 5$, то выводится оценка ***3***, если $k < 3$, то выводится оценка ***2***.

2 ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ С/С++

Язык Си++ - это универсальный язык программирования, способный реализовать объектно-ориентированный подход. Двумя его основными предшественниками были языки Си и Симула. Важнейшие свойства объектно-ориентированного программирования в языке Си++ являются развитием идей, заложенных в языке Симула.

Язык *Си* был создан *Деннисом Ричи* прежде всего для операционной системы *Unix*. Постепенно Си приобрел широкое признание, появились его реализации для практически всех операционных систем, и он стал стандартным

языком системного и, во многом, прикладного программирования.

От Си язык Си++ унаследовал эффективность, необходимую для системного программирования, начиная от написания драйверов внешних устройств до разработки компиляторов и систем управления базами данных.

Совместимость с Си обусловили еще одну важную область применения Си++ - программирование сложных графических пользовательских интерфейсов. Интерфейсы прикладных программ с такой графической средой, как Microsoft Windows были разработаны, прежде всего, на Си. С развитием и распространением Си++ разработка графических пользовательских интерфейсов перешла к нему.

К настоящему времени ситуация с использованием различных языков продолжает меняться. Средства быстрой разработки графических приложений средствами типа Visual Basic, Powerbuilder и т.п. удовлетворили существенную часть потребности в разработке простых интерфейсов. Например, язык Java в последнее время стал все более широко употребляться для разработки интерфейсов. Тем не менее, Си++ позволяет разработать более сложные программы, поскольку с его помощью можно использовать всю мощь базовой графической среды.

Обработка сложных структур данных – текста, бизнес-информации, графических образов – еще одна область применения Си++, обусловленная его объектно-ориентированными свойствами. Объекты языка отлично подходят для представления сложных структур данных и манипуляций с ними. Для хранения сложных структур данных можно использовать объектно-ориентированные системы управления базами данных, интерфейсы которых опять-таки рассчитаны на Си++.

Построение распределенных систем, сетей, сопряжение различных программ, идеально подходит для Си++. В общем, везде, где требуется высокоэффективная программа, работающая со сложными структурными данными, естественным является выбор языка Си++.

Кроме эффективности, мощности и выразительности языка, преимуществом Си++ является возможность многократного использования программ и модулей. Для каждой из перечисленных областей и многих других, не упомянутых здесь, на языке Си++ можно создать классы и библиотеки классов, реализующие функциональность часто встречающихся объектов данной области.

Например, для обработки текстов можно создать классы для работы со строками, словами, параграфами и т.д. Для графических систем – библиотеки различных форм, окон, движений и т.п. Объектная модель обуславливает удобство создания многократно используемых элементов и, что еще более существенно, легкость их адаптации к конкретной задаче.

Сказанное не означает, что другие языки не имеют права на существование. Во-первых, существует большой класс задач, для которых самое главное – быстрота разработки, а эффективность, а иногда и даже надежность, не играют большой роли. Во-вторых, существуют специализированные языки в определенных предметных областях. В-третьих, для составления программ, которые

должны работать на разных машинах без перекомпиляции также удобнее использовать другие языки (например, Java).

Чтобы не тратить слишком много места и времени, скажем, что существует масса языков программирования, и почти каждый из них способен решать поставленную задачу. Выбор зависит от самой задачи, квалификации и знаний программистов, других программ, с которыми данная программа взаимодействует и других факторов. Просто Си++, при прочих равных, лучше.

Как мы уже отмечали, язык Си++ - это универсальный язык программирования. С его помощью можно разрабатывать программы, используя разные стили программирования: процедурный (структурный), объектно-ориентированный и т.д. Процедурное (структурное) программирование поддерживается тем, что программа на языке Си++ состоит из функций, которые вызывают друг друга. В таком варианте Си++ используется как «улучшенный» Си. Тем не менее, в определенных обстоятельствах процедурный подход имеет право на существование. Объектно-ориентированный подход обеспечивается механизмом классов. В данном разделе мы будем уделять основное внимание структурному стилю программирования на языке Си/Си++.

Несколько слов о стандарте языка. Разработчиком языка Си++ считается является *Бьерн Страуструп*. Благодаря его активной позиции и высокому авторитету, вариантов языка Си++ было не так уж много, тем не менее, почти каждая компания, разрабатывая компилятор Си++, добавляла или исключала те или иные возможности. Страдал в конечном итоге рядовой программист, для которого переход с одного компилятора на другой, а иногда даже переход с одной версии того же самого компилятора на другую, выливался в серьезную переработку программы. С появлением стандарта, стало понятно, какие свойства языка можно реализовать на всех компиляторах, а какие – только на каком-то особенном компиляторе. Поэтому, если и используются нестандартные свойства, то используются осмысленно, с пониманием всех последствий.

Не менее важным результатом стандартизации явилось то, что в процессе выработки и утверждения стандарта, язык был уточнен и дополнен рядом существенных возможностей. В обсуждении стандарта приняло участие огромное количество программистов, как практиков, так и теоретиков. В целом процесс стандартизации способствовал популяризации Си++, а сам язык стал мощнее.

Последнее техническое заседание комитета состоялось в конце 1997 года. На настоящее время стандарт утвержден Международной организацей по стандартизации ISO (International Organization for Standardization)[1].

Россия является членом Международной организации по стандартизации ISO. Поэтому принятие стандарта ISO означает автоматическое принятие его в России. Процессом стандартизации и распространением стандартов в России занимается ВНИИ Сертификации (адрес узла в Интернете:www.vniis.ru).

Важной причиной выбора языка программирования C/C++ для изучения

[1] Стандарт ANSI/ISO принят в 1997г. Это международный стандарт, его номер ISO/IES 14882. Считается, что к этому стандарту относятся версии компиляторов языка C++ старше 4.02.

является то, что данный язык признан как в рамках стандарта ANSI[2] (ANSI C), так и в рамках международного стандарта (ISO), а потенциальные возможности, заложенные в концепцию построения языка, позволяют применять его практически для всех существующих сегодня операционных систем и оболочек.

ANSI C на сегодняшний день поддерживается практически всеми распространёнными компиляторами языка C. Любая программа, написанная только с использованием стандарта и не допускающая специфических аппаратных допущений, гарантированно должна работать на любой платформе с достаточно стандартной реализацией языка C.

Как и любой язык программирования, язык C/C++ является формальным языком. Он служит для описания данных и алгоритмов их обработки на ЭВМ.

Несмотря на огромную разницу между естественными и формальными языками, у них есть много общего.

Например, изучение естественного языка является сложным процессом, включающим как обретение элементарных автоматических навыков, так и восприятие сложных абстрактных понятий. При этом возможность относительно свободного использования языка как средства общения появляется уже на ранних стадиях этого процесса, когда вообще ещё не имеет смысла говорить о знании языка. Так, подавляющее большинство населения любого крупного города общается между собой, используя разговорный язык той страны или той местности, в которой расположен этот город. Практически все, кто проживает в городе, свободно владеет разговорным языком, а вернее, навыками разговорной речи, но полностью знает язык лишь часть его жителей.

Аналогичная ситуация наблюдается и с языками программирования. Первые опыты программирования (подобно использованию навыков разговорной речи) не требуют особых познаний в области формальных языков. *Для составления работающих программ достаточно иметь интуитивные представления об алгоритмах и устройстве компьютера*. Часто бывает достаточно ознакомиться с несколькими работающими программами или даже с фрагментами таких программ, чтобы, в буквальном смысле используя образцы, успешно описывать собственные алгоритмы.

Однако грамотная речь невозможна без знания языка, а профессиональное программирование требует глубоких знаний в области языков программирования.

C/C++ является языком "общения" человека с компьютером. Основным

---

[2] Американским национальным институтом стандартов (*American National Standards Institute).*

В 1983 году Американский национальный институт стандартов сформировал комитет X3J11 для создания спецификации стандарта C. В 1989 году стандарт был утверждён как ANSI X3.159-1989 «Язык программирования C».

В 1990 году, стандарт ANSI C (с небольшими изменениями) был принят Международной организацией по стандартизации (ISO) как ISO/IEC 9899:1990.

"читателем" текстов на языке С++ является *транслятор*. Это особая программа, в обязанности которой входит проверка правильности текста программы и его последующий перевод на язык процессора – основного устройства ЭВМ, который и обеспечивает выполнение программы. У процессора «свой взгляд» на программу. Он не имеет никакого представления о содержательной стороне описываемых алгоритмов. Процессору важны *адреса, регистры, прерывания.*

С/С++ - это сложный, логически стройный и красивый язык. Его хорошее знание приводит к мастерскому владению этим языком, что позволит решать любые задачи.

Само по себе программирование без специального инструментария, называемого *интегрированной средой программирования* (ИСП), само по себе невозможно. Поэтому важным является сам факт выбора ИСП. Это зависит от целей и рассматриваемых задач и, заложенных в ИСП, возможностей. Факт самого изучения языка С/С++ от ИСП не зависит. Важно, чтобы эта среда поддерживала стандары ANSI/ISO.

Поэтому в дальнейшем излагаемый материал построен так, чтобы была возможность изучить основополагающие конструкции языка С/С++. Их изучение позволит, в свою очередь, без особого труда адаптировать полученные программы в любой ИСП для С/С++.

2.1 Структура программы

Самая короткая програимма на языке С/С++ (в дальнейшем – Си) выглядит так:

```
//Элементарная программа
void main( ) { }
```

Здесь первая строчка – комментарий к программе. Иначе, это просто сообщение для человека, которое никак не учитывается в программе компилятором[3].

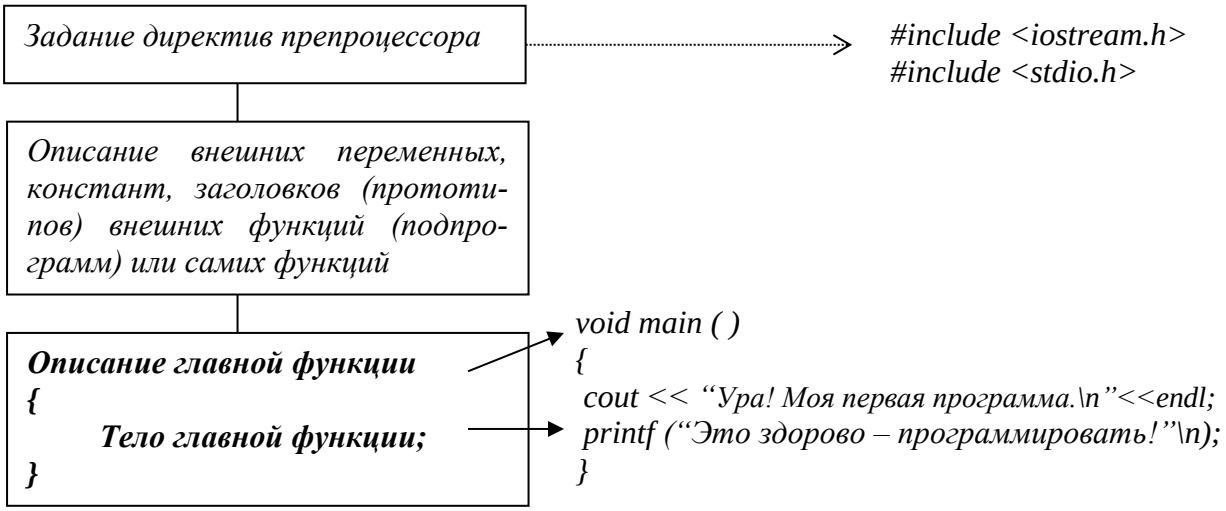
main( ) – имя *главной функции* в программе. В языке Си принято имена программ и подпрограмм называть *функциями*. Программа всегда начинает выполняться именно с этой функции. По аналогии с математикой (например, ***f(x,y)***), функция может иметь аргументы, которые перечисляются внутри круглых скобок через запятые. В данном случае аргументов нет. И так же, как в математике, функция *имеет какое либо значение* (например, ***z=f(x,y)***: величина ***z*** принимает значение функции ***f***, зависящей от аргументов ***x*** и ***y***). В программировании принято говорить: «*функция возвращает значение*». На то, какое будет это значение, указывает служебное слово, стоящее перед именем функции. Ес-

[3] Все комментарии, если они расположены в одной строке, помечаются символом двойного прямого слэша (//). Если требуется выделить в качестве комментария блок текста, то вначале блока выставляются символы /* (слэш, звездочка), а в конце – наоборот */ (звездочка, слэш).

ли это *void*, как в примере, то говорят, что функция не возвращает никакого значения.

Фигурные скобки обозначают *тело функции*. Скобка ***{*** означает начало (*begin*) функции, скобка ***}*** – ее конец (*end*). Внури скобок записываются те действия, которые должна выполнить функция. Скобки - обязательный компонент функции. Пустые фигурные скобки означают, что никакие действия программа не производит. Таким образом, наша программа ничего не делает!

Рассмотрим общие правила построения программы на языке Си, которая что-либо «делает». Синтаксис структуры программы выглядит так:



С правой стороны синтаксической структуры приведен пример текста программы на языке Си.

Если мы хотим, чтобы программа что-то «делала», то в ней необходимо создать некоторый синтаксический объект и послать ему сообщение. Сказанное равносильно тому, что для выполнения какой-либо работы мы назначаем конкретного человека. Действительно, фраза: «Эту работу нужно сделать Иванову» более конкретна, чем фраза: «Эту работу нужно сделать». В этом примере синтаксическим объектом, которому посылается сообщение, является объект «Иванов».

Объекты в программе могут быть созданы программистом, а могут быть заранее предопределены и включены в состав интегрированной системы программирования. В приведенном примере структуры программы мы воспользуемся готовыми предопределенными объектами: в первом случае это будет объект *cout* предопределенного класса *iostream (*поток ввода-вывода), а во втором – предопределенная функция форматного вывода *printf ( )* стандартного ввода-вывода (*stdio*). Предопределенные классы и функции определены в файлах заголовков с расширением *.h* – от латинского *head* (голова). Поэтому первое, что надо сделать при написании программы– ***включить файлы заголовков*** для использования объектов в нашей программе. Подключение файлов делается специальной директивой (указанием) ***# include*** < >, называемой *директивой препроцессора*.

Таким образом,

```
#include <iostream.h>
#include <stdio.h>
void main( ) { }
```

означает, что в программе будет использоваться глобальный объект потокового ввода-вывода (например, *cout*) и глобальная функция стандартного ввода-вывода данных (например, *printf ( ))*. Глобальными они называются потому, что доступ к ним возможен из любой части программы. Эти синтаксические объекты выполняют вывод на консоль (экран монитора). В функции *main( )* мы можем к ним обратиться и послать сообщение:

```
#include <stdio.h>
#include <iostream.h>
void main( )
{
cout << "Ура! Моя первая программа."<<endl;
printf ("Это здорово – программировать!\n");
}
```

Таким образом, программа посылает объекту *cout* сообщение: «Вывести (<<) строку *Ура! Моя первая программа.*» и сообщение: «Вывести (<<) перевод строки (*endl*) », а для функции *printf( )* сообщение: « Вывести на экран *Это здорово – программировать!* и перевести каретку на новую строку (\n)». Обратите внимание, что сообщения заключаются в двойные аппострофы (“ ”). В ответ на эти сообщения и объект *cout* и функция *printf( )* выполнят указанные действия, то есть выводят на консоль соответствующие строки.

Из рассмотренного примера и обобщенной структуры программы следует, что программа на языке Си состоит из одной или нескольких функций. Главной и обязательной функцией в ней является функция ***main( )*** – с нее начинается выполнение программы. Других функций может и не быть, или они могут быть либо предопределены в заголовочных файлах, либо созданы самими программистами и объявлены как глобальные для доступа к ним из любого места программы.

Функции в программах могут определяться как в одном физическом файле вместе с функцией ***main( )***, так и в различных физических файлах. Во втором случае говорят о многофайловых программах.

На начальных стадиях изучения Си мы будем предполагать, что вся программа состоит только из одного файла с главной функцией ***main( )*** и вызываемых ей функций из заголовочных файлов.

2.2 Компиляция и выполнение программы

Программа на языке Си – это обычный текст, который может быть составлен в любом текстовом редакторе. Для того чтобы компьютер смог выполнить программу, ее нужно перевести на язык машинных инструкций. Эту задачу выполняет компилятор. Он читает тектовый файл с программой, анализирует его,

проверяет на наличие синтаксических ошибок. Если ошибок не обнаружено, то компилятор создаст исполняемый файл. Откомпилированную программу можно выполнять многократно с различными исходными данными.

Все перечисленные функции и еще множество дополнительных возможностей при компилировании, отладке и исполнении программы наиболее удобно выполнять в интегрированной среде программирования. Рассмотрим пример компиляции и выполнения программы в какой-либо ИСП. Пусть, например, такой средой будет любая версия Visual C++.

Общий порядок создания программы в этой среде таков.

1. ***Создается новый проект:*** меню *File* ► атрибут *New* ► диалоговое окно, закладка *Project* ►тип выполняемого файла *Win 32 Consol Application,* отметить кнопку *Creat new workspace* ► в поле *Project name* набрать имя проекта (например, *test*) ► в поле *Location* задать имя каталога, в котором будут храниться все файлы, относящиеся к данному проекту ► нажать кнопку «*OK*».

2. ***Создается новый файл:*** меню *File* ► атрибут *New* ► диалоговое окно, закладка *File* ►тип создаваемого файла *text file* ► в поле *File name* набрать имя файла (например, *first.cpp*) ► поле *Location* должно показывазадать на ранее определенное имя каталога файлов проекта ► нажать кнопку «*OK*».

3. ***Набирается текст программы:*** вышеуказанный пример в пункте 2.1.

4. ***Компилируется текст:*** клавиша *F7* или меню *Build* ► пункт *Build test.exe.*

В нижней части экрана появится окно с сообщениями об ошибках. Если таковые есть, то двойной щелчек по сообщению об ошибке укажет на место в программе, где эта ошибка допущена. После исправления всех ошибок система выдаст сообщение об успешной компиляции и компановке (сообщение *Linking*).

5. ***Вызывается исполнение программы:*** клавиша *Ctrl+F5* или меню *Build* ► пункт *Execute test.exe.* На окне монитора появится консольное окно. В нем будут выведены три строки:

Ура! Моя первая программа.
Это здорово – программировать!
Press any key to continue

Последняя надпись озночает, что программа выполнилась и лишь ожидает нажатия произвольной клавиши для закрытия консольного окна.

Итак, главное из сказанного.

Первое – «скелет программы». ***Второе –*** алгоритм работы в ИСП.

«Говорить» мы научились. Теперь – научимся тому, «как говорить».

2.3 Алфавит языка программирования

Как и любой естественный язык, язык Си имеет свой алфавит (или множество литер). Он основывается на множестве символов таблицы кодов ASCII (*American Standard Code for Information Interchange* - американская стандартная

кодировочная таблица для печатных символов и некоторых специальных кодов)[4] и включает:

- строчные и прописные буквы латинского алфавита (мы их будем называть буквами);
- цифры от *0* до *9* (назовём их буквами-цифрами);
- символ «_» (подчерк - также считается буквой);
- набор специальных символов: « *" { } , | [ ] + - % / \ ; ' : ? < > = ! & # ~ ^ . * »*;
- прочие символы.

Алфавит С++ служит для построения слов, которые в С++ называются лексемами. Различают пять типов лексем:

- ✓ идентификаторы;
- ✓ ключевые слова;
- ✓ знаки (символы) операций;
- ✓ литералы;
- ✓ разделители.

Почти все типы лексем (кроме ключевых слов и идентификаторов) имеют собственные правила словообразования, включая собственные подмножества алфавита.

Лексемы разделяются разделителями, к числу которых относятся *пробел*, символы горизонтальной и вертикальной *табуляции*, символ *новой строки*, перевода *формата* и *комментарии*.

2.3.1 Идентификаторы

Идентификаторы – это, по-существу, имена, которые присваиваются переменным, функциям, или другим лексемам языка Си. Основные правила построения идентификаторов из букв алфавита следующие:

1) Первым символом идентификатора может быть только буква.
2) Следующими символами идентификатора могут быть буквы, буквы-цифры и буквы-подчерки.
3) Длина идентификатора неограниченна (фактически же длина зависит от реализации системы программирования).

Вопреки правилам словообразования, в Си существуют ограничения относительно использования подчерка в качестве самой первой буквы в идентификаторах. Особенности реализации делают нежелательными для использования идентификаторы, которые начинаются с этого символа.

2.3.2 Ключевые слова и имена

Часть идентификаторов Си входит в фиксированный словарь ключевых слов. Эти идентификаторы образуют подмножество ключевых слов (они так и

---

[4] ASCII представляет собой кодировку для представления десятичных цифр, латинского и ***национального*** алфавитов, знаков препинания и управляющих символов.

называются *ключевыми* словами). Прочие идентификаторы после специального объявления становятся именами. Имена служат для обозначения переменных, типов данных, функций и меток. Ключевые слова ***нельзя*** применять для объявления пользовательских идентификаторов. Ниже приводится список некоторых ключевых слов:

asm, auto, break, case, catch, char, class, const, continue, default, 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 и др.

В случае примения таких слов для объявления пользовательских имен компилятор ИСП выдаст ошибку.

2.3.3 Символы операций и разделители

Множество лексем, соответствующее множеству символов операций и разделителей, строится на основе набора специальных символов и символов-букв алфавита. Единственное правило словообразования для этих категорий лексем заключается в задании фиксированного множества символов операций и разделителей.

Следующие последовательности специальных символов и букв алфавита образуют множество символов операций (часть из них в зависимости от контекста может быть использована в качестве разделителей):

,	**!**	**!=**	**|**	**|=**	**%**	**%=**	**&**
&&	**&=**	**( )**	*****	***=**	+	++	+=
-	--	-=	->	->*	.	.*	/
/=	::	<	<<	<=	<<=	>	>>
>=	>>=	= =	**?:**	**[ ]**	^	^=	~
||	#	# #	**sizeof**	**new**	**delete**	**typeid**	**throw**

Кроме того, к числу разделителей относятся следующие последовательности специальных символов:

...	**;**	**{ }**

2.3.4 Литералы

В C++ существует четыре типа литералов:

- целочисленный литерал,
- вещественный литерал,
- символьный литерал,
- строковый литерал.

Это особая категория слов языка. Для каждого типа литералов используются собственные правила словообразования. Мы не будем приводить здесь эти правила. Ограничимся лишь общим описанием структуры и назначения каждого типа литералов. После этого правила станут более-менее понятны.

Целочисленный литерал служит для записи целочисленных значений и является соответствующей последовательностью цифр (возможно со знаком минус). Целочисленный литерал, начинающийся с *0*, воспринимается как вось-

миричное целое число. В этом случае цифры *8* и *9* не должны встречаться среди составляющих литерал символов. Например, запись *03* соответствует десятичному числу 3, азапись *010* – десятичному числу *8*. Целочисленный литерал, начинающийся с *0x* или *0X*, воспринимается как шестнадцатеричное целое число. В этом случае целочисленный литерал может включать символы от *A* или *a*, до *F* или *f*, которые в шестнадцатиричной системе эквивалентны десятичным значениям от 10 до 15. Непосредственно за литералом может располагаться в произвольном сочетании один или два специальных суффикса: *U* (или *u*) и *L* (или *l*). Эти суффиксы обозначают специальны формат числа (например «беззнаковый» или «длинный»).

Вещественный литерал служит для отображения вещественных (действительных) значений. Он фиксирует запись соответствующего значения в обычной десятичной или научной нотациях. В научной нотации мантисса отделяется от порядка литерой *E* или *e*). Например, запись *0.5e-2* соответствует вещественному числу *0.005*. Непосредственно за литералом могут располагаться один из двух специальных суффиксов: *F* (или *f*) – вещественный (*float*) и *L* или *l* – длинный (*long*).

Значением *символьного литерала* является соответствующие значения ASCII-кода. Это не только буквы, буквы-цифры или специальные символы алфавита Си. Символьный литерал представляет собой *последовательность из одной или нескольких литер, заключённых в одинарные кавычки*. Символьный литерал служит для представления литер в одном из форматов представления. Например, литера *Z* может быть представлена литералом *'Z'*, а также литералами *'\132'* и *'\x5A'*. Любая литера может быть представлена в нескольких форматах представления: обычном формате, восьмиричном или шестнадцатиричном. Допустимый диапазон для обозначения символьных литералов в десятичном представлении ограничен десятичными числами от *0* до *255*. Допустимый диапазон для обозначения символьных литералов в восьмиричном представлении ограничен восьмиричными числами от *0* до *377*. Допустимый диапазон для обозначения символьных литералов в шестнадцатиричном представлении ограничен шестнадцатиричными числами от *0x0* до *0xFF*. Литеры, которые используются в качестве служебных символов при организации формата представления или не имеют графического представления, могут быть представлены с помощью ещё одного специального формата. Ниже приводится список литер, которые представляются в этом формате. К их числу относятся литеры, не имеющие графического представления, а также литеры, которые используются при организации структуры форматов.

Список таких литер организован по следующему принципу: сначала приводится представление литеры в специальном формате, затем – эквивалентное представление в шестнадцатеричном формате, далее – обозначение или название литеры, за которым приводится краткое описание реакции на литеру (смысл литеры).

\0	***\x00***	***null***	*пустая литера*
\a	***\x07***	***bel***	*сигнал*

\b	***\x08***	***bs***	*возврат на шаг*
\f	***\x0C***	***ff***	*перевод страницы*
\n	***\x0A***	***lf***	*перевод строки*
\r	***\x0D***	***cr***	*возврат каретки*
\t	***\x09***	***ht***	*горизонтальная табуляция*
\v	***\x0B***	***vt***	*вертикальная табуляция*
\\	***\x5C***	\	*обратная косая черта*
\'	***\x27***	'	*апостроф*
\"	***\x22***	"	*двойная ковычка*
\?	***\x3F***	***?***	*вопросительный знак*

Строковые литералы являются последовательностью (возможно, пустой) литер в одном из возможных форматов представления, заключённых в двойные кавычки. Строковые литералы, расположенные последовательно, соединяются в один литерал, причём литеры соединённых строк остаются различными. Так, например, последовательность строковых литералов *"\xF" "F"* после объединения будет содержать две литеры, первая из которых является символьным литералом в шестнадцатиричном формате *'\F'*, второй – символьным литералом *'F'*. Строковый литерал и объединённая последовательность строковых литералов заканчиваются пустой литерой, которая используется как индикатор конца литерала.

Таким образом, как сказано в справочном руководстве по C++, файл с программой на Си состоит из последовательности объявлений различных синтаксических единиц языка программирования. Синтаксическую единицу, отвечающую за выполнение конкретного задания, в Си принято называть *функцией*.

В свою очередь, функция состоит из заголовка, который включает спецификаторы объявления, описатели и инициализаторы и тела.

Тело функции представляет собой блок, построенный на основе алфавита языка из различных синтаксических конструкций, заключаемый в фигурные скобки.

2.4 Типы данных

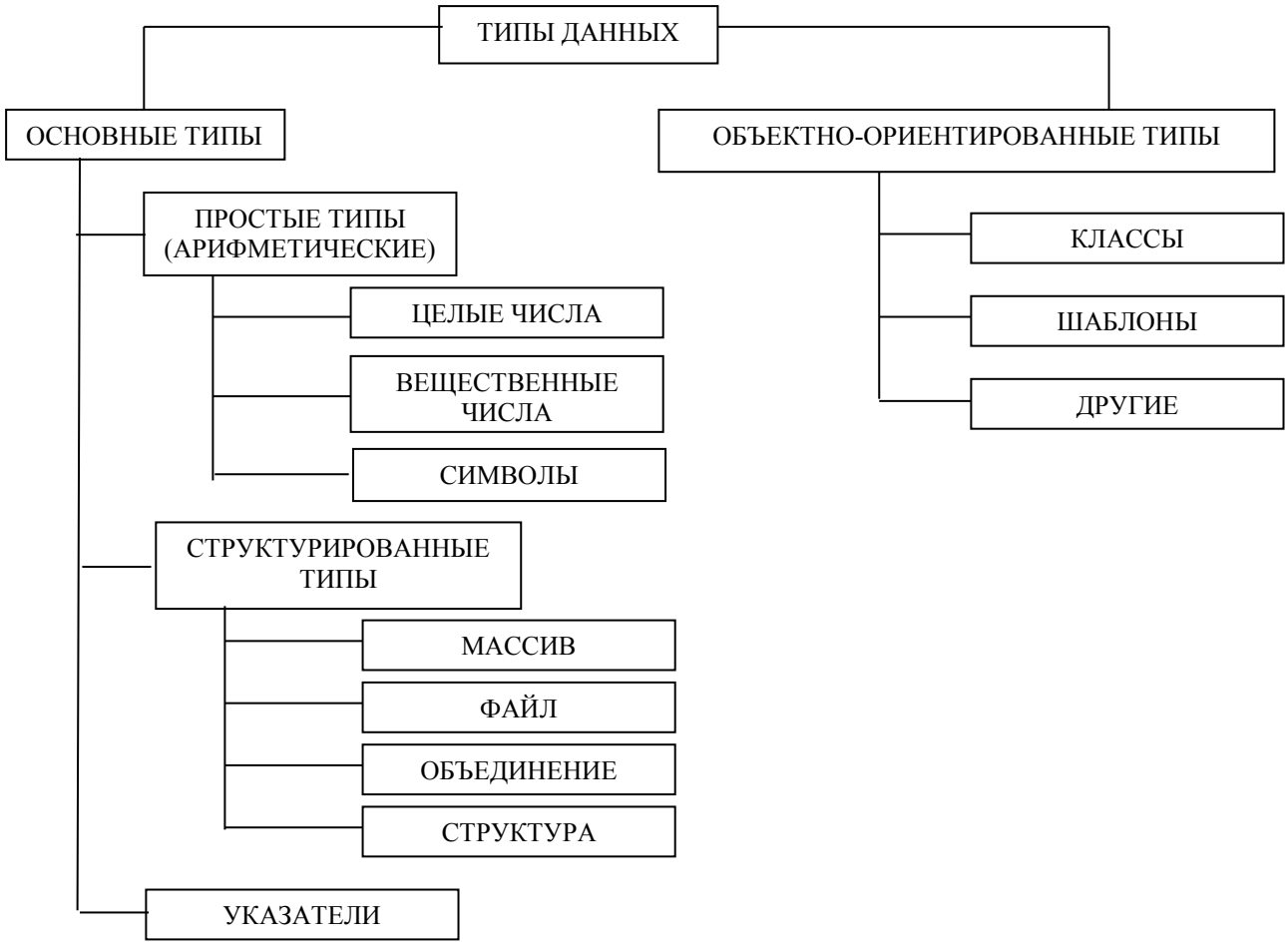
2.4.1 Типы данных и объявление переменных

Определение. *Переменная* – величина, которая может быть изменена при выполнении программы.

Когда речь идет об описании какой-либо переменной, то имеют в виду задание этой переменной ее уникального *имени*, а также *типа*, соответствующего тем данным, с которыми эта переменная будет оперировать (целые или вещественные числа, символы и т.д.).

Определение типа для переменной необходимо для того, чтобы определить какое количество памяти компьютера необходимо будет выделить, чтобы оперировать теми или иными данными.

Условно, все типы данных языка С++ можно представить в виде древовидной структуры, представленной ниже.



Структурированные типы данных и особый тип указатель, а также объектно-ориентированные типы мы рассмотрим в последующем изложении материала. Здесь мы начнем изучение с представления и назначения простых (арифметических)типов данных для описания переменных.

2.4.2 Простые типы данных

Машинное представление программы предполагает, что каждой переменной, как было сказано выше, отводится некоторый участок памяти. Размер этого участка и интерпретация его содержимого определяется типом.

Помимо непосредственно типа данных в языке Си существует такое понятие, как модификатор типа. Поэтому, описание типа переменной может использоваться в сочетании с парой модификаторов: *signed, unsigned.*

Эти модификаторы изменяют формат представления данных, но не влияют на размеры выделяемых областей памяти.

Модификатор типа *signed* указывает, что переменная может принимать как положительные, так и отрицательные значения. Возможно, что при этом самый левый бит области памяти, выделяемой для хранения значения, используется для представления знака. Если этот бит установлен в 0, то значение пе-

ременной считается положительным. Если бит установлен в 1, то значение переменной считается отрицательным.

Модификатор типа *unsigned* указывает, что переменная принимает неотрицательные значения. При этом самый левый бит области памяти, выделяемой для хранения значения, используется так же, как и все остальные биты области памяти – для представления значения.

Основные характеристики целочисленных типов выглядят так:

Тип данных	**Назначение**	**Байты**	**Биты**	**Диапазон значений**	
				Min	*Max*
signed int	Целое со знаком	2	16	-32768	32767
unsigned int	Целое без знака	2	16	0	65535
signed short	Короткое целое со знаком	2	16	-32768	32767
unsigned short	Короткое целое без знака	2	16	0	65535
signed long	Длинное целое со знаком	4	32	-2147483648	2147483647
unsigned long	Длинное целое без знака	4	32	0	4294967295
float	Вещественное число, с плавающей точкой	4	32	3.4E-38	3.4E+38
double	Вещественное двойной точности	8	64	1.7E-308	1.7E+308
long double	Длинное вещественное число двойной точности	10	80	3.4E-4932	3.4E+4932
signed char	Символьный тип со знаком	1	8	- 128	127
unsigned char	Символьный тип без знака	1	8	0	255

Плавающие типы используются для работы с вещественными числами, которые могут быть представлены двумя способами: в форме записи с десятичной точкой и в форме "научной нотации". Например, записи *297.7* или *0.002355* – представление чисел в форме записи с десятичной точкой. А записи *2.977E2* или *2.355E-3* – представление тех же чисел, но в форме научной нотации. В научной нотации слева от символа *E* записывается мантисса, справа – значение экспоненты, которая всегда равняется показателю степени 10. Поэтому представление *2.977E2* можно трактовать так: $2.977 \cdot 10^2$. А представление *2.355E-3* трактуется как $2.355 \cdot 10^{-3}$.

Таким образом, имена типов данных и их сочетания с модификаторами типов используются для представления данных различных размеров в знаковом и беззнаковом представлении.

Все эти типы образуют множество простых (арифметических) типов.

2.4.3 Объявление переменных

Как было сказано выше, процедура ввода имени переменной предполагает не только создание отличного от любого ключевого слова идентификатора, но и кодирование дополнительной информации о характеристиках переменной, с которой будет связано объявляемое имя.

К основным характеристикам переменной, помимо ее имени, относятся *тип*, *класс памяти*, *время жизни* переменной. Только в общем случае, переменные характеризуется своими именами, которые могут быть любыми, за исключением используемых ключевых имен.

Синтаксис объявления переменных таков:

Тип переменной Имя_переменной [= начальное значение];

Все описанные переменные в программах должны принимать какое-либо значение. Это значение с помощью оператора присваивания (=) может быть *присвоино им явно* при объявлении, например *int a=0* (переменной целого типа с именем *a* присвоено значение *0*), или вычисляться в ходе выполнения программы, например,

```
{
float b; //объявлена переменная b вещественного типа
b = sin(0); //переменная b принимает значение результата вычисления
            //функции sin(0)
}
```

2.5 Операции, выражения, операторы

Так как «операция» и «выражение» тесно взаимосвязанные понятия, то вначале дадим определение выражения

Определение. *Выражение* – это формула, состоящая из одного или более числа *операндов* (переменных, констант и др. конструкций языка), соединенных знаком *операции*.

Определение. *Операция* – это конструкция языка Си, состоящая из одного или более символов, фиксирующих определенное действие.

Порядок выполнения операций при вычислении выражения определяется их приоритетом (рангом) и может регулироваться с помощью круглых скобок. Заключенное в скобки выражение интерпретируется компилятором в первую очередь.

Кратко символы операций мы рассматривали при изучении алфавита языка Си (см. п.2.3.3). Более подробно рассмотрим приоритет операций и их градацию на группы с помощью сводной таблицы.

СВОДНАЯ ТАБЛИЦА ОСНОВНЫХ ОПЕРАЦИЙ ПО ГРУППАМ

ГРУППА	ЗНАК	НАЗНАЧЕНИЕ	ПРИМЕР ТИПА И ЗНАЧЕНИЯ ДАННЫХ	ПРИМЕР ВЫРАЖЕНИЯ	РЕЗУЛЬТАТ
1	2	3	4	5	6

Присваивания	=	Переменной или выражению, стоящему слева от знака присваивается значение переменной или выражения, стоящего справа от знака	***int*** a, b, c;	**A) обычное**
				b=2; c=b;
				B) множественное
				b=c=a=5;
				C) составное
				a*=2, b+=2, c/=2
					b=2, c=2
					
					b=5,c=5,a=5
					
					a=a*2, b=b+2, c=c/2

Арифметические	* / % + -	Умножение Деление Остаток деления Сложение Вычитание	***int*** a=5, b=3, c; ***float*** a=5,b=3,c; ***int*** a=5, b=3, c; ***int*** a=7, b=3, c; ***float*** a=5,b=3,c; ***int*** a=5, b=3, c;	c=a*b c=a/b c=a/b c=a%b c=a+b c=a-b	c=15 c=1.666667 c=1 c=2 c=8.0 c=2
Унарные	++ – – 	Инкремент Декремент Составные	***int*** a, b, c;	c=++a c=a++ c=– –b c=b– – c=a– –*2 c=a+++b	c=a+1,a=a+1 c=a+1,a=a c=b-1,b=b-1 c=b-1,b=b c=(a-1)*2,a=a c=(a+1)+b, a=a, b=b

Отношения	= =
	!=
	<
	>
	<=
	
	>=
		Равно
		Не равно
		Меньше
		Больше
		Меньше или равно
		Больше или равно
			int a, c;	c= =a;
				c!=a;
				c<a;
				c>a;
				c<=a;
				
				c>=a;
					Значение левого от знака выражения (или переменной) относительно значения правого выражения

Логические	**!** **&&** ||	Логическое НЕ Логическое И Логическое ИЛИ	***int*** a, c;	!a; c&&a; c||a;	Принять не а Принять и с и а Принять или с или а
Поразрядные	~ >>	Поразрядное отрицание Поразрядный сдвиг влево	Действует на один операнд целого типа Действует на левый операнд целого типа	***unsigned char*** E='\0301', F; F=~E; c=5; c<<2;	E=11000001 ('⊥') F=00111110 ('>') c=20, т.к. 5 = =101<<2=10100

	<<	Поразрядный сдвиг вправо	Действует на левый операнд целого типа	c=5; c>>2;	c=1, т.к. 5 = =101>>2=001
	^	Поразрядное исключающее ИЛИ	Поразрядная обработка битовых кодов целых операндов	***char*** a=’A’, ***char*** z=’Z’ a=a^z; z=a^z; a=a^z;	a=01000001 z=01011010 a=00011011 z=01000001='A' a=01011010='Z'
	|	Поразрядное ИЛИ	Применима к операндам целого типа	c=5|6; b=5|4	101|110=111=**7** 101|100=101=**5**
	&	Поразрядное исключающее И	Применима к операндам целого типа	c=5&6; b=5&4	101&110=100=**4** 101&100=100=**4**
Условное выражение	**?:**	Трехместная операция формирования условного выражения Операнд1 **?** Операнд2 **:** Операнд3	Три операнда. Операнд1: арифметическое или логическое выражение-условие. Операнд2: результат по выполнению условия Операнд3: результат по невыполнению условия Операнды 2 и 3 одного типа.	***char*** c=5; ***int*** a=2, b=3; c=(c>0)?a:b; c=(c<0)?a:b;	c=a=2 c=b=3
Преобразование типа	(тип)	Преобразует первоначальный тип переменной в новый тип	Применима к любым типам	***int*** a=1; ***float*** b; b=(float)a;	Теперь b=1.0, т.к. целое значение переменной ***a*** преобразовано в вещественный тип

Операции имеют приоритет выполнения, то есть очередность выполнения в составе выражения. Приоритет символов операций можно представить таблицей:

Ранг	**Операции**	**Группа**
1.	**( ) [ ] -> .**	Выбор компонентов
2.	**!** (НЕ-логическое отрицание), **~** (побитовое отрицание), **+ +** (инкремент), **- -** (декремент), **&** (получение адреса), ***** (обращение по адресу), **(тип)** (преобразование типа), **sizeof** (вычисление размера в байтах)	Унарные
3.	*****(умножение), **/** (деление), **%** (остаток от деления)	Мультипликативные - бинарные
4.	**+(**сложение), **-** (вычитание)	Аддитивные - бинарные
5.	<<(сдвиг влево), >>(сдвиг вправо)	Поразрядного сдвига

		- бинарные
6.	< (меньше), <= (меньше или равно), >= (больше или равно), > (больше), = = (равно), **!=** (не равно)	Бинарные
7.	**&** (поразрядная конъюнкция «И»)	Бинарные
8.	^ (поразрядная исключающее «ИЛИ»)	Бинарные
9.	| (поразрядная дизъюнкция «ИЛИ»)	Бинарные
10.	**&&** (конъюнкция «И»)	Бинарные
11.	|| (дизъюнкция «ИЛИ»)	Бинарные
12.	**?:** (условная операция)	Трехместная
13.	=, *****=, /=, **%**=, +=, -=, **&**=, ^=, |=, <<=, >>=	Присваивания - бинарные
14.	**,** (операция запятая)	

Еще одним ключевым понятием в Си, кроме операции и выражения, является понятие *оператора*.

Определение. Любая конструкция языка Си, заканчивающаяся точкой с запятой (***;***), называется *оператором*.

Следует различать понятия «выражение» и «оператор». Оператор (в чистом виде) не использует в своей конструкции знаков операций. С другой стороны, если после выражения стоит точка с запятой, то это выражение конструктивно является оператором.

Например, *float maks; sin(x); printf( );* – это операторы, а *maks = sin(x);* – одновременно и оператор, и выражение.

Операторы являются основными «строительными» блоками программы. Программа состоит из последовательности операторов.

По функциональному назначению можно выделить пять типов операторов:

1. *описания* (например, *float m;//*описана переменная *m* вещественного типа);
2. *присваивания* (например, *m = sin(x)+25;//*переменной *m* присвоено значение выражения, стоящего справа от операции присваивания «=»);
3. *управления* (например, *goto n1;//*передать управление на метку *n1*);
4. *вызова функций* (например, *fun1(a,b)//*вычислить функцию *fun1* от двух аргументов-переменных *a* и *b;*);
5. *пустой оператор* (допускается использовать оператор, состоящий только из единственного символа – «*;*»).

Операторы можно объединять в блоки. Такой объединенный в единое целое блок или участок программы также рассматривается как оператор, называемый *составным оператором*.

Определение. Группа, последовательно расположенных операторов, заключенных в фигурные скобки ***{ }***, называется *составным оператором*.

Для составного оператора характерно то, что не во всех случаях после закрывающейся фигурной скобки ставится точка с запятой, например, при организации циклов.

2.6 Преобразование данных

Некоторые операции могут выполняться над разнотипными операндами. В этом случае перед выполнением операции компилятор приводит операнды к одному типу. Существуют основные правила приведения типов.

Явное приведение. Используется операция явного приведения, задающая переменной новый тип. В этом случае перед именем переменной, внутри скобок операции указывается требуемый тип. Например, *int a;…; (float)a=c+0.5;* Здесь переменная *a* вначале была установлена как переменная целого типа. Но в процессе выполнения программы оказалось, что она должна оперировать с вещественными данными. Поэтому внутри скобок операции приведения типа для переменной *a* задан тип *float*.

При явном приведении типов существуют определенные правила такого приведения: не всякий тип может быть приведен к другому типу. Типы *char* и *short* могут быть преобразованы в тип *int*. Тип *float* – в тип *double*. Тип *int* – в тип *float*;

Если один из операторов двойной точности (*double*), то и другие операторы, и результат преобразуются к двойной точности. Аналогично если один из операторов имеет тип длинного числа *long*.

Если все операторы одного типа, то такого же типа будет и результат;

При выполнении операции присваивания результат приводится к типу переменной, расположенной слева от знака равно (=). Например, ***int*** *a=3, c;* ***float*** *b=6.4; … ; c= a+b;* Здесь, несмотря на то, что в выражении участвуют разного типа переменные, значение переменной *c* будет целым числом (*c=9*) , но (ВНИМАНИЕ!) округленное до ближайшего целого.

Сказанное выше говорит о том, что при работе с типами данных необходимо аккуратно пользоваться преобразованием типов и помнить о правилах этих преобразований.

2.7 Понятие о препроцессоре

Определение. *Препроцессор* (макропроцессор) – это составная часть пакета интегрированной системы программирования, которая обрабатывает исходный текст программы до того, как он пройдет через компилятор. Препроцессор работает на первом шаге компиляции программ, выполняет подстановки для макровызовов и подключает указанные файлы.

Директивы препроцессора в программе начинаются с символа «*#*». Различают несколько деректив: *подключения файла* – ***#include***, *замещения* (замены*) лексических единиц* – ***#define*** и *отмены* этого *замещения* – #***undef***, *условной компиляции* – #***if***, #***elif***, #***else***, ***#ifdef***, ***#ifndef***, *выдачи дополнительных указаний* компилятору – #***pragma***, выдачи сообщения и *завершения компиляции* – #***error***.

Любая строка вида *#include <имя_файла>* или *#include "имя_файла"* заменяется содержимым файла с именем «*имя_файла*». Если имя соответственного файла заключено кавычками, то файл ищется среди исходных файлов пользовательской программы. Если такового не оказалось или имя файла заключено

в угловые скобки (< >), то поиск осуществляется в системных библиотеках. Признаком системной библиотеки является ее расширение, состоящее из одной буквы *h* после точки. Например, имя *stdio.h* означает системную библиотеку стандартного ввода/вывода данных.

Директива макроподстановки имеет вид: *#define идентификатор подстановка.* Она вызывает замену в оставшейся части программы названного идентификатора на текст подстановки. Например, подстановка вида *#define N 1000* заменяет каждое вхождение идентификатора *N* в тексте программы на число *1000*; подстановка *#define sin(x) ((x>0) ? x : cos(x))* заменяет каждое вхождение выражения *sin(y)* в тексте программы на выражение *((y>0) ? y : cos(y))*. При этом параметр *x* макроподстановки изменяется на фактический параметр *y* в последующем тесте программы. Текст подстановки заключают в скобки для обеспечения большей надежности программы. Подстановка эффективна в тех случаях, когда требуется проверить работу программы с новыми данными, например, для новой математической функции, используемой в программе неоднократно. Вместо того, чтобы переписывать эту функцию несколько раз в программе перед ее компиляцией, достаточно один раз изменить макроподстановку, не затрагивая основную часть программы. Более подробно о директиве замещения см. п. 2.14.5.

Директива ***#undef*** отменяет определение макроподстановки идентификатора, после нее идентификатор пкрестает быть определенным.

Исходный файл можно компилировать не целиком, а частями, используя директивы условной компиляции. Предположим, что *M* – это макроимя, тогда:

#if *M>5*
текст1
#elif *M>2*
текст2
#else
текст3
#endif

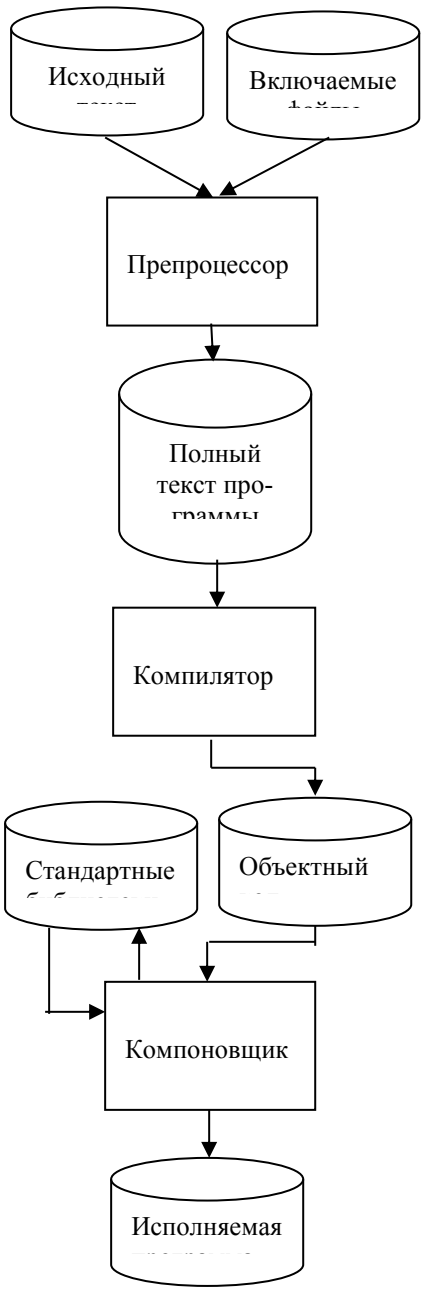
если *M>5* компилироваться будет текст1, иначе если *M>2* компилироваться будет текст2, в противном случае компилироваться будет текст3. Блок условной компиляции завершается директивой ее окончания ***#endif.***

Директив ***#elif*** может быть несколько, либо вообще ни одной. Директива ***#else*** может быть опущена.

Директива ***#ifdef*** *МАКРОИМЯ* – модификация условия компиляции. Если указанное после нее макроимя определено, то условие считается выполненным. Напротив, для директивы ***#ifndef*** *МАКРОИМЯ* условие выполняется, если макроимя неопределенно.

Возможности директивы ***#pragma*** у разных компиляторов различные. Например, с ее помощью можно организовать вставку дополнительной информации для отладчика, или не выдачу предупреждений при компиляции. Конкретные возможности директивы можно узнать, воспользовавшись справкой («хэлпом») конкретной среды программирования.

Конструкция
***#ifndef** M*
*#**error** "Макроимя неопределенно"*
#endif
выдаст соответствующее сообщение и не даст откомпилировать исходный файл, если макроимя *M* будет не определенно.



Для определения и описания препроцессорных директив в программах существуют ограничения.

Во-первых, препроцессорная директива размещается в одной строке. Во-вторых, символ «#», вводящий каждую директиву препроцессора, должен быть первым отличным от пробела символом в строке с препроцессорной директивой. В-третьих, как правило, препроцессорные директивы объявляются вначале программы.

Исходный текст программы на Си проходит три обязательных этапа обработки:

- Препроцессорное преобразование текста;
- Компиляцию;
- Компоновку.

Только после успешного завершения этих этапов формируется исполняемый машинный код программы (EXE-файл).

Схему получения исполняемого кода можно проиллюстрировать так, как это показано на рисунке слева.

Более подробно о директивах препроцессорной обработки можно узнать в книге «Подбельский В.В., Фомин С.С. Программирование на языке Си. – М: Финансы и статистика, 2001 г., стр.136-164».

2.8 Ввод/вывод информации

Стандартные библиотеки Си содержат различные функций ввода-вывода данных. Многие из них позволяют не только совершать определенные операции над определенными данными, но и, как бы, совмещать собственные уникальные действия с аналогичными действиями других функций этих библиотек. То есть, одно и то же действие над данными можно совершить, используя раз-

нообразные подходы – главное, чтобы конечная цель при этом была достигнута. Казалось бы, зачем такое многообразие подобных средств? Почему допустимы накладки в действиях? Ведь, по существу, подобный подход приводит к разрастанию программного обеспечения языка программирования. Ответ достаточно прост. Так же как и человек способен выразить собственную мысль разными способами, так и разработчики программного обеспечения стремятся предоставить пользователю более насыщенный и совершенный инструментарий разработки программ, не вычеркивая при этом прежних разработок, накапливая общий багаж знаний о языке программирования. В результате нам, пользователям, остается только определить, какой из механизмов для достижения конечной цели решения задачи будет для нас более удобным и приемлемым. Но чтобы сделать этот выбор следует изучить возможные механизмы реализации и сравнить их друг с другом.

Первыми в ряду изучаемых функций будут функции ввода-вывода данных. Это не случайно, так как благодаря им, пользователь получает простой и мощный инструментарий диалогового общения с программой. И, хотя в языке Си таких групп функций довольно много, мы рассмотрим наиболее характерные при разработке простых программ.

2.8.1 Форматный ввод/вывод

Чтобы воспользоваться ресурсами функций форматного ввода-вывода, следует в самом начале программы задать директиву препроцессора *#include* <*stdio.h*>. По этой директиве, на этапе компиляции программы, компилятор воспользуется описанием функций, хранящихся в заголовочном файле библиотеки стандартного ввода-вывода *stdio.h* (standard input-output head). Рассмотрим основные правила работы сначала с функцией вывода данных на экран *printf( )*, а затем с возможностью и правилами ввода данных посредством функции этой библиотеки *scanf( )*. Обратите внимание, что как и в математике, понятие «функция» предполагает ее имя, последующие за именем скобки, внутри которых могут располагаться аргументы. Аргументов может быть несколько или не быть вообще. О том, что это функции именно форматного ввода-вывода свидетельствует наличие буквы *f* (format) в конце имени функции перед скобками.

2.8.1.1 Функция *printf( )*

Вывод (печать) данных на экран осуществляется благодаря такому описанию синтаксиса функции *printf( )*:

printf(“форматная строка”,список аргументов);

Форматная строка ограничивается кавычками и может включать в себя произвольный текст, управляющие символы и спецификации преобразования данных. Например, задание оператора *printf(“Ура! Моя первая программа”)*, приведет к выводу на экран текста *«Ура! Моя первая программа»* в одной строке. Здесь, в качестве параметров функции использовалась только текстовая

строка. Если же выполнить функцию *printf(“Ура! \nМоя первая программа”);* – то результатом ее работы будет вывод текста на экран в следующем виде:

Ура!
Моя первая программа

Такое (построковое) представление произошло потому, что в форматной строке был задан управляющий форматом вывода управляющий символ ***\n***, переводящий курсор (точку вывода) на следующую строку экрана.

Кроме этого управляющего символа используются также следующие символы:

\t – горизонтальная табуляция;
\r – возврат каретки (курсора) к началу строки (без перевода каретки на новую строку);
\\ – обратная косая черта \, применяемая обычно для задания пути к файлам;
\’ – апостроф (символ одиночной кавычки ‘);
\” – кавычка (символ двойной кавычки “);
\0 – нулевой символ;
\a – сигнал-звонок;
\b – возврат на одну позицию (на один символ влево);
\f – перевод (прогон) страницы;
\v – вертикальная табуляция;
\? – знак вопроса.

Использование указанных символов приводит к достаточно эффективной организации вывода данных. Например, вызов функции вывода в таком виде:

printf(“\” Ура! \a\”\n\”Моя первая программа\a\” ”);

приведет к выводу на экране сообщения:

“Ура!”
“Моя первая программа”

и при этом после вывода каждой строки последует краткий звуковой сигнал.

Рассмотрим еще один пример:

printf(“Переменная x=%d”,x);

Здесь *%d* – спецификация преобразования данных. Вообще, спецификации предназначены для управления формой внешнего представления значений аргументов функции вывода. То есть для того, чтобы определить, какого типа значение должно быть выведено, какой переменной это значение должно принадлежать и как оно должно выглядеть на экране. Обобщенный формат спецификации преобразования имеет вид:

% ширина_поля . точность модификатор спецификатор

Обязательными являются только два: *%* и *спецификатор*.

Спецификаторами являются:

d – для целых десятичных чисел;

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

f – для вещественных чисел в формате с фиксированной точкой;

e – для вещественных чисел в форме с плавающей точкой.

Теперь ясно, что в предыдущем примере на экран будет выведено целое число.

Рассмотрим еще пару примеров использования спецификаторов:

Пусть

int *x=22;* ***float*** *c=534.557, e=31.4 ;*

Тогда в результате работы функции

printf(" x=%d c=%f e=%e",x,c,e);

на экране будет строка

x=22 c=534.557 e=3.1400E+01.

Если же воспользоваться функцией

printf(" x=%5d c=%6.2f e=%8.2e",x,c,e);

то ее результатом на экране будет строка

x= 22 c= 534.56 e= 3.14E+01.

Обратите внимание, что у вывода целого числа *x* появилось три сдвига вправо после знака равенства (плюс две выводимых цифры – всего пять знаков). У вещественного числа *c* выведено всего 6 символов (вместе с пробелом перед цифрой 5), причем значение двух после точки округлено; вывод величины *e* изменился до двух знаков после точки (всего 8 символов). Такой форматный вывод стал возможен благодаря указанию ширины поля вывода и его точности (количества знаков после запятой для вещественных чисел). Под шириной поля вывода подразумевается количество всех выводимых символов (не только чисел).

2.8.1.2 Функция *scanf( )*

Функция применяется для ввода данных с клавиатуры. Ее синтаксис имеет вид:

scanf("форматная строка", список аргументов);

В отличие от форматной строки функции вывода здесь форматная строка имеет вид почти аналогичный виду спецификации вывода, т.е.:

% ширина_поля модификатор спецификатор

при этом обязательными являются *%* и *спецификатор*.

Аргументами для функции ввода могут быть только *адреса объектов* программы, в частном случае – адреса переменных. Для обозначения адреса переменной перед ее именем указывается знак *&*.

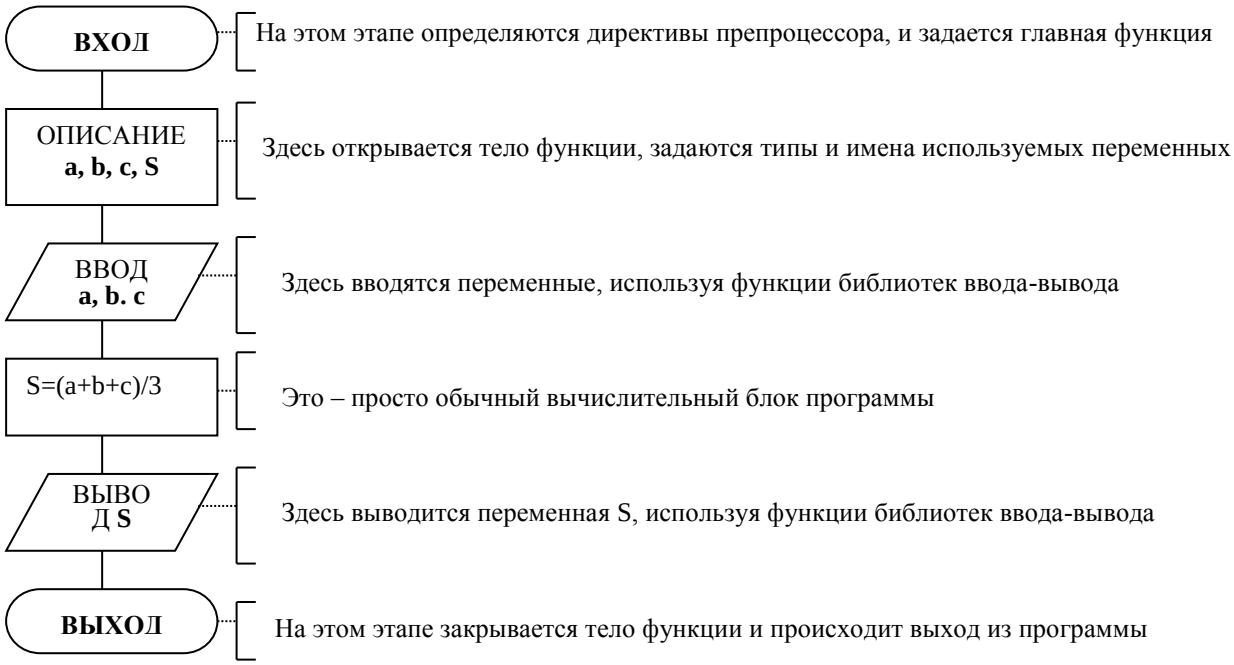
Таким образом, чтобы, например, ввести с клавиатуры три числа разного типа (пусть *10* – целое, *22* – вещественное, *33.3* – вещественное) следует воспользоваться такой нотацией функции ввода:

scanf(“%d%f%f”, &a,&b,&c);

В результате, *по адресам памяти*, отведенным под хранение значений, соответствующих переменных *a, b* и *c*, будут занесены соответственно значения *10, 22.0, 33.3*, вводимые с клавиатуры. Обратите внимание, как от примененного спецификатора изменилось фактическое значение переменной *b*.

В качестве примера рассмотрим возможный алгоритм и текст программы для реализации задачи о нахождении среднего арифметического суммы трех, вводимых с клавиатуры, чисел *a, b, c*. Числа *a* и *b* – целые, число *c* – вещественное число. Результат вычислений требуется сохранить в переменной *S*, и вывести ее содержимое на экран так, чтобы при этом курсор вывода был переведен на новую строку экрана, и произошла выдача звукового сигнала.

Алгоритм программы имеет вид:



Тогда, реализуемая этот алгоритм, программа может быть составлена так.

```
#include<stdio.h>
void main(void)
{
  int a,b; float c,S;
  printf(“Введите три числа \n”);
  scanf(“%d%d%f”,&a,&b,&c);
  S=(a+b+c)/3;
  printf(“Среднее арифметическое, S=%f\n\a”,S);
}
```

На этом обзор функций форматного ввода-вывода можно закончить. Сказанного выше вполне достаточно, чтобы организовать простейший интерфейс взаимодействия пользователя с работающей программой. Считается, что применение этих функций – устаревший стиль программирования, сохранившийся от первоначальной версии языка С. Однако это не умоляет их значения при

ознакомлении с возможностями языка программирования в целом. Более того, эти функции и эта библиотека продолжают входить в набор стандартных библиотек всех современных версий языка С/С++.

Из сказанного следует, что для организации даже элементарных программ на языке Си необходимо иметь четкое представление о типах данных, правилах использования операций и выражений, правилах взаимодействия этих конструкций языка с компилятором на этапе препроцессорной обработки. Также необходимо знать хотя бы элементарные функции ввода/вывода для организации простейших интерфейсов взаимодействия человека и компьютера в процессе работы программы.

2.8.2 Потоковый ввод/вывод

При работе с данными, хранящимися на каких-либо носителях информации (диск, дискета, перфолента и т.п.) изначально не придавалось значения тому, как организована суть обработки этих данных. Достаточно было того, что эти данные можно было сформировать, распознать и обработать, не задумываясь над механизмами этой обработки. И, хотя более естественное понятие «независимой обработки данных» от каких-либо специфических свойств их машинного представления лежало на поверхности, долгое время разработчики языка Си не могли найти приемлемого решения. Так было до тех пор, пока не появился язык С++, а вместе с ним и определение «потоковый ввод-вывод». Если вдуматься в этот термин, то можно понять его естественную суть так: все, что мы передаем в естественной форме (на естественном языке) – это не что иное, как поток (последовательность) закодированной в каком-либо виде информации, а с точки зрения информатики – последовательность байтов. Так почему бы не реализовать эту «естественность» для ЭВМ-реализации? В результате и было найдено решение такой задачи. Не вдаваясь в специфичность этого решения, рассмотрим лишь малую толику тех возможностей, которые предоставляют нам некоторые функции и операторы потокового ввода-вывода. Название потока произошло от того, что информация вводится и выводится в виде последовательности байтов – символ за символом. В языке С++ существуют два класса реализации потоков. Класс *istream* реализует поток ввода, класс *ostream* реализует поток вывода. Эти классы определены в библиотеке, заголовочный файл которой носит имя ***iostream.h*** (*i – input* (вход) , *o – output* (выход), *stream* – поток). Так же, как и в случае с форматным вводом-выводом, рассмотрим действие двух операторов: потокового ввода ***cin*** и потокового вывода ***cout***. Для этих операторов в том же файле ***iostream.h*** определены операции ввода-вывода данных. Для операции вывода определена операция *вставки* или *включения* (записи) данных в поток – <<. Для операции ввода данных используется операция, называемая *извлечения* (чтения) данных из потока – **>>.** Не следует бояться того, что знаки этих операций совпадают со знаками побитового сдвига (см. п.2.3.3). Опять же, благодаря появлению С++, появилась возможность и переопределять (*перегружать*) действие операторов и функций. По-

этому в сочетании с операторами ***cin*** и ***cout*** эти операции будут пониматься как *вставка* и *извлечение*, а не как *побитовый сдвиг*. Для обеих этих конструкций характерно свойство преобразования последовательности символов потока в значение типизированного объекта. Чтобы понять последнее утверждение просто рассмотрим ранее приведенные примеры, с использованием этих структур.

2.8.2.1 Оператор *cout*<<

Для более понятного представления операторов потокового ввода/вывода приведем несколько примеров их сравнений с функциями форматного ввода/вывода. Выражение и результаты действия этих конструкций эквивалентны.

Форматный ввод/вывод	**Потоковый ввод/вывод**
printf("*Ура! Моя первая программа"****);***	***cout<<"****Ура! Моя первая программа"****;***
printf("\" ****Ура! ****a*\"****n****\"****Моя первая программа****a****\" "****);***	***cout <<"\" ****Ура! ****a****\"****n****\"****Моя первая программа****a****\" ";*** или: ***cout <<"\" ****Ура! ****a****\"****n****\";*** ***cout << "\"****Моя первая программа****a****\" ";***

Обратите внимание, что и в этих конструкциях используются те же управляющие символы, что и в форматном вводе/выводе.

Значительно проще с помощью операторов потокового ввода/вывода описание вывода тогда, когда требуется вывести значение переменной:

printf("*Переменная x=****%d****",x);*** ~ ***cout <<"****Переменная x="* << *x;*

Обратите внимание, здесь в правом выражении «исчез» признак спецификации преобразования типа данных *%d*. Теперь, например, выражение:

printf(" x=%d c=%f e=%e",*x,c,e****);***

можно записать, например, так:

cout << " x=" << x <<" c=" << c << " e=" << ***e;***

и результат будет тем же, что и при описании переменных ***x, c, e*** для вывода с использованием функции ***printf( ).***

Так же, как и для функций стандартного ввода/вывода, в операторах потокового ввода/вывода допускается вывод значения выражения после знака <<. Например, если *a=5* и *b=6*, то в результате действия выражения ***cout << a+b*** на экран будет выведено значение *11*.

2.8.2.2 Оператор *cin*>>

Еще «проще» обстоит дело при вводе данных в память компьютера, например, с клавиатуры. Здесь отпадает надобность в спецификаторах. Компилятор «сам распознает» ранее заданный тип данных переменных по их именам. Поэтому эквивалентны выражения:

scanf("*%d%f%f"****, &****a,&b,&c****);*** и ***cin >> a >> b >> c;***

Обратите внимание, что в правом выражении «исчез» не только специфи-

катор типа *%mun* конкретной переменной, но и признак о том, что данные должны быть занесены в память по адресу (&) хранения этой переменной (компилятор все "проделает автоматически"). Не менее «просто» с использованием этих конструкций организован ввод/вывод для переменных сложных (составных) типов данных, например, массивов. Но об этом – позже.

Для сравнения приведем текст программы, которую мы рассмотрели в качестве примера при изучении функций форматного ввода/вывода. Он теперь выглядит так:

```
#include<iostream.h>
void main(void)
{
 int a,b; float c,S;
 cout<<"Введите три числа\n";
 cin>>a>>b>>c;
 S=(a+b+c)/3;
 cout<<"Среднее арифметическое, S=" << S<<"\a";
}
```

Обратите внимание, что в этом примере, в строке *S=(a+b+c)/3*, происходит преобразование целых типов в правой части по правилу большего приоритета типа слева от знака равенства[5].

Некоторые неудобства опраторы потокового ввода-вывода доставляют тогда, когда требуется более жестко отформатировать ввод или вывод данных вещественного типа, указав требуемое количество знаков до и после запятой в них.

Для ограничения количества символов после запятой для плавающих данных, в библиотеке потокового ввода-вывода используются, так называемый, специальный механизм флагов (*манипуляторов*), синтаксис которого общем виде выглядит так:

cout.флаг-функция(значение).

Поэтому, чтобы, например, вывести только 4 знака после запятой, нужно перед основным выводом задать флаг-вывод вида ***cout.precision(4);*** Для указанного выше примера при выводе среднего арифметического чисел следовало бы поступить так:

```
... {...;
    S=(a+b+c)/3;
  cout.precision(4);
    cout<<"Среднее арифметическое, S=" << S;
   }
```

Ниже приведен полный набор манипуляторов потокового ввода-вывода и примеры их применения.

---

[5] На этом примере можно более детально понять смысл преобразования типов данных. Например, объявите переменную *c* также целого типа, введите данные так, чтобы результатом правой части было вещественное число (*a=2, b=3, c=5*), и посмотрите, каков будет результат. Подумайте тнад вариантами получения точного результата.

Манипуляторы:

endl	– при выводе перейти на новую строку;
ends	– вывести нулевой байт (признак конца строки символов;
flush	– немедленно вывести и опустошить все промежуточные буферы;
dec	– выводить числа в десятичной системе (действует по умолчанию);
oct	– выводить числа в восьмиричной системе счисления;
hex	– выводить числа в шестнадцатиричной системе;
*setw(**int** n)*	– установить ширину поля вывода в *n* символов (*n*- целое число);
*setfill(**int** n)*	– установить символ-заполнитель; этим символом выводимое значение будет дополняться до необходимой ширины;
*setprecision(**int** n)*	– установить количество цифр после запятой при выводе вещественных чисел;
*setbase(**int** n)*	– установить систему счисления для вывода чисел; *n* может принимать значение 0, 2, 8, 10, 16, где 0 означает систему счисления по умолчанию, т.е. 10.

Примеры:

1) Вывести одно и то же число 100 в разных системах счисления:

```
int x=100;
cout<<"В десятичной системе:          "<<dec<<x<<endl
    <<"В восьмеричной системе:        "<<oct<<x<<endl
    <<"В шестнадцатиричной системе: "<<hex<<x<<endl;
```

2) Вывести число в поле общей шириной 6 символов (3 цифры до запятой, десятичная точка и 2 цифры после запятой):

```
double x;
cout<<setw(6)<<setprescision(2)<<x<<endl;
```

2.8.2.3 Задачи для самопроверки

2.9 Управление выполнением программы

2.9.1 Организация ветвлений

Ранее, при изучении базовых понятий языка программирования, все иллюстрирующие примеры были просты и сводились, по-существу, к примерам, основанным на элементарных алгоритмических конструкциях – задачах, имеющих *линейный алгоритм* решения. На практике, как правило, таких простых задач не существует. Любая реальная задача, в той или иной степени, требует воплощения в себе методов, при которых предполагается альтернативное ре-

шение какой-либо ее части. Для этого в языке Си предусмотрены специальные конструкции, называемые «*операторами управления*». Эти операторы предназначены для решения задач, в основе которых лежат алгоритмические конструкции управления ветвления (выбора) и цикла. Рассмотрим, как такие конструкции реализуются на языке программирования.

2.9.1.1 Простые и составные операторы

Ранее мы говорили о различных типах операторов, участвующих в организации вычислительного процесса. Операторы, которые участвуют в процессах, связанных с ветвлениями относятся к группе *операторов управления*. В этой группе большую роль играют операторы, состоящие не из единичного действия, а *из совокупной последовательности дискретных действий*. Можно мысленно представить, что такая последовательность единичных операторов (группа), в конечном итоге, выполняет какое-то одно единственное и конкретное решение. Следовательно, по большому счету, такое объединение простых операторов можно отождествить с одним оператором, но состоящим из нескольких операторов элементарных. Мы говорили, что в этом случае такой оператор в конструкции языка Си называется *составным оператором*.

Мы также говорили, что для составного оператора характерно то, что не во всех случаях после закрывающейся фигурной скобки ставится точка с запятой.

Рассмотрим пару примеров.

Пример 1. Вычислить значение функции *Sin(x)* при $x=\pi/2$, а **затем** – значение *Cos(x)* при $x=\pi/4$.

Тогда в теле главной функции *main( )* на языке Си можно было бы написать так:

```
....
main( )
{....
 x=PI/2;
 y=sin(x);
 x= PI /4;
 y=cos(x);
……
}
```

Из определения оператора нам известно, что любая конструкция языка Си, заканчивающаяся точкой с запятой (**;**), называется *оператором*. Поэтому каждая строчка в теле приведенной выше функции – есть оператор, причем, простой, то есть, состоящий из конкретного дискретного действия.

Пример 2. Вычислить значение функции *Sin(x)*, если $x=\pi/2$, и вычислить значение *Cos(x)*, если $x=\pi/4$.

Легко заметить, что в такой постановке задачи явно просматриваются два отдельных блока. Каждый из этих блоков состоит из двух операторов: оператора инициализации переменной и оператора – выражения (выражается значение

одной переменной через значение ранее инициализированной переменной). Такие блоки в программе на языке Си, согласно определению составного оператора можно представить так:

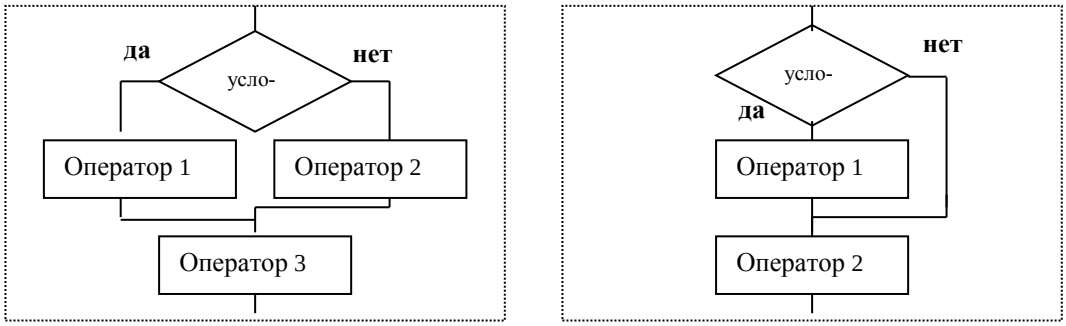
```
 main( )
{….
  {x= PI /2;  y=sin(x);};  //первый блок
  {x= PI /4;  y=cos(x);};  //второй блок
……
}
```

Иначе говоря, дискретные действия программы на каждом ее шаге представлены не отдельно взятыми дискретными (простыми) операторами, а дискретными блоками (составными операторами), представляющими собой совокупность простых операторов, заключенных в фигурные скобки.

В большинстве программ, использующих управляющие структуры, составные операторы играют главенствующую роль, так как выражают логику и смысл программы в целом.

2.9.1.2 Условный оператор

Оператор условия (ветвления) реализует одну из схем:



На схемах «Оператор 1,2, или 3» может быть как простым, так и составным. В языке Си такие схемы реализуются с помощью оператора ***if…else***.

Общая форма записи (синтаксис) для реализации первой (слева) схемы выглядит так:

if (логическое условие) {оператор 1;} else {оператор 2;}

Трактуется это так: сначала вычисляется «*логическое выражение*» и, если оно верно, то выполняется «*оператор 1*». Если «*логическое выражение*» не верно, то выполняется «*оператор 2*» (*else* – «иначе»). И тот и другой оператор может быть как «простым», так и «составным».

Частным случаем синтаксиса данного представления может быть запись, реализующая вторую схему (справа):

if (логическое условие) {оператор 1;};{оператор 3;}

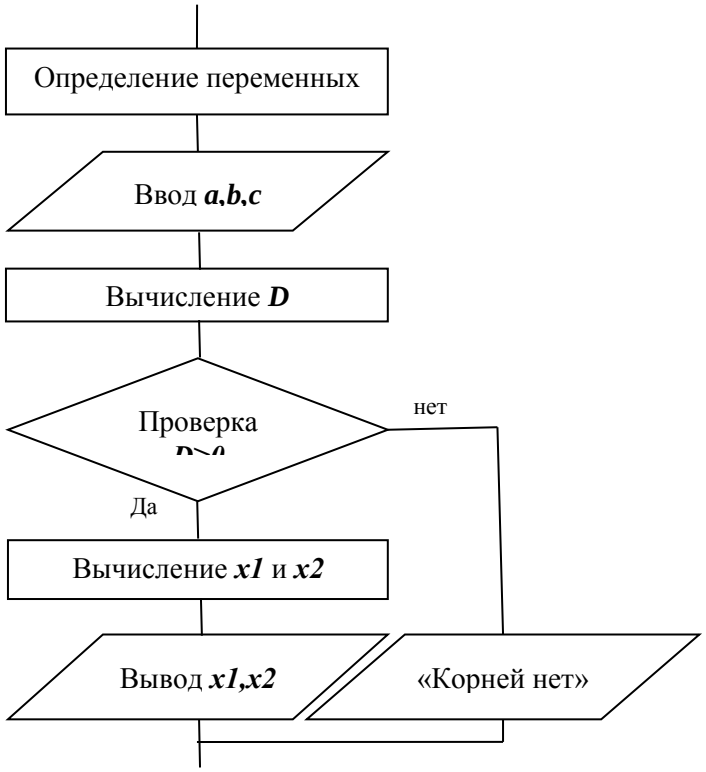
В этой записи вместо оператора «*else*» применяется пустой оператор «**;**». Причем важно обратить внимание на то, что «*оператор 2*» будет выполняться не только тогда, когда не выполняется логическое условие («нет»), но и в том случае, когда оно выполняется («да»), сразу после исполнения «*оператора 1*». То есть «*оператор 2*» выполняется всегда!

В приведенных случаях применения условного оператора отметим еще одну существенную деталь: перед конструкцией «*else*» после *составного* оператора точка с запятой (*;*) не ставиться.

Пример 3. Для трех вводимых вещественных чисел определить действительные корни решения квадратного уравнения.

Здесь основным условием проверки истинности решения является проверка значения дискриминанта на его положительность. Если он не отрицателен, то корни есть и их можно вычислить по известным формулам. В противном случае – действительных корней нет.

Блок-схема алгоритма программы выглядит так:



Примером реализации такого алгоритма может быть вариант программмы вида:

```
#include <iostream.h>
#include <math.h>
void main(void)
{
  float a,b,c;
 double x1,x2,D;
  cout<<"Введите a , b ,c\n";
  cin>>a>>b>>c;
  D=b*b-4*a*c;//
  if(D>=0)
   {
    x1=(-b+sqrt(D))/(2*a);
```

В этом примере роль условия выполняет проверка на не отрицательность, ранее вычисленного значения переменной *D*. Роль «*оператора 1*» выполняет составной оператор, состоящий из операторов вычисления величин *x1* и *x2* и вывода на экран их значений посредством оператора *cout*<<. Роль «*оператора 2*» выполняет простой оператор *cout*<< сообщающий об отсутствии корней.

Особенности текста программы.

```
   x2=(-b-sqrt(D))/(2*a);
   cout.precision(4);
   cout<<" x1="<<x1<<" x2="<< x2;
  }
 else
   cout<<"корней нет"<<"\n";

}
```

1. В программе используются операторы потокового ввода-вывода. Поэтому необходимо подключить файл ***iostream.h***;

2. При использовании математических функций в программе, например, извлечения корня ***sqrt( )*** или возведения числа в степень $x^N \equiv pow(x,N)$, необходимо подключить математическую библиотеку из заголовочного файла ***math.h***;

3. Переменные, работающие с такими функциями, объявляются в программах как переменные, имеющие, как минимум, тип ***double*** (двойной точности).

Подведем некоторые итоги.

- Конструкция ***if…else*** – *условный оператор* – используется для выбора одного из двух направлений дальнейшего хода программы;
- Выбор последовательности действий программы осуществляется в зависимости от значения логического условия, заключенного в скобках выражения, записанного после *if*.
- Действия программы после оператора *else*, выполняются в том случае, если значение выражения логического условия равно нулю (т.е. условие ложно, не соответствует заданным требованиям).
- Во всех остальных случаях выполняются действия, следующие сразу за условием;
- Если при соблюдении или несоблюдении условия надо выполнить сразу несколько действий программы, то эти действия следует объединить в группу – заключить в фигурные скобки, т.е. образовать составной оператор;
- При помощи вложенных одна в другую нескольких конструкций *if* можно реализовать множественный выбор.

2.9.1.3 Оператор-переключатель

Часто возникающая в программировании задача – выбор одного варианта из многих. Как было сказано выше, это можно сделать с помощью нескольких вложенных друг в друга операторов *if….else*, например, так:

```
if(условие 1) оператор 1;
else
 {if (условие 2) оператор 2;
  else
   {if(условие 3) оператор 3;
    else
     {……..}
```

Такое представление не совсем удобно в реализации, так как требует внимательного применения синтаксических структур оператора условия, отслеживая соответствие между скобками *{ }*, логику перехода между операторами по значениям условия: «*ис-*

```
    }
 }
оператор N;
```

тина» или «ложь».

Более удобный способ – использование оператора-переключателя (*выбора*) ***switch***, синтаксис которого выглядит так:

```
switch(«выражение»)
   {
     case «константа 1»: «оператор 1»;[break;]
     case «константа 2»: «оператор 2»;[break;]
       ……
     case «константа n»: «оператор n»;[break;]
     [default : «оператор n+1»;]
   }
«оператор k»;
```

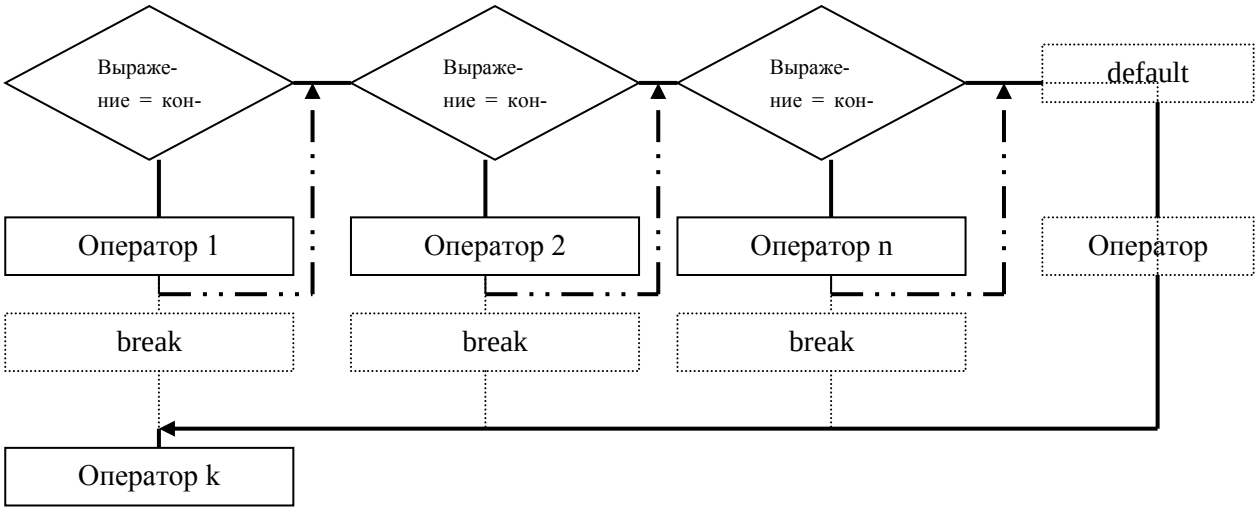
Этот оператор выполняется так. Сначала вычисляется значение «*выражения*», тип которого должен быть одним из простых (*int, char*).

Вычисленное значение (полученное число или символ) сравнивается со значением ***N*** (*N=1,2,…,n*) константных выражений «*константа N*». При совпадении этих двух значений выполняется соответствующий «*оператор N*» (который может быть составным). Затем управление сразу передается оператору «*оператор k*», следующий после *switch*, в том случае, если в соответствующей ветви имеется необязательный оператор *break*.

Если такого оператора в ветви нет, то управление передается последовательно на следующую ветвь до тех пор, пока не будет выполнен «*оператор n*».

Если значение «*выражения*» не совпало ни с одним из значений «констант», то управление передается на ветвь, помеченную *default*, и выполняется «*оператор n+1*». При ее отсутствии осуществляется выход из оператора *switch*, т.е. управление передается оператору «*оператор k*».

Схематично работу оператора *switch* можно проиллюстрировать так:



Штрихованными линиями отмечены связи и блоки в случае, если имеются необязательные синтаксические конструкции оператора-переключателя.

Причем, если таковые имеются, то исключаются переходы, помеченные на схеме линиями, состоящими из чередования точек и тире.

Пример 4. Проиллюстрировать в программе работу оператора *switch* с использованием оператора *break* и без него. В качестве проверяемых констант использовать символы *'S'*, *'C'* и *'T'*. В качестве операторов для этих констант использовать операторы форматного вывода соответствующих фраз: *"вычисление синуса", "вычисление косинуса", "вычисление тангенса"*, а в противном случае, оператор вывода фразы *"вычисление котангенса"*.

Вариант такой программы может быть таким.

```
#include <stdio.h>
void main(void)
{
  char s;
  printf("Введите первую букву\n->");
  scanf("%c",&s);
  switch(s)
  {
      case 'S': printf("\nвычисление синуса\n"); break;
      case 'C': printf("вычисление косинуса\n"); break;
      case 'T': printf("вычисление тангенса\n"); break;
      default: printf("вычисление котангенса\n");
  }
 printf("конец программы");
}
```

Если всюду в ветвях использовать оператор ***break***, то при вводе в качестве контанты, например, символа *'S'*, результатом работы программы будет:

```
Введите первую букву
->S
вычисление синуса
конец программы
```

Если оператор ***break*** всюду убрать, то при вводе того же символа *'S'* результат будет таков:

```
Введите первую букву
->S
вычисление синуса
вычисление косинуса
вычисление тангенса
вычисление котангенса
конец программы
```

Пример 5. Простейший калькулятор.

Известно, что в простейшем калькуляторе выполняются четыре арифметических действия: «+», «-», «*», «/». В зависимости от этих действий ***изменяется уже существующее*** значение. В калькуляторе также проверяется ***условие деления на нуль*** и ***корректность*** ввода ***знака операции***.

Тогда, основной блок программы калькулятора, отвечающий за одно из указанных действий, можно представить так.

```
#include <stdio.h>
#include <iosream.h>
void main(void)
{
 char s; float a=0, b;
 printf("Введите операнд-> : ");cin>>b;
printf("\nи операцию -> : ");cin>>s;
switch(s)
 {
  case '+': a=a+b; printf("Результат = %f\n",a);break;
  case '-': a=a-b; printf("Результат = %f\n",a);break;
  case '*': a=a*b; printf("Результат = %f\n",a);break;
  case '/': if(b==0) {cout<<”деление на нуль”;break;}
            else {a=a/b; printf("Результат = %f\n",a);break;}
  default: printf("такой операции нет\n";break;
 }
printf("конец программы");
}
```

Таким образом:

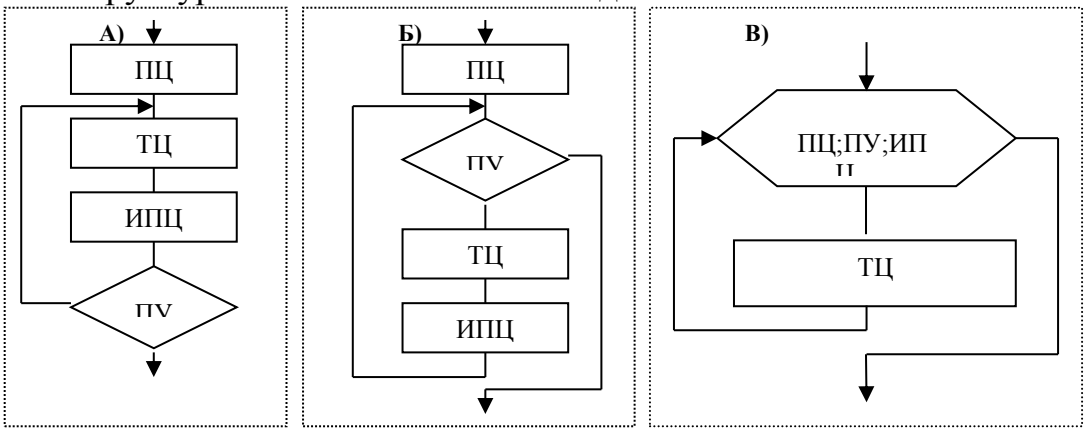
- Конструкция «*оператор-переключатель*» предназначена для выбора одного из нескольких возможных направлений дальнейшего хода программы;
- Выбор последовательности действий осуществляется в зависимости от равенства значения *переменной-селектора* (выражение в скобках сразу после ключевого слова *switch*) *константе*, стоящей после ключевого слова *case*;
- Если после выбранной последовательности действий встретиться ключевое слово *break* (оператор *безусловного завершения действий*), то управление в программе передается оператору, стоящему сразу после всей конструкции *switch*;
- Если после последовательности действий в цепочке *case* оператор безусловного завершения действия *break* отсутствует, то управление передается на следующий шаг сравнения переменной-селектора и очередной константы;

- Если значение переменной-селектора не равно ни одной из констант, записанных после *case*, то выполняются действия, указанные после ключевого слова *default*;
- В качестве переменной-селектора можно использовать переменную целочисленного типа: целого (*int*) типа или символьного (*char*) типа.

2.9.1.4 Задачи для самопроверки по организации ветвлений

2.9.2 Организация циклических вычислений

Изучая основы теории алгоритмов (см. п.1.5), мы говорили, что сколько-нибудь более или менее сложная программа не обходится без использования алгоритмов циклических структур. Язык Си предполагает три типа циклических операторов для реализации циклов с *постусловием* (А), *предусловием* (Б) и частного случая цикла с предусловием - *параметрического* цикла (В). Схематично эти структуры соответственно выглядят так:



Здесь, напомним, ПЦ – параметр цикла; ТЦ – тело цикла; ИПЦ – изменение параметра цикла; ПУ – проверка условия. Для реализации таких алгоритмических структур в языке Си применяются операторы ***do…while***, ***while*** и ***for***.

Рассмотрим эти операторы более детально.

2.9.2.1 Операторы цикла *wile* и *do…while*

Синтаксис оператора реализации цикла с постусловием:

do {операторы;} while (условие);

Смысл трактовки этого оператора в следующем: «делаем (*do*) «оператор» (составной оператор) до тех пор, пока (*while*) не выполнится логическое «условие» (условие-истина).

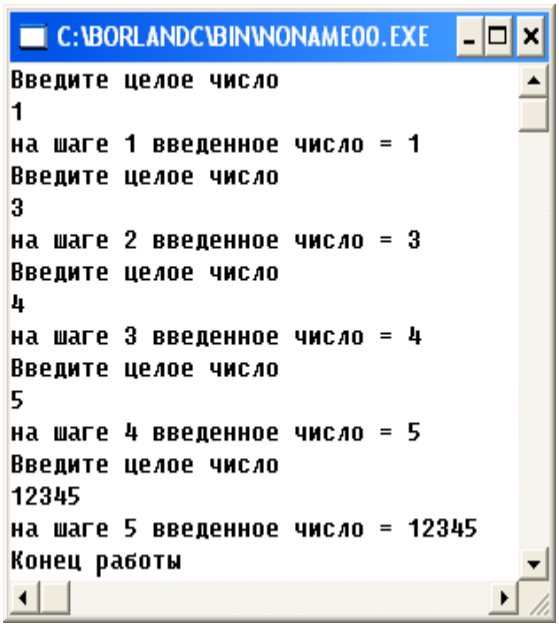
Мы уже знаем, что отличительной особенностью циклического алгоритма с пост-условием является то, что в этом цикле выполнение операторов, входящих в его тело, произойдет хотя бы один раз. Причем такой цикл может быть с заданным числом повторений и с неопределенным числом повторений. Согласно блок-схеме, словесный алгоритм цикла с пост-условием и с заданным числом повторений трактуется и записывается в такой последовательности:

- задать начальное значение параметру цикла (его можно назвать переменной-счетчиком итераций). Этот параметр является переменной либо целого типа, либо переменной символьного типа;
- записать ключевое слово *do*, после которого открыть с помощью фигурной скобки { тело цикла;
- последовательно записать операторы, составляющие тело цикла; последним оператором тела цикла должен быть оператор, изменяющий счетчик. После него закрывается фигурная скобка }, определяющая конец тела цикла;
- осуществить проверку логического условия – сравнить достиг ли параметр цикла требуемого значения. Если результат проверки истинный (т.е. параметр цикла не достиг искомого значения), то повторить выполнение тела цикла. Если результат проверки ложный, т.е. параметр цикла достиг искомого значения, то выйти из цикла.

Рассмотрим два примера, поясняющих сказанное:

***Пример* 1. Цикл с заданным числом повторений.** Ввести с клавиатуры и вывести на экран последовательность целых чисел.

```
#include <stdio.h>
void main(void)
{
 int i, k=1;
 do
   {printf("Введите целое число\n");
    scanf("%d",&i);
    printf("на шаге %d введенное
число = %d\n",k,d);
    k++; // k=k+1
   }
  while(k<=5);
 printf("Конец работы");
}
```

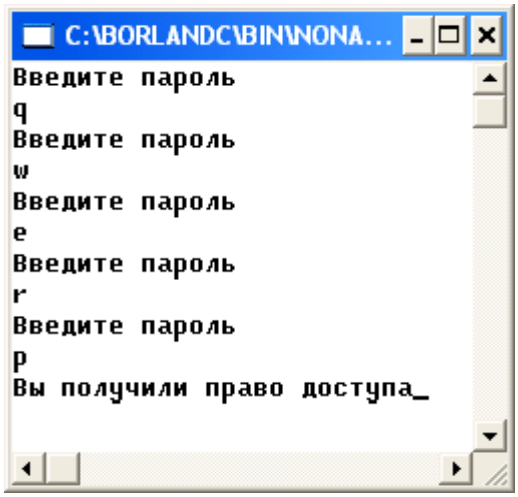


Результатом работы такой программы будет вид экрана, приведенный справа.

Здесь очевидно, что логическим условием проверки является сам параметр цикла – переменная-счетчик итераций.

***Пример 2.* Цикл с неизвестным числом повторений.** Водить символ с клавиатуры до тех пор, пока он не будет угадан.

```
#include <stdio.h>
void main(void)
{
 char pas;
 do
  { printf("Введите пароль\n");
```



```
    scanf("%s",&pas);
  }
 while(pass!='p');
printf("Вы получили право доступа");
}
```

Интерфейс такой программы показан на экране справа.

Очевидно, что в данной программе цикл может повторяться сколь угодно раз. Параметром цикла является не счетчик итераций, как в предыдущей программе, а значение вводимой переменной.

Пример 3. **Также – цикл с неизвестным числом повторений.**

Написать программу, которая определяет минимальное число во введенной с клавиатуры последовательности *положительных* чисел (длина последовательности неограничена). Завершением ввода служит ввод любого *отрицательного* числа.

```
#include<stdio.h>
void main(void)
{ int a, min=10000;
   printf('\nВводите числа. Завершение - отриц. число\n');
do
 { scanf(%i,"&a);
   if ((a<min)&&(a>0))
   min=a;
 }
while (a>0);
printf('Минимальное число = %i\n,"min);
}
```

Особенность этого примера в то, что проверяемое условие внутри тела составного оператора цикла – также составное условие*:* ***if ((a<min)&&(a>0))***

Синтаксис оператора реализации цикла с пред-условием:

while (условие) {оператор;}

Трактуется он так: «до тех пор, пока «условие» имеет истинное значение, выполнять «оператор» – тело цикла, иначе (условие приняло ложное значение) – выйти из цикла.

Отличительной особенностью циклического алгоритма с предусловием является то, что в этом цикле выполнение операторов, входящих в его тело, может не произойти вообще, если при первой же проверке условие выполнения тела цикла окажется недопустимым. Как и для цикла с постусловием такой цикл может быть с заданным числом повторений и с неопределенным числом повторений. Согласно блок-схеме словесный алгоритм цикла с предусловием и с заданным числом повторений трактуется и записывается в такой последовательности:

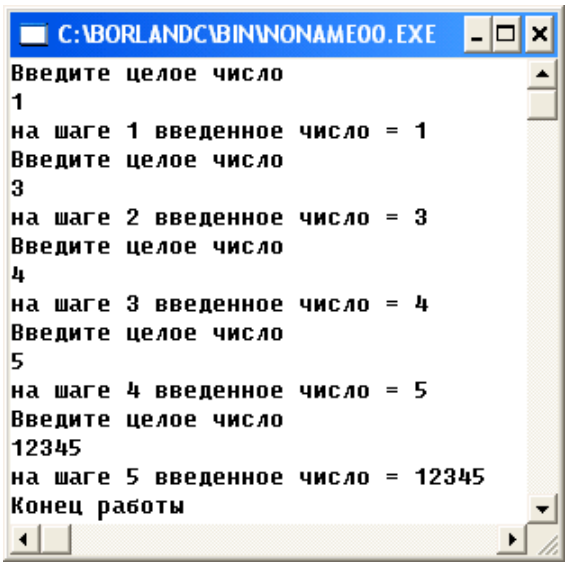
- задать начальное значение параметру цикла;
- записать логическое условие проверки выполнения тела цикла;

- открыть фигурную скобку и последовательно записать операторы, составляющие тело цикла; Последним оператором тела цикла должен быть оператор, изменяющий параметр цикла, после которого закрывается фигурная скобка;

Сказанное рассмотрим на примерах (результаты работы программ аналогичны результатам примера 1 и примера 2).

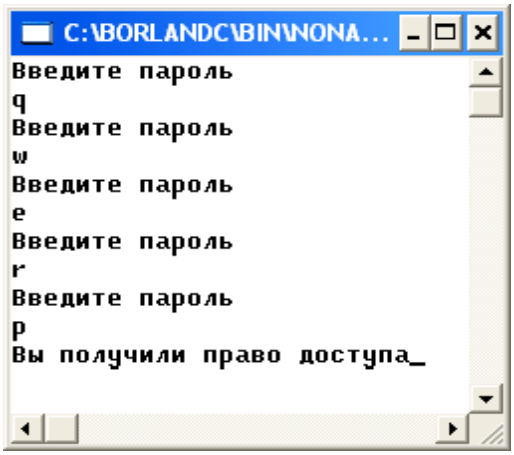
Пример 4. **Цикл с заданным числом повторений.**

```
#include <stdio.h>
void main(void)
{
  int i, k=1;
  while(k<=5)
    {printf("Введите целое число\n");
     scanf("%d",&i);
     printf("на шаге %d введенное
 число = %d\n",k,d);
     k++; // k=k+1
    }
  printf("Конец работы");
}
```



Пример 5. **Цикл с неизвестным числом повторений**

```
#include <stdio.h>
void main(void)
{
  char pas;
  while(pass!='p')
    { printf("Введите пароль\n");
      scanf("%s",&pas);
    }
 printf("Вы получили право доступа");
}
```



2.9.2.2 Оператор *for*

Наиболее сложна в смысле синтаксиса, но и наиболее популярна в С++, форма реализации параметрического цикла. Для этих целей служит оператор *for*. Его синтаксис эквивалентен следующему фрагменту:

```
«установка начального значения ПЦ»;
while(«условие ПЦ»)
 {
   {«оператор 1»;}
    «ИПЦ»;
 }
«оператор 2»;
```

То есть, перед вхождением в цикл осуществляется установка параметра цикла (счетчика) – ПЦ, фиксирующего признаки начала и окончания цикла. Затем проверяется значение «условие ПЦ». Повторение тела цикла (ТЦ) осуществляется до тех пор, пока «условие ПЦ» не станет ложным. При этом, после каждого выполнение ТЦ счетчик (ПЦ) либо увеличивается, либо уменьшается на заданную величину до заданного предела, указанного в «условии ПЦ».

Синтаксис оператора *for* имеет вид:

for(«выражение 1»; «условие»; «выражение 2»)
{«оператор 1»;}
«оператор 2;»

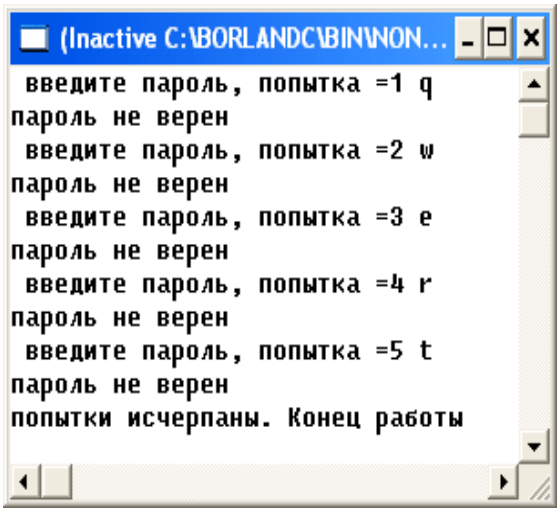
«Выражение 1» и «выражение 2» могут состоять из нескольких выражений, разделенных запятыми. «Выражение 1» определяет действия, выполняемые до начала цикла, т.е. определяет начальные условия для цикла. Логическое выражение «условие» определяет условия окончания или продолжения цикла: цикл продолжается, если значение этого логического выражения истинно. «Выражение 2» обычно задает необходимые для следующей итерации изменения параметров или любых переменных цикла. После его выполнения управление вновь передается на проверку «условия». Так происходит до тех пор, пока значение «условия» не станет ложным. Таким образом, «выражение 1» вычисляется только один раз, а «условие» и «выражение 2» вычисляются после каждого выполнения тела цикла.

Особенностью данного оператора является и то, что все три его составляющие, разделенные точкой с запятой (или какие-либо из них) могут отсутствовать. Однако, точка с запятой, как разделитель «смыслового» предназначения составляющих должна быть всегда!. Например, этот оператор может выглядеть так *for(; ;)*. Если в операторе «условие» отсутствует, то оно считается всегда истинным. В этом случае нужны специальные средства выхода из цикла.

Рассмотрим пример, иллюстрирующий работу данного оператора.

Пример 1. Пусть, например, требуется так составить программу, что за определенное количество раз, например, не более 5, нужно «угадать» ключ пароля – символ *p* (результат приведен справа).

```
#include <stdio.h>
void main(void)
{ char pas; int k;
  for(k=1; k<=5; k++)
  { printf(" введите пароль, попытка
                =%k ",k);
    scanf("%s",pas);
    if (s!='p') printf("пароль не верен
                          \n");
    }
    if (k>5) printf("попытки исчерпа-
          ны. Конец работы\n");
}
```



Пример 2. Усложним задачу.

Пусть, например, требуется (по аналогии с предыдущим примером) ввести символьный пароль для продолжения работы программы. Ключом такого пароля пусть, по-прежнему, остается символ *p*. Известно, что каждый символ в ASCII – кодировке имеет целочисленное представление. Требуется так составить программу, что правильность ввода «пароля» проверялась бы не только на вводимый символ, но и было бы ограничено количеством повторных вводов. То есть, за определенное количество раз, например не более 5, нужно «угадать» ключ пароля.

Конечно, данный пример – не совсем, может быть, удачен, но листинг его программы позволит нам, помимо оператора *for* изучить и еще некоторые «конструктивные» возможности языка С++.

```
#include <stdio.h>
void main(void)
{
  char pas=0; int k;
  for(pas=pas,k=0;(k<=4)&&(pas!='p'); k++)
  {
    printf("введите пароль, попытка = %d",k+1);
    scanf("%s",pas);
  }
  if ((k>=5)&&(pas!='p'))
      printf("попытки исчерпаны. Конец работы\n");
  else
     printf("Вы получили доступ.Конец работы\n");
}
```

В этой программе применяются две переменные: переменная *pas* символьного типа, предназначенная для определения "пароля", и целочисленная переменная *k*, предназначенная для ограничения количества попыток задания "пароля". Так как pas имеет тип char, то для нее допустимы выражения вида: *pas =0* (присваивание числа: теперь в *pas* содержится число *0*), *pas ='d'* (присваивание символа: теперь в *pas* содержится символ *d*). В то же время, недопустимо выражение вида *pas=d* (присваивание значения, содержащегося в переменной *d*, если *d* ранее не была объявлена, как переменная типа *char* и ей не было присвоено какого-либо значения – символа).

При организации цикла ввода "пароля", вначале устанавливаются начальные значения сразу для двух переменных: *pas=0*, *k=0*. Затем проверяется совместное условие на количество разрешений попыток ввода пароля *k<=4* (по условию не более 5) и проверка (&&) на соответствие значения вводимого символа символу *'p'*: *pas!='p'*. Это условие трактовать можно так: "*пока значение величины k меньше 5, и при этом, пока значением величины pas не является символ p*". Если оба условия имеют место, то выполняется тело цикла – вводится новое значение переменной *pas*, увеличивается значение переменной *k* (*k++*), и управление вновь передается на проверку условия. Если хотя бы одно

из условий не выполняется (т.е. будет либо *k=5*, либо *pas='p'*), то программа выйдет из цикла и выполнится одна из двух ветвей, зависящих от значений указанных величин.

Итак, особенность параметрического цикла в том, что в качестве его параметров можно использовать не один, а несколько, одновременно проверяя условия их выполнения и одновременно изменяя их значения. Эти параметры перечисляются (указываются) внутри оператора *for( )* через запятые, и их функциональное назначение определяется разделителем «;». Так же, как и в цикле с предусловием, тело цикла, при использования оператора *for( )*, может не выполниться ни разу, если заданные начальные условия совпадут с условиями проверки.

2.9.2.3 Вложенные циклы

Ранее (см. п. 1.6) мы рассматривали алгоритмы сложных вычислительных процессов. На практике часто встречаются задачи, в которых циклические алгоритмы вложены друг в друга. Мы говорили, что в таком алгоритме различают термины «внешний» и «внутренний» циклы.

В цикл, называемым внешним, могут входить один или несколько циклов, называемых внутренними. Организация как внешнего, так и внутреннего циклов осуществляется по тем же правилам, что и организация простого цикла. Параметры внешнего и внутреннего циклов разные и изменяются не одновременно. То есть, при одном значении параметра внешнего цикла, параметр внутреннего цикла поочередно принимает все значения.

Примером задач вложенности циклов может служить **задача**, например, **нахождения** какого-либо **элемента прямоугольной матрицы.**

Матрица имеет два определяющих ее элемента-параметра: номер строки и **номер столбца.**

Если обозначить через ***i*** - номер строки, через ***j***-номер столбца, через a_{ij} - элемент, находящийся на пересечении строки и столбца, то матрицу ***A***, размерности 3×3, можно представить так:

$$A_{i,j} = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix}$$

В качестве примера программной реализации рассмотрим следующий пример.

Пример 3. Разработать программу вычисления таблицы Пифагора (***таблицы умножения***) размерностью (*5×5*):

Здесь ***i=5*** - строки и ***j=5*** - столбцы.

Внешний цикл по *i* – номеру строки.

Внутренний цикл по *j* – номеру столбца.

Используя цикл ***for*** можно получить такой вариант реализации:

…

```
{ int i, j;

for(i=1;i<=5;i++)
  for(j=1;j<=5;j++)
    i*j;
...
}
```

Также вспомним пример 1 из п.1.6.

Пример 4. Разработать алгоритм программы определения потенциальной возможности радиоэлектронной станции (РЛС) по дальности обнаружения целей на малых высотах при разной высоте антенны по формуле $D_ц = 4.12(\sqrt{H_ц} + \sqrt{h_a})$, где высота антенны может быть $h_a \in \{6,9,12,15\}$*м.*, а высоты целей лежат в диапазоне $H_ц \in \{100,200,300,400,500\}$*м.*

Решение. Проведем предварительные исследования и формализуем данную задачу.

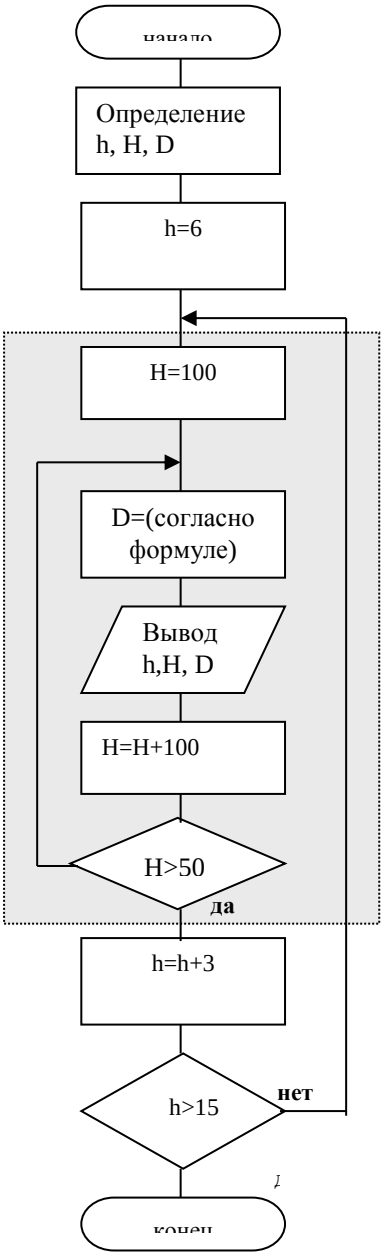
Заметим, что величины, определяющие как высоты антенны, так и высоты цели, могут принимать различные значения. Поэтому целесообразно ввести индексацию для этих величин (т.е. каждой из них присвоить какой-либо номер). Примем в качестве индекса для высот антенны параметр *i={1, 2, 3, 4}*, а для высот цели параметр *j={1, 2, 3, 4, 5}*. Тогда *h(1)=6, h(2)=9, h(3)=12, h(4)=15,* а *H(1)=100, H(2)=200, H(3)=300, H(4)=400, H(5)=500.*

Таким образом, очевидно, что для вычислений значений *D* следует вначале, установив одну из высот антенны, например *h(1)*, получить все значения дальности обнаружения цели *D(1,j)* для нее, изменяя последовательно значения высот цели *H(j)*. Как только это будет получено, следует изменить индекс *i* и повторить процедуру вычисления значений *D*, но теперь уже для новой антенны. И так продолжать до тех пор, пока все номера индекса *i* не будут исчерпаны.

Из приведенных рассуждений следует, что алгоритм должен иметь два цикла: внешний по параметру *i* и внутренний по параметру *j* для каждого *i*.

Мы помним некоторые математические закономерности для высот антенны и цели. Величина каждой следующей высоты антенны отличается от предыдущей на *3* единицы, а высота полета цели на *100* единиц. И так как нам не требуется непосредственно хранить значения величин *h(i)* и *H(j)* в памяти ЭВМ, то их имена (*h* и *H*) можно использовать в качестве изменяемых параметров, имеющих начальное и конечное значение и явно выраженный шаг изменения. При этом *h* – параметр внешнего цикла, а *H* – параметр внутреннего цикла. Таким образом,

$h_{нач} = 6, шаг\ \Delta h = 3, h_{кон} = 15;\quad H_{нач} = 100, шаг\ \Delta H = 100, h_{кон} = 500.$



Теперь все готово для определения всех компонент алгоритма:

А) используемыми в нем переменными будут переменные *h, H* и *D*. Причем первые две – это целые числа, а последняя – вещественное число;

Б) входными данными являются начальные значения для переменных *h* и *H*, причем, эти значения не требуют ввода в память ЭВМ, например, с клавиатуры, а могут быть заданны непосредственным описанием в программе;

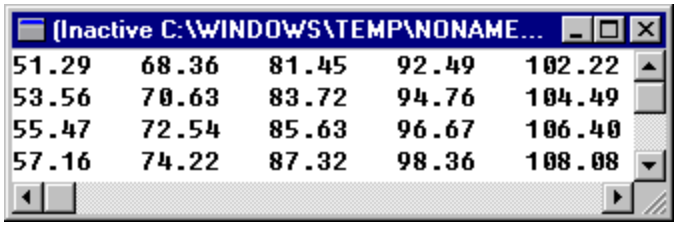
В) вычислительный блок (тело цикла) предполагает вычисление величины *D* и изменение параметров циклов;

С) выходными данными являются значения величин *h, H* и *D* на каждом шаге внутреннего цикла для шага цикла внешнего. Процесс их вывода также может быть организован в теле цикла.

На основании детального анализа задачи можно приступать к разработке блок-схемы ЭВМ-алгоритма. Вариант алгоритма с постусловием представлен слева.

Разработаем программу реализации данной задачи, но при этом используем не иллюстративный алгоритм с постусловием, а алгоритм, использующий конструкцию ***for.***

Результат работы программы и ее вариант может быть таким:




```
#include<stdio.h>
#include<math.h>
#include<conio.h>
void main(void)
{ float D;   int h, H, i=0, j=0;
  for(h=6;h<=15;h=h+3)        //внешний цикл
   {  for(H=100;H<=500;H=H+100)     //внутренний цикл
      {  D=4.12*(sqrt(H)+sqrt(h));
         printf("%.2f\t",D);
         if((j+1)%5==0) //в строке 5 элементов
           printf("\n");
        j++; /* меняем индекс j в теле внутреннего  цикла*/
      } //конец внутреннего цикла
```

```
        i++; j=0;  // меняем индекс i в теле внешнего  цикла, устанавливая j на
                   первый столбец
    }    // конец внешнего цикла
}
```

2.9.2.4 Операторы *break* и *continue*

В некоторых случаях при выполнении циклов требуется либо завершить их в целом, либо пропустить часть тела цикла и выполнить следующую его итерацию.

Оператор *break* завершает выполнение цикла. Например:

```
int sum=0, i=1;
for(;;)
{
 if(i>100)
  break;
  sum=sum+1;
  i=i+1;
}
```

В этом примере цикл выполнится 100 раз, последовательно присваивая переменной *sum* значения 1+2+3+…+100. Как только это произойдет, цикл будет завершен. Заметим, что изначально цикл был определен на неизвестное (бесконечное) число повторений: *for(;;).*

С оператором *break* мы уже сталкивались ранее, когда рассматривали оператор-переключатель (см. п.2.9.1.3). Смысл оператора *break* заключается в безусловном выходе из любого участка (блока) программы, в котором он находится.

Оператор *continue* заставляет пропустить остаток тела цикла (все операторы, следующие за ним) и выполнить следующую его итерацию. Например:

```
int sum=0;
for(int i=1;i<=100;i++)
 { if (i%7==0)
    continue;
  sum=sum++;
 }
```

В этом примере вычисляется сумма всех целых чисел от 0 до 100, которые не делятся на 7. Если в процессе вычислений встретится число, которое делится на 7 без остатка, то оператор тела цикла *sum=sum++;* пропускается и управление передается на следующую итерацию.

2.9.2.5 Операторы перехода и возврата

Безусловное изменение последовательности выполнения операторов в программе можно выполнить с помощью оператора перехода ***goto*** *метка;* Например, при вводе числа с клавиатуры требуется вычислить абсолютную его

величину. Тогда один из вариантов выполнения такого условия может быть представлен так:

```
cin>>x;
 if(x>=0)
  goto p;
 x=-x;
p: abs=x;
```

В настоящее время считается, что применение оператора *goto* – это «плохой» стиль программирования, так как он очень легко запутывает логику выполнения программы. Язык С++ позволяет обойтись без него с помощью гибкости других операторов управления. Поэтому лучше не использовать этот оператор, или использовать только в самом крайнем случае.

В больших, многофайловых и многофункциональных программах оперируют данные, которые являются результатом работы отдельных функций. Поэтому, чтобы получить для последующей обработки такой результат, в С++ предусмотрен оператор возврата «***return*** *выражение*».

Например, в отдельной функции вычисляется факториал числа, а для последующей обработки возвращается его значение, уменьшенное на единицу. Тогда такая функция может быть представлена так:

```
…
n=5;
int fact(int n)
{int k;
 if (n==1)
    k=1;
 else
   k=n*fact(n-1);
return k-1;
}
```

Об особенностях описания и вызова функций в программах, в частности, рекурсивных (т.е., вызывающих самих себя), мы поговорим позже. Здесь важно отметить, что если число, которое «получит» функция *fact( )* будет больше единицы (например, 5), то переменная *k* последовательно примет значение 5*4*3*2*1. А это и есть факториал числа 5. За счет оператора *return* функция *fact( )* «вернет» факториал числа, уменьшенный на *(k-1)* единицу.

2.9.2.6 Задачи для самопроверки по организации циклов

2.10 Массивы, **указатели и операции с адресами**

2.10.1 Понятие указателя

Указатель **–** это переменная, содержащая адрес другой переменной. То есть, ***значением*** переменной типа указатель ***является целое число, равное адресу*** того объекта, на который ссылается указатель.

Указатель существенно связан с типом объекта, на который он ссылается. Если в описании перед обозначением объекта поставить символ **"*"**, то оно будет описывать указатель на объект того же типа и класса памяти, которые соответствуют данному обозначению без звездочки.

Унарная операция **"*"**, называемая ***операцией косвенной адресации***, рассматривает свой операнд как адрес объекта и обращается по этому адресу, возвращая его содержимое.

Унарная операция **"&"**, называемая ***операцией нахождения адреса*** и, будучи примененной к переменной, возвращает ее адрес.

Например, рассмотрим последовательность операторов:

```
int x,y,*px,*py;     // Описание целочисленных переменных x,y
                     // и  указателей на целые значения px и py.
px = &x;             // Значением переменной px станет адрес переменной x.
y  = *px;            // Переменная y приобретает значение "того",
                     // на что указывает px, т.е. значение переменной x.
*px = 0;            // Переменная x получает значение 0.
y = *px + 1;         // Переменная y получает значение на 1 большее
                     // значения x.
*px += 1;            // Увеличение содержимого x на единицу.
*px++;               // Увеличение содержимого x на единицу, при
                     // этом постфиксная операция ++ не изменяет px,
                     // пока объект по адресу px не будет получен.
*++px;               // Префиксная операция ++(--) увеличивает
*--px;               // (уменьшает) px до получения значения x.
py = px;             // Копирование содержимого указателя px в ука-
                     // затель py в результате чего py указывает на
                     // то же, что и px.
```

Над указателями можно выполнять следующие операции:

1. *Присваивание значения указателя другому указателю того же типа*.
2. *Инициализация указателя*. Операторы: ***char a; char *pa=&a;*** описывают символьную переменную ***a*** и указатель ***pa*** на объект типа ***char***, а также инициализируют ***pa*** так, чтобы он указывал на ***a***.
3. *Сложение и вычитание указателей одного и того же типа*.
4. *Сложение и вычитание указателя и целого*.

Приведем несколько примеров.

Пусть ***p** - указатель на объект любого типа. Тогда оператор ***p++;*** увеличивает ***p*** так, что он указывает на следующий объект того же типа.

Пусть ***i*** - переменная целого типа. Оператор ***p += i;*** увеличивает указатель ***p*** так, чтобы он указывал на объект, отстоящий на **i** "единиц" памяти, занимае-

мых объектом данного типа, от объекта, на который указывает ***p***.

Еще пример. При объявлении ***long*** **xptr* и ***char*** **cptr* копилятором для хранения переменных типа *long* будет выделено по *четыре* байта, а для хранения переменных типа *char* по *одному* байту. Пусть, начиная с адреса памяти 100 расположены два целых числа (переменная *x* по адресу 100 и переменная *y* по адресу 104), а начиная с адреса 200 расположены переменные типа *char: s* по адресу 200, *k* по адресу 201 и *m* по адресу 202. Тогда:

**xptr=&x;//*инициализация указателя адресом переменной *x (xptr=100)*

*xptr=xptr+1;//*теперь *xptr=104*, т.е. адрес переменной *y*

**cptr=&s;//* инициализация указателя адресом переменной *s (xptr=200)*

cptr=cptr+1;// теперь *cptr=201*, т.е. адрес переменной *k*

cptr=cptr+1;// теперь *cptr=202*, т.е. адрес переменной *m*

Иными словами, прибавление или вычитание любого целого числа, увеличивает или уменьшает адрес на количесво байтов, соответствующее типу.

5. *Сравнение указателей одного и того же типа*. Если ****p*** и ****q*** - указатели на объекты *одного типа*, то к ним применимы операции отношения **(<, >=, >, <=, !=, ==)**.

Рассмотрим пример, опираясь на утверждения:

- Отношение ***p!=q*** истинно, если ***p*** и ***q*** указывают на разные объекты;
- Отношение ***p==q*** истинно, если ***p*** и ***q*** указывают на один и тот же объект.

Пример.

```
int x=10,y=10,*xptr=&x,*yptr=&y;// инициализация величин и указателей
if(xptr==yptr)          // сравнение величин указателей
 cout<<"адреса равны"<<endl;
else
 cout<<"адреса не равны"<<endl;
if(*xptr==*yptr)              // сравнение значений на которые они указывают
 cout<<"значения равны"<<endl;
else
 cout<<" значения не равны"<<endl;
```

В данном примере результатом первого сравнения будет «*ложь*» и на экран выведится сообщение «*адреса не равны*». Результатом второго сравнения будет «*истина*», что приведет к сообщению «*значения равны*».

6. *Присваивание указателю нуля (NULL) и сравнение указателя с нулем (NULL)*. Например, отношение ***p!=NULL*** истинно, если указатель ***p*** отличен от ***NULL***.

2.10.2 Массивы

Как правило, в реальном мире, мы сталкиваемся не с простейшими, единичными объектами, а с их совокупностью. Например, группа разных людей, объединенная одинаковыми функциональными обязанностями; или календарь какого-то года, состоящий из месяцев, имеющих постоянное название; или разные недели в месяце, состоящие из одних и тех же дней. Такие примеры – бесконечны. Их объединяет одно: каждый из ключевых объектов (например, год) состоит из строго определенных данных (например, месяцев, или дней). То есть, мы имеем совокупный набор одинаковых, по сути, данных. Поэтому, основная проблема языка Си – как описать такой набор данных, как единое целое?

Определение. ***Массив*** – это расположенные в памяти вплотную друг за другом элементы одного типа. Каждый массив имеет имя. Основные **свойства** массива:

- все элементы массива имеют один тип;
- элементы массива могут быть любого типа, в том числе типа, определенного пользователем. Элементами массива не могут быть только функции и элементы типа *void*;
- если в качестве элементов массива используются массивы, то речь идет о многомерных массивах (двумерных, трехмерных и т.д.). Одномерный массив иногда называют *вектором*, двухмерный массив - *матрицей*;
- индекс первого элемента всегда равен 0.
- имя массива (без квадратных скобок) является константой-указателем, содержащим адрес первого элемента массива;

В программах массив объявляется подобно простой переменной, но после имени массива указывается число его элементов в квадратных скобках:

<тип элементов массива> <имя массива>[<количество элементов>].

Пример. *int myArray[8];* //объявлен целочисленный массив из восьми эллементов

Массив, как и переменную, можно инициализировать при объявлении. Значения для последовательных элементов массива отделяются друг от друга запятыми и заключаются в фигурные скобки: *int iArray[8] = {7, 4, 3, 5, 0, 1, 2, 6};*

Обращение к отдельным элементам массива производится путем указания индекса элемента в квадратных скобках, например:

myArray[3] = 11; // в третий элемент массива помещено число 11.

myArray[i] = iArray[7-i]; //в содержимое массива с номером элемента *i* помещено содержимое из элемента с номером *7-i*.

Индекс должен быть целым выражением, значение которого не выходит за пределы допустимого диапазона. Поскольку индексация массивов начинается в

C/C++ всегда с нуля (т. е. первый элемент имеет индекс 0), то, если массив состоит из *N* элементов, индекс может принимать значения от *0* до *N-1*.

Итак, из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве.

Примеры:

int *days[12]; //массив-вектор из 12 элементов, принимающих целые значения (*тип *int).*

days[0]=31; *//первому элементу массива присвоено значение 31.*

days[1]=28; *//второму элементу массива присвоено значение 28.*

Здесь, в первой строчке мы объявили массив из 12 элементов и присвоили ему имя *days*. Остальные строки – *явное* присваивание значений элементам массива по их номеру. Чтобы присвоить конкретному элементу массива его значение используется операция индексации *[ ]*.

Из данного примера видно, что первый элемент массива имеет индекс 0, второй – индекс 1 и т.д.

Довольно частая шибка при составлении программ – это не учет того, что граница объявленного массива не контролируется компилятором. Например, если мы захотим распечатать некоторый 13-й элемент массива, в котором объявлено 12 элементов (*cout<<days[13];*), то компилятор не выдаст ошибки. Но что при этом произойдет вообще неизвестно: либо произойдет сбой программы, либо будет распечатано случайное число.

В качестве примера программы с массивами рассмотрим классический пример сортировки по возрастанию элементов массива, состоящего из 100 элементов методом простого последовательного перебора и сравнения элементов («пузырек»), используя вложенный цикл *for*.

```
int array[100];
…….
for (int i=0;i<99;i++)
  { for (int j=i+1;j<100;j++)
     if (array[j]<array[i])
       { int tmp=array[j];
         array[j]=array[i];
         array[i]=tmp;
       }
   }
```

Многомерные массивы формализуются списком константных выражений, следующих за идентификатором массива, причем каждое константное выражение заключается в свои квадратные скобки. Каждое константное выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных выражения, трехмерного – три и т.д. Например:

```
int a[2][3];      // объявление двумерного массива в виде матрицы с двумя
                  //строками и тремя столбцами, со следующей индексацией:
                  // a[0][0]  a[0][1]   a[0][2]
                  // a[1][0]  a[1][1]   a[1][2]
```

Такое объявление может представлять матрицу целых чисел, например, состоящую из двух строк и трех столбцов.

```
int w[3][3] = { { 2, 3, 4 },{ 3, 4, 8 },{ 1, 0, 9 } };
```

Как и в случае с одномерными массивами, такое задание элементов массива *w[3][3]* называется явным. Здесь следует помнить, что *w[0][0]=2, w[0][1]=3, w[0][2]=2* и т.д. Списки, выделенные в фигурные скобки, соответствуют строкам массива, в случае отсутствия скобок инициализация будет выполнена неправильно.

Интересной особенностью инициализации многомерных массивов является возможность не задавать размерность всех элементов массива кроме последнего. Приведенный выше пример может быть переписан так:

```
int w[ ][3] = { { 2, 3, 4 },{ 3, 4, 8 },{ 1, 0, 9 } };
```

В одномерных массивах отсутствие размерности в операции индексации, как правило, применяется при явном задании строки символов. Например, описание:

```
char str[ ] = "объявление символьного массива";
```

задает символьный массив из 31 элемента типа *char*. Здесь *str[0]* – это символ «*о*», *str[1]* – это символ «*б*», *str[30]* – это символ «*а*»,

Следует учитывать, что в символьном литерале находится на один элемент больше, так как последний из элементов является управляющей последовательностью «\0», поэтому *str[31]* – это символ «\0». Более детально символьные массивы будут рассмотрены в разделе по обработке строк в языке C/C++.

Рассмотрим пример листинга программы ввода элементов матрицы A размерностью 3×3, вывода ее на экран и изменения элемента с заданными индексами.

Листинг	**Результат**

`#include<iostream.h>`
`#include<stdio.h>`
`void main ( )`
`{`
`  int a[3][3], i,j;`
`  cout<<"Вод элементов матрицы A:\n";`
`  for(i=0;i<3;i++)`
`  for(j=0;j<3;j++)`
`  {`
`  printf("a[%d][%d]=",i+1,j+1);`	

```
        cin>>a[i][j];
     }

  cout<<"\n Вывод матрицы A:";
    for(i=0;i<3;i++)
     for(j=0;j<3;j++)
     { if(j%3==0)
         cout<<"\n\t"<<a[i][j];
       else
         cout<<"\t"<<a[i][j];
     }

  cout<<"\n\n Какой элемент матрица A изменить?:\n";
  cout<<"i=";cin>>i;
  cout<<"j=";cin>>j;
  printf(" Введите значение элемента a[%d][%d]=",i,j);cin>>a[i-1][j-1];

  cout<<"\n Вывод новой матрицы A:";
    for(i=0;i<3;i++)
     for(j=0;j<3;j++)
     { if(j%3==0)
         cout<<"\n\t"<<a[i][j];
       else
         cout<<"\t"<<a[i][j];
     }
}
```

```
 Ввод элементов матрицы A:
a[1][1]=1
a[1][2]=2
a[1][3]=3
a[2][1]=4
a[2][2]=5
a[2][3]=6
a[3][1]=7
a[3][2]=8
a[3][3]=9

 Вывод матрицы A:
        1       2       3
        4       5       6
        7       8       9

 Какой элемент матрицы A изменить?:
i=2
j=2
 Введите значение элемента a[2][2]=10

 Вывод новой матрицы A:
        1       2       3
        4       10      6
        7       8       9
```

2.10.3 Связь указателей и массивов.

Имя массива является ***указателем-константой***, значением которой служит ***адрес первого элемента массива*** *(с индексом 0)*.

Из этого утверждения следует, что доступ к первому элементу массива может быть осуществлен или непосредственно через индекс этого элемента в массиве:

<имя массива>[0],

или через указатель на имя массива, которое в свою очередь и является адресом первого элемента этого массива:

***<имя массива>**

Таким образом, в общем случае доступ к любому заданному элементу массива можно осуществить двумя способами:

- через индекс элемента:

<имя массива>[<номер элемента>]

- через указатель на массив и номер его элемента:

***(<имя массива>+<номер элемента>).**

Рассмотрим сказанное на примерах.

char *mas[ ]="понедельник";*//объявление и инициализация символьного массива.

char **cptr=&mas[0];*//инициализация указателя *cptr* адресом начала массива.

Последующий вывод на экран в таком виде:

cout<<cptr<<" "<<cptr+4<<" "<<cptr+10<<endl;

cout<<mas<<" "<<mas+4<<" "<<mas+10<<endl;

приведет к выводу одинаковых результатов;

понедельник дельник к

понедельник дельник к

Здесь мы определили адрес начального элемента массива, с которого он начинается, по этому адресу записана буква «*п*». Последующее прибавление позволяет сдвигать адрес на соответствующее количество байт вправо. Еще из этого примера очевидно, что само имя массива является указателем на адрес его первого элемента.

Аналогично, одинаковый результат даст и такой вывод:

*cout<<*cptr<<" "<<*(cptr+4)<<" "<<*(cptr+10) <<endl;*

cout<<mas[0]<<" "<<mas[4]<<" "<<mas[10]<<endl;

на экране будет:

п д к

п д к

В этом примере мы обращаемся к конкретному адресу элемента массива либо через указатель на элемент (первый случай), либо непосредственно через индекс этого элемента (второй случай).

При работе с указателями на адреса нужно быть внимательным с обращением к элементам массива. Например, неккоректной будет запись вида:

*cout<<*cptr<<" "<<*cptr+4<<" "<<*cptr+10<<endl;*

Здесь вначале идет обращение по адресу первого элемента и на экране будет его значение (буква «*п*»). Последующее приращение – это добавление десятичного числа к значению адреса, то есть, обращение совершенно по другому адресу ппамяти, в котором в данное время располагается какая-то другая, неизвестная нам величина. Поэтому и на экран может быть выведено не значение, например, как мы ожидали четвертого элемента массива (**cptr+4*), а совсем неизвестное значение.

Рассмотрим еще несколько любопытных примеров, показывающих применение указателей для символьных массивов.

Пример1. Вывести на экран заданную строку сисволов несколькими способами.

```
#include <iostream.h>
void main( )
{
  char x[ ]="Пример строки";// пусть это - заданный массив символов.
int i=0;
cout<< "\n Вывод первого символа строки (1 способ): "<< x[0];
cout<< "\n Вывод первого символа строки (2 способ):  "<< *x;
cout<< "\n Вывод всей строки без цикла (1 способ):  "<< x;
cout<< "\n Вывод всей строки без цикла (2 способ): "<< &x[0];
cout<< "\n Вывод всей строки с  циклом (1 способ):  ";
while (x[i]!='\0')// пока не встретиться элемент массива, в котором содер-
                                жится признак окончания строки (/0)
     cout << x[i++];// выводим элемент массива по его индексу
cout<< "\n Вывод всей строки с  циклом (2 способ):  ";
i=0;
  while (*(x+i)!='\0') // пока не встретиться адрес, содержимое которого -
                                признак окончания строки (/0)
     cout << *(x+i++); // выводим элемент массива по его адресу
}
```

В языке С++ нет специального типа данных "строка". Вместо этого каждая символьная строка в памяти компьютера представляется в виде одномерного массива типа ***char***, последним элементом которого является символ **'\0'.** Изображение строковой константы может использоваться по-разному. Если строка применяется для инициализации массива типа ***char***, например, так:

char array[ ] = "инициализирующая строка";,

то адрес первого элемента строки становится значением указателя-константы (имени массива) ***array***.

Если строка используется для инициализации указателя типа ***char*** **:**

char *pointer = "инициализирующая строка";,

то адрес первого элемента строки становится значением указателя-переменной ***pointer***.

Если использовать строку в выражении, где разрешено применять указатель, то используется адрес первого элемента строки:

```
char * string;
string = "строковый литерал";
```

В данном примере значением указателя **string** будет не вся строка "строковый литерал", а только адрес ее первого элемента. Более подробно о работе с символьными массивами (строками) мы рассмотрим позже.

Пример 2. Программа поиска максимального элемента массива. В программе для наглядности использованы два равнозначных способа доступа к элементу массива.

```
...
```

```
void main( )
{int a[4]={3,6,9,2}, *pa=a;              //объявление массива и указателя на него
      int i, max_a=a[0];
      for(i=0;i<4;i++)
        if(*(a+i)>max_a       //доступ к элементу массива через его имя
            max_a = *(pa+i); //доступ к элементу  массива через его адрес
            cout<<max_a;
}
```

Очевидно, что запись тела цикла в приведенном примере эквивалентна записи:

```
for(i=0;i<4;i++)
  if(a[i]>max_a)
      max_a = a[i];  //доступ к элементу массива по индексу
```

Полезен будет следующий пример инициализации массива и ввода/вывода на экран его элементов.

Пример 3. Доступ к элементам массива:

а) по имени-указателю:

```
void main( )
 {int a[4];int i;
   for(i=0;i<4;i++)
        scanf(%d,"(a+i));
   for(i=0;i<4;i++)
        printf(%d\n,"*(a+i));
 }
```

Обратите внимание, что в функции *scanf* знак & (операция взятия адреса) не используется. Это связано с тем, что имя массива, как уже говорилось, и есть указатель, значение которого равно адресу первого по счету (нулевого) элемента массива.

В функции *printf* мы обращаемся к значению элемента массива и используем указатель на этот объект.

б) Эквивалентная запись программы с доступом **по индексу массива:**

…

```
void main( )
{int a[4];int i;
   for(i=0;i<4;i++)
        scanf(%d,"&a[i]);
   for(i=0;i<4;i++)
        printf(%d\n,'a[i]);
}
```

Рассмотрим **несколько примеров** совместного использования указателей

и массивов.

Пример 1. Ввод пяти элементов одномерного массива без использования цикла.

```
#include<iostream.h>
int a[5],*uk;
main ( )
{
  uk = &a[0];
  cin >> *(uk) >> *(uk+1) >> *(uk+2) >> *(uk+3) >> *(uk+4);
  cout << *(uk) << " " << *(uk+1) << " " << *(uk+2) << " " \
 << *(uk+3) << " " << *(uk+4);
}
```

Пример 2. Определение разности между наибольшим и наименьшим элементами массива.

```
#include<iostream.h>
main ( )
{
  int u,y,s,a[20],*ukazatel=&a[0];
  /* ---------------------- */
  for (int w=0; w<=5; w++)
  {
    cout << "\n Вводите элемент массива... ";
    cin >> *(ukazatel+w);
  }
  u = y = *ukazatel;
  for (w=0; w<=5; w++)
  {
    if  (y<=*(ukazatel+w)) y = *(ukazatel+w);
    else
      if (u>=*(ukazatel+w)) u = *(ukazatel+w);
  }
  cout << y-u << endl;
}
```

Пример 3. Вычисление среднего арифметического элементов массива.

```
#include<iostream.h>
main ( )
{
  int a[20],*ukazatel = &a[0];
  for (int w=0; w<=3; w++)
  {
    cout << "Введите очередной элемент массива: ";
    cin >> *(ukazatel+w);
```

```
  }
  int y = 0;
  for (w=0; w<=3; w++)
    y = y + *(ukazatel+w);
  cout << "Среднее арифметическое  = " << float(y)/w;
}
```

Пример 4. Определение последнего элемента заданного числового массива, меньшего 3.

```
#include<iostream.h>
main ( )
{
  int a[20],*ukazatel = &a[0];
  cout << "Программа выводит последнее число, меньшее 3" << endl;
  for (int w=0; w<=3; w++)
  {
    cout << "Вводи число... ";
    cin >> *(ukazatel+w);
  }
  cout << "\n Элемент: ";
  int y = 3,f=0;
  for (w=3; w>=0; w--)
 if  (y > *(ukazatel+w))
 {
          cout << *(ukazatel+w); f=1; break;
 }
  if (!f) cout << "таких элементов нет.";
}
```

И еще один пример, иллюстрирующий эквивалентность использования в программах имени массива и указателя на его адрес.

Пример 4. Сортировка (“пузырек”) массива символов.

```
#include<iostream.h>
#define n 10 //предопределенная константа n=10
main ( )
{
  char m[n],*p=m,temp;
  /* ------- */
  cout << "\n Ввод " << n << " символов для сортировки" << endl;
  for (unsigned i=0; i<n;i++)
    cin>>*(m+i);
  cout << "\n Исходный массив m[i],i=1," << n << ":\n";
  for (i=0; i<n; i++)
    cout << *(p+i);
```

```
  cout << "\n Начало сортировки:\n";
  for (unsigned k=0; k<n-1; k++)
  {
    for (i=k+1; i<n; i++)
      if  (*p>*(m+i))
      {
        temp = *p; *p = *(m+i); *(m+i) = temp;
      }
  }
  cout << " Упорядоченный массив m[i],i=1," << n << ":\n";
  cout << "(распечатка с использованием *(m+i))\n";
  for (i=0; i<n; i++)
    cout << *(m+i) << ",";
  cout << "\n Упорядоченный массив m[i],i=1," << n << ":\n";
  cout << "(распечатка с использованием *(p+i))\n";
  for (i=0; i<n; i++)
    cout << *(p+i) << ",";//здесь ошибка!
}
```

Результат работы программы:

```
Ввод 10 символов для сортировки
fywaproldv
Исходный массив m[i],i=1,10:
f,y,w,a,p,r,o,l,d,v,
Начало сортировки:
Упорядоченный массив m[i],i=1,10:
(распечатка с использованием *(m+i))
a,d,f,l,o,p,r,v,w,y,
Упорядоченный массив m[i],i=1,10:
(распечатка с использованием *(p+i))
y, ,н, , ,ц,Ь,y, ,o,
```

В конце программы упорядоченный массив ***m[ ]*** распечатывается дважды, причем второй вариант (с обращением к элементам массива ****(p+i))*** является *ошибочным*, поскольку указатель ***p*** на массив ***m[ ]*** изменялся в цикле (конструкция ***p++***;) и к рассматриваемому моменту указывает уже не на начало массива **m**, а на его последний элемент.

2.10.4 Массивы указателей

Для понимания конструкций, состоящих из набора звездочек, скобок и имен типов, нужно аккуратно применять синтаксические правила, учитывающие последовательность выполнения операций.

Например, определение:

int *array[6];

вводит ***массив указателей на объекты типа*** **int**. Имя массива – ***array***, он состоит из шести элементов, тип каждого **int .**

Определение:

int (*ptr)[6];

вводит ***указатель ptr на массив из шести элементов, каждый из которых имеет тип*** **int**. Возможность создания массивов указателей позволяет экономить память при использовании многомерных массивов.

По определению массива, его элементы должны быть однотипные и одинаковые по длине. Пусть необходимо определить массив для представления списка из двадцати пяти фамилий. Если определять его как двумерный массив типа ***char***, то в определении элементов массива необходимо задать предельные размеры каждого из двух индексов, например: ***char spisok[25][20];***. Таким образом определено, что количество фамилий в списке не более 25 и что длина каждой фамилии не превышает 19 символов (букв). При определении массива одну из его предельных размерностей (значение самого левого индекса) можно не указывать. В этом случае количество элементов массива определяется, например, явной инициализацией (рис.1):

char spisok[ ][20] = {"ИВАНОВ", "ПЕТРОВ", "СИДОРОВ"}; .

Массив spisok	Массивы char[20]																	
И	В	А	Н	О	В	0	0	0	0	0	0	0	0	0	0	0	0	\0
П	Е	Т	Р	О	В	0	0	0	0	0	0	0	0	0	0	0	0	\0
С	И	Д	О	Р	О	В	0	0	0	0	0	0	0	0	0	0	0	\0

Рис.1. Схема размещения в памяти элементов массива *spisok*

Теперь в массиве ***spisok*** только 3 элемента, каждый из них длиной 20 элементов типа ***char*** (20 байт). В противоположность этому при определении и инициализации этими же символьными строками одномерного массива указателей типа ***char ****** память распределяется гораздо рациональнее (рис.2):

char *pointer [3] = {"ИВАНОВ", "ПЕТРОВ", "СИДОРОВ"}

Массив pointer	Строковые константы							
* →	И	В	А	Н	О	В	\0	
* →	П	Е	Т	Р	О	В	\0	
* →	С	И	Д	О	Р	О	В	\0

Рис.2. Схема размещения в памяти элементов с помощью массива указателей

Применение указателей и их массивов позволяет рационально решать, в частности, задачи сортировки сложных объектов с неодинаковыми размерами.

Пример 1. Задача сортировки строк матрицы.

Матрица с элементами типа ***double*** представлена двумерным массивом ***array [n][m]***, где ***n*** и ***m*** – целочисленные константы. Предположим, что нужно упорядочить строки матрицы в порядке возрастания сумм их элементов. Чтобы не переставлять сами строки исходного массива, введен вспомогательный одномерный массив указателей ***double *par[n]***. Инициализируем его элементы адресами массивов строк. В качестве значений элементов массива используем номера строк. Матрицу напечатаем три раза: до и после сортировки с помощью вспомогательного массива указателей и (после сортировки) с использованием основного имени массива.

Приведем текст этой программы.

```
#include <iostream.h>
void main( )
{
  const int n=5; //Количество строк.
  const int m=7; //Количество столбцов.
  double array[n][m]; //Основной массив.
  for (int i=0;i<n;i++)
      for (int j=0;j<m;j++)
        array[i][j]=n-i; //Заполнение массива.
  double *par[n]; //Массив указателей.
  for (i=0;i<n;i++)  //Цикл перебора строк.
      par[i]=(double*)array[i];
  //Печать массива через указатели.
  cout << "\n До перестановки элементов массива  указателей: ";
  for (i=0;i<n;i++)
    { cout<< "\n строка " << i+1 <<": ";
      for (int j=0;j<m;j++)
              cout << "\t" << par[i][j];
    }
  //Упорядочение указателей на строки массива.
  double si,sk;
  for (i=0;i<n-1;i++)
      { for (int j=0, si=0;j<m;j++)
            si+=par[i][j];
            for (int k=i+1;k<n;k++)
               { for (j=0,sk=0;j<m;j++)
                        sk+=par[k][j];
                 if (si>sk)
                    { double *pa=par[i];
                      par[i]=par[k];
                      par[k]=pa;
```

```
                    double a=si;
                    si=sk;
                    sk=a;
                   }
                }
            }
//Печать массива через указатели.
  cout << "\nПосле перестановки элементов массива:";
  for (i=0;i<n;i++)
    { cout << "\n строка " << i+1 <<": ";
      for (int j=0;j<m;j++)
              cout << "\t" << par[i][j];
    }
  cout << "\nИсходный массив остался неизменным: ";
  for (i=0;i<n;i++)
    { cout << "\nstroka " << i+1 <<": ";
      for (int j=0;j<m;j++)
               cout << "\t" << array[i][j];
    }
}
```

```
До перестановки элементов массива  указателей:
строка 1:       5       5       5       5       5       5       5
строка 2:       4       4       4       4       4       4       4
строка 3:       3       3       3       3       3       3       3
строка 4:       2       2       2       2       2       2       2
строка 5:       1       1       1       1       1       1       1
После перестановки элементов массива:
строка 1:       1       1       1       1       1       1       1
строка 2:       2       2       2       2       2       2       2
строка 3:       3       3       3       3       3       3       3
строка 4:       4       4       4       4       4       4       4
строка 5:       5       5       5       5       5       5       5
Исходный массив осталс  неизменным:
stroka 1:       5       5       5       5       5       5       5
stroka 2:       4       4       4       4       4       4       4
stroka 3:       3       3       3       3       3       3       3
stroka 4:       2       2       2       2       2       2       2
stroka 5:       1       1       1       1       1       1       1
```

2.10.5 Задачи для самопроверки по работе с массивами и указателями

2.11 Работа со строками

По некоторым оценкам, до 70% времени компьютер при выполнении программ тратит на манипуляцию с текстовыми строками: копирует их из од-

ного места памяти в другое, проверяет наличие в строке определенных слов, сцепляет или усекает символы и т.д.

Да и человек, в процессе информационного обмена большую часть информации получает в результате обработки текста. Текст, как правило, представим в виде совокупности лексических единиц (*лексем*). Каждая лексема – это, в смысле естественного языка, отдельное слово, состоящее из последовательности символов. С точки зрения машинной реализации, ***совокупная последовательность символов*** образует ***строку***.

В отличие от некоторых других языков программирования, в языке C/C++ нет специального типа данных, инициализирующего строки. Работа с набором символов, образующим строку, происходит либо посредством инициализации одномерного массива символов, либо посредством инициализации указателя на символьный тип. Например, абсолютно идентичны записи вида:

```
char str[ ]="string";
char str[6]={"s", "t", "r", "i", "n", "g"};
char*str="string";
```

Здесь в первом примере мы определили адрес начала символьного массива, задав ему имя *str* и, начиная с его первого элемента, последовательно поместили набор из шести символов. Во втором примере, мы однозначно определили размер символьного массива и каждому из его элементов явно присвоили значение. В третьем случае, используя именной указатель символьного типа, мы разместили в памяти по его адресу последовательность символов.

Обратим внимание, что *во всех случаях речь идет о значении элемента, находящегося по некоторому адресу*. Зная теперь, что строка – это последовательность символов, строго следующих друг за другом и также последовательно размещенных в памяти, можно получить доступ к этим элементам, указав лишь адрес начала их размещения в памяти.

Поэтому при работе со строками последний из приведенных примеров инициализации строки – предпочтительней. Более того, в последнем случае, при компиляции после последнего символа (*g*) будет размещен специальный признак (*\0*) – *нуль-терминатор*, который позволяет всегда определить длину строки. Точнее, после последовательного перебора символов этот признак не позволит выйти за пределы строки. Если же в строке не будет этого признака, то ее обработка может оказаться сколь угодно долгой.

В отличие от числовых переменных, при работе со строками недопустимы такие простые операции как сравнение или копирование друг в друга с помощью оператора присваивания (=). Например, если объявлено *char s1[ ]="Visual", s2[ ]="C++"*, то неверным будут, например такая инструкция:

```
if(s2=="C++")
 s2=s1;
```

Так как строка представляет собой массив символов, то для решения такой задачи необходимо поочередно сравнивать или копировать все элементы символьного массива.

Поэтому в языке C/C++ предусмотрена достаточно богатая коллекция спе-

циальных функций по обработке символьных строк.

1. Библиотечные функции обработки текстовых строк.
2. Разбор задач по обработке строковых данных.
3. Примеры применения функций обработки текстовых строк в программах

2.11.1 Библиотечные функции обработки текстовых строк

Прототипы функций по обработке строк содержатся в заголовочном файле <***string.h***>. В качестве параметров этим функциям, как правило, передаются указатели на строки. Хотя в данной библиотеке сосредоточен достаточно большой набор различных функций по обработке строк, мы рассмотрим лишь некоторые и, в частности, те из них, которые будут использоваться нами в разрабатываемых программах.

1) Функция *strcat( )*

Функция *strcat* добавляет одну строку к другой.

Синтаксис: ***char*** **strcat (**char** *dest, **char** *src);*

Пример:

```
#include <iostream.h>
#include <string.h>
void main (void)
{
char *c = "C++", *vsl = "Visual";
strcat (vsl,c);
cout<<vsl<<endl;
}
```

В результате на экране будет сообщение: «***VisualC++***».

2) Функция *strcmp( )*

Функция *strcmp* сравнивает одну строку с другой.

Синтаксис: ***int*** *strcmp (**char** *s1, **const char** *s2);*

Функция strcmp( ) выполняет беззнаковое сравнение строк *s1* и *s2*, начиная с первого символа в каждой строке и продолжая сравнение последующих символов до тех пор, пока не встретятся не совпадающие символы или строки не кончатся.

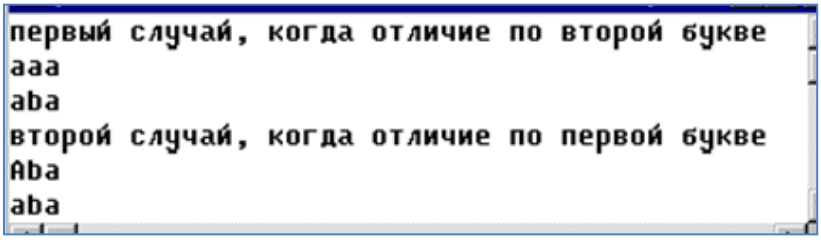
Возвращаемые значения *strcmp* могут быть:

- ***< 0*** , если *s1* меньше *s2* (раньше по алфавиту);
- ***= 0***, если *s1* равно *s2* (все символы строго совпали);
- ***> 0***, если *s1* больше *s2* (по алфавиту позднее).

Пример и результат работы программы:

```
#include <stdio.h>
#include <string.h>
void main (void)
```

```
{
char *buf1 = "aaa", *buf2 = "aba", *buf3 = "Aba"; int prt;
printf("первый случай, когда отличие по второй букве\n");
prt = strcmp (buf2,buf1);
if (prt>0) printf ("%s\n%s\n",buf1,buf2);
else printf ("%s\n%s\n",buf2,buf1);
printf("второй случай, когда отличие по первой букве\n");
prt = strcmp (buf2,buf3);
if (prt>0) printf ("%s\n%s\n",buf3,buf2);
else printf ("%s\n%s\n",buf2,buf3);
}
```



3) Функция *strlen( )*

Функция *strlen* вычисляет длину строки.

Синтаксис: ***int strlen (const char** *s);*

Возвращаемое значение *strlen( )* – число символов в строке *s*, не считая нулевого окончания.

Пример.

Если было объявлено: … ***char** *str=”string”; **int** x;* , то вызов функции *x=**strlen**(str)* и ***printf**(“x=%d”,x);* позволит получить на экране результат: ***x=6.***

В этом примере длина элементов массива символов *”string”* равна 6. Докажем, что, исходя из определения строки, после последнего символа стоит нуль-терминатор. Добавим в программу цикл последовательного вывода символов, написав следующую программу:

```
#include <stdio.h>
#include <string.h>
void main( )
{char *str="string"; int x;
 x=strlen(str);  printf("x=%d\n",x);
 for(int i=0;i<7;i++)
   printf("индекс массива=%d, указатель на %d элемент массива
          =%s\n",i,i,str+i);
}
```

Результат программы -

```
x=6
индекс массива=0, указатель на 0 элемент массива =string
индекс массива=1, указатель на 1 элемент массива =tring
индекс массива=2, указатель на 2 элемент массива =ring
индекс массива=3, указатель на 3 элемент массива =ing
индекс массива=4, указатель на 4 элемент массива =ng
индекс массива=5, указатель на 5 элемент массива =g
индекс массива=6, указатель на 6 элемент массива =
```

свидетельствует о том, что после последнего элемента с индексом 5 (символ «*g*») стоит еще один с индексом 6 (седьмой символ – нуль-терминатор «\0»).

4) Функция *strcpy( ).*

Функция *strcpy* копирует содержимое одной строки в другую.

Синтаксис: ***char*** *strcmp (****char*** **s1, **const char** *s2);*

Пример:

```
#include <iostream.h>
#include <string.h>
void main (void)
{
char *s1, *s2 = "Visual";
strcpy (s1,s2);
cout<<"s1="<<s1<<endl;
}
```

В результате выполнения программы на экране будет получен результат:

s1=Visual.

5) Функция *strtok( ).*

Суть большинства реальных задач по обработке текстовой информации сводится к тому, чтобы в заданном фрагменте текста найти нужное слово, установить его адрес или позицию в данном тексте. Это означает, что поисковая программа должна «уметь» выделять отдельные слова (лексемы) или строки по признаку их окончания (нуль-терминатору). В реализации C/C++ такая обработка отдельных слов возможна благодаря функции *strtok( )*.

Легко заметить, что любой текст состоит из слов, разделенных либо пробелами, либо знаками препинания. Если каким либо образом выделить все эти признаки разделения и отслеживать их наличие (отсутствие), то можно определить конкретное слово. Именно на этом и строится механизм функции *strtok( )*.

Функция *strtok( )* просматривает одну строку на лексемы, которые выделены ограничителями, определяемыми во второй строке.

Синтаксис: ***char*** **strtok (****char*** **s1, **const char** *s2);*

Функция *strtok( )* рассматривает строку *s1*, как состоящую из последовательности нулей или лексем, выделенных с помощью символов из строки *s2*.

Первый вызов функции *strtok( )* возвращает указатель на первый символ лексемы в строке *s1* и записывает нулевой символ в строку *s1* непосредственно сразу за выделенной лексемой. Последующие вызовы со значением *NULL* в ка-

честве первого аргумента будут обрабатывать строку *s1* таким же образом, пока не кончатся все лексемы.

Строка разделитель *s2* от вызова к вызову может меняться.

Возвращаемое значение функции *strtok( )* – указатель на лексему, находящуюся в *s1*. Когда закончатся все лексемы, содержащиеся в строке *s1*, функция *strtok( )* возвратит *нулевой указатель (NULL)*.

Работу функции *strtok( )* удобно проиллюстрировать следующим примером:

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

void main (void)
{
 char *s1= "мои адреса на mail: xxx2014@mail.ru, на google:
            yyy_2014@gmail.com";
 char *s2=" ",*s3;
 s3=strtok(s1,s2);

 cout<<s3<<"\n"<<s3+1<<"\n"<<s3+3<<"\n"<<s3+4<<"\n";
 getch( );
}
```

```
мои
ои

адреса на mail: xxx2014@mail.ru, на google: yyy_2014@gmail.com
```

Здесь в качестве разделителя лексем в *s2* мы выбрали пробел. Этот блок программы показывает, что функция *strtok( )* работает почти так же, как и функция копирования *strcpy( )*, только результатом ее работы будет не число (*<0, 0, >0*), а последовательность символов между символом *\0* и одним из символов разделителей, указанных в *s2* То есть в *s3* копируется вся строка *s1*, но при первом вхождении в *strstok( )* в *s3* передается первая лексема, отделенная от других пробелом, после которой приписывается *NULL*-терминатор по правилу хранения массива символа для строк. Сказанное подтверждает побайтовый вывод *s3*. Следовательно, чтобы «избавиться» от всех *NULL*-терминаторов и вывести последовательно все лексемы, необходимо заменить эти *NULL*-терминаторы символами-разделителями. И делать это нужно до тех пор, пока не встретиться последний терминатор, фиксирующий истинное окончание строки. Сделать это можно так.

```
void main ( )
{
```

```
char *s1= "мои адреса на mail: xxx2014@mail.ru, на google:
           yyy_2014@gmail.com";
char *s2=",,:, ",*s3; // в качестве разделителей выбраны запятая двоето-
                      чие и пробел
s3=strtok(s1,s2);
int i=1
do
 {cout<<"лексема"<<i<<" = "<<s3<<endl;
  s3=strtok(NULL,s2);
  i++;
  }
while(s3!=NULL);

}
```

Результат работы программы:

```
лексема 1 = мои
лексема 2 = адреса
лексема 3 = на
лексема 4 = mail
лексема 5 = xxx2014@mail.ru
лексема 6 = на
лексема 7 = google
лексема 8 = yyy_2014@gmail.com
```

6) Функция *strstr( )*.

Функция *strstr( )* ищет в строке *s1* подстроку *s2*. Возвращает указатель на тот элемент в строке *s1*, с которого начинается подстрока *s2*.

Синтаксис: ***char** *strstr (**char** *s1, **const char** *s2);*

Пример:

```
#include <iostream.h>
#include <string.h>
void main ( )
{
 char *s1= "мои адреса на mail: xxx2014@mail.ru, на google:
            yyy_2014@gmail.com";
 char *s2="@",*s3;
s3=strstr(s1,s2);
cout<<s3<<endl;
}
```

Результат:

```
@mail.ru, на google: yyy_2014@gmail.com
```

Таким образом, единичное использование функции *strstr( )* находит первое вхождение подстроки *s2* в строке *s1*. Чтобы найти и все последующие такие вхождения, приведенную программу достаточно доработать, например, так.

```
void main ( )
{
 char *s1= "мои адреса на mail: xxx2014@mail.ru, на google:
yyy_2014@gmail.com";
 char *s2="@",*s3;
 do
  { s3=strstr(s1,s2);
    cout<<s3<<endl;
    strcpy(s1,s3+1);
   }
  while(s3!=NULL);
}
```

Результат:

```
@mail.ru, на google: yyy_2014@gmail.com
@gmail.com
```

Здесь на каждом шаге нахождения подстроки *s2* в строку *s1* переписывается ее остаток, сдвинутый на один байт (все символы, стоящие в массиве сразу после *s2*). Так происходит до тех пор пока в остатке не окажется последний символ символьного массива – *NULL*-терминатор.

Использование двух последних функций наталкивает на мысль об их применении в поиске каких-либо слов в тексте. Пусть, например, нужно найти и составить список всех почтовых *e-mail* адресов. Тогда искомая программа может выглядеть, например, так.

```
void main ( )
{
 char *s1= "мои адреса на mail: xxx2014@mail.ru, на google:
            yyy_2014@gmail.com";
 char *s2=",, ",*s3; // в качестве разделителей выбраны запятая и пробел
 s3=strtok(s1,s2);
 int i=1;
 do
  {if(strstr(s3,"@")!=NULL)
    cout<<"лексема"<<i<<" = "<<s3<<endl;
   s3=strtok(NULL,s2);
   i++;
  }
 while(s3!=NULL);
}
```

Результат:

```
лексема 5 = xxx2014@mail.ru
лексема 8 = yyy_2014@gmail.com
```

Здесь программа выделяет из текста только те лексемы, в которых подстрокой выступает символ @ – признак e-mail адресов.

2.11.2 Примеры применения функций обработки текстовых строк в программах

Используя функции по работе с символьными массивами, рассмотрим несколько примеров полезных программ.

- **Ввести и вывести на экран, в порядке ввода, нумерованный список из пяти «фамилий».**

Особенность задачи состоит в том, что к каждому элементу символьного массива приписан его номер. Такой список можно представить в виде матрицы из двух столбцов (первый из которых числового типа, а второй – символьного) и пяти строк.

Пример программы:

```
typedef char fam[20];
void main( )
{ fam fio[5]; int i;
  cout<<"Введите список фамилий\n";
  for(i=0;i<5;i++)
   cin>>fio[i];
  cout<<"\n\n Вывод нумерованного списка после ввода:\n";
  for(i=0;i<5;i++)
   cout<<i+1<<"\t"<<fio[i]<<"\n";
}
```

Результат:

```
Введите список фамилий
Иванов
Сидоров
Петров
Антонов
Данин


 Вывод нумерованного списка после ввода:
1       Иванов
2       Сидоров
3       Петров
4       Антонов
5       Данин
```

Особенность программы состоит в создании нового пользовательского типа ***fam*** – символьного массива из 20 элементов. Это возможно при использовании ключевого слова ***typedef***.

Синтаксис ***typedef char fam[20]*** читается так: с помощью ***typedef*** объявлен

новый тип ***fam[20]***, фактический тип которого ***char***. Тогда ***fam*** *fio[5]* – и есть объявление списка из пяти элементов символьного типа, длина которых не более 20 символов или объявление матрицы размерностью 2×5.

- **Отсортировать список «фамилий», полученный в результате выполнения программы предыдущего примера, по возрастанию (по алфавиту от А до Я).**

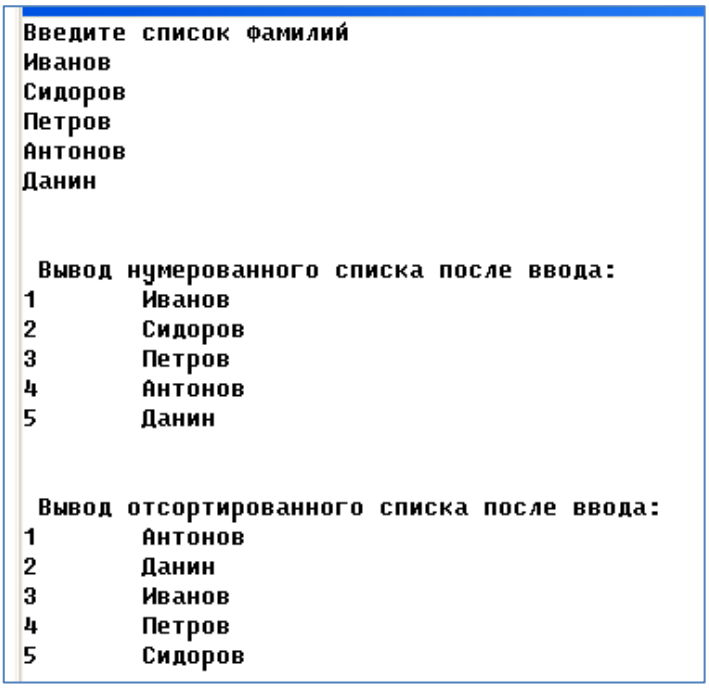
Так же, как и в ранее рассмотренных примерах сортировки массивов чисел, мы не будем задаваться вопросом написания программы, использующей какой-либо эффективный алгоритм сортировки. Различные алгоритмы рассмотрим отдельно позднее. Здесь применим самый простой – «пузырьковый» алгоритм, обычный перебор массивов и последовательное их сравнение друг с другом.

Для этого, как известно, необходима промежуточная переменная для хранения временного результата сравнения. Так как сортировка производится по символьному массиву *char fam[20]*, то и временная переменная должна быть такого же типа. Обозначим ее *char name[20]*.

Полностью программа может выглядеть так.

```
typedef char fam[20];
void main( )
{ fam fio[5]; int i;
 char name[20]; //переменная для хранения временного результата перестановки
  cout<<"Введите список фамилий\n";
  for(i=0;i<5;i++)
   cin>>fio[i];
  cout<<"\n\n Вывод нумерованного списка после ввода:\n";
  for(i=0;i<5;i++)
   cout<<i+1<<"\t"<<fio[i]<<"\n";
                    // Упорядочение списка:
  for(i=0;i<4;i++)
   for(int j=i+1;j<5;j++)
   {if(stricmp(fio[j],fio[i])<=0)
    {strcpy(name,fio[i]);
     strcpy(fio[i],fio[j]);
     strcpy(fio[j],name);
    }
   }
  cout<<"\n\n Вывод отсортированного списка после ввода:\n";
  for(i=0;i<5;i++)
   cout<<i+1<<"\t"<<fio[i]<<"\n";
}
```

Результат:



Здесь в блоке упорядочения списка используются две строковые функции: *stricmp( )* и s*trcpy( )*. Первая из них сравнивает два символьных массива – предыдущий и следующий из введенного списка. При этом применяется функция, игнорирующая регистр символов *stricmp( )*. Вторая функция, s*trcpy( )*, копирует содержимое одной переменной в другую. Ее применение необходимо по той причине, что, как было отмечено выше, применение обычной операции присваивания (*name=fio[i]*) для символьных массивов недопустимо.

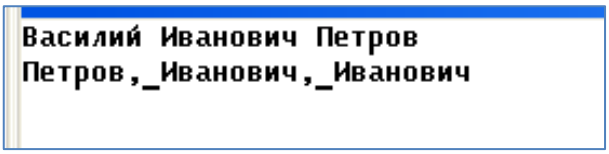
- **Составить программу, которая считывает имя, отчество и фамилию, а выводит на экран фамилию и через запятые имя и отчество.**

На первый взгляд – очень простая задача. Для ее решения всего лишь нужно воспользоваться функцией конкатенации *strcat* (добавления одной строки к другой). Однако, правила работы с символьными массивами кроют в себе свои особенности. Например, если запустить на выполнение такую программу:

```
void main( )
{ char *im,*ot1,*fio, s[]=",_"; //блок описания переменных
  cin>>im>>ot1>>fio;

 strcat(fio,s);
 strcat(fio,im);
 strcat(fio,s);
 strcat(fio,ot1);
 printf("%s\n",fio);
},
```

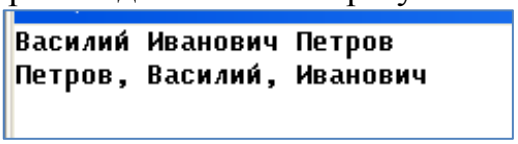
то можно получить неожиданный результат, например:



Все дело в «Блоке описания переменных»: *char *im,*ot1,*fio, s[]=",_";* и в «адресной арифметике».

Мы знаем из определения массива, что его имя является адресом первого элемента, точкой входа в адрес памяти, где последовательно хранятся все его элементы. Объявлением *char *im,*ot1,*fio* мы определили три указателя на значение используемого в объявлении типа, но не на конкретный адрес в памяти, где начинается переменная данного типа. Может оказаться так, что все три объявленные указателя будут использовать один и тот же участок памяти, а какие-либо операции с этими указателями приведут и к изменению содержимого данного участка. Это и произошло в нашем примере: уже после первой операции *strcat* область, где хранилось имя оказалась измененной.

Чтобы избежать подобного результата, нужно более жестко конкретизировать «Блок описания переменных». Например, так: *char im[20],ot1[20],fio[30],s[]=", "* Здесь, указывая имена символьных массивов и определяя их размер, мы заведомо определяем для них свой участок памяти. И в этом случае та же программа даст искомый результат:



- **Задача инвертирования массива. Дан массив символов (строка) размерностью *N*. Необходимо получить строку, в которой символы расположены в обратном порядке.**

Эта задача может быть решена различными способами, рассмотрим один из них.

В предыдущем примере мы косвенно говорили об «адресной арифметике» и как связаны указатели и массивы символов. Воспользуемся этим свойством. Пусть дан массив символов, например, строка «*программист*». В ней 11 символов. По правилу образования массивов эти символы имеют индексацию от 0 до 10. Доступ к каждому из них возможен различными способами: например, к букве «*п*» – *m[0]*, или *m, или *(m+0)*; к букве «*р*» – *m[1]*, или *(m+1)*. Именно этим свойством доступа мы и воспользуемся.

Теперь об алгоритме. Заметим, что если в слове три буквы («*Ура*»), то нужна только одна перестановка («*у*» и «*а*» меняются местами). Если четыре буквы («*Урал*») – нужны две перестановки (сначала меняются местами «*у*» и «*л*», затем – «*р*» и «*а*»). Если пять букв («*Урале*») – снова две перестановки (буква «*а*» при этом остается на месте); шесть букв («*Уралец*») – три перестановки и т.д. То есть, в любом случае перестановок должно быть не больше половины от количества *N* букв в слове. В языке C/C++ операция над целыми числами *N/2* (взять целое от деления) позволит нам найти количество переста-

новок.

Таким образом, искомая программа может выглядеть, например, так.

```
#include <iostream.h>
#include <string.h>
void main( )
{ char m[1000], s;
  int i,N;
  cin>>m;
  N=strlen(m); //определение количества символов в массиве
  for(i=0;i<N/2;i++)
  {
   s=*(m+i);
   *(m+i)=*(m+(N-i-1));
   *(m+(N-i-1))=s;
  }
 cout<<m;
}
```

Результат:

```
введите слово -> программист

инвертированный массив: тсиммаргорп
```

- **Определить является ли слово палиндромом. Палиндром – слово (предложение) одинаково читающееся в обоих направлениях.**

Алгоритм прост. Сначала решается задача инвертирования исходного массива. Полученный результат сохраняется в новом массив. Затем, используя функцию сравнения строк, определяется, является ли слово палиндромом.

Искомая программа может выглядеть так.

```
void main( )
{ char m[1000],m1[1000],s,c;
  int i,N;
 do
  {
   cout<<"\n Введите слово:-> ";
   cin>>m1;
   strcpy(m,m1);// копирование (сохранение) исходного массива
   N=strlen(m1); //определение длины массива
   for(i=0;i<N/2;i++)
    {
      s=*(m1+i);
      *(m1+i)=*(m1+(N-i-1));
      *(m1+(N-i-1))=s;
    }
```

```
  cout<<" Результат инверсии: "<<m1;
  if(strcmp(m,m1)==0) // сравнение исходного и инвертированного масси-
вов
   cout<<"\n\nЭто слово - ПАЛИНДРОМ";
  else
   cout<<"\n\nЭто слово - НЕ ПАЛИНДРОМ";
  cout<<"\n\n Продолжить? Да - любой символ, Нет -'q' -> "; cin>>c;
 }while(c!='q');
}
```

Результат:

```
Введите слово:-> ПОТОК
Результат инверсии: КОТОП

Это слово - НЕ ПАЛИНДРОМ

Продолжить? Да - любой символ, Нет -'q' -> y

Введите слово:-> ПОТОП
Результат инверсии: ПОТОП

Это слово - ПАЛИНДРОМ

Продолжить? Да - любой символ, Нет -'q' -> q
```

Здесь цикл *do…while* используется лишь для организации интерфейса многократной проверки вводимых слов на принадлежность их к палиндромам.

- **Проверить является ли фраза палиндромом. Например, «А роза упала на лапу Азора».**

Алгоритм решения этой задачи может быть таким.

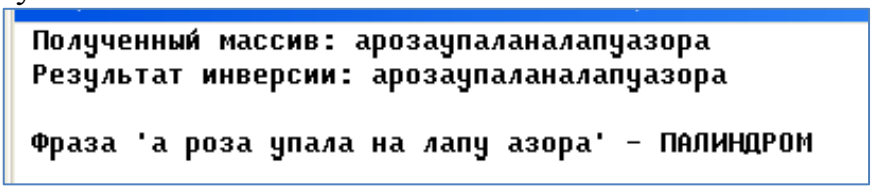
1) В предложении последовательно выделяются все лексемы. Тем самым избавляемся от пробелов или других знаков препинания.
2) На каждом шаге выделения лексем каждая последующая лексема сцепляется с предыдущей лексемой. Тем самым получаем одно слово.
3) Инвертируем полученное слово.
4) Сравниваем исходное слово и слово, полученное в результате инверсии, и делаем вывод, является ли исходная фраза палиндромом.

Код программы может быть таковым.

```
  #include <iostream.h>
  #include <string.h>
  void main( )
  {
   char s1[]="а роза упала на лапу азора",m[1000],c;
   char *s2=" ",*s3, s4[1000],k[100]; // в качестве разделителя выбран
пробел
   int i,N;
  strcpy(k,s1);//копия исходной фразы
```

```
        s3=strtok(s1,s2);   //выделение первой лексемы
        strcpy(s4,s3);      //сохранение ее в массиве, отвечающем за сцепленный массив
        do
         {
          s3=strtok(NULL,s2); //выделение очередной лексемы
          strcat(s4,s3);  //сцепка очередной лексемы с предыдущей
         }
        while(s3!=NULL);
   cout<<" Полученный массив"<<s4<<endl;       //исходное предложение
                                                после удаления пробелов
        strcpy(m,s4);       //сохранение исходного (без пробелов) массива в массиве m
        N=strlen(s4);      //определение его длины
        for(i=0;i<N/2;i++) //блок инвертирования
         {
          c=*(s4+i);
          *(s4+i)=*(s4+(N-i-1));
          *(s4+(N-i-1))=c;
         }
        cout<<" Результат инверсии: "<<s4;
        if(strcmp(m,s4)==0) //сравнение исходной фразы с инвертированной
         cout<<"\n\n Фраза'"<<k<<"' - ПАЛИНДРОМ";
        else
         cout<<"\n\ Фраза '"<<k<<"' - НЕ ПАЛИНДРОМ";
      }
```

Результат:



Большинство прототипов строковых функций (если не оговорено особо) находится в заголовочном файле ***string.h*** [http://it.kgsu.ru/C++/oglav.html].

Таблица 1. Основные функции работы со строками

Функция	Назначение
int getchar( )	Возвращает значение символа(если он есть), который пользователь набрал на клавиатуре. После ввода символа нужно нажать клавишу **Enter**. Заголовочный файл - **stdio.h**

int getch( )	Аналогично предыдущему, только символ на экране не отображается. Используется чаще для организации задержки выполнения программы. Заголовочный файл - **conio.h**
int putchar(int c)	Выводит символ **c** на экран. В случае успеха возвращает сам символ **c**, в противном случе - **EOF**. Заголовочный файл - **stdio.h**
char *gets(char *s)	Читает символы, включая пробелы и табуляции, до тех пор, пока не встретится символ новой строки, который заменяется нулевым символом. Последовательность прочитанных символов запоминается в области памяти, адресуемой аргументом **s**. В случае успеха возвращает аргумент **s**, в случае ошибки - нуль. Заголовочный файл -**stdio.h**
int puts(const char *s)	Выводит строку, заданную аргументом **const char *s**. Заголовочный файл - **stdio.h**
char *strcat(char *dest, const char *scr)	Объединяет исходную строку **scr** и результирующую строку **dest**, присоединяя первую к последней. Возвращает**dest**.
char *strncat(char *dest, const char *scr, int maxlen)	Объединяет **maxlen** символов исходной строки **scr** и результирующую строку **dest**, присоединяя часть первой к последней. Возвращает **dest.**
char *strchr(const char *s, int c)	Ищет в строке **s** первое вхождение символа **c**, начиная с начала строки. В случае успеха возвращает указатель на найденный символ, иначе возвращает нуль.
char *strrchr(const char *s, int c)	Аналогично предыдущему, только поиск осуществляется с конца строки.
int strcmp(const char *s1, const char *s2)	Сравнивает две строки. Возвращает отрицательное значение, если **s1<s2**; нуль, если **s1==s2**; положительное значение, если **s1>s2**. Параметры - указатели на сравниваемые строки.
int stricmp(const char *s1, const char *s2)	Аналогично предыдущему, только сравнение осуществляется без учета регистра символов.
int strncmp(const char *s1, const char *s2, int maxlen)	Аналогично предыдущему, только сравниваются первые **maxlen** символов.
int strnicmp(const	Аналогично предыдущему, только сравниваются пер-

char *s1, const char *s2, int maxlen)	вые **maxlen** символов без учета регистра.
char *strcpy(char *dest, const char *src)	Копирует исходную строку **src** и завершающий ее нулевой символ в строку результата **dest**. Возвращает **dest.**
char *strncpy(char *dest, const char *src, int maxlen)	Аналогично предыдущему, только копируются первых **maxlen** символов.
int strcspn(const char *s1, const char *s2)	Возвращает длину максимальной начальной подстроки строки **s1**, не содержащей символов из второй строки **s2.**
char *strdup(const char *s)	Копирует строку во вновь выделенный блок памяти, ***самостоятельно*** выделяя из кучи необходимое для размещения копии количество байтов. Возвращает указатель на сдублированную строку. Удалить эту строку можно с помощью функции **free( )**, указав в качестве параметра указатель на эту строку. Если памяти недостаточно - возвращается нуль.
int strlen(const char *s)	Возвращает длину строки **s** - количество символов, предшествующих нулевому символу.
char *strlwr(char *s)	Преобразует все прописные (большие) буквы в строчные (малые) в строке **s.**
char *strupr(char *s)	Преобразует все строчные (малые) буквы в прописные (большие) в строке **s.**
char *strnset(char *s, int c, int n)	Заполняет строку **s** символами **c**. Параметр **n** задает количество размещаемых символов в строке.
char *strpbrk(const char *s1, const char *s2)	Ищет в строке **s1** первое вхождение любого символа из строки **s2**. Возвращает указатель на первый найденный символ или нуль - если символ не найден.
char *strrev(char *s)	Изменяет порядок следования символов в строке на обратный (кроме завершающего нулевого символа). Функция возвращает строку **s.**
char *strset(char *s, int c)	Заменяет все символы строки **s** заданным символом **c.**
int strspn(const char *s1, const char *s2)	Вычисляет длину максимальной начальной подстроки строки **s1**, содержащей только символы из строки **s2.**

char *strstr(const char *s1, const char *s2)	Ищет в строке **s1** строку **s2**. Возвращает адрес первого символа вхождения строки **s2**. Если строка отсутствует - возвращает нуль.
char *strtok(char *s1, const char *s2)	Делит исходную строку **s1** на лексемы (подстроки), разделенные одним или несколькими символами из строки **s2**.

В таблице 2 приведены некоторые функции, которые также могут использоваться при работе со строками.

Таблица 2. Функции взаимосвязи чисел и строк

Функция	Назначение
double atof(const char *s);	Преобразует строку **s** в число с плавающей точкой типа **double**. Заголовочный файл - **math.h**
int atoi(const char *s);	Преобразует строку **s** в число типа **int**. Возвращает значение или нуль, если строку преобразовать нельзя. Заголовочный файл - **stdlib.h**
long atol(const char *s);	Преобразует строку **s** в число типа **long**. Возвращает значение или нуль, если строку преобразовать нельзя. Заголовочный файл - **stdlib.h**
char *itoa(int value, char *s, int rad);	Преобразует значение целого типа **value** в строку **s**. Возвращает указатель на результирующую строку. Значение **rad** - основание системы счисления, используемое при преобразовании (от 2 до 36). Заголовочный файл - **stdlib.h**
char *ecvt(double value, int ndig, int *dec, int *sign);	Преобразует значение **value** типа **double** в завершающуюся нулем строку. Возвращает адрес статического буфера, который перезаписывается при каждом вызове этой функции. Чтобы сохранить результат, можно воспользоваться, например, функцией **strcpy( )**. Значение **ndig** - количество цифр результата. Значение **dec** - указатель на целое значение, где размещается позиция десятичной точки (***результирующая строка не содержит символа десятичной точки***). Отрицательное значение означает, что десятичная точка находится слева от первой цифры строки. Заголовочный файл -**stdlib.h**
char *gcvt(double value, int ndec, char *buf);	Преобразует значение **value** типа **double** в завершающуюся нулем строку, которая адресуется аргументом **buf**. Способ представления выбирается функцией (обычный или экспоненциальный). Значение **ndec** - ко-

	личество десятичных разрядов (не гарантируется, что будет именно столько разрядов в числе), не более 18. Заголовочный файл - **stdlib.h**

Более подробно с применением многих из приведенных в таблицах функций можно ознакомиться в электронном пособии «Язык программирования С++. Начала» по адресу http://it.kgsu.ru/C++/oglav.html.

2.11.3 Задачи для самопроверки по работе с символьными массивами

1.1 Контрольные вопросы

1) Какова особенность языка С/С++ при обработке строковых данных. Привести примеры задания строк.

ОТВЕТ:

В отличие от некоторых других языков программирования, в языке С/С++ нет специального типа данных, инициализирующего строки.

Работа с набором символов, образующим строку, происходит либо посредством инициализации одномерного массива символов, либо посредством инициализации указателя на символьный тип.

Например, абсолютно идентичны записи вида:

char *str[]=”string”;*
char *str[6]={“s”, “t”, “r”, “i”, “n”, “g”};*
char*str=”string”;*

1.2 Разработка и анализ задач в среде программирования

Задача 1. Связь массива указателей и массивов символов.

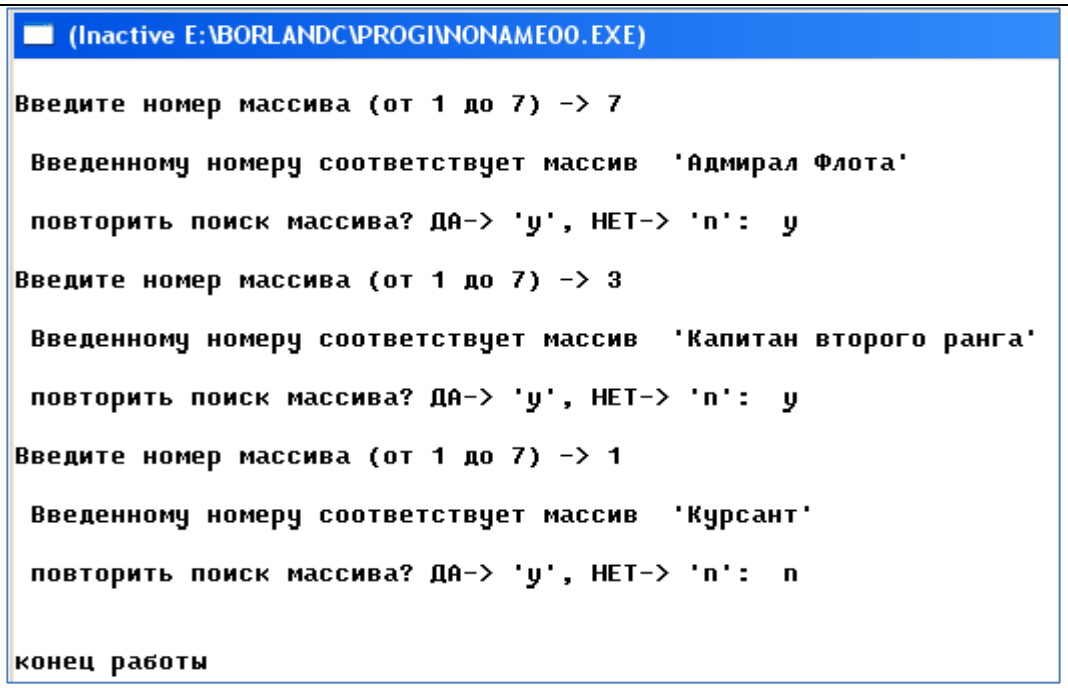
Разработать программу определения воинского звания по номеру массива, в котором это звание записано

Программа

```
#include<stdio.h>
#include<iostream.h>
void main(void)
{  int k; char s;
  char *day[]={"Курсант",
      "Капитан третьего ранга",
      "Капитан второго ранга",
      "Капитан первого ранга",
      "Контр-адмирал",
      "Вице-адмирал",
      "Адмирал Флота"};
do
{printf("\nВведите номер массива (от 1 до 7) -> ");
  scanf("%d",&k);   k=k-1;
  if(k<0||k>6)
```

```
        printf("\nНеверно задан номер, такого массива нет");
      else
        printf("\n Введенному номеру соответствует массив  '%s'",day[k]);
       printf ("\n\n повторить поиск массива? ДА-> 'y', НЕТ-> 'n':\t");
        cin>>s;
   }while(s=='y');
   printf("\n\nконец работы");
   }
}
```

Результат



Задача 2. Особенности связи массива указателей и массивов символов.

Разработать программу определения посимвольного вывода элементов массива указателей на символьные массивы

☝ В конце каждой «строки» стоит признак ее окончания ‘\0’ – символ NULL

☝ В памяти все символьные массивы расположены последовательно друг за другом!!!

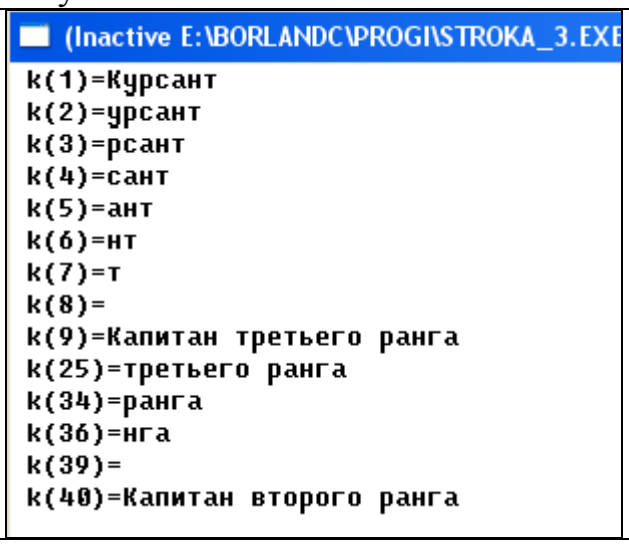
Программа

```
#include<stdio.h>
#include<iostream.h>
void main(void)
{  int k; char s;
  char *day[]={"Курсант",
```

```
        "Капитан третьего ранга",
        "Капитан второго ранга",
        "Капитан первого ранга",
        "Контр-адмирал",
        "Вице-адмирал",
        "Адмирал Флота"};

  // посимвольный вывод
  for(k=0;k<9;k++)
    printf(" k(%d)=%s\n",k+1, *day+k);     //k(8)=NULL
  printf(" k(%d)=%s\n",k+16,*day+16);
  printf(" k(%d)=%s\n",k+25,*day+25);
  printf(" k(%d)=%s\n",k+27,*day+27);
  printf(" k(%d)=%s\n",k+30,*day+30);     //k(39)=NULL
  printf(" k(%d)=%s\n",k+31,*day+31);
  }
```

Результат



2.Использование строковых функций.

2.1 Контрольные вопросы

1. В каком заголовочном файле расположены функции по обработке строк. Привести примеры строковых функций. Каковы их некоторые особенности.

ОТВЕТ:

Прототипы функций содержатся в заголовочном файле <***string.h***>. В качестве параметров этим функциям, как правило, передаются указатели на строки.

1) **Функция *strcat( )***

Функция *strcat* добавляет одну строку к другой.

2) **Функция *strcmp( )***

Функция *strcmp* сравнивает одну строку с другой.

Функция strcmp( ) выполняет без знаковое сравнение строк *s1* и *s2*, начиная с первого символа в каждой строке и продолжая сравнение последующих символов до тех пор, пока не встретятся не совпадающие символы или строки не кончатся.

Возвращаемые значения *strcmp* могут быть:

- ***< 0*** , если *s1* меньше *s2* (раньше по алфавиту);
- ***= 0***, если *s1* равно *s2* (все символы строго совпали);
- ***> 0***, если *s1* больше *s2* (по алфавиту позднее).

3) **Функция *strlen( )***

Функция *strlen* вычисляет длину строки.

4) **Функция *strcpy( ).***

Функция *strcpy* копирует содержимое одной строки в другую.

2. В чем отличительные особенности функций *gets( )* и *puts( )* от других функций и операторов ввода/вывода символьных массивов?

ОТВЕТ:

При вводе символов, состоящих из нескольких слов, разделенных друг от друга пробелами функции стандартного ввода/вывода *printf( ), scanf( )* и операторы *cin>>, cout<<* пробел воспринимают как нуль-терминатор и воспринимают отдельные слова, как самостоятельные массивы символов. Функции *gets( )* и *puts( )* различают пробел, считают его символом. Поэтому набор слов воспринимается ими как поток до символа нуль-терминатора в конце этого потока.

2.2 Разработка и анализ задач в среде программирования

Задача 3. Исследовать результат работы функций gets( ) и puts( ) и операторов «cin>>» и «cout<<»

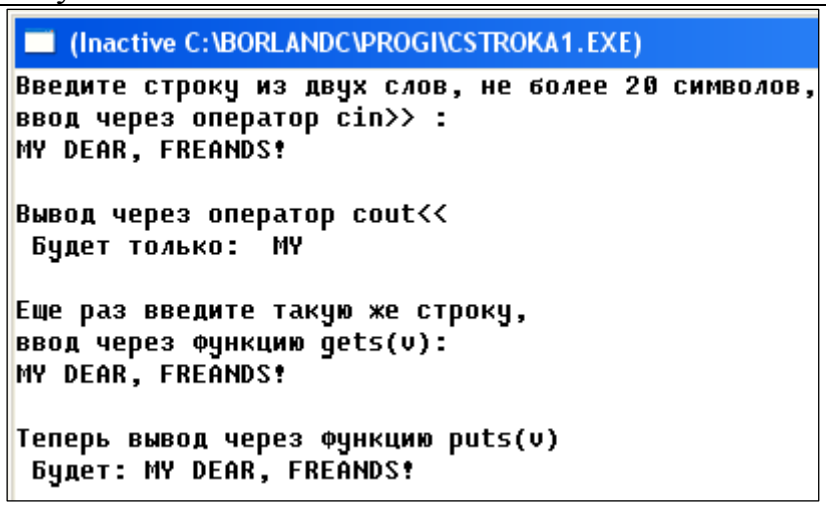
Программа

```
include <stdio.h>
#include <iostream.h>
void main( )
{
 char v[20];
  cout<<"Введите строку из двух слов, не более 20 символов,\n";
  cout<<"ввод через оператор cin>> :\n";
  cin>>v;
  printf("\nВывод через оператор cout<<\n Будет только: \t");
  cout<<v;

  cout<<"\n\nЕще раз введите такую же строку,\n";
  cout<<"ввод через функцию gets(v):\n";
  gets(v);
  printf("\nТеперь вывод через функцию puts(v)\n Будет:\t");
```

```
  puts(v);
}
```

Результат



Задача 4. Строковые функции.

Функция

Возвращаемые значения: ***strcmp*** *возвращает следующие значения:*

значения могут быть ***< 0*** *, если* ***s1*** *меньше* ***s2*** *(****раньше*** *по алфавиту)* ***;***

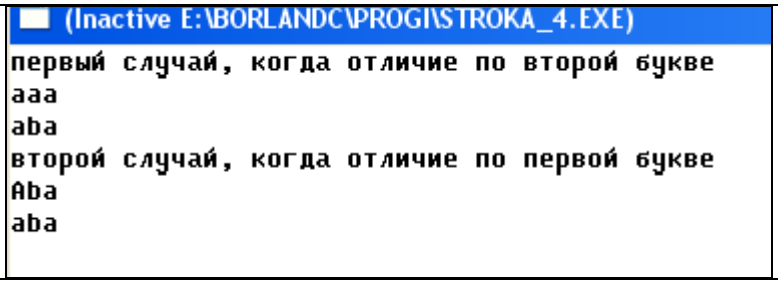
= 0, *если* ***s1*** *равно* ***s2*** *(все символы строго* ***совпали****);*

> 0, *если* ***s1*** *больше* ***s2*** *(по алфавиту* ***позднее****).*

Программа

```
#include <stdio.h>
#include <string.h>
void main (void)
{
char *buf1 = "aaa", *buf2 = "aba", *buf3 = "Aba"; int prt;
printf("первый случай, когда отличие по второй букве\n");
prt = strcmp (buf2,buf1);
if (prt>0) printf ("%s\n%s\n",buf1,buf2);
else printf ("%s\n%s\n",buf2,buf1);
printf("второй случай, когда отличие по первой букве\n");
prt = strcmp (buf2,buf3);
if (prt>0) printf ("%s\n%s\n",buf3,buf2);
else printf ("%s\n%s\n",buf2,buf3);
}
```

Результат


(Inactive E:\BORLANDC\PROGI\STROKA_4.EXE)
первый случай, когда отличие по второй букве
aaa
aba
второй случай, когда отличие по первой букве
Aba
aba

Задача 5. Строковые функции.

Функция

Функция Функция ***strlen*** *вычисляет длину строки*

Синтаксис: ***#include <string.h>***

size_t strlen (const char *s);

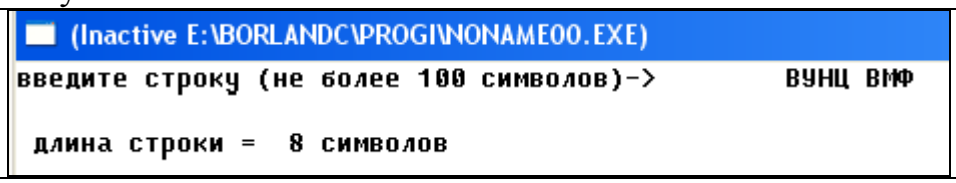
Описание: ***strlen*** *вычисляет длину строки в* ***s****.*

Возвращаемое значение: ***strlen*** *возвращает число символов в строке* ***s****, не считая нулевого окончания.*

Программа

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
void main (void)
{char s1[100]; int x;
cout<<"введите строку (не более 100 символов)->\t";
gets(s1);  x=strlen(s1);
cout<<"\n длина строки =  ";
printf("%d символов",x);
}
```

Результат



Задача 6. Строковые функции.

Функция

Функция Функция ***strcpy*** *копирует одну строку в другую*

Синтаксис: ***#include <string.h>***

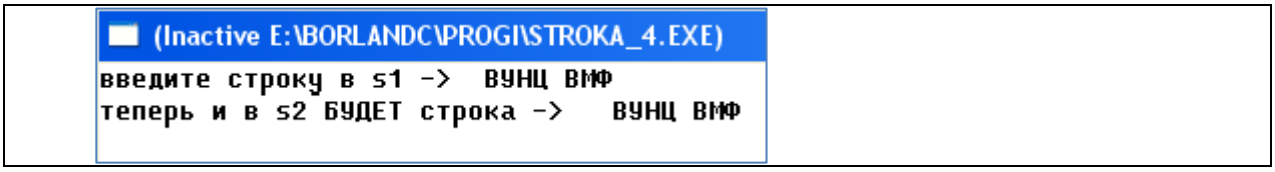
strcpy (char *s1; char *s2);

Описание: ***strcpy*** *копирует* ***s2 в s1****.*

Программа

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
void main (void)
{char *s1,*s2;
cout<<"введите строку ->\t";
gets(s1);  strcpy (s2,s1);
printf("теперь и в s2 содержится строка ‘%s ’",s2);
}
```

Результат



2.12 Структурированные типы данных

Данный раздел в основном посвящен важнейшему для современных систем программирования типу данных – структуре (*structure*). На понятии «*структура*» базируются все системы объектно-ориентированного программирования. Тип данных «*класс*», по сути, является усовершенствованным типом данных «*структура*». Различие между типом данных «*структура*» и типом данных «*класс*» в языке C++ минимально. Поэтому без четкого представления, что такое структура и как использовать этот тип данных при разработке программ, невозможно освоение методов объектно-ориентированного программирования.

Поэтому, рассмотрим более детально, что представляет собой тип данных «*структура*» в языке C/C++.

2.12.1 Структуры и объединения

Предположим, что нам нужно создать карточку сведений по результатам сдачи экзаменов во время сессии студентами учебной группы вида:

№ группы	№ п/п	Фамилия	Оценки			Средний балл
			Математический анализ	Линейная алгебра	История	

Здесь очевидно, что данные, сгруппированные в запись для одного студента, имеют различные характеристики по типу. Тип для полей *«№ группы», «№ п/п», «Оценки»* эти значения могут быть целыми числами; поле *«Фамилия»* однозначно имеет символьный тип; поле *«Средний балл»* может принимать действительные (вещественные) числовые значения.

В языках программирования такие данные называют данными структурированного (смешанного) типа, а для их описания используют специальные конструкции. В языке Си такими конструкциями являются *структуры* (*structure*) и *объединения* (*union*). Их последующее развитие привело к созданию *классов* – базовых понятий объектно-ориентированного программирования.

Итак, ***структурой*** называется группа данных различных типов и (или) предназначения, которые представляют собой единый информационный элемент.

Структуры помогают в организации сложных данных, поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое. Синтаксис объявления структурного типа с одновременным объявлением переменных объявляемого типа “структура” имеет вид:

struct *<**имя** структурного типа >*
{
*<**тип**Элемента1>< элемент1;>*
*<**тип**Элемента2 ><элемент2;>*
……
}*<**имя** переменной1>, <**имя** переменной2>,…;*

Вначале объявления структуры записывается ключевое слово ***struct***. Это ключевое слово в дальнейшем определяет **новый тип данных** переменных, агрегированных (сгруппированных) в этом типе. Затем идет имя структурного типа, произвольно задаваемое пользователем. Затем в фигурных скобках, перечисляются элементы структуры (переменные) с указанием их типов. Элементы структуры иногда называют ***полями или членами структуры.***

После перечисления членов структуры, заключенных в фигурные скобки, следуют имена объявляемых конкретных переменных структурного типа.

В описании структуры могут отсутствовать или имя структурного типа, или имена объектов (имена переменных). В том случае, если в объявлении структуры отсутствует *<имя структурного типа>*, то объявление представляет собой описание конкретного структурного объекта. Если отсутствуют имена объектов, то данное объявление является описанием структурного типа. Далее имя структурного типа может использоваться в теле программы для объявления объектов (переменных) данного структурного типа.

Итак, определение структуры в программе[6] на C/C++ может быть сделано по разному. Вернемся к нашему примеру и определим структуру, например, так:

struct ***{*** ***int*** *gr, nm, ma, la, hist;* ***char*** *fam[20];* ***float*** *sb;* ***};***	***struct student*** ***{*** ***int*** *gr, nm, ma, la, hist;* ***char*** *fam[20];* ***float*** *sb;* ***};***

Здесь во втором случае структуре присвоено *имя структурного типа* (***student***).

Структура, как и любая переменная, может быть объявлена как внутри

[6] Здесь и далее приводятся примеры программ для предполагаемой выше карточки сведений о сессии

функции, так и вне ее:

```
void main( )
{
struct
{
int gr, nm, ma, la, hist;
char fam[20];
float sb;
};
….
}
```

```
struct student
{
int gr, nm, ma, la, hist;
char fam[20];
float sb;
}
void main( )
{
….
}
```

Как было сказано выше, объявляя структуру, мы объявили (определили) **новый тип** для переменных, которые хотим использовать в программах. Осталось только их объявить.

Объявление переменных структурированного типа также может осуществляться по-разному. Это зависит от «вкуса» программиста и от оптимизации им кода своей программы. Рассмотрим несколько примеров.

```
void main( )
{
struct
{
int gr, nm, ma, la, hist;
char fam[20];
float sb;
}one, g[3];
….
}
```
①

```
struct student
{
int gr, nm, ma, la, hist;
char fam[20];
float sb;
}one, g[3];
void main( )
{
….
}
```
②

```
void main( )
{
struct
{
int gr, nm, ma, la, hist;
   char fam[20];
   float sb;
};
structone, g[3];
….
}
```
③

```
struct student
{
int gr, nm, ma, la, hist;
   char fam[20];
   float sb;
}
void main( )
{
struct student one, g[3];
// или
student one, g[3];
….
}
```
④

В первом и во втором примерах переменные *one* (если запись такого вида только одна) и *g[3]* (массив из трех записей) объявлены вместе с объявлением структуры; в третьем и четвертом примерах они объявляются как имеющие

структурированный тип. При этом (пример 4), если структурированный тип имеет имя (*student*), то и тип переменных можно задавать через имя этого структурированного типа.

Объединение (ключевое слово ***union***) представляет собой частный случай структуры. Формат описания объединения такой же, как и у структуры, за исключением того, что вместо ключевого слова *struct* используется ключевое слово *union*. Так же, как и при описании структур, нужно понимать различие между именем типа объединения и объектом объединения. Имя типа объединения является именем типа, создаваемого пользователем. Объявленный тип можно далее использовать для объявления переменных типа “объединение”. Под “объектом объединения” понимается конкретная переменная типа “объединение”, размещаемая в оперативной памяти.

Отличие объекта-объединения от объекта-структуры заключается в том, что все члены структурного объекта одновременно находятся в оперативной памяти, а под объект объединения выделяется участок оперативной памяти, в котором в каждый момент времени может находиться только один из членов объекта объединения. Причем размер этого участка памяти равен размеру максимального из элементов объекта объединения.

Пример объявления типа “объединение *uAlfa*”:

```
union uAlfa
{
 int Pi;
 double Ro;
 char Fi;
};
```

После такого объявления имя **типа** *uAlfa* можно использовать для объявления объектов типа “*union uAlfa*” в программах:

```
uAlfa Beta,Gamma;        //Объявление объектов Beta, Gamma
                         //типа “union uAlfa”
```

И инициализация объектов объединения, и доступ к ним аналогичны инициализации и доступу к объектам структуры.

Как было сказано выше, под объект объединения в оперативной памяти отводится участок размером, равным максимальному размеру, отводимому под тип для какого-либо из его членов. При этом значение, хранимое в этом участке памяти, интерпретируется по-разному, в зависимости от того, какой из членов описанного объединения используется для доступа к хранимому значению.

2.12.2 Действия над структурными объектами

- Над структурными объектами можно выполнять следующие операции:
- взятие адреса объекта **&**;

- объявление указателя на объект;
- выделение блока памяти под структурный объект оператором *new* (только в языке С++);
- объявление ссылки на структурный объект (только в языке С++);
- объявление массива структурных объектов;
- передача объекта функции в качестве аргумента;
- возврат из функции в качестве результата работы;
- присваивание значения одного объекта другому (в этом случае структурные объекты должны быть одного типа);

Приведем пример присваивания значения одного объекта другому:

```
struct sAlfa {              //Объявление структурного типа sAlfa
      int Ro;
      double Pi;
      char Fi;};
sAlfa Beta={10, 12.5, «Y»};   //Объявление объекта Beta
                              //с инициализацией его членов
      sAlfa Gamma;            //Объявление объекта Gamma типа sAlfa
      Gamma=Beta;             //Операция присваивания значения
                              //объекта Beta объекту Gamma
      cout<<Gamma.Pi;         //На экран выводится число 10
      cout<<Gamma.Ro;         //На экран выводится число 12.5
      cout<<Gamma.Fi;         //На экран выводится символ «Y»
```

Так же, как и на обычную переменную, на объект можно объявить ссылку.

При этом ссылка является вторым именем объекта, его псевдонимом. Так, например, для вышеописанного объекта *Beta* можно объявить ссылку следующим образом:

```
sAlfa &rDzeta=Beta;     //Объявление ссылки rDzeta на объект  Beta
```

После такого объявления ссылку *rDzeta* можно использовать в любых операторах вместо имени объекта *Beta*, например:

```
cout<<rDzeta.Pi;        //На экран выводятся значения переменных -
cout<<rDzeta.Ro;        //членов объекта Beta - 10, 12.5, «Y»
cout<<rDzeta.Fi;
```

Ссылку на объект можно использовать для передачи этого объекта функции в качестве параметра. Этим достигается возможность изменения значений членов объекта, объявленного вне функции операторами функции.

Более подробно о передаче параметров в функциях ниже. Здесь отметим, что при передаче функции переменной (в том числе и объекта) по значению, функция может считывать значение переменной-параметра, но изменить значе-

ние переменной, объявленной вне функции, находясь внутри функции в этом случае невозможно. Использование параметра-ссылки снимает эту проблему.

2.12.3 Массивы структур

Допускается создавать массивы структурных объектов. Массивы структурных объектов могут быть полезны в том случае, когда необходимо сохранить информацию о множестве реальных однотипных объектов. Так, например, двумерный массив объектов можно использовать для сохранения параметров набора пикселей (точек экрана компьютера). Сохраненную таким образом информацию можно использовать для многократного вывода сохраненного изображения на экран. Как известно, каждый пиксел экрана компьютера характеризуется в основном тремя параметрами – координатой *X*, координатой *Y* и отображаемым цветом, данные о котором представляют собой целочисленное значение. Исходя из этого, можно создать следующий структурный тип для хранения параметров одного пикселя:

```
struct sPixel{              //Объявление структурного типа для
    int X;          //хранения параметров пикселя
    int Y;
    int Color;};
```

Далее можно создать двумерный массив объектов типа *sPixel* следующим образом:

```
sPixel Picture[100][100];//Объявление двумерного массива объектов
                         //типа sPixel
```

Доступ к элементам массива объектов производится также, как и к элементам обычного массива, т.е. с помощью операции индексации:

```
Picture[20][10].Color=255;    //Элементу [20][10] матрицы пикселей
                              // задано конкретное значение цвета
```

Инициализировать массив объектов можно, так же как и обычный массив, перечислением значений в фигурных скобках. При этом, если при инициализации массива задаются значения всех элементов массива структур, то достаточно одной пары фигурных скобок.

2.12.4 Правила доступа к структурированным переменным

Теперь все готово для работы с переменными структурированного типа. В предыдущем пункте косвенно мы осуществляли доступ к структурным переменным. Более детально правила доступа к структурным переменным (полям внутри структуры), рассмотрим на примерах листингов программ.

Листинг 1. **Пример ввода-вывода одной записи.**

```
#include<stdio.h>
#include<iostream.h>
struct student
{
```

```
int gr, nm, ma, la, hist;
char fam[20];
float sb;
} one;    //объявлена одна переменная типа структуры student

void main( )
{
      printf ("Ведите поля записи, с вычислением среднего балла:\n");
    cout<<" Номер группы ->          ";   cin>>one.gr;
    cout<<" Номер п/п -> ";               cin>>one.nm;
    cout<<" Фамилия -> ";                 cin>>one.fam;
    cout<<" Оценка по мат.анализу -> "; cin>>one.ma;
    cout<<" Оценка по лин.алгебре -> "; cin>>one.la;
    cout<<" Оценка по истории ->     "; cin>>one.hist;
     one.sb= (one.ma+ one.la+ one.hist)/3.0;
     printf ("\nВывод результатов сессии:\n");
     printf (" №гр\t№п/п\tФамилия\tМА\tЛА\tИст\tСр.балл\n");
     printf (" %d\t%d\t%s\t%d\t%d\t%d\t%f \n",
         one.gr,one.nm,one.fam,one.ma,one.la,one.hist,one.sb);
}
```

Результат работы программы:

```
Ведите пол  записи, с вычислением среднего балла:
 Номер группы ->          123
 Номер п/п ->             1
 Фамили  ->               Иванов
 Оценка по мат.анализу -> 3
 Оценка по лин.алгебре -> 3
 Оценка по истории     -> 5

Вывод результатов сессии:
 №гр    №п/п    Фамили  МА      ЛА      Ист     Ср.балл
 123    1       Иванов  3       3       5       3.666667
```

Из рассмотренного примера листинга программы следует то, что для доступа к структурной переменной (например, ***fam*** – члену структуры *student*) необходимо сначала указать имя переменной структурированного типа (***one***), затем поставить знак точки «**.**» и, затем, указать имя структурной переменной (***fam***): ***one.fam***.

Напомним еще одну особенность, связанную с преобразованием типа данных. При вычислении среднего балла *one.sb= (one.ma+ one.la+ one.hist)/3.0* в знаменателе вместо целого числа 3 стоит действительное число 3.0. Напомним, что переменная левой части этого выражения имеет тип *float*. Следовательно, и правая часть должна быть преобразована к этому типу. Если бы в знаменателе было указано целое число, то выражение дроби (целое/целое) давало бы целый результат от деления, отбрасывая остаток. Это бы привело к не-

верному результату. Поэтому либо числитель (он всегда целый), либо знаменатель, либо и то, и другое следует преобразовать в вещественный тип. В данной программе это проще сделать со знаменателем.

Листинг 2. **Пример ввода-вывода нескольких записей.**

```
#include<stdio.h>
#include<iostream.h>
#define N 2 //директива замещения – играет роль определения количества
                записей
struct student
{
 int gr, nm, ma, la, hist;
char fam[20];
float sb;
} g[N]; // определяет количество (массив) записей типа структуры student

void main( )
{
int i;
for(i=1;i<=N;i++)   // блок ввода записей
{
      printf ("Ведите поля записи, с вычислением среднего балла %d-го студента:\n",i);
      cout<<" Номер группы ->          "; cin>>g[i].gr;
      cout<<" Номер п/п ->             "; cin>>g[i].nm;
      cout<<" Фамилия -> ";              cin>>g[i].fam;
      cout<<" Оценка по мат.анализу -> ";      cin>>g[i].ma;
      cout<<" Оценка по лин.алгебре -> "; cin>>g[i].la;
      cout<<" Оценка по истории ->     "; cin>>g[i].hist;
      g[i].sb= (g[i].ma+ g[i].la+ g[i].hist)/3.0;
}
      printf ("\nВывод результатов сессии:\n\n");
      printf (" №гр\t№п/п\tФамилия\tМА\tЛА\tИст\tСр.балл\n\n");
for(i=1;i<=N;i++) // блок вывода результатов
{
      printf (" %d\t%d\t%s\t%d\t%d\t%d\t%f \n",
          g[i].gr,g[i].nm,g[i].fam,g[i].ma,g[i].la,g[i].hist,g[i].sb);
}
}
```

Результат работы программы:

я

я

```
Ведите пол  записи, с вычислением среднего балла 1 студента:
 Номер группы ->          121
 Номер п/п ->             1
 Фамили  ->               Иванов
 Оценка по мат.анализу -> 3
 Оценка по лин.алгебре -> 3
 Оценка по истории     -> 5

Ведите пол  записи, с вычислением среднего балла 2 студента:
 Номер группы ->          121
 Номер п/п ->             2
 Фамили  ->               Петров
 Оценка по мат.анализу -> 2
 Оценка по лин.алгебре -> 3
 Оценка по истории     -> 5

Вывод результатов сессии:

 №гр    №п/п    Фамили  МА      ЛА      Ист     Ср.балл

 121    1       Иванов  3       3       5       3.666667
 121    2       Петров  2       3       5       3.333333
```

2.12.5 Примеры использования структур в программах

Рассмотрим несколько полезных примеров по работе со структурированными типами данных.

Предположим, что мы получили структуру из *N* (например, пяти) записей вида:

```
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
121     1       Иванов  3       3       5       3.666667
121     2       Петров  2       3       5       3.333333
122     1       Сидоров 4       4       3       3.666667
122     2       Годин   5       5       3       4.333333
123     1       Антонов 5       2       2       3.000000
```

Рассмотрим следующие **задачи.**

Пример 1. *Вывести на экран результаты сессии конкретного студента.*

Алгоритм решения прост. Достаточно в блоке вывода произвести сравнение всех записей по полю *fam[20]* с вводимым массивом символов такого же размера, например *fio[20]*. Для поиска такого элемента используем функцию *strcmp( )*(см. п.2.11.2).

К исходному коду программы (листинг 2) добавим такой блок:

```
…
char fio[20];
cout<<"\nВведите фамитлию ->: ";cin>>fio;
printf("\n№гр\t№п/п\tФамилия\tМА\tЛА\tИст\tСр.балл\n");
```

```
        printf("-----------------------------------------------------\n");
        for(i=0;i<N;i++)
       { if(strcmp(fio,gr[i].fam)==0)
printf("\n%d\t%d\t%s\t%d\t%d\t%d\t%f\n",gr[i].ng,gr[i].np,gr[i].fam,gr[i].m
                                        a,gr[i].la,gr[i].ist,gr[i].sb);
```

...

Результат:

```
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-----------------------------------------------------------
121     1       Иванов  3       3       5       3.666667
121     2       Петров  2       3       5       3.333333
122     1       Сидоров 4       4       3       3.666667
122     2       Годин   5       5       3       4.333333
123     1       Антонов 5       2       2       3.000000

Введите фамилию ->: Антонов

№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-----------------------------------------------------------
123     1       Антонов 5       2       2       3.000000
```

Пример 2. *Вывести на экран всех тех студентов, с указанием номера их группы, у которых есть хотя бы одна двойка по какому-либо предмету. Наименование предмета – также вывести на экран.*

Особенность алгоритма решения этой задачи в следующем: в каждой записи нужно проверить каждый предмет по отдельности на наличие в нем двойки, продолжая проверку, даже если двойка по одному из предметов уже найдена.

Для этого достаточно к исходному коду программы добавить, например, такой блок проверки.

```
…
for(i=0;i<N;i++)
 {
  if(gr[i].ma==2)
   {
    printf("%s\t%d\t%d\tМатан.\n",gr[i].fam,gr[i].ng,gr[i].ma);
   }
  if(gr[i].la==2)
   {
    printf("%s\t%d\t%d\tЛ.алг.\n",gr[i].fam,gr[i].ng,gr[i].la);
   }
  if(gr[i].ist==2)
   {
    printf("%s\t%d\t%d\tИстор.\n",gr[i].fam,gr[i].ng,gr[i].ist);
   }
```

…

Результат:

```
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
121     1       Иванов  3       3       5       3.666667
121     2       Петров  2       3       5       3.333333
122     1       Сидоров 4       4       3       3.666667
122     2       Годин   5       5       3       4.333333
123     1       Антонов 5       2       2       3.000000

Фамилии №гр     оценка  предмет
-------------------------------
Петров  121     2       Матан.
Антонов 123     2       Л.алг.
Антонов 123     2       Истор.
```

Пример 3. *Вывести на экран всех тех студентов, с указанием номера их группы, у которых средний балл, например, ≤ 3,5 (или ≥ 3,5). Средний балл и условие поиска (≤ , ≥) вводятся с клавиатуры.*

Особенность алгоритма в следующем. Кроме проверки на каждом шаге числового значения среднего балла, нужно еще проверять и символьное значение знака – условия проверки. Поэтому, учитывая, что знак условия – это последовательность не более чем двух символов, нужно ввести в программу определение переменной символьного массива. И, далее, используя функцию сравнения строк, последовательно проверить возможные варианты ввода символов условного знака проверки, а уж затем, в каждом варианте – осуществить проверку всех записей в структуре на величину указанного среднего балла.

Решение задачи можно достичь, применив, например, такой вариант блока проверки.

```
…
char z[2]; float b;
cout<<"\nВведите условие проверки(знак) и балл -> : ";cin>>z>>b;
printf("\n№гр\tФамилии\tСр.балл\n");
printf("----------------------\n");

if(strcmp(z,"<=")==0)
 {for(i=0;i<N;i++)
   if(gr[i].sb<=b)
     printf("%d\t%s\t%3.1f\n",gr[i].ng,gr[i].fam,gr[i].sb);
 }
if(strcmp(z,">=")==0)
 {for(i=0;i<N;i++)
   if(gr[i].sb>=b)
     printf("%d\t%s\t%3.1f\n",gr[i].ng,gr[i].fam,gr[i].sb);
  }
```

else
cout<<"Таких записей в списке нет!";
…
Результат:

```
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
121     1       Иванов  3       3       5       3.666667
121     2       Петров  2       3       5       3.333333
122     1       Сидоров 4       4       3       3.666667
122     2       Годин   5       5       3       4.333333
123     1       Антонов 5       2       2       3.000000

Введите условие проверки(знак) и балл ->: >= 3.5

№гр     Фамилии Ср.балл
----------------------
121     Иванов  3.7
122     Сидоров 3.7
122     Годин   4.3
```

Пример 4. *Отсортировать всех студентов (общий список) по убыванию среднего балла.*

Предлагаемый алгоритм не является самым эффективным среди алгоритмов поиска, но он нагляден и удобен в программной реализации.

Пусть задан массив из пяти чисел в следующем порядке *5, 4, 7, 1, 6*. Тогда, чтобы отсортировать эти числа по убыванию, необходимо выполнить следующие шаги.

	0	**1**	**2**	**3**	**4**	Индексы символьного массива
Шаг **0**	5	4	7	1	6	Сравнение первого и последующих элементов. *Первый и максимальный из элементов меняются местами.*
Шаг **1**	7	4	5	1	6	Первый элемент найден. Сравнение второго и последующих элементов. *Второй и максимальный из оставшихся элементов меняются местами.*
Шаг **2**	7	6	5	1	4	Второй элемент найден. Сравнение третьего и последующих элементов. *Третий и максимальный из оставшихся элементов меняются местами.*
Шаг **3**	7	6	5	1	4	Третий элемент найден. Сравнение четвертого и последующих элементов. *Четвертый и максимальный из оставшихся элементов меняются местами.*
Шаг **4**	7	6	5	4	1	Четвертый элемент найден. Сравнение пятого и последнего элементов. *Пятый и максимальный из оставшихся двух элементов ме-*

						няются (не меняются) местами.
Результат	7	6	5	4	1	Все элементы найдены.

Из данной схемы алгоритма следует, что для его программной реализации следует ввести еще один цикл – внешний, фиксирующий шаги проверки чисел во внутреннем цикле. Кроме того, для нашего примера потребуются также временные переменные для перестановки всех элементов структуры: переменная типа *int* для перестановки целых чисел; переменная типа *float* для перестановки действительного числа среднего балла и переменная типа *char* для перестановки массива символов фамилии.

Блок программы сортировки структуры по среднему баллу может, например, выглядеть так.

```
…
char smax[20]; float max; int j,nn;
 for(j=0;j<N;j++)
   for(i=0;i<N;i++)
    if(gr[i].sb<=gr[i+1].sb)
     {
      nn=gr[i].ng;
      gr[i].ng=gr[i+1].ng;
      gr[i+1].ng=nn;
         nn=gr[i].np;
         gr[i].np=gr[i+1].np;
         gr[i+1].np=nn;
      strcpy(smax,gr[i].fam);
      strcpy(gr[i].fam,gr[i+1].fam);
      strcpy(gr[i+1].fam,smax);
         nn=gr[i].ma;
         gr[i].ma=gr[i+1].ma;
         gr[i+1].ma=nn;
      nn=gr[i].la;
      gr[i].la=gr[i+1].la;
      gr[i+1].la=nn;
         nn=gr[i].ist;
         gr[i].ist=gr[i+1].ist;
         gr[i+1].ist=nn;
      max=gr[i].sb;
      gr[i].sb=gr[i+1].sb;
      gr[i+1].sb=max;
     }
cout<<”Результат сортировки по среднему баллу:”;
printf("\n№гр\№п/п\tФамилии\tМА\tЛА\tИст\tСр.балл\n");
printf("-----------------------------------------------------\n");
for(i=0;i<N;i++)
```

```
        printf("%d\t%d\t%s\t%d\t%d\t%d\t%f\n",gr[i].ng,gr[i].np,gr[i].fam,gr[i
 ].ma,
                                        gr[i].la,gr[i].ist,gr[i].sb);
    …
```

Здесь переменная *j* и цикл по *j* фиксируют шаги упорядочения элементов структуры по убыванию среднего балла; переменные *nn*, *max* и *smax[20]* (соответственно, целого, вещественного и символьного типов) – временные переменные, отвечающие за перестановку элементов соответствующих типов в структуре.

Результат:

```
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
121     1       Иванов  3       3       5       3.666667
121     2       Петров  2       3       5       3.333333
122     1       Сидоров 4       4       3       3.666667
122     2       Годин   5       5       3       4.333333
123     1       Антонов 5       2       2       3.000000

Результат сортировки по среднему баллу:
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
122     2       Годин   5       5       3       4.333333
121     1       Иванов  3       3       5       3.666667
122     1       Сидоров 4       4       3       3.666667
121     2       Петров  2       3       5       3.333333
123     1       Антонов 5       2       2       3.000000
```

Пример 5. *Отсортировать всех студентов одной группы по убыванию среднего балла. Номер группы вводится с клавиатуры*

Если решена задача предыдущего примера (отсортирован весь список студентов по среднему баллу), то предлагаемая задача решается очень просто. Следует только в блок вывода результата добавить условие проверки и новую переменную, отвечающую за сравнение текущего номера группы с запрашиваемым номером, например:

```
…
int nom;
cout<<"\nВведите номер группы -> "; cin>>nom;
cout<<"Результат сортировки по среднему баллу в одной группе:";
printf("\n№гр\№п/п\tФамилии\tМА\tЛА\tИст\tСр.балл\n");
printf("-------------------------------------------------------\n");
for(i=0;i<N;i++)
 if(gr[i].ng==nom);
  printf("%d\t%d\t%s\t%d\t%d\t%d\t%f\n",gr[i].ng,gr[i].np,gr[i].fam,gr[i].ma,
                                        gr[i].la, gr [i].ist,gr[i].sb);
…
```

Результат:

```
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
121     1       Иванов  3       3       5       3.666667
121     2       Петров  2       3       5       3.333333
122     1       Сидоров 4       4       3       3.666667
122     2       Годин   5       5       3       4.333333
123     1       Антонов 5       2       2       3.000000

Введите номер группы -> 122

Результат сортировки по среднему баллу в одной группе:
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
122     2       Годин   5       5       3       4.333333
122     1       Сидоров 4       4       3       3.666667
```

Однако, анализируя результат этого и предыдущего примеров, можно заметить один недостаток начального формирования данных. Если, списки студентов заведомо формировались неупорядоченно, например, по номеру в группе или по алфавиту, то и последующие результаты в программе также могут оказаться неупорядоченными, не смотря на то, что по какому-либо одному критерию мы их упорядочили. В примерах упорядочили по среднему баллу, но не упорядочили по номеру, как в списке в целом, так и в списке группы. В любом случае в таких задачах определяется первый (главный) критерий, с которого начинается упорядочивание. Затем уточняются дополнительные критерии. Рассмотрим еще несколько примеров упорядочивания.

Пример 6. *Отсортировать всех студентов (общий список) по убыванию среднего балла и упорядочить этот список по номеру.*

Простой пример. Если уже упорядочен список по убыванию среднего балла (пример 4), то достаточно для решения этой задачи в блоке вывода присвоить переменной, отвечающей за номер по порядку в группе (*gr[i].np*), номер индекса в отсортированном списке: *gr[i].np=i+1*. В программе это можно сделать, например, так.

```
…
        cout<<”Вывод нумерованного списка, отсортированного по среднему баллу:”;
        printf("\n№п/п\№гр\tФамилии\tМА\tЛА\tИст\tСр.балл\n");
        printf("-------------------------------------------------------\n");
        for(i=0;i<N;i++)
        printf("%d\t%d\t%s\t%d\t%d\t%d\t%f\n",
i+1,gr[i].ng,gr[i].fam,gr[i].ma,
                                        gr[i].la,gr[i].ist,gr[i].sb);
…
```

Результат:

```
№гр     №п/пп   Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
121     1       Иванов  3       3       5       3.666667
121     2       Петров  2       3       5       3.333333
122     1       Сидоров 4       4       3       3.666667
122     2       Годин   5       5       3       4.333333
123     1       Антонов 5       2       2       3.000000

Вывод нумерованного списка, отсортированного по среднему баллу:
№п/п    №гр     Фамилии МА      ЛА      Ист     Ср.балл
-------------------------------------------------------
1       122     Годин   5       5       3       4.333333
2       121     Иванов  3       3       5       3.666667
3       122     Сидоров 4       4       3       3.666667
4       121     Петров  2       3       5       3.333333
5       123     Антонов 5       2       2       3.000000
```

Задачи поиска и сортировки элементов массивов и структур одни из наиболее распространенных задач в программировании. Многие из них сложны и требуют специальной организации или применения более эффективных алгоритмов. Некоторые из этих способов и алгоритмов будут рассмотрены позже.

2.12.6 Задачи для самопроверки по структурированным типам данных

Контрольные вопросы

ВОПРОС 1. Дать определение понятия «структура» или «структурированные типы данных».

ОТВЕТ: *Структура* - это составной объект, в который входят элементы любых типов.

В отличие от массива, который является однородным объектом, структура может быть *неоднородной*, то есть состоять из *объектов данных разного типа*. Часто такие объединенные данные называют *записями*, а каждое из конкретных данных, входящих в структуру, называют *полями*.

ВОПРОС 2. В чем различия между структурами и массивами?

ОТВЕТ: отличия структуры от массива следующие:

1)элементы структуры могут быть не одного и того же типа;

2) структурная переменная представляет собой переменную нового типа, задаваемого самим программистом при описании структуры;

ВОПРОС 3. Как объявляется структура (синтаксис)?

ОТВЕТ: Объявление структуры осуществляется с помощью ключевого слова ***struct***, за которым следует *имя типа* структуры (тег структуры) и *список элементов* структуры, заключенных в фигурные скобки. В структуре обязательно должен быть указан хотя бы один компонент. Определение структур имеет следующий вид:

struct [имя_типа структуры]***{*** тип и имя элемента ***1;***
тип и имя элемента ***2;***
…… …***;***
тип и имя элемента ***n;***
}[имя структурной переменной]***;***

ВОПРОС 4. Приведите примеры 2-х способов объявления структур в программах. В чем их различия?

ОТВЕТ: Сама структура описывается двумя способами, например:

1. *struct*	**2. *struct* cursant**
{	**{**
char zv[10];	***char*** zv[10];
char fam[20];	***char*** fam[20];
int math,fiz,inf;	***int*** math,fiz,inf;
float sb;	***float*** sb;
};	**};**

Здесь в первом случае объявлена просто структура, состоящая из 6-ти полей, а во втором случае ей присвоено *имя типа структуры* (**cursant**).

ВОПРОС 5. Приведите примеры различных способов объявления структурных переменных в программах. В чем их различия?

ОТВЕТ: Синтаксис задания структурных переменных также может быть представлен несколькими способами, например:

1. *struct*	**2.** struct **cursant**
{	{
char zv[10];	***char*** zv[10];
char fam[20];	***char*** fam[20];
int math,fiz,inf;	***int*** math,fiz,inf;
float sb;	***float*** sb;
}one, gr[10];	} **one, gr[10];**
или непосредственно в программе определить эти переменные *one* и *gr[10]:*	**или**
void main(void)	void main(void)
{….	{….
struct one, gr[10];….	struct **cursant** one, gr[10];….
	}
	или
	void main(void)

}	{…. **cursant** one, gr[10];…. }

Здесь в обоих случаях задания структур определяются по две *структурные переменные*, первая из которых (*one*) представляет единственную запись, состоящую из 6-ти полей (примерно так, как было показано в таблице предыдущего примера), а вторая (*gr[10]*) – это массив из 10-ти различных записей с одинаковыми полями. Благодаря таким переменным (*gr[10]*) можно представлять базу данных в виде списков записей (например, классный журнал, оценочная ведомость, реестр жителей, кадровая карточка и т.п.).

Отметим, что в первом случае описания структуры не указано *имя ее типа*. Это означает, что задание структурных переменных может производиться только двумя способами: или сразу после закрывающейся фигурной скобки описания структуры, или в тексте программы с помощью конструкции ***struct*** *one, gr[10]*. Такая запись говорит, что переменные *one* и *gr[10]* имеют структурированный тип.

Во втором случае описания структуры ей присвоено *имя типа структуры* (***cursant***). Это позволяет задавать структурные переменные тремя способами. Первый способ аналогичен заданию в первом случае описания структуры. Второй способ формируется с помощью конструкции *struct* ***cursant*** *one, gr[10]* и говорит о том, что определяются две *структурные переменные* **типа *cursant*.** Третий способ задания с помощью конструкции ***cursant*** *one, gr[10]* сообщает о том, что определяются *две переменные* **типа *cursant***, который и является признаком заданной структуры.

Иначе говоря, *определив имя типа структуры (пользовательский тип) мы получили возможность присваивать этот тип переменным так же, как и в случаях задания для них предопределенных типов*, например, *int, float* или *char*.

ВОПРОС 6. Как осуществляется доступ к компонентам структуры? Приведите пример.

ОТВЕТ: Доступ к компонентам структуры осуществляется с помощью указания имени структуры и следующего через точку имени выделенного компонента, например, если мы объявили структуру вида:

```
:        struct dan {  char fio[15];
                       int year;
                       int gruppa;} kurs;
```

то тогда в программе мы можем задать значения полям этой структуры, например, так:

kurs.fio=’Иванов’;

kurs.year = 1985;

scanf(“%d”,&***kurs.gruppa***);

 *Далее вопросы занятия рекомендуется провести в форме самостоятельного решения примеров по составлению программ по обработке структурированных данных*

2.ИСПОЛЬЗОВАНИЕ СТРУКТУР В ПРОГРАММАХ

Задача 1 Требуется разработать программу, позволяющую обрабатывать данные по успеваемости для одного курсанта.

1) Введите программу, показанную ниже.

2) Попробуйте отредактировать ее вывод под свою фамилию

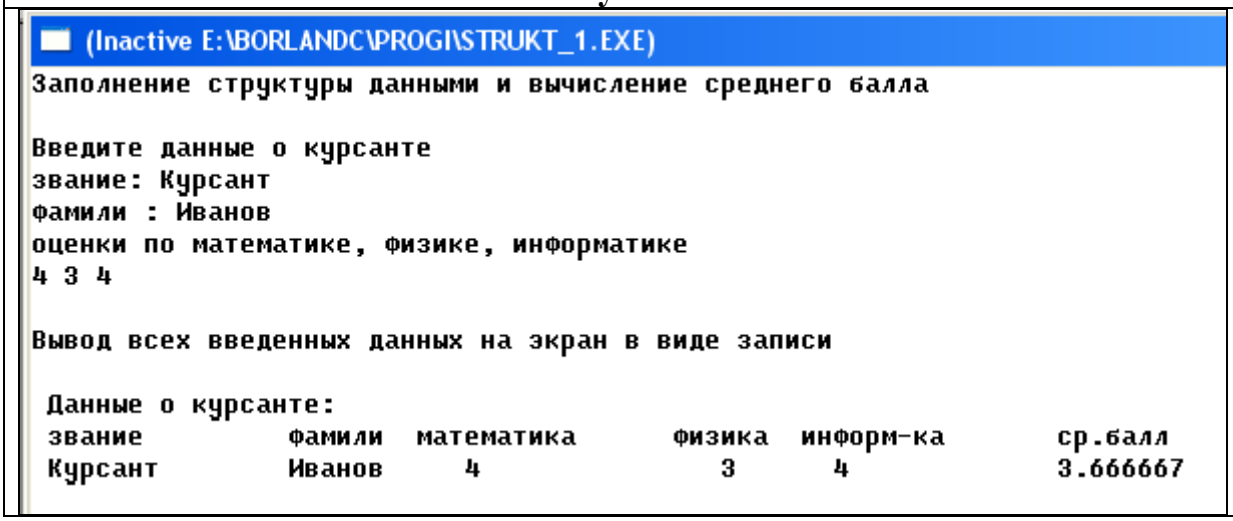
Программа

```
##include<stdio.h> //библиотека форматного ввода-вывода
struct cursant      //задание структуры и описание ее полей
{char zv[10];  //поле для задания звания - массив символов из 10 элементов
char fam[20]; //поле для задания фамилии - массив символов из 20 элементов
int math,fiz,inf; //три поля целого типа для задания оценок по 3-м предметам
float sb;     //поле вещественного типа для определения среднего балла
};
void main(void)
{   cursant k;
printf("Заполнение структуры данными и вычисление среднего балла\n");
printf("\nВведите данные о курсанте");
printf("\nзвание: ");scanf("%s",&k.zv);
printf("фамилия: "); scanf("%s",&k.fam);
printf("оценки по математике, физике, информатике\n");
scanf("%d%d%d", &k.math, &k.fiz, &k.inf);
k.sb=(k.math+k.fiz+k.inf)/3.0; //вычисление ср. балла

printf("\nВывод всех введенных данных на экран в виде записи\n");
printf("\n Данные о курсанте:\n",i+1);
printf(" звание\t\tфамилия\tматематика\tфизика\tинформ-ка\tср.балл\n");
printf(" %s",k.zv);
printf("\t%s",k.fam);
printf("\t   %d",k.math);
printf("\t\t   %d",k.fiz);
```

```
printf("\t  %d",k.inf);
printf("\t\t%f",k.sb);
}
```

Результат



Задача 2 Требуется разработать программу, позволяющую обрабатывать данные по успеваемости для трех курсантов, нахождении данных об успеваемости курсанта по запросу его фамилии.

1) Введите программу, показанную ниже.
2) Попробуйте отредактировать ее вывод под свою фамилию

Программа

```
#include<stdio.h>
#include<string.h>
struct cursant
{char zv[10];
char fam[20];
int math,fiz,inf;
float sb;
} gr[3];

void main(void)
{
int g,i;char r, k,*fio;
float ng;

printf("заполнение структуры данными и вычисление среднего балла\n");

for(i=0;i<=2;i++)
{printf("\nведите данные о %d-м курсанте",i+1);
printf("\nзвание: ");
scanf("%s",gr[i].zv);
printf("фамилия: ");
scanf("%s",gr[i].fam);
printf("оценки по математике, физике, информатике\n");
scanf("%d%d%d",&gr[i].math,&gr[i].fiz,&gr[i].inf);

gr[i].sb=(gr[i].math+gr[i].fiz+gr[i].inf)/3.0;
}

printf("\nвывод всех введенных данных на экран в виде записи\n");
printf("\n\t\tзвание  фамилия  математика физика  информ-ка  ср.балл");
```

```
for(i=0;i<=2;i++)
{printf("\n%d-й курсант\t",i+1);
printf("%s",gr[i].zv);
printf("  %s",gr[i].fam);
printf(" \t%d",gr[i].math);
printf("\t%d",gr[i].fiz);
printf("\t%d",gr[i].inf);
printf("\t%f",gr[i].sb);
}

printf("\n\nвывод данных об оценках по запрашиваемой фамилии - ");
printf("\nвведите фамилию курсанта\n");
scanf("%s",fio); //по адресу fio,  вводится фамилия запроса поиска
printf("   его оценки:\n");
printf("математика  физика  информ-ка   ср.балл");
for(i=0;i<=1;i++) // цикл поиска и вывода записей, удовлетворяющих условию
{if(strcmp(fio,gr[i].fam)==0) //сравнение на совпадение двух строк
{printf("\n");
printf("\t%d",gr[i].math);
printf("\t%d",gr[i].fiz);
printf("\t%d",gr[i].inf);
printf("\t%f",gr[i].sb);
}
}
}
```

Результат

```
(Inactive E:\BORLANDC\PROGI\STRUKT_2.EXE)
введите данные о 2-м курсанте
звание: курсант
фамили : Петров
оценки по математике, физике, информатике
3 3 3

введите данные о 3-м курсанте
звание: курсант
фамили : Сидоров
оценки по математике, физике, информатике
5 5 4

вывод всех введенных данных на экран в виде записи

                звание   фамили   математика физика  информ-ка  ср.балл
1-й курсант     курсант  Иванов         3       4       5       4.000000
2-й курсант     курсант  Петров         3       3       3       3.000000
3-й курсант     курсант  Сидоров        5       5       4       4.666667

вывод данных об оценках по запрашиваемой фамилии -
введите фамилию курсанта
Сидоров
   его оценки:
математика  физика  информ-ка   ср.балл
        5       5       4       4.666667
```

ЗАДАЧА 3. Вывести на экран сведения о тех обучаемых, средний бал которых по результатам экзаменационной сессии больше (меньше) 4.5 балла. В противном случае выдавать сообщение об отсутствии таких записей.

Ответ(*продолжение программы*)

```
printf("\n\nфамилии только тех, у кого средний балл > 4.5\n");
 for(i=0;i<=2;i++) //цикл поиска записей, удовлетворяющих условию запроса
 {if(gr[i].sb>4.5) //проверка условия поиска
   printf("это курсант %s\n",gr[i].fam);
    if(i==2)  printf("Данные исчерпаны\n");
 }
```

ЗАДАЧА 4. Выводить на экран полные сведения о тех обучаемых, которые по результатам экзаменационной сессии имеют хотя бы одну двойку по какому-либо предмету.

Ответ:

```
printf("\n\nсведения о тех, кто имеет двойки\n");
for(i=0;i<=2;i++) //цикл поиска записей, удовлетворяющих условию запроса
{if((gr[i].fiz==2)||(gr[i].math==2)||(gr[i].inf==2))   //проверка условия поиска
  {
    printf("%s",gr[i].zv);
    printf("\t  %s",gr[i].fam);
    printf("\t  %d",gr[i].math);
    printf("\t  %d",gr[i].fiz);
    printf("\t  %d",gr[i].inf);
    printf("\t  %f\n",gr[i].sb);
  }
 if(i==2)
 printf("Данные исчерпаны\n");
}
```

Результат работы фрагмента:

```
сведения о тех, кто имеет двойки
сержант   Петров          2       2       3       2.333333
курсант   Сидоров         5       5       2       4.000000
Данные исчерпаны
```

2.13 Файловые типы данных

2.13.1 Основные понятия файловых данных

Данные, с которыми работает программа, могут быть, в общем случае, двух видов. Это данные, которые получаются в результате вычислений и данные, которые поступают в программу извне, например, вводятся в программу с клавиатуры или других внешних носителей. Если объем данных небольшой и нет необходимости в их длительном хранении, то достаточным для программы может быть только их ввод с клавиатуры и вывод результатов обработки на экран. Однако, при обработке больших массивов информации этот способ становится не эффективным и затруднительным, и это заметно уже на этапе отладки программы. При рассмотрении примеров предыдущих глав мы неоднократно убеждались в этом на практике.

При использовании любого языка программирования большое значение имеет то, насколько в нем заложены возможности, благодаря которым пользователь может работать с данными, хранящимися на внешних – долговременных

носителях информации. Из курса информатики известно, что таким образом хранящиеся данные принято называть (*физическими*) *файлами*. Напомним, что существуют различные определения файлов, одним из которых может быть следующее.

Определение. *Файл* (*физический*) – это поименованная область памяти, предназначенная для хранения данных.

Под данными можно понимать не только те единицы информации, с которыми работает какая-либо программа, но и саму программу, как совокупность кодированных единиц информации. Поэтому компилятор любого языка программирования должен «уметь» распознавать тип обрабатываемых данных (по существу, тип физического файла: текст, таблица, графика и т.д.).

По умолчанию компилятор языка C/C++ всегда настроен для обработки данных в текстовом формате. Это означает, что он способен обрабатывать в режиме «*чтение/запись*» данные текстовых файлов. Обработка данных, как и все вычислительные процессы, происходит в оперативной памяти компьютера. Данные с физического файла переписываются из физического файла в некоторую область оперативной памяти (по некоторому свободному от других данных адресу), обрабатываются там, а затем возвращаются на отведенное им место в физическом файле.

Поэтому для обработки данных физического файла в языке Си предусмотрена специальная *конструкция*, позволяющая установить *связь* между программой и физическим файлом. Эту конструкцию называют *логическим файлом* и она, по существу, представляет новый тип данных. Поэтому возможно следующее определение логического файла

Определение. *Файл* (*логический*) – это представитель структурированных типов данных языка программирования, предназначенный для обработки данных физического файла в адресном пространстве оперативной памяти компьютера.

Таким образом, *за связь программы с данными*, хранящимися в физическом файле, отвечает *логический файл – специальная конструкция* языка программирования.

Часто логический файл в терминах языка программирования называют *потоком*. Различают понятия входного и выходного потоков.

Определение. Если данные ***считываются из*** физического файла ***в*** программу, то это – *входной поток*; если данные ***записываются из*** программы ***в*** физический файл, то это – *выходной поток*.

Работу с каким-либо физическим файлом можно осуществлять тогда, когда все подготовлено для этого. Это означает, что в программе должна иметься такая *конструкция* логического файла, которая позволяла бы, прежде всего, *открыть* данный физический файл. Как только файл открыт для доступа к его данным, так сразу компилятором выделяется *специальная область в оперативной памяти* для хранения его содержимого. Это сделано для ускорения процессов обработки файловых данных (чтение, запись, дополнение, удаление и т.д.). Эту область памяти принято называть *буфером файла*.

Определение. *Буфер файла* – это специальная область оперативной памяти, предназначенная для временного хранения данных, считанных с физического файла, или данных предназначенных для хранения в нем.

Из этого определения следует вывод о том, что при программировании задач, в которых происходит работа с файловыми данными, следует внимательно относиться к этой временной области обрабатываемых данных *в данный момент работы* программы. Иначе говоря, следует следить за тем, чтобы эта область оперативной памяти (буфер файла) была в нужное время открыта, а по окончании работы с ней – освобождена для других данных и программ (закрыта).

При создании какого-либо физического файла формируются два основных признака его конструкции: *указатель текущей позиции* файла (начиная с первой) и *признак его конца*.

Все данные в файлах расположены последовательно друг за другом. Каждому типу данных, хранящихся в файле, соответствует определенное количество байтов. Следовательно, для конкретного значения можно (по его типу) определить и его позицию размещения в файле. В этом и состоит суть указателя текущей позиции.

Практически во всех языках программирования признаком конца файла является обозначение *EOF* (*End Of File*). Благодаря этому признаку, точно так же, как и благодаря указателю текущей позиции, можно совершать отдельные операции над файловыми данными.

Сказанного выше достаточно, чтобы начать работу в СИ-программе с некоторым физическим файлом.

Основные действия для работы в СИ-программе с физическим файлом:

1. определить логический файл – конструкцию языка Си, указывающую на файл физический;
2. открыть физический файл – установить связь с созданным логическим файлом;
3. целесообразно проверить существует ли физический файл;
4. если существует, то приступить к работе с файловыми данными, в противном случае требуется создать физический файл;
5. по окончании работы следует закрыть файл, освободив буфер файла в оперативной памяти.

В зависимости от выбранного компилятора языка программирования (более ранних или более поздних версий языка C/C++) возможен различный набор функции по обработке файловых данных. Но базовыми функциями для всех компиляторов являются функции стандартного ввода/вывода данных. Поэтому, на начальном этапе изучения функций и правил при работе с файловыми данными будет достаточным изучение материала излагаемого ниже.

Определение логического файла делается с помощью такого оператора описания:

FILE *<имя указателя>,

где:

- *FILE* – по существу, *имя агрегированного типа данных*, благодаря которому создается *специальная структурная переменная* для хранения сведений об открываемом файле. Структура – этой переменной описана в заголовочном файле *stdio.h*;
- **<имя* указателя> – произвольный идентификатор, который в дальнейшем в программе будет использоваться в качестве имени *логического* файла. Например, если объявить конструкцию *FILE *fp*, то это означает, что объявлен логический файл *fp*, содержание которого пока не определено. Объявление логического файла в тексте программы, так же как и переменных любых типов, определяется областью его видимости и временем жизни. То есть, например, его можно объявить как глобальной переменной (до объявления функции), или локальной (внутри тела функции).

Связь между объявленным логическим файлом и файлом физическим устанавливается в теле функции и делается это с помощью функции открытия файла *fopen( )*.

Ее синтаксис имеет вид:

fopen(<"имя физического файла">,<"режим использования">),

где под *именем физического файла* понимается не только непосредственно имя, но и полный путь к тому месту, где этот файл храниться.

Например, запись «*fopen("C:\\Temp\\file1.txt",<"режим использования">)*» означает, что открываемый файл с именем *file1.txt* хранится на диске *C*, в каталоге *Windows*, в подкаталоге *Temp*.

Обратите внимание, что в указании пути вместо одного слеша используется два слеша «\\». Такой нотации требует синтаксис языка Си, так как мы знаем, что один *слеш* используется для указания компилятору правил использования форматной строки вывода: например, «\» – слияние разорванного текста программы, «\n»–перевод каретки, а «\t»–признак табуляции в строке.

Режим использования физического файла – это параметр, определяющий, как будут обрабатываться файловые данные.

Например, физический файл будет *открыт на чтение* из него данных, или, наоборот – *для записи в него* данных и т. д.

Этот режим может принимать следующие, наиболее часто употребляемые ***значения***:

- ✓ "***a***" – файл открывается для записи в конец файла (добавление), если физический файл не существует, то он вначале создается;
- ✓ "***r***" – файл открывается только для чтения из него данных. Физический файл при этом должен существовать, иначе работа программы завершится аварийно по признаку отсутсвия файла;
- ✓ "***w***" – файл открывается только для записи. При этом, если файл отсутствует, то он создается. Если такой файл уже создан, то он перезаписывается без со-

хранения в нем старых данных, так как запись осуществляется с начала физического файла.

- ✓ "***r+***" Открывает существующий файл для чтения и записи.
- ✓ "***w+***" Создает новый файл для чтения и записи. Перезаписывает любой существующий файл с тем же именем.
- ✓ "***a+***" Открывает файл в режиме чтения и записи для добавления новой информации в конец файла. Если файл не существует, он создается, и любой существующий файл с тем же именем перезаписывается.

Таким образом, например, такая запись внутри тела функции

fp=fopen("C:\\Windows\\Temp\\file1.txt","r")

означает, что открываемый файл с именем *file1.txt* хранится на диске *C*, в каталоге *Windows*, в подкаталоге *Temp*, открывается на чтение из него данных, а в программе на языке Си работа с этими данными будет происходить через логический файл *fp*. Таким образом, ***мы связали физический и логический файлы***: логический файл *fp* приобрел значение библиотечной функции языка Си *fopen( ).*

Именно в том случае, когда при открытии используется режим типа чтения (*"r"*), следует делать проверку наличия физического файла, чтобы не произошло аварийного выхода из программы. Делается это с помощью проверки результата работы функции *fopen( )*, например, так:

```
if(fp==NULL)
   printf("файл не существует");
else {/*работа с физическим файлом через его указатель fp */}.
```

В этом примере константа логического нуля *NULL* означает, что результат работы функции *fopen( )* (значение которой было присвоено ранее объявленному логическому файлу *fp*) пустой, то есть такой файл не существует.

И, наконец, как только обмен данными завершится, следует с помощью функции *fclose(<имя логического файла>)* освободить буфер файла.

Таким образом, примерный «**скелет**» программы по работе с фалами следующий:

```
#include<stdio.h>
FILE *fp;
void main(void)
{ ....
   fp=fopen("C:\\Temp\\file1.txt","r");
if(fp==NULL)
  printf("файл не существует");
else {/*работа с физическим файлом через его указатель*/};
....;
fclose(fp);
....;
}
```

2.13.2 Использование библиотечных функций по работе с файлами

Как было отмечено выше, функции по работе с файлами расположены в заголовочном файле C/C++ <*stdio.h*>. Ниже приведена справочная таблица по этим функциям. Дополнительные сведения о них можно узнать, воспользовавшись литературой или справочной системой интегрированной среды программирования. Более детально рассмотрим только некоторые функции, которые будут использоваться нами в примерах программ (в предположении, что *логический* файл *fp* создан).

Кроме функций ***fopen( )*** и ***fclose( )*** часто используются функции:

- ***fscanf(****fp,"форматная строка",arg1,arg2,...****)*** – вводит данные из открытого в режиме чтения файла *fp* в переменные *arg1*, *arg2*,…, в соответствии с форматной строкой.

Например, запись

fscanf(fp,"%s%d",g[i].fam,&g[i].ma);

означает, что из открытого в режиме чтения файла, на который указывает логический файл *fp*, в структурную переменную *g[i].fam* заносится первое данное, хранящееся в файле в виде последовательности символов, а в структурную переменную *g[i].ma* по ее адресу в памяти (&) заносится второе данное, хранящееся в файле в виде целого числа;

- ***fprintf(****fp,"форматная строка",arg1,arg2,...****)*** – пересылает данные в открытый в режиме записи файл *fp* значения переменных *arg1*, *arg2*,…, в соответствии с форматной строкой.

Например, запись

fprintf(fp,"%s\t%d", g[i].fam,g[i].ma);

означает, что в открытом в врежиме записи файле, на который указывает логический файл *fp*, будут размещены последовательно, начиная с первой его позиции, значение структурной переменной *g[i].fam* (последовательность символов) и значение структурной переменной *g[i].ma* (целое число), и эти данные разделены друг от друга признаком табуляции (\t).

- ***fgets(****m,c_max,fp****)*** – последовательное чтение символов из файла, на который указывает логический файл *fp* до начала новой строки (признак ‘\0’), или заданное количество символов, уменьшенное на 1. Прочитанные символы помещаются в буфер, на который указывает *m*, начиная с ячейки, адресуемой первым параметром.
- ***fputs(****s,fp****)*** – запись заданной строки в файл.

Например, прочитать из одного файла строку, ограниченную определенным количеством символов и поместить ее в другом файле. Исходная строка в файле *stroka.txt* – «***Отборочный турнир***».

Листинг программы.

```
#include<stdio.h>
FILE *fp;
void main( )
{  char *m;
   fp=fopen("c:\\temp\\stroka.txt","r");
```

```
    fgets(m,15,fp); // чтение только 15 символов, причем последний символ –
                    конец строки
    fclose(fp);
    printf("%s",m); //Результат на экране: «Отборочный тур»

    fp=fopen("c:\\temp\\stroka1.txt","w"); // Создание файла stroka1.txt и от-
                                           крытие его на запись
    fputs(m,fp); // Запись в файл результата чтения
    fclose(fp);
}
```

Другие функции приведем в следующей таблице.

Таблица. Функции по обработке файловых данных заголовочного файла ***stdio.h***.

Функция	**Описание**
int ***fclose****(<указатель на файл>);*	Закрывает поток ввода/вывода
int ***feof*** *(<указатель на файл>);*	Проверка достижения конца файла. Возвращает истину (ненулевое значение), если указатель находится в конце файла, и ложь - в противном случае
int ***ferror*** *(<указатель на файл>);*	Возвращает код ошибки; 0 – отсутствие ошибки
int ***fflush*** *(<указатель на файл>);*	Запись на диск содержимого буфера для заданного файла.
int ***fgetc*** *(<указатель на файл>);*	Возвращает очередной символ из файла
int ***fgetpos*** *(<указатель на файл>,<указатель на объект>);*	Возвращает текущую позицию в файле и копирует ее значение в указанный объект
char ***fgets****(<указатель на буфер>,<максимальное число читаемых символов>,<указатель на файл>);*	Последовательное чтение символов из файла до начала новой строки, или заданное количество символов, уменьшенное на 1. Прочитанные символы помещаются в буфер, начиная с ячейки, адресуемой первым параметром.
***`fopen`*(<константа-указатель>, <константа-режим>)*	Открывает указанный физический файл в режимах чтения/записи
int ***fprintf*** *(<указатель на файл>, <формат>, <аргументы>);*	Записывает данные в файл по заданному формату. Возвращает количество записанных байтов. Если обнаружены ошибки, то возвращает EOF.
int ***fputc****(<символ>,<указатель на файл>);*	Запись заданного символа в файл
int ***fputs****(<строка>, <указатель на файл>);*	Запись заданной строки в файл.

int ***fread*** *(<указатель на буфер>,<размер в байтах одного элемента>, <количество читаемых элементов>, <указатель на файл>);*	Чтение данных из файла, начиная с текущего положения указателя. Возвращает число прочитанных элементов. Число прочитанных байтов равно результату функции, умноженному на число байтов в элементе. В случае ошибки возвращается нуль.
int ***fscanf*** *(<указатель на файл>,<формат>,<список адресных аргументов>);*	Чтение данных из файла, преобразование их в соответствующий формат и размещение в ячейки, заданные адресными аргументами.
int ***fseek*** *(<указатель на файл>,<количество байтов>,<начальная позиция отсчета>);*	Перемещение указателя файла на заданное количество байтов. В случае успеха возвращается нуль, в случае ошибки - ненулевое значение. Третий параметр может принимать следующие значения: SEEK_SET - отсчет идет от начала файла, SEEK_END - отсчет идет от конца файла, SEEK_CUR - отсчет идет от начала текущей позиции.
void ***rewind*** *(<указатель на файл>);*	Установка указателя на начало файла.
long ***ftell****(<указатель на файл>);*	Возвращает текущую позицию в файле как длинное целое

2.13.3 Примеры чтения-записи данных

Пример 1. **Создание физического файла** *file1.txt* на диске *d* и запись туда одного целого *i* и одного вещественного *j* чисел, разделенных признаком табуляции *\t*.

```
#include<stdio.h>
FILE *fp;
void main(void)
{
 int i=100; float j=3.14;
 fp=fopen("d:file1.txt","w");
 fprintf(fp,"%d\t%f",i,j);
 fclose(fp);
}
```

Пример 2. **Чтение информации из физического файла** *file1.txt*, размещенного на дискете *d* и запись данных, хранящихся в нем, в переменную целого типа *i* и в переменную вещественного типа *j*, с последующим выводом этих чисел на экран в виде столбца.

Напомним, что в этом режиме файл должен существовать. Поэтому:

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

```
FILE *fp;
void main(void)
{ int i; float j;
 fp=fopen("a:file1.txt","r");
 if(fp==NULL)  {printf("файл не существует"); exit (0);}
 else {fscanf(fp,"%d%f",&i,&j); printf("число i=%d\nчислоj=%f",i,j);}
 fclose(fp);
}
```

Здесь функция *exit (0)* из заголовочного файла *stdlib.h* вызовет безусловный выход из программы.

Пример 3. (см. примеры работы с двумерными массивами, п. 2.10.2)

А) Ввод с клавиатуры и запись числовой матрицы в текстовый файл.

```
….
FILE *fa,*fb;
  void main ( )
  { int a[3][3],b[3][3],i,j;
  cout<<" Введите элементы матрицы A:\n";
   for(i=0;i<3;i++)
    for(j=0;j<3;j++)
     { printf("a[%d][%d]=",i+1,j+1);cin>>a[i][j];
      }
  fa=fopen("c:\\temp\\fmatr_A.txt","w");
  cout<<"\n Вывод матрицы A на экран и ее запись в файл
              fmatr_A.txt:";
   for(i=0;i<3;i++)
    for(j=0;j<3;j++)
     {if(j%3==0)
       {cout<<"\n\t"<<a[i][j];fprintf(fa,"\n\t%d",a[i][j]);}
      else
       {cout<<"\t"<<a[i][j];fprintf(fa,"\t%d",a[i][j]);}
      }
  fclose(fa);
….
```

В) Чтение числовой матрицы из текстового файла, изменение элемента, запись новой матрицы в новый текстовый файл.

```
….
fa=fopen("c:\\temp\\fmatr_A.txt","r");
    for(i=0;i<3;i++)
     for(j=0;j<3;j++)
      { fscanf(fa,"%d",&b[i][j]);
       }
  fclose(fa);
```

```
    cout<<"\n\n Какой элемент матрицы A изменить?:\n";
    cout<<"i=";cin>>i;
    cout<<"j=";cin>>j;
    printf(" Введите значение элемента a[%d][%d]=",i,j);
  cin>>b[i-1][j-1];

    fb=fopen("c:\\temp\\fmatr_B.txt","w");
    cout<<"\n Вывод элементов новой матрицы B на экран и ее за-
              пись в файл  fmatr_B.txt:";
   for(i=0;i<3;i++)
    for(j=0;j<3;j++)
     {if(j%3==0)
       {cout<<"\n\t"<<b[i][j];fprintf(fb,"\n\t%d",b[i][j]);}
      else
       {cout<<"\t"<<b[i][j];fprintf(fb,"\t%d",b[i][j]);}
      }
    fclose(fb);
  ...
```

C) Вывод указанных элементов матрицы из файла

```
…
  fa=fopen("c:\\temp\\fmatr_A.txt","r");
  fb=fopen("c:\\temp\\fmatr_B.txt","r");
  cout<<"\n\n Вывод на экран элементов главной диагонали матрицы
           A  из файла fmatr_A.txt:\n";
  cout<<"\t";
   for(i=0;i<3;i++)
    for(j=0;j<3;j++)
     {if(j==i)
       printf("\ta[%d][%d]=%d",i+1,j+1,a[i][j]);
      }
  cout<<"\n Вывод на экран элементов второй строки матрицы B из
           файла fmatr_B.txt:\n";
  cout<<"\t";
   for(i=0;i<3;i++)
    for(j=0;j<3;j++)
     {if(i==1)
       printf("\tb[%d][%d]=%d",i+1,j+1,b[i][j]);
      }
  fclose(fa);
  fclose(fb);
…
```

Пример 4*. **Код Цезаря.** Во второй строке текстового файла *fcezar.*txt храниться текст (не более 255 символов), который необходимо закодировать. Кодирование осуществляется по правилу: каждая из букв исходного текста замещается буквами того же алфавита, сдвинутыми влево или вправо от исходной буквы на количество позиций *k*, указанных в первой строке файла. Если *k<0*, то сдвиг влево; при *k>0* – сдвиг вправо; при *k=0* сдвига нет. Алфавит имеет замкнутый круг. То есть после последней буквы снова идет первая буква алфавита.

Пример. Если сдвинуть исходный текст русского алфавита «КОДИРОВАНИЕ» на 4 символа вправо, то результатом будет слово «ОТИМФТЖДСМЙ» без учета буквы «Ё».

Алгоритм решения. Направление сдвига определяет левую или правую границу алфавита. При сдвиге *вправо* на *k* позиций точка отсчета для алфавита кодирования смещается на *k позиций влево* от последней (правой) буквы. И наоборот, при сдвиге *влево* на *k* позиций точка отсчета для алфавита кодирования смещается на *k позиций вправо* от первой (левой) буквы.

Полный листинг программы для случая, когда направление сдвига заранее неизвестно, и результаты ее работы при различных значениях *k* выглядят, например, так:

Листинг программы

```
#include "iostream.h"
#include "stdio.h"
#include "string.h"

FILE *fc;
void main( )
{ char m2[256];
  int i,k,s;

            // блок чтения данных из файла
fc=fopen("c:\\temp\\fcezar.txt","r");
fscanf(fc,"%d",&k);
  printf(" в первой строке направление и число сдвига: k = %d\n",k);
for(i=0;i<=255;i++)
  fgets(m2,256,fc);
fclose(fc);

cout<<"\n исходное слово                ->\t"<<m2<<"\n";

            // блок кодирования при сдвиге символов вправо
if(k>0)
 {
```

* Задачи подобного типа, приведенные в примерах 4 и 5, в различной интерпретации, часто встречаются в олимпиадных задачах по информатике и в заданиях категории С (повышенной сложности) материалов ЕГЭ.

```
 for(i=0;i<strlen(m2);i++)
 {if(m2[i]==32)
    m2[i]=m2[i];
  else
   {
    if(m2[i]>'Я'-k)
    { s='Я'-m2[i];
      m2[i]=(char)('A'+k-1-s);
    }
    else
     m2[i]=(char)(m2[i]+k);
  }
 }
 printf(" при сдвиге ВПРАВО на k=%d символов ->\t%s\n\n",k,m2);
 }

                // блок кодирования при сдвиге символов влево
  if(k<0)
   {
   for(i=0;i<strlen(m2);i++)
   {if(m2[i]==32)
     m2[i]=m2[i];
    else
     {
      if(m2[i]<'A'-k)
       { s='A'-k-m2[i];
         m2[i]=(char)('Я'-s+1);
       }
      else
       m2[i]=(char)(m2[i]+k);
   }
  }
  printf(" при сдвиге ВЛЕВО k=%d символов -> \t%s\n\n",k,m2);
  }

                // блок кодирования при отсутствии сдвига символов
  if(k==0)
    cout<<" если НЕТ сдвига                ->\t"<<m2;
}
```

Результат

```
в первой строке направление и число сдвига: k = 4

исходное слово                      ->   АБВГДЕЖЗ ЦЧШЪЩЬЭЮЯ
при сдвиге ВПРАВО на k=4 символов ->   ДЕЖЗИЙКЛ ЪЫЬЮЭАБВГ
```

```
в первой строке направление и число сдвига: k = -5

исходное слово                      ->   АБВГДЕЖЗ ЦЧШЪЩЬЭЮЯ
при сдвиге ВЛЕВО на k=-5 символов ->   ЫЬЭЮЯАБВ СТУХФЧШЩЪ
```

```
в первой строке направление и число сдвига: k = 0

исходное слово                      ->   АБВГДЕЖЗ ЦЧШЪЩЬЭЮЯ
если НЕТ сдвига                     ->   АБВГДЕЖЗ ЦЧШЪЩЬЭЮЯ
```

Пример 5. **Вывод текстовых строк из файла.**

В текстовом файле *fPushk.txt* хранится фрагмент произведения А.С. Пушкина «Евгений Онегин». В первой строке этого файла указано число *j* – количество следующих текстовых строк (например, не более 10). Каждая строка заканчивается символом перевода каретки («\n» – «*Enter*»). В каждой строке может быть, например, не более 255 символов.

Вывести на экран содержимое этого файла.

Примеры содержимого текстового файла, листинга программы и результатов ее работы могут выглядеть, например, так:

Листинг программы

```
#include<iostream.h>
#include<stdio.h>
FILE *fs;
void main ( )
{ int i, j;  char s[10][250];
  fs=fopen("c:\\temp\\fPushk.txt","r");
   fscanf(fs,"%d",&j); //j=5
   for(i=0;i<=j;i++)
     fgets(s[i],250,fs);
  fclose(fs);
   for(i=0;i<=j;i++)
    cout<<s[i]<<"\n";
}
```

Содержимое файла *fPushk.txt*

```
5
"Мой дядя самых честных правил,
 Когда не в шутку занемог,
 Он уважать себя заставил
 И лучше выдумать не мог."
          А.С.Пушкин. Евгений Онегин.
```

Результат

```
"Мой дядя самых честных правил,
 Когда не в шутку занемог,
 Он уважать себя заставил
 И лучше выдумать не мог."
          А.С.Пушкин. Евгений Онегин.
```

(**Вопрос**. Какой размерности ($k \times m$) может быть символьный массив *s[k][m]* в данной интерпретации программы и почему?)

Функции по обработке файловых данных существенно упрощают про-

граммы, рассмотренные нами при изучении структур. Основная проблема – многократный ввод данных с клавиатуры при добавлении каких-либо новых блоков программы для решения задач при дополнительных условиях. Следующий пример показывает использование файловых данных в структуре.

Пример 6. **Вывести на экран структуру**, данные для которой хранятся в файле *C:\Temp\str.txt* вида:

9
121 Иванов 3 3 5
121 Петров 2 3 5
122 Сидоров 4 4 3
122 Годин 5 5 3
123 Антонов 5 2 2
121 Пухов 5 5 5
122 Волков 4 4 4
123 Дашко 5 4 5
123 Антипов 5 2 5

Здесь в первой строке указано количество записей для данных структуры, в последующих строках – значения этих данных.

Тогда программа для данного примера может выглядеть так.

```
#include<stdio.h>
struct student
{
   int ng, ma, la, hist;
   char fam[20];
   float sb;
} g[100]; //объявлена структура, в которой можно хранить 100 записей

FILE *fp;//объявлен указатель на логический файл


void main( )
{  int i,N;
   fp=fopen("c:\\temp\\str.txt","r");//открыт физический файл на чтение из него данных

                    //Блок чтения данных из файла
   fscanf(fp,"%d",&N);//чтение первой строки в переменную N
   for(i=0;i<N;i++)//чтение последующих N строк и заполнение данных структуры

  {fscanf(fp,"%d%s%d%d%d",&g[i].ng,&g[i].fam,&g[i].ma,&g[i].la,
          &g[i].hist);
    g[i].sb=(g[i].ma+g[i].la+g[i].hist)/3.0;//вычисление среднего балла
   }
```

```
   fclose(fp);//Закрытие файла

                    // Блок вывода результатов
   printf ("\nВывод результатов сессии:\n\n");
   printf ("\nВсего студентов: -> : %d", N);
   printf ("\n №гр\tФамилия\tMA\tЛА\tИст\tСр.балл\n");
   printf("-----------------------------------------------------\n");
   for(i=0;i<N;i++)
    {
     printf (" %d\t%s\t%d\t%d\t%d\t%f \n",g[i].ng,g[i].fam, g[i].ma, g[i].la,
              g[i].hist,g[i].sb);
    }
}
```

Результат.

```
Вывод результатов сессии:


Всего студентов: -> : 9
 №гр    Фамилии MA      ЛА      Ист     Ср.балл
-----------------------------------------------------
 121    Иванов  3       3       5       3.666667
 121    Петров  2       3       5       3.333333
 122    Сидоров 4       4       3       3.666667
 122    Годин   5       5       3       4.333333
 123    Антонов 5       2       2       3.000000
 121    Пухов   5       5       5       5.000000
 122    Волков  4       4       4       4.000000
 123    Дашко   5       4       5       4.666667
 123    Антипов 5       2       5       4.000000
```

Данные, хранящиеся в физическом файле, останутся неизменными, если заведомо не задаться целью – изменить этот файл. Это очень удобно при многократной обработке данных или при отладке программы, оперирующей исходными данными при новых условиях задачи.

Таким образом, мы рассмотрели некоторые возможности языка C/C++, позволяющие связывать программы с обработкой данных, хранящихся в различных физических файлах.

При этом мы рассматривали базовые возможности заголовочного файла *stdio.h* по обработке файловых данных. Это не единственная возможность. Стандартная библиотеки языка C++ содержит также три класса для работы с файлами. Это – класс входных файловых потоков ( ), класс выходных файловых потоков( ) и класс двунаправленного файлового потока ( ). Возможности этих классов можно рассматривать только после изучения основ объектно-ориентированного подхода в программировании.

2.13.4 Задачи для самопроверки по файловым типам данных

2.14 Функции в языке Си

Любая программа на любом алгоритмическом языке программирования состоит из отдельных модулей (одного или нескольких), которые могут содержаться либо в структуре одного исполняемого файла, либо в нескольких взаимосвязанных файлах, обращение к которым происходит по мере надобности. За реализацию таких модулей отвечают специальные конструкции языка. Чаще всего их называют *подпрограммами* или *процедурами*, но каждая из них, по-существу, является отдельной программой, выполняющей специфичную роль. При этом одна из таких программ является основной (ведущей), а остальные – вспомогательными или дополнительными.

В терминах языка Си любую программу для реализации модуля принято называть функцией. Например, изученная нами функция *printf( )* реализует модуль программы выдачи сообщения на экран, а функция *scanf( )* – модуль программы ввода значений с клавиатуры по определенному адресу в памяти.

Синтаксис и понятие функции в языке программирования C/C++ очень близки к терминологии, принятой в математике. Если в математических терминах написать *Cos (x)* или $y=Sin(\pi/4)$, или $g=f(x,y)$, то очевидно, что в первом случае речь идет о значении функции косинуса в точке x; во втором случае говорится о значении, которое приобретает величина y, после вычисления функции синуса в точке $\pi/4$; в третьем случае говорится о значении, которое приобретает величина g, после вычисления функции f , величина которой, в свою очередь, зависит от двух аргументов – x и y. Итак, в этих примерах мы рассмотрели такие понятия как*:*

- *значение (результат) непосредственно самой функции (Cos (x));*
- *значение переменной (y), получаемой в результате выполнения функции в конкретной точке*;
- *значение переменной (g), получаемой в результате выполнения функции (f(x,y)), значение которой, в свою очередь, зависит от нескольких аргументов*.

При этом понятен и синтаксис таких записей: функция определена именем и следующими после имени скобками; в скобках указываются через запятую (если более одного) аргументы (или переменные) от которых зависит значение функции; произвольным переменным могут присваиваться значения функций. Запомним ключевые понятия: *имя, скобки, значение, параметры*.

Все эти понятия присущи синтаксису языка Си при описании функций. Стандарт ANSI языка предполагает следующую форму определения функции (см. структуру программы на Си, п.2.1):

[тип_результата] ***имя_функции (****[список формальных параметров]****)***
{
*/****тело функции****: описание данных, операторы;*

*[return([выражение]);] */*

}

Здесь:

- *тип_результата* – необязательный элемент, он задается в том случае, если от данной функции требуется получить результат (значение), который будет использован в вычислениях другими переменными или функциями. Этот тип, определяется результатом работы функции и может быть, например, *void* (пустой тип), *int, char, float* и т.д. По умолчанию, *тип_результата* всегда *int*;
- *имя_функции* – любое имя, задаваемое пользователем, кроме зарезервированных имен, например, *printf, scanf, getch, sqrt* и др. Имя для главной функции (основной программы) также зарезервировано словом ***main***;
- *список формальных параметров* – список аргументов, с которыми производятся вычисления в функции. Этот элемент необязателен, так как возможны случаи, когда нет необходимости задавать параметры. Более подробно параметры рассмотрим ниже;
- *тело функции* – стандартная последовательность описания данных, операторов и выражений по обработке этих данных. Внутри тела функции возможна конструкция, при которой происходит обращение к другой функции;
- *return([выражение])* – специальная функция-оператор. Необязательная конструкция, применяемая только тогда, когда при описании функции задан тип, возвращаемого ею значения (кроме *void*). Очевидно назначение – вернуть то значение, которое будет использоваться в других выражениях и функциях основной программы и которое является результатом работы данной функции. Следует отметить, что результатом может быть и значение, получаемое в результате выполнения некоторого *выражения,* вычисляемого внутри скобок оператора возврата, например, *return(a*b/c)*. Значение, возвращаемое функцией, может быть любого типа, в том числе типа, объявленного пользователем. Функция не может возвращать массив и функцию, но может возвращать указатель на массив или указатель на функцию (о массивах и указателях см. п.2.10).

Обобщая изложенное, можно дать следующее определение.

Определение. *Функция* – это логически самостоятельная *именованная часть программы*, в которой могут быть использованы некоторые заданные параметры и которая может возвращать некоторое значение.

К функции можно обратиться из любой другой функции – ***вызвать*** функцию.

В языке С++ функции играют двоякую роль.

Во-первых, любая программа в языке С++ представляет собой одну большую функцию. Эта функция всегда имеет стандартное имя *main* и называ-

ется **главной** функцией. Операционная система при загрузке программы всегда вызывает именно эту главную функцию *main*. Главная функция программы, так же, как и любая не главная, может принимать параметры, в данном случае главная функция получает параметры от операционной системы, и может возвращать значение в операционную систему. В каждой программе может быть только одна главная функция.

Во-вторых, любая не главная функция выполняет некоторый фиксированный набор действий. Однажды описанная функция может вызываться программой неограниченное число раз. Тем самым отпадает необходимость многократного описания в программе одних и тех же действий.

Любая функция может содержать в своем теле вызовы (но не описания) других, не главных функций. Для того, чтобы функцию можно было вызвать, она должна быть предварительно описана. Описание функции должно быть приведено вне любой другой функции. Функция может быть описана и в отдельном файле.

Написанные и откомпилированные функции могут объединяться в библиотеки. Любая современная среда программирования, в том числе и среда программирования Си/С++, имеет множество встроенных библиотек функций. Библиотечные функции обычно обеспечивают реализацию большей части задач, которые могут стоять перед программой.

Функции, описанные в программе или помещенные в библиотеку видны из всех файлов программы (программа может быть построена из нескольких файлов). Единственное условие для возможности вызова функции – это наличие дополнительного объявления (прототипа) функции перед ее вызовом (описание прототипов функций приведено ниже).

Иначе говоря, функция – это независимая последовательность операторов для выполнения некоторой задачи. При конструировании многофункциональных программ в теле основной программы могут быть использованы как библиотечные функции, так и пользовательские. Связь между функциями осуществляется через аргументы (заданные параметры) и возвращаемые значения. Если программа состоит из нескольких функциональных модулей, то синтаксис стандарта Си требует такого их представления (возможны два способа):

1-й способ:

```
#include<...>
...
 [тип_рез.] name ([список форм. парам.]); /*прототип функции с именем
                                                    name          */
...
main( )       //главная функция
{              /*тело функции main: описание данных, операторы;
 обращение к функции name( );
 [return([выражение]);]
                                                                  */
}
```

[тип_рез.] ***name (****[список форм. парам.]****)*** ***//описание функции***
{ */*тело функции* ***name****: описание данных, операторы;*
[return([выражение]);]
**/*
};

2-й способ отличается от первого тем, что и прототип и непосредственно само описание используемой в программе функции (функций) задаются до главной функции, например:

#include<…>
[тип_рез.] ***name (****[список форм. парам.]****)*** */*****прототип функции** *с именем* ***name и ее описание*** */*
{ */*тело функции* ***name****: описание данных, операторы;*
*[return([выражение]);] */*
};
main( ) *//главная функция*
{ */*тело функции main: описание данных, операторы;*
обращение к функции name( );
[return([выражение]);] */*
}

Многие программисты используют второй способ описания, однако, на наш взгляд, первый, хотя и требует некоторых повторений в описании заголовка функций, предпочтителен, так как позволяет проследить логическую последовательность взаимодействия отдельных модулей и блоков программы.

2.14.1 Прототипы функции

В приведенных выше фрагментах программ описание неглавной функции *name( )* приводится в одном физическом файле с описанием главной функции *main( ).* При втором способе текст описания неглавной функции предшествует тексту функции *main( )*. В таких случаях при компиляции программы проблем не возникает, компиляция проходит безошибочно. Ошибка компиляции возникнет в том случае, если описание неглавной функции расположить по тексту после описания функции *main( )* (первый способ) или привести его в другом физическом файле.

Дело в том, что компилятор обрабатывает текст программы последовательно, строку за строкой. И в тот момент, когда компилятор доходит до строки, содержащей вызов функции, например, *name( ),* компилятор уже должен иметь некоторую информацию о вызываемой функции, в частности, тип возвращаемого функцией значения, число и типы ее параметров. В тех случаях, когда описание функции приводится после описания функции *main( )* или приводится в другом файле, или когда производится вызов какой-либо библиотечной функции, в языке С++ используются краткое объявление функции, называемое ***прототипом*** функции. Прототип функции в таких случаях необходим

для того, чтобы предварительно сообщить компилятору информацию о вызываемой функции.

Прототип функции можно получить, взяв из описания функции ее заголовок и завершив этот заголовок ***точкой с запятой***.

Информация, содержащаяся в прототипе функции, используется компилятором для проверки правильности оператора вызова функции.

Область действия прототипа зависит от места его расположения. Прототип вызываемой функции может быть приведен и в теле вызывающей функции, непосредственно перед оператором вызова функции. В этом случае область действия прототипа ограничивается телом функции, в которой этот прототип приведен. Прототип может быть записан вне какой-либо функции. В этом случае область действия прототипа распространяется на весь файл, в котором этот прототип приведен, т.е. любая функция, описанная в данном файле ниже прототипа, может содержать вызов соответствующей функции. На практике часто прототипы функций приводятся в заголовочных файлах, т.е. файлах с расширением ***.h***. Эти заголовочные файлы подключаются в программе с помощью директивы препроцессора ***#include*** в тех файлах, в которых имеются соответствующие вызовы функций. Для стандартных библиотек функций, входящих в состав среды программирования C++, имеются заголовочные файлы, содержащие прототипы функций. Эти заголовочные файлы подключаются к программе при необходимости вызова библиотечных функций.

2.14.2 Передача параметров в функциях

Определение. *Параметры функции* – это переменные (или другие объекты), значения которых передаются в функцию в качестве входных данных и над которыми в функциях выполняются заданные действия.

Например, форма такого определения:

```
include<...>
 void name (int k,int m);
main( )
{  int a=5,b=6;
    name(a,b);
...
 }
```

означает, что главная функция *main( )* обратилась с вызовом в функцию *name( )* и передала в нее для дальнейшей обработки два целых числа.

При работе с функциями различают два понятия параметров: *формальные* параметры и *фактические* параметры.

Определение. *Формальные* параметры – это абстрактные переменные, которые используются для описания функции. *Фактические* параметры (аргументы) – это переменные, конкретные значения которых подставляются в функцию вместо формальных параметров в момент ее вызова.

В рассмотренном выше примере формальными параметрами для функции *name( )* являются переменные *k* и *m*. В момент вызова из функции *main( )* функции *name( )* с передачей в нее двух значений (переменных *a* и *b*), переменным *k* и *m* соответственно присваиваются значения 5 и 6 (*k=5, m=6*). В данном примере переменные *a* и *b* являются фактическими параметрами.

Количество фактических параметров, их тип и порядок аргументов при их передаче в вызываемую функцию должно совпадать с количеством, типом и порядком формальных параметров при описании заголовка вызываемой функции.

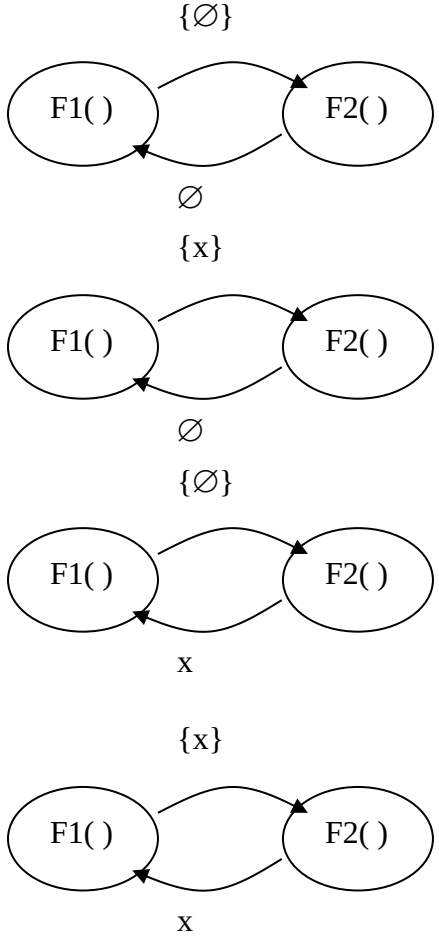
Формальных параметров при определении функции может быть несколько. Все они перечисляются при объявлении функции через запятую в ее заголовке. Синтаксис языка Си допускает, например, следующие варианты объявления функций при описании в их заголовках формальных параметров:

*void fun1(int **s**, float **g**);*
float fun2(int, float, float);

Здесь в первом случае объявлена функция *fun1( )*, формальные параметры которой (два) и их тип (***int*** *s,* ***float*** *g*) определены *явно* в заголовке. Во втором случае формальные параметры (три) в функции *fun2( )* определены не явно. Это означает, что их количество, тип и имена должны быть объявлены в теле функции при ее описании.

2.14.3 Способы обращения к функциям

При работе в программах, использующих в своей структуре несколько функций, всегда возникает условие отношения между двумя функциями, одна из которых является вызывающей (основной), а другая – вызываемой. При этом различают понятия использования вызываемой функции в основной программе либо как *оператор*, либо как *выражение*. Например, запись в теле основной функции *y=fun1( );* – выражение, а запись *fun2( );* – оператор.



Возможны следующие способы обращения к функциям:

Здесь (и ниже) функция F1( )–вызывающая, функция F2( ) –вызываемая. Такая графическая интерпретация 1 означает, что F1( ), вызывая F2( ), не передает в нее ни каких параметров, а F2( ) не возвращает в F1( ) никакого значения для последующей его обработки. F2( ) используется в F1( ) как *оператор*.

В данном примере (интерпретация 2) F1( ), вызывая F2( ), передает в нее в качестве фактических параметров значения каких-либо переменных, а F2( ) не возвращает в F1( ) никакого значения для последующей его обработки. И здесь F2( ) используется в F1( ) как *оператор*.

Интерпретация 3. F1( ), вызывая F2( ), не

передает в нее в каких-либо аргументов, но F2( ) возвращает в F1( ) какое-либо значение для последующей его обработки в F1( ). Здесь F2( ) используется в F1( ) как *выражение*.

Интерпретация 4. F1( ), вызывая F2( ) передает в нее в качестве аргументов какие-либо значения фактических параметров, и F2( ) возвращает в F1( ) какое-либо значение для последующей его обработки в F1( ). Здесь F2( ) используется в F1( ) тоже как *выражение.*

Рассмотрим пример, иллюстрирующий сказанное.

```
#include<...>
 void F1(void);
 void F2 (int, int);
 int F3(void);
float F4(int,float);
 void main(void)
 {  int a=5,b=6, x; float y, z=3.14;
    F1( );
    F2(a,b);
    x=F3( );
 ...y=F4(x,z)/x+a*b;
 printf("x=%d\t y= %f", x,y);
  }
```

В данной программе объявлены прототипы четырех функций в соответствии со схемой вызова, описанной выше. Все они последовательно вызываются из главной функции в качестве операторов или выражений с помощью передачи (не передачи) фактических параметров и с контролем (или без него) возвращаемого значения. Последующее описание функций должно быть, например, таким:

```
void F1(void)
{  printf ("Привет\n");
}
void F2 (int k, int m)
{ printf ("Выражение b-a=%d\n",m-k);
}
int F3(void)
{int a=6, b=4, c;
c=a*b;
return c;
}
float F4(int k, float m)
{ float n=k*m;
 return n;
}
```

В результате работы программы на экране будет получено сообщение следующего вида:

Привет
Выражение b-a=1
x=24 y=33.139999

Первая строка этого результата будет получена в функции *F1( )*, вторая – в функции *F2( )*, третья в главной функции, после получения значения для параметра *x* в функции *F3( )* и его использования в функции *F4( )*.

Приводя этот пример, мы предполагаем, что и главная функция и все другие функции находятся в одном физическом файле. Кроме того, здесь следует обратить внимание на то, как используются переменные, которые участвуют в вычислениях в каждой отдельно взятой функции. Например, переменные с одними и теми же именами *k* и *m* принадлежат и функции *F2( )*, функции *F4( )*, но имеют разные типы и разные значения. Хотя тип переменных *a* и *b* не меняется, они также относятся к двум функциям *main* и *F3( ),* изменяя в них свое значение. Эта особенность использования одних и тех же переменных в разных функциях относится к важному свойству программирования на C/C++ - *области видимости переменных* (см. ниже).

Здесь отметим, что область видимости параметров (переменных) функции ограничена телом функции, т.е. параметры функции могут использоваться только в операторах, образующих тело функции. Поэтому использование формальных параметров позволяет не изменять значения переменных (фактических параметров) в теле вызывающей функции.

Еще несколько замечаний о фактических и формальных параметрах при обращении к функциям в случае их передачи из одной функции в другую.

При несоответствии типов фактических и формальных параметров, компилятор производит преобразование типов фактических параметров к типам соответствующих формальных параметров. Без проблем происходит преобразование от младших типов к старшим, т.е. когда, например, параметр типа *char* преобразуется к одному из типов *int, float, double*, или когда параметр типа *int* преобразуется к типу *long, float* или *double*. Возможно возникновение ошибок при выполнении программы (но не на этапе компиляции) если в операторе вызова функции фактический параметр имеет более старший тип, чем тип формального параметра в описании функции. В таких случаях возможно искажение значений передаваемых функции данных. Поясним сказанное примерами.

Вначале рассмотрим пример вызова функции, имеющей один формальный параметр типа *char*. Как известно, данные типа *char* занимают в памяти один байт, т.е. 8 бит. Ячейка памяти типа *char* может содержать целое число в диапазоне от –128 до +128. Однако при вызове функции с формальным параметром типа *char* в качестве фактического параметра можно записать любое число, целое или дробное. В том случае, если фактический параметр выходит за пределы диапазона –128…+128 или является числом с плавающей запятой, компилятор не выдает ошибки, но выдает предупреждающее сообщение о том, что было произведено преобразование типа и что возможна потеря данных. На

примере рассмотрим, каким образом происходит преобразование фактических параметров в таких случаях:

void FunC(char); /*Прототип функции, имеющей один формальный параметр типа *char* */

FunC(258); /*Вызов функции *FunC( )* с передачей ей в качестве фактического параметра десятичного числа 258. Реально, функции передается число 2

В приведенном примере функции передается число 258, двоичное представление которого имеет вид 100000010, т.е. занимает 9 бит. В таких случаях, когда функции в качестве фактического параметра передается число, занимающее в памяти больше, чем 8 бит, происходит простое отбрасывание лишних старших бит. В приведенном примере отбрасывается один бит, функции передается число в двоичном представлении 00000010, в десятичном представлении это число 2. В данном примере функции можно передать и любое отрицательное число. При этом нужно иметь в виду, что отрицательное число функции передается в его дополнительном коде (мы предполагаем, что читателю известен способ хранения отрицательных чисел в оперативной памяти компьютера). Рассмотрим вызов той же функции *FunC( )*, но с фактическим параметром -508:

FunC(-508); /*Вызов функции *FunC( )* с передачей ей в качестве фактического параметра отрицательного числа -508. Реально функции передается число 4 */

Дополнительный код числа –508 в двоичном формате имеет вид 1111111000000100. Это двухбайтовое число. При передаче его функции *FunC( )* отбрасывается 8 старших бит его двоичного представления, в результате чего функции передается двоичное число 00000100, т.е. десятичное число 4.

Аналогичная процедура отбрасывания лишних разрядов при передаче функции целочисленных значений реализуется и в тех случаях, когда функция имеет формальный параметр типа *short, int* или *long*. Отличие существует только в количестве передаваемых функции и отбрасываемых двоичных разрядов. Для примера рассмотрим функцию, имеющую параметр типа *short*:

void FunD(short); /*Прототип функции, имеющей один формальный параметр типа *short* */

FunD(65539); //Вызов функции *FunD( )* с передачей ей в качестве фактического параметра десятичного числа 65539.

Максимальное значение, которое можно записать в ячейку памяти типа short равно 65535. Поэтому в результате отбрасывания избыточных старших разрядов в двоичном представлении числа 65639, реально функции передается в качестве фактического параметра число 3.

Данные типа *short int* как и *short*, занимают в памяти компьютера два байта, т.е. 16 бит. Десятичное число 65539 в двоичном представлении имеет

вид 1000000000000011, т.е. занимает 17 бит. При передаче функции *FunD( )* этого числа в качестве фактического параметра, лишний, семнадцатый бит отбрасывается, в результате чего функции передается двоичное число 0000000000000011, т.е. десятичное число 3.

В операторе вызова функции, имеющей параметр типа *char, short* или *int* в качестве фактического параметра можно записать и число с плавающей запятой, т.е. число типа *double* или *float*. При этом дробная часть такого фактического параметра отбрасывается полностью, а целая часть числа преобразуется в полном соответствии с вышеописанными правилами преобразования целочисленных фактических параметров. В качестве примера возьмем те же самые вышеописанные функции *FunC( )* и *FunD( )*, но с фактическими параметрами типа *double*.

void FunC(char); /*Прототип функции, имеющей один формальный параметр типа char */

FunC(258.777); /*Вызов функции *FunC( )* с передачей ей в качестве фактического параметра числа типа double */

Отбрасывается дробная часть десятичногопредставления числа. У двоичного представления числа 258 отбрасывается старший бит. В результате таких преобразований функции передается двоичное число 00000010, т.е. десятичное число 2

FunC(-508.888); /*Вызов функции *FunC( )* с передачей ей в качестве фактического параметра отрицательного числа типа double.*/

Отбрасывается дробная часть десятичного представления числа. У двоичного представления числа -508 отбрасывается 8 старших бит. В результате функции передается двоичное число 00000100, т.е. десятичное число 4.

void FunD(short); /*Прототип функции, имеющей один формальный параметр типа *short* */

FunD(65539,999); /*Вызов функции *FunD( )* с передачей ей в качестве фактического параметра числа типа *double*.*/

Отбрасывается дробная часть десятичного представления числа. У двоичного представления числа 65539 отбрасывается старший бит, семнадцатый бит. В результате таких преобразований функции передается двоичное число 0000000000000011, т.е. десятичное число 3.

2.14.4 Использование в качестве фактических параметров функции вызовов других функций

В качестве фактических параметров в операторе вызова функции могут использоваться вызовы других функций. В этих случаях функции передается значение, возвращаемое другой функцией, используемой в качестве фактического параметра. Продемонстрируем сказанное примером программы. В этой программе в качестве фактического параметра *FunC( )* используем библиотечную функцию *sin( ).*

#include<*iostream.h*>

```
#include<math.h>      /* заголовочный файл, содержащий прототипы библиотечных тригонометрических функций */

double FunC(double Hi)
{ return(2*Hi);} //Функция возвращает удвоенное значение ее параметра

void main( )
{
  double Pi,Ro; //Объявление переменных Pi,Ro
  cin>>Pi; //Ввод с клавиатуры значения переменной Pi
  Ro=FunC(sin(Pi)); /*Оператор, содержащий вызов функции FunC( ). В качестве фактического параметра функции FunC( ) использована функция sin(Pi).
  cout<<Ro; /*Вывод на экран значения переменной Ro, равного удвоенному значению, возвращаемому функцией sin(Pi). Так, например, если введем с клавиатуры значение для Pi, равное 1.57 (sin(1.57)=1), то функция FunC( ) возвратит число 2.
}
```

2.14.5 Применение макросов в функциях

Рассматривая директивы препроцессора (см. п.2.7) мы рассматривали директиву замещения *#define*. Часто эту директиву препроцессора называют ***макросом***, макрокомандой или макроопределением. В языке C/C++ макросы являются некоторым подобием функций. Так же, как и функция, макрос имеет имя. Так же, как и функция, однажды написанный макрос может многократно использоваться в программе. Так же, как и функция, макрос может иметь параметры.

В общем случае формат макроопределения имеет вид:

#define** имя_макроса(список_параметров) **строка_замещения

Как видно, всякое описание макроса начинается со знака #, за которым следует ключевое слово *define*. За ключевым словом *define* следует собственно *описание макроса*, состоящее из *имени* макроса и *строки замещения*. Строку замещения иногда называют макроподстановкой или макрорасширением. Так же, как и имя функции, имя макроса задается произвольно и может состоять из букв латинского алфавита, цифр и знака подчеркивания. Так же, как и при описании функции, после имени макроса может быть приведен в скобках список формальных параметров макроса. Но в отличие от функции, здесь формальные параметры приводятся без указания их типов. Перечисленные в скобках формальные параметры используются в операторах строки замещения. Этот механизм использования параметров в макросе аналогичен механизму использованию параметров функции, т.е. при вызове макроса формальные параметры замещаются фактическими параметрами, которые и обрабатываются операторами строки замещения.

Строка замещения представляет собой произвольное выражение, включающее переменные, константы, операторы и даже вызовы ранее определенных функций и макросов.

После объявления макроса, его имя может использоваться далее в тексте программы неограниченное число раз. Каждое вхождение в программу имени макроса препроцессор заменяет текстом строки замещения, т.е. макроподстановкой.

Макрос может не иметь параметров, в этом случае объявление макроса имеет вид:

#define** имя_макроса **строка_замещения

Приведем пример использования простейшего макроса с параметрами:

```
#include<iostream.h>
#define max(a,b) (a<b ? b : a) //Определение макроса max(a,b) с параметрами a,b
void main(void)
{
  int x=5, y=10, z;
  z =max(x,y);
  cout<<z;
}
```

Здесь выражение *max(x,y)* препроцессор заменяет на выражение *(x>y ? x : y)*, согласно которому переменной *z* присваивается максимальное из значений переменных *x, y*. На экран выводится значение,присвоенное переменной *z*, число 10.

Следующий пример демонстрирует макрос, строка замещения которого содержит вызов библиотечной функции *sin( )*.

```
#include<iostream.h>
#include<math.h>
#define macs(x,y)  p*sin(x*y) // Определение макроса macs с формальными
                              параметрами x,y.
void main(void)
{
int p=5;
double w;
w=macs(2.0,4.0);
cout<<w;
}
```

Строка замещения макроса представляет собой произведение переменной *p* на значение, возвращаемое функцией *sin(x*y),* где *x,y* – формальные параметры макроса. Обращаем внимание на то, что переменная *p* не объявлена до описания макроса, и тем не менее ошибки при работе препроцессора не возникает. Это объясняется тем, что, как было сказано выше, препроцессор чисто механи-

чески заменяет встречающееся ему в тексте программы имя макроса на строку замещения и тем самым подготавливает текст программы для обработки компилятором. После обработки препроцессором текста программы строка *w=macs(2.0,4.0)* преобретает вид: *w= p*sin(2.0 * 4.0)*. На экран выводится значение, возвращаемое макросом *macs(2.0*4.0)*, в данном случае это число 4.94679

Основное отличие макроса от функции заключается в следующем.

1. Тело функции в программе всегда существует в единственном числе, независимо от количества операторов вызова функции, имеющихся в программе. При каждом вызове функции происходит обращение к операторам этого единственного экземпляра. При этом функции передаются фактические параметры, записанные в операторе вызова. Недостатком использования в программе функций является то, что на процесс вызова функции затрачивается значительное машинное время, необходимое для сохранения в стеке состояния регистров процессора, адреса возврата в вызывающую функцию и для передачи фактических параметров из вызывающей функции в вызываемую.

2. Иначе происходит работа макросов. Каждый раз, когда препроцессор обнаруживает в тексте программы идентификатор макроса, он подставляет вместо имени макроса его макроподстановку, т.е. выражение, стоящее в правой части определения макроса. Далее, обработанный таким образом препроцессором, исходный текст программы поступает на вход компилятора. В результате, в откомпилированной программе, количество блоков исполняемого кода макроса равно количеству вызовов этого макроса в исходном тексте программы. При этом не затрачивается машинное время на вызов макроса, операторы макроса выполняются последовательно, один за другим, как если бы это был обычный код программы. Недостатком такого подхода является увеличение объема исполняемого кода программы.

Еще одним отличием макросов от функций является то, что описание макроса может быть приведено как вне функций, так и в теле какой-либо функции. При этом, независимо от места определения макроса, область действия его распространяется на весь нижерасположенный текст программы в данном файле. Рассмотрим пример.

```
#include<iostream.h>
void FunA(void)    //Описание функции FunA( )
{
#define sqr(x) x*x//Определение макроса sqr(x) в теле функции FunA( )
}                          //Макрос возвращает квадрат параметра x
int FunB(int a,int b) //Описание функции FunB( )
{
 return sqr(a)+sqr(b);
}
void main(void)
{
 cout<<FunB(4,5);
```

}

Макрос *sqr(x)*, описанный в функции *FunA( )*,виден и в функции *FunB( )*. Функция *FunB( )* возвращает сумму квадратов двух параметров. На экран выводится сумма квадратов двух фактических параметров 4 и 5, число 41

В то же время область действия макроса не выходит за пределы файла, в котором он описан. В том случае, когда один и тот же макрос нужно использовать в нескольких файлах программы, описание макроса можно привести в отдельном заголовочном файле, т.е. в файле с расширением ***.h***, и затем директивой препроцессора *#include* подключить этот заголовочный файл ко всем файлам, в которых необходимо использовать этот макрос. Это возможно потому, что препроцессор объединяет текст файла, имя которого объявляется в директиве *#include* с основным текстом программы.

Любая директива препроцессора, в том числе и определение макроса, должна располагаться в исходном тексте программы в одной строке. В том случае, когда описание макроса не умещается в одной строке на экране, строку можно продолжить обратной косой чертой. Так, например, эквивалентны описания:

#define** macs(x,y) **p*sin(x*y) и ***#define** macs(x,y) **\p*sin(x*y)***

Объявленный макрос можно аннулировать директивой препроцессора *#undef*. Так, например, если в тексте программы встретится директива:

***#undef** macs(x,y),*

то далее по тексту программы макрос *macs(x,y)* становится неопределенным, и последующие ссылки на него будут приводить к ошибке при компиляции.

Макросы могут быть вложенными, т.е. имя ранее объявленного макроса может входить в макрорасширение последующего макроса. Приведем пример.

```
#include<iostream.h>
#include<Math.h>
#define pro(x,y)  x*y//Определение макроса pro(x,y), возвращающего произ-
                     ведение двух его параметров
 #define macsP(x,y) p*sin(pro(x,y)) //Определение макроса macsP(x,y). Стро-
                     ка замещения этого макроса содержит ранее определенный макрос pro(x,y)
void main(void)
{
 int p=5;
 double w;
 w=macsP(2.0, 4.0);
 cout<<w;
}
```

В программе происходит вызов макроса *macsP* с параметрами *x=2.0* и *y=4.0*. Обработку этого макроса препроцессор выполняет за два прохода. При первом проходе выражение: *w=macsP(2.0, 4.0)* замещается препроцессором на выражение: *w= p*sin(pro(2.0,4.0))*. Затем, на втором проходе, препроцессор обнаруживает вложенный макрос: *pro(2.0,4.0)* и замещает его на выражение: *2.0*4.0*. После обработки препроцессором описываемый оператор приобретает вид: *w=p*sin(2.0*4.0)*. На экран выводится значение, возвращаемое макросом *macsPro(2.0, 4.0)*, число *4.94679.*

Как было сказано ранее, замену имени макроса его макрорасширением препроцессор выполняет чисто механически. Это нужно иметь в виду при написании макросов, представляющих собой сложные выражения. В некоторых случаях, для того, чтобы получить требуемый результат, необходимо в описании макрорасширения макроса, заключать в круглые скобки или отдельные формальные параметры, или отдельные выражения, составляющие макрорасширение. Приведем конкретный пример.

```
#include<iostream.h>
#define sqr(x) x*x          //Определение макроса sqr(x). Макрос возвращает
                            квадрат параметра x.
void main(void)
{
  int a=4,b=5;
 cout<<sqr(a);      //В данном случае макрос срабатывает правильно, на
                    экран выводится квадрат фактического параметра макро-
                    са a – число 16.
 cout<<sqr(a+b);
#define sqr1(x) (x)*(x)
 cout<<sqr1(a+b);
}
```

В операторе *cout<<sqr(a+b)* в качестве фактического параметра макроса *sqr* использована сумма двух переменных *a+b*, поскольку мы хотим получить квадрат суммы двух переменных. В принципе это допустимо. Но препроцессор, при обработке текста программы, вместо макрокоманды *sqr(a+b)* подставляет ее макрорасширение, которое будет иметь вид: *a+b*a+b.* В результате на экран будет выводиться неверный результат – число 29. Для получения требуемого результата формальные параметры в определении макроса в этом случае нужно заключить в круглые скобки. При такой записи вызов макроса *sqr1(a+b)* заменяется его макроподстановкой *(a+b)*(a+b*). На экран выводится квадрат суммы двух переменных *a* и *b*, число 41.

В строке замещения макроса формальный параметр может быть записан со знаком #. Этот символ преобразует стоящий после него параметр к типу «строка», т.е. препроцессор в этом случае, при замене макрокоманды ее макрорасширением, подставляет не значение фактического параметра, а его текстовое представление. Эту операцию удобно использовать, например, в операторах вывода на экран. Приведем пример.

```
#include<iostream.h>
#define prn(Teta) cout<<#Teta<<«=«<<Teta        //Описание макроса prn.

void main(void)
{
int Alfa=5,Beta=10;
prn(Alfa);
prn(Beta);
prn(Alfa+Beta);
}
```

Первое вхождение в макроподстановку формального параметра *Teta* приводится со знаком #, это означает, что препроцессор на этом месте, в строке вызова макроса, подставит не значение фактического параметра, а его текстовое представление. Объявление переменных *Alfa* и *Beta*. Вызов макроса prn с фактическим параметром *Alfa*. На экран выводится строка *Alfa=5.* Вызов макроса prn с фактическим параметром *Beta*. На экран выводится строка *Beta=10.* Вызов макроса *prn* с фактическим параметром *Alfa+Beta*. На экран выводится строка *Alfa+Beta=15*

2.14.6 Главная функция программы *main( )*

Как было сказано выше, любая программа, написанная на языке Си, представляет собой функцию ***main( )***, которая называется главной функцией. Функция *main( )* вызывается операционной системой при запуске программы. Стандарт языка С++ предусматривает две формы описания функции *main( )*:

1) *без параметров*:

<*тип*> ***main( )***

{….Тело функции….***}***

2) *с двумя параметрами*:

<*тип*> ***main(int*** argc, ***char****argv[ ]***)***

{….Тело функции….***}***

Функция *main( )* может не возвращать никакого значения, в этом случае указывается тип возвращаемого значения *void*, или может возвращать целочисленное значение. Принято, что в последнем случае, при успешном завершении программы функция *main( )* должна возвратить нулевое значение. Возврат функцией *main( )* ненулевого значения будет означать аварийное завершение программы.

Функция *main( )* может быть описана с двумя параметрами. Причем типы этих параметров жестко фиксированы. Первый параметр должен быть типа *int*, а второй – типа *char* [ ]*, т.е. должен быть массивом указателей типа *char* (подробно о массивах указателей в п.2.10.4). Имена формальных параметров

функции *main( )* могут быть произвольными, но принято для первого параметра использовать имя *argc*, для второго – *argv*.

При таком задании параметров функции *main( )* операционная система, при запуске программы, в первый параметр заносит количество слов в командной строке запуска программы, а в указатели – элементы массива второго параметра - адреса слов, из которых состоит командная строка запуска программы. Слова в командной строке запуска программы должны отделяться друг от друга пробелами. Первый указатель, *argv[0]*, содержит адрес первого слова командной строки запуска программы, т.е. имя программы. Причем заносится в этот параметр полное имя программы, т.е. имя исполняемого файла с указанием пути. Соответственно, второй указатель *argv[1]* будет содержать адрес второго слова командной строки, третий указатель *argv[2]* будет содержать адрес третьего слова командной строки, и т. д. Рассмотрим сказанное на примере программы.

```
#include<iostream.h>
void main(int argc, char*argv[ ])
{cout<<argv[0]<<endl;          //Вывод на экран полного имени исполняемого файла программы.
  for(int i=1;i<argc;i++)
     cout<<argv[i]<<endl;      // Вывод на экран арументов, задаваемых при вызове программы в командной строке.
}
```

Если создать исполняемый файл этой программы с именем *Program.exe* и запустить его из директория *C:\MyProgram* с помощью командной строки:

Program.exe Alfa Beta Gamma,

то на экран будут выведены строки:

C:\MyProgram\Program.exe
Alfa
Beta
Gamma

Иначе говоря, применение функции *main( )* с параметрами в программах, позволяет осуществлять запуск их из командной строки с указанием аргументов или команд подобно тому, как это делается в командной строке *command.com*

2.14.7 Рекуррентный вызов функций

В языке C++ функция может вызвать сама себя. Этот процесс называется рекуррентным вызовом или рекурсией. Пример рекуррентного вызова функции рассмотрим на фрагменте программы. В этом примере возможность рекурсии функции используется для вывода экран передаваемого функции *FunR( )* слова в обратном порядке; в качестве параметра передается слово “*лаз*”, а на экран функция выводит слово “*зал*”.

Напомним (см. п.2.11), что строка символов, заключенная в кавычки, на самом деле является указателем на первый элемент этой самой строки, реально размещенной в оперативной памяти. Поэтому, несмотря на то, что в описании функции *FunR( )* в качестве формального параметра указан указатель типа *char**, при вызове этой функции в качестве фактического параметра фигурирует строка символов “лаз”.

```
#include<iostream.h>
void FunR(char* Eta)
{ Eta++;                  //Инкремент параметра-указателя.
   if(*Eta!=«\0»)         //Если указатель указывает не на конец строки, выполняется самовызов функции FunR( ),т.е. функция вызывает сама себя.
  FunR(Eta);
   Eta--;                 //После возврата из самовызова указатель перемещается на предыдущий символ строки-параметра и указываемый символ выводится на экран.
 cout<<*Eta;              // Результат – слово «зал».
}
void main(void)
{
FunR("лаз");              // Вызов рекурсивной функции с параметром «лаз»
}
```

Функция *FunR( )* производит вложенные самовызовы до тех пор, пока указатель на переданную строку – параметр не будет указывать на конец строки. По достижении конца строки функция выводит на экран последний символ. Затем, после каждого возврата из очередного самовызова функции, происходит перемещение указателя на предыдущий символ и вывод его на экран. Таким образом, любую переданную в качестве параметра строку символов функция *FunR( )* выводит на экран в обратном порядке.

Достоинством программы, использующей рекурсию, является компактный программный код, поэтому рекурсивные функции используются при работе со сложными типами данных, такими как связные списки, очереди, деревья.

Недостатком рекурсивного вызова функций является значительные затраты машинного времени, связанные с процессом вызова функции. Это снижает скорость вычислений. Кроме того, при каждом новом рекурсивном вызове функции выделяется оперативная память стека для размещения локальных переменных, а так же для сохранения адреса возврата из функции и состояния регистров процессора, что может вызвать переполнение стека.

2.14.8 Функции с переменным числом параметров

Иногда возникает необходимость написания в какой-то степени универсальной функции. Универсальной в том смысле, что функция в разных случаях должна выполнять один и тот же набор действий, но с разным числом передаваемых ей параметров. В языке C++ существует возможность написания функ-

ции, которая может принимать при ее вызове в разных ситуациях разное число параметров. Формат описания такой функции иллюстрируется следующим примером:

```
void FunA(int Par1, int Par2…)      //Пример описания функции с
    { Тело функции }                //переменным числом параметров
```

В этом примере функция *FunA( )* принимает два фиксированных параметра *Par1* и *Par2* типа *int*, за которыми следует многоточие, означающее то, что функция может принимать при ее вызове произвольное количество параметров. Между многоточием и последним фиксированным формальным параметром может стоять запятая или любое количество пробелов.

При вызове такой функции количество фактических параметров не может быть меньшим количества фиксированных формальных параметров в описании функции. Так, например, вызов приведенной в примере функции должен содержать два или более фактических параметра. Проверка соответствия типов фактических параметров типам формальных производится при компиляции только для фиксированных параметров.

Для переменных фактических параметров существует следующее правило преобразования типов при передаче их функции:

- фактические параметры типов *char* и *short* передаются как *int*;
- фактические параметры типа *float* передаются как *double*.

Все фактические параметры при вызове функции располагаются в оперативной памяти последовательно, друг за другом. За последним фиксированным параметром следуют переменные фактические параметры. Такой порядок размещения фактических параметров позволяет организовать доступ к переменным параметрам с помощью указателя. Этот метод сходен с методом обращения к элементам массива с помощью указателя, описанным в разделе, посвященном массивам (см. п. 2.10). Суть этого метода заключается в том, что вначале объявленному указателю присваивается значение адреса последнего фиксированного параметра. Далее, если произвести увеличение значения указателя на 1, он будет содержать адрес первого переменного параметра, при увеличении значения указателя на 2, он будет содержать адрес второго переменного параметра и т.д. Продемонстрируем этот метод доступа к переменным параметрам функции на примере. В приведенном примере функция имеет один фиксированный параметр типа *int*. Используем этот параметр для передачи функции при ее вызове нескольких передаваемых ей переменных параметров. В данном примере функция с переменным числом параметров *FunA( )* выполняет только то, что выводит на экран значения всех передаваемых ей фактических параметров.

```
#include<iostream.h>
void FunA(int Par1...)      //Описание функции с переменным числом параметров, только один из которых фиксированный.
```

```
{
 int *pEta=&Par1;          //Объявление указателя pEta с инициализацией
                           его адресом единственного фиксированного па-
                           раметра Par1.
 for(int k=0;k<Par1;k++)   //Оператор цикла выводит на экран значения
                             фактических переменных параметров.
 {
  pEta++;                  //Инкремент значения указателя. Результатом
                           этой операции является то, что указатель pEta
                           указывает на следующий переменный параметр.
  cout<<*pEta<<" ";        //На экран выводится  значение очередного фак-
                           тического переменного параметра.
 }
}
void main( )
{
 FunA(4,11, 22, 33, 44);
}
```

В результате работы программы на экран выводятся числа 11 22 33 44.

При использовании функций с переменным числом параметров нужно иметь в виду, что компилятор не контролирует содержимое указателя, используемого для доступа к параметрам функции. Так, например, в вышеприведенном примере фиксированному параметру *Par1* в операторе вызова функции можно задать значение, равное 5. Ошибки компилятор не выдаст, будет выполнено 5 циклов вывода на экран. При этом в первых четырех проходах цикла указатель *pEta* последовательно будет указывать на фактические параметры 11, 22, 33, 44. На последнем же, пятом проходе цикла указатель *pEta* будет указывать *на область памяти, выходящую за пределы области размещения фактических параметров*, на экран будет выводиться случайное значение, содержащееся в этом участке памяти.

К недостаткам функций с переменным числом параметров можно отнести то, что все переменные фактические параметры функции должны быть одного типа с тем, который имеет последний фиксированный параметр. Это связано с тем, что операция инкремента указателя увеличивает его реальное значение на величину, равную размеру типа указателя. И поэтому для попадания каждый раз, после инкремента указателя, на начало очередного фактического параметра, эти параметры должны быть одного типа, а именно типа используемого указателя.

Еще одной сложностью при разработке таких функций является то, что заранее не известно количество фактических параметров, которое будет использовано в операторе вызова этой функции. Поэтому обычно применяют один из двух возможных способов определения количества фактических параметров в операторе вызова функции:

- в один из фиксированных параметров записывается число фактических параметров в вызове функции;
- последнему переменному фактическому параметру задается какое-нибудь условное значение, например нулевое, которое будет являться признаком окончания списка фактических параметров.

Вышеприведенный фрагмент программы демонстрирует первый способ доступа к переменным параметрам функции.

Рассмотрим фрагмент программы, демонстрирующий второй способ определения количества переменных параметров функции. В функции *FunC( )* первый, фиксированный параметр используется только для определения адреса начала списка фактических параметров функции, его значение здесь может быть произвольным. Естественно, что и значение первого, фиксированного параметра, может использоваться в теле функции. В приведенном фрагменте программы это значения не используются в целях упрощения примера программы. Функция выводит на экран значения ее переменных параметров. Признаком окончания списка параметров является нулевое значение параметра. Нулевое значение параметра на экран в данном примере не выводится.

```
#include<iostream.h>
void FunC(int Par1...)
{
int *pEta=&Par1;          //Объявление указателя pEta с инициализацией
                          его адресом начала списка параметров функции
while(*(++pEta)!=0)       //Цикл вывода на экран значений переменных
                          параметров функции. В цикле производится
                          проверка значения каждого очередного параметра.
                          Если значение параметра не равно нулю, оно
                          выводится на экран, в противном случае
                          происходит выход из цикла.
 {
  cout<<*pEta<<" ";
 }
}

void main( )
{
FunC(1, 11, 22, 0);           //На экран выводятся числа 11 22
FunC(1, 11, 22, 33, 0);       //На экран выводятся числа 11 22 33
FunC(1, 11, 22, 33, 44, 0);   //На экран выводятся числа 11 22 33 44
}
```

2.14.9 Задачи для самопроверки по работе с функциями

Можно, например, использовать примеры из п.2.11.2 и п. 2.12.15.

2.15. Правила доступа и области видимости переменных

2.15.1 Исходные файлы и объявление переменных

Обычная СИ-программа представляет собой определение функции *main( )*, которая для выполнения необходимых действий вызывает другие функции. Приведенные выше примеры программ представляли собой один исходный файл, содержащий все необходимые функции для выполнения программы вцелом. Связь между функциями осуществлялась по данным посредством передачи параметров и возврата значений функций. Но компилятор языка С++ позволяет также разбить программу на несколько отдельных частей (исходных файлов), оттранслировать каждую часть отдельно, и затем объединить все части в один выполняемый файл при помощи редактора связей.

При такой структуре исходной программы функции, находящиеся в разных исходных файлах могут использовать ***глобальные внешние переменные.*** Все функции в языке С++ по определению внешние и всегда доступны из любых файлов. Например, если программа состоит из двух исходных файлов, как показано на рис.7.1, то функция *main( )* может вызывать любую из трех функций *fun1, fun2, fun3*, а каждая из этих функций может вызывать любую другую.

file1.c	file2.c
main ( ) ***{ ...*** ***}*** ***fun1***( ) ***{ ...*** ***}***	***fun2***( ) ***{ ...*** ***}*** ***fun3***( ) ***{ ...*** ***}***

Рис.2.15.1. Пример программы из двух файлов

Для того, чтобы определяемая функция могла выполнять какие либо действия, она должна использовать переменные. В языке С++ все переменные должны быть объявлены до их использования. Объявления устанавливают соответствие имени и атрибутов переменной, функции или типа. Определение переменной вызывает выделение памяти для хранения ее значения. *Класс* выделяемой памяти определяется *спецификатором класса памяти*, и определяет *время жизни* и *область видимости* переменной, связанные с понятием блока программы.

В языке С++ блоком считается последовательность объявлений, определений и операторов, заключенная в фигурные скобки. Существуют два вида блоков – составной оператор и определение функции, состоящее из составного оператора, являющегося телом функции, и предшествующего телу заголовка функции (в который входят имя функции, типы возвращаемого значения и формальных параметров). Блоки могут включать в себя составные операторы, но не определения функций. Внутренний блок называется вложенным, а внешний блок – объемлющим.

Время жизни - это интервал времени выполнения программы, в течение которого программный объект (переменная или функция) существует. Время жизни переменной может быть ***локальным*** или ***глобальным***. Переменная с глобальным временем жизни имеет распределенную для нее память и определенное значение на протяжении *всего времени* выполнения программы, начиная с момента выполнения объявления этой переменной. Переменная с локальным временем жизни имеет распределенную для него память и определенное значение *только во время* выполнения блока, в котором эта переменная определена или объявлена. При каждом входе в блок для локальной переменной распределяется новая память, которая освобождается при выходе из блока.

Все функции в C++ имеют глобальное время жизни и существуют в течение всего времени выполнения программы.

Область видимости – это часть текста программы, в которой может быть использован данный объект. Объект считается видимым в блоке или в исходном файле, если в этом блоке или файле известны имя и тип объекта. Объект может быть видимым в пределах блока, исходного файла или во всех исходных файлах, образующих программу. Это зависит от того, на каком уровне объявлен объект: на внутреннем, т.е. внутри некоторого блока, или на внешнем, т.е. вне всех блоков.

Если объект объявлен внутри блока, то он видим в этом блоке, и во всех внутренних блоках. Если объект объявлен на внешнем уровне, то он видим от точки его объявления до конца данного исходного файла.

Объект может быть сделан глобально видимым с помощью соответствующих объявлений во всех исходных файлах, образующих программу.

Спецификатор класса памяти в объявлении переменной может быть *auto*, *register*, *static* или *extern*. Если класс памяти не указан, то он определяется по умолчанию из контекста объявления.

Объекты классов *auto* и *register* имеют локальное время жизни. Спецификаторы *static* и *extern* определяют объекты с глобальным временем жизни.

При объявлении переменной на внутреннем уровне может быть использован любой из четырех спецификаторов класса памяти, а если он не указан, то подразумевается класс памяти *auto*.

Переменная с классом памяти *auto* имеет локальное время жизни и видна только в блоке, в котором объявлена. Память для такой переменной выделяется при входе в блок и освобождается при выходе из блока. При повторном входе в блок этой переменной может быть выделен другой участок памяти.

Переменная с классом памяти *auto* автоматически не инициализируется. Она должна быть проинициализирована явно при объявлении путем присвоения ей начального значения. Значение неинициализированной переменной с классом памяти auto считается неопределенным.

Спецификатор класса памяти *register* предписывает компилятору распределить память для переменной в регистре, если это представляется возможным. Использование регистровой памяти обычно приводит к сокращению времени доступа к переменной. Переменная, объявленная с классом памяти *register*,

имеет ту же область видимости, что и переменная *auto*. Число регистров, которые можно использовать для значений переменных, ограничено возможностями компьютера, и в том случае, если компилятор не имеет в распоряжении свободных регистров, то переменной выделяется память как для класса *auto*. Класс памяти *register* может быть указан только для переменных с типом *int* или указателей с размером, равным размеру *int*.

Переменные, объявленные на внутреннем уровне со спецификатором класса памяти *static*, обеспечиваю возможность сохранить значение переменной при выходе из блока и использовать его при повторном входе в блок. Такая переменная имеет глобальное время жизни и область видимости внутри блока, в котором она объявлена. В отличие от переменных с классом *auto*, память для которых выделяется в стеке, для переменных с классом *static* память выделяется в сегменте данных, и поэтому их значение сохраняется при выходе из блока.

Пример объявления переменной *i* на внутреннем уровне с классом памяти *static* ( ):

исходный файл file1.c	исходный файл file2.c
main( )	***fun2***( )
{ *...*	***{ static int*** *i=0; ...*
}	***}***
fun1( )	***fun3***( )
{ static int *i=0; ...*	***{ static int*** *i=0; ...*
}	***}***

В приведенном примере объявлены три разные переменные с классом памяти *static*, имеющие одинаковые имена *i*. Каждая из этих переменных имеет глобальное время жизни, но видима только в том блоке (функции), в которой она объявлена. Эти переменные можно использовать для подсчета числа обращений к каждой из трех функций.

Переменные класса памяти *static* могут быть инициализированы константным выражением. Если явной инициализации нет, то такой переменной присваивается нулевое значение. При инициализации константным адресным выражением можно использовать адреса любых внешних объектов, кроме адресов объектов с классом памяти *auto*, так как адрес последних не является константой и изменяется при каждом входе в блок. Инициализация выполняется один раз при первом входе в блок.

Переменная, объявленная локально с классом памяти extern, является ссылкой на переменную с тем же самым именем, определенную глобально в одном из исходных файлов программы. Цель такого объявления состоит в том, чтобы сделать определение переменной глобального уровня видимым внутри блока.

Пример объявления переменной *i*, являющейся именем внешнего массива длинных целых чисел, на локальном уровне:

исходный файл file1.c	исходный файл file2.c
main( )	***long*** *i[MAX]={0};*
{ *...*	***fun2****( )*

```
}
fun1( )
{ extern long i[ ]; ...
}
```

```
{ ...
}
fun3( )
{ ...
}
```

Объявление переменной *i[ ]* как *extern* в приведенном примере делает ее видимой внутри функции *fun1*. Определение этой переменной находится в файле *file2.c* на глобальном уровне и должно быть только одно, в то время как объявлений с классом памяти extern может быть несколько.

Объявление с классом памяти *extern* требуется при необходимости использовать переменную, описанную в текущем исходном файле, но ниже по тексту программы, т.е. до выполнения ее глобального определения. Следующий пример иллюстрирует такое использование переменной с именем *st*.

```
main( )
{ extern int st[ ]; ...
}
static int st[MAX]={0};
 fun1( )
{   ...
}
```

Объявление переменной со спецификатором *extern* информирует компилятор о том, что память для переменной выделять не требуется, так как это выполнено где-то в другом месте программы.

При объявлении переменных на глобальном уровне может быть использован спецификатор класса памяти *static* или *extern*, а так же можно объявлять переменные без указания класса памяти. Классы памяти *auto* и *register* для глобального объявления недопустимы.

Объявление переменных на глобальном уровне – это или определение переменных, или ссылки на определения, сделанные в другом месте программы. Объявление глобальной переменной, которое инициализирует эту переменную (явно или неявно), является определением переменной. Определение на глобальном уровне может задаваться в следующих формах:

1. Переменная объявлена с классом памяти *static*. Такая переменная может быть инициализирована явно константным выражением, или по умолчанию нулевым значением. То есть обявления *static int i=0* и *static int i* эквивалентны, и в обоих случаях переменной *i* будет присвоено значение *0*.

2. Переменная объявлена без указания класса памяти, но с явной инициализацией. Такой переменной по умолчанию присваивается класс памяти *static*. То есть объявления *int i=1* и *static int i=1* будут эквивалентны.

Переменная объявленная глобально видима в пределах остатка исходного файла, в котором она определена. Выше своего описания и в других исходных файлах эта переменная невидима (если только она не объявлена с классом *extern*).

Глобальная переменная может быть определена только один раз в пределах своей области видимости. В другом исходном файле может быть объявлена другая глобальная переменная с таким же именем и с классом памяти *static*, конфликта при этом не возникает, так как каждая из этих переменных будет видимой только в своем исходном файле.

Спецификатор класса памяти *extern* для глобальных переменных используется, как и для локального объявления, в качестве ссылки на переменную, объявленную в другом месте программы, т.е. для расширения области видимости переменной. При таком объявлении область видимости переменной расширяется до конца исходного файла, в котором сделано объявление.

В объявлениях с классом памяти *extern* не допускается инициализация, так как эти объявления ссылаются на уже существующие и определенные ранее переменные.

Переменная, на которую делается ссылка с помощью спецификатора extern, может быть определена только один раз в одном из исходных файлов программы.

2.15.2 Объявления функций

Функции *всегда определяются глобально*. Они могут быть объявлены с классом памяти *static* или *extern*. Объявления функций на локальном и глобальном уровнях имеют одинаковый смысл.

Правила определения области видимости для функций отличаются от правил видимости для переменных и состоят в следующем.

1. Функция, объявленная как *static*, видима в пределах того файла, в котором она определена. Каждая функция может вызвать другую функцию с классом памяти *static* из своего исходного файла, но не может вызвать функцию определенную с классом *static* в другом исходном файле. Разные функции с классом памяти *static* имеющие одинаковые имена могут быть определены в разных исходных файлах, и это не ведет к конфликту.

2. Функция, объявленная с классом памяти *extern*, видима в пределах всех исходных файлов программы. Любая функция может вызывать функции с классом памяти *extern*.

3. Если в объявлении функции отсутствует спецификатор класса памяти, то по умолчанию принимается класс *extern*.

Все объекты с классом памяти *extern* компилятор помещает в объектном файле в специальную таблицу внешних ссылок, которая используется редактором связей для разрешения внешних ссылок. Часть внешних ссылок порождается компилятором при обращениях к библиотечным функциям C++, поэтому для разрешения этих ссылок редактору связей должны быть доступны соответствующие библиотеки функций.

2.15.3 Инициализация глобальных и локальных переменных

При инициализации необходимо придерживаться следующих правил:

1. Объявления, содержащие спецификатор класса памяти *extern*, не могут содержать инициаторов.

2. Глобальные переменные всегда инициализируются, и если это не сделано явно, то они инициализируются нулевым значением.

3. Переменная с классом памяти *static* может быть инициализирована константным выражением. Инициализация для них выполняется один раз перед началом программы. Если явная инициализация отсутствует, то переменная инициализируется нулевым значением.

4. Инициализация переменных с классом памяти *auto* или *register* выполняется всякий раз при входе в блок, в котором они объявлены. Если инициализация переменных в объявлении отсутствует, то их начальное значение не определено.

5. Начальными значениями для глобальных переменных и для переменных с классом памяти *static* должны быть константные выражения. Адреса таких переменных являются константами и эти константы можно использовать для инициализации объявленных глобально указателей. Адреса переменных с классом памяти *auto* или *register* не являются константами и их нельзя использовать в инициаторах. Например:

```
int global_var;
int func(void)
{ int local_var;                          /* по умолчанию auto  */
static int *local_ptr=&local_var;     /*  так неправильно   */
static int *global_ptr=&global_var; /*  а так правильно   */
register int *reg_ptr=&local_var;    /*  и так правильно   */
}
```

В приведенном примере глобальная переменная *global_var* имеет глобальное время жизни и постоянный адрес в памяти, и этот адрес можно использовать для инициализации статического указателя *global_ptr*. Локальная переменная *local_var*, имеющая класс памяти *auto* размещается в памяти только на время работы функции *func*, адрес этой переменной не является константой и не может быть использован для инициализации статической переменной *local_ptr*. Для инициализации локальной регистровой переменной *reg_ptr* можно использовать неконстантные выражения, и, в частности, адрес переменной *local_ptr.*

2.15.4 Управление памятью

Сказанное выше говорит о том, что в C/C++ существует механизм управления памятью. Этот механизм предполагает три способа выделения памяти для используемых в программе данных: *автоматический, статический и динамический.*

2.15.4.1 Автоматические переменные

Если переменные объявлены внутри тела функции, то всякий раз, когда вызывается эта функция, под переменную, в зависимости от ее типа, автоматически отводится память. Когда функция завершается, память автоматически освобождается. Поэтому такие переменные называются ***автоматическими***.

Аналогично, автоматическими переменными будут и переменные объявленные и инициализированные внутри блока (последовательности операторов, заключенных в фигурные скобки). Память для них выделяется при входе в блок и освобождается при его завершении.

2.15.4.2 Статические переменные

Для всех переменных, объявленных вне функции, память выделяется статически, один раз вначале программы. Переменная уничтожается, освобождая память только тогда, когда выполнение программы завершается. Переменная может быть объявлена как статическая и внутри функции, с использованием ключевого слова *static*. В этом случае, один раз инициализированная она не изменяется между вызовами функции. Например:

```
double glob;
void fun( )
{
  static bool x=false;
   if(!x)
     x=true;
……….....
}
```

В этом примере статическими являются переменные *glob* и *x*. Первая существует до тех пор, пока программа не завершится. Вторая переменная –*x*, объявленная внутри функции, как принимающая значение *false* и инициализированная при первом входе как *true*, сохранит это значение, сколько бы теперь не проводилось вызовов данной функции. Память под нее теперь также будет освобождена только по завершении программы в целом.

2.15.4.3 Модель памяти программы

Рассматривая классическую архитектуру вычислительной машины, нельзя обойти вниманием важнейший компонент, предназначенный для хранения информации – оперативное запоминающее устройство (ОЗУ).

Рассмотрим упрощенную схему распределения оперативной памяти (ОП), поддерживаемую современными ПК. С введением 32 разрядов для нумерации адресов доступной памяти исчезло многообразие и сложность «моделей памяти», а вместе с ними и адреса типа *near*, *far* и *huge*[7].

В современных операционных системах (ОС) каждой запущенной программе предоставляется свое адресное пространство размером, равным

$V_{mem}=2^{32}=2^2\cdot2^{10}\cdot2^{10}\cdot2^{10}=4294967296=4$ *Гбайт*.

Естественно, что многие компьютеры просто физически не имеют такого объема ОЗУ и, следовательно, не могут предоставить его программе. Кроме то-

[7] Подробнее о моделях памяти и указанных типах смотри, например: Романовская Л.М. и др. Прграммирование в среде Си для ПЭВМ ЕС. – М.: Финансы и статистика, 1991. с.132-151.

го, в многозадачных ОС одновременно могут выполняться десятки и сотни программ, каждая из которых может претендовать на указанный размер ОП. Выходом из такого положения служит предоставление каждой программе не реального, а виртуального адресного пространства, образующегося с помощью специальных программно-аппаратных преобразований.

В результате модель памяти, которая выделяется каждой программе, выглядит единообразно и, по большому счету, не зависит от установленного объема ОЗУ в конкретном ПК. Такой подход имеет два основных преимущества: во-первых, программисту не надо задумываться над особенностями строения памяти в зависимости от конфигурации или загрузки ПК; во-вторых, программный код изолируется в своем адресном пространстве и не может повлиять на работоспособность других программ, выполняющихся на этом же ПК.

Для понимания основных процессов, обсуждаемых в рамках нашего курса достаточно представить память, выделяемую программе в виде рисунка, приведенного ниже. С использованием такой модели памяти программисту достаточно только указать адрес байта, чтобы получить к нему доступ.

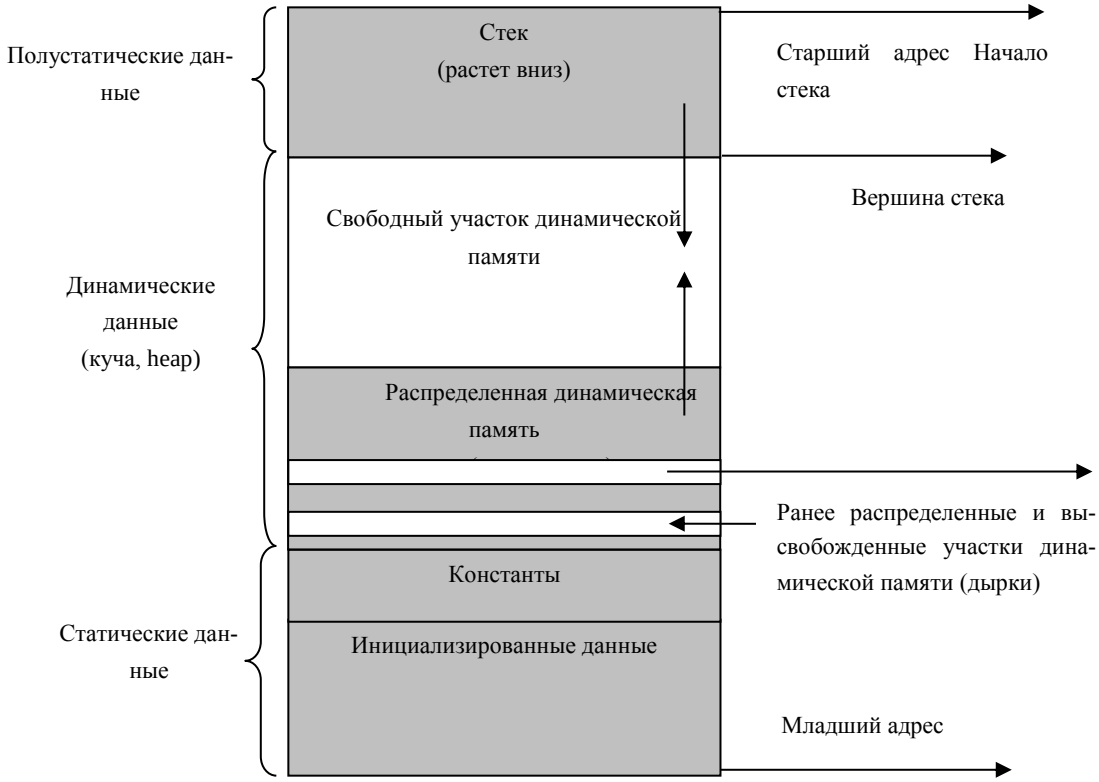


Рис. 1. Упрощенная модель памяти программы

Язык C++ имеет развитые средства управления динамической памятью. Рассмотрим основные из них.

2.15.4.4 Динамические структуры

В динамически распределяемой области могут использоваться те же самые типы данных, что и в статической, в том числе, определяемые программи-

стом. Основным отличием технологии использования динамических данных является то, что создание и удаление переменных (выделение и высвобождение памяти) может происходить не только перед началом выполнения и, соответственно, после завершения программы, а в произвольный момент времени.

Рассмотрим простейший случай использования динамической памяти на примере задачи:

1) выделить память для хранения целого числа;
2) вывести адрес выделенной памяти и значение, хранящееся в ней;
3) записать в выделенную память число 7;
4) вывести адрес выделенной памяти и значение, хранящееся в ней;
5) высвободить занятую память.

Для решения задачи используем следующие средства и приемы:

- в статической области памяти выделим переменную «с» типа указатель на целое (строка5);
- в строке 6:
  1. автоматически – с помощью оператора *sizeof( )* - определим размер памяти, необходимый для хранения целого числа;
  2. выделим память этого размера – *malloc( );*
  3. осуществим операцию приведения типа выделенной памяти - *(int*)* -укажем, что будем использовать её для хранения целых чисел.
- в строках 7 и 9 осуществим вывод требуемой информации;
- в строке 8 занесем в выделенный участок значение 7;
- высвободим выделенную память в строке 10.

Листинг программы:

```
1   #include <stdio.h>
2   #include <malloc.h>
3   void main( )
4   {
5     int *c;
6     c=(int*) malloc(sizeof(int));
7     printf("c=%x, *c=%d \n",c,*c);
8     *c=7;
9     printf("c=%x, *c=%d \n",c,*c);
10    free(c);
11  }
```

Состояние памяти программы, перед выполнением 6-й строки, иллюстрируется рисунком 11.2.

Переменная *c* расположена в статической области по адресу *0x0012ff7c* и содержит значение *0x00000001*, которое ссылается на область памяти, недоступную для программы, что отображается знаками вопроса в окне «*Memory*» и сообщениями «*CXX0030*: *Error: expression cannot be evaluated*» (Ошибка: переменная не может быть вычислена») в окне «*Watch*».

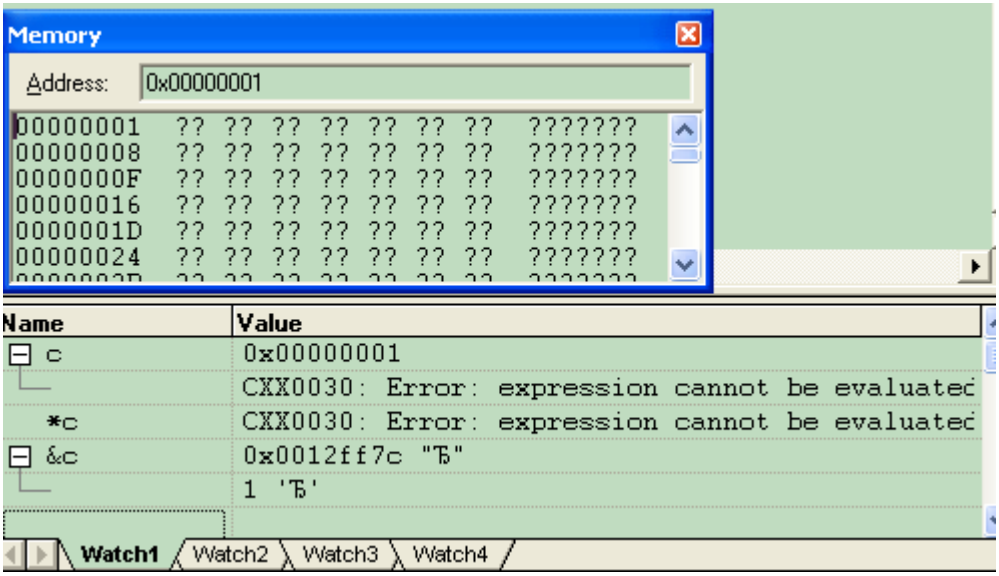


Рис. 2 Состояние памяти программы перед выполнением 6-й строки

После выполнения 6-й строки картина меняется (рис. 3). Программе выделена память по адресу *0x00420e10*, который теперь содержится в переменной *c*. Выделенная память заполнена символами *CD* и может быть проинтерпретирована как число (- *842150451*).

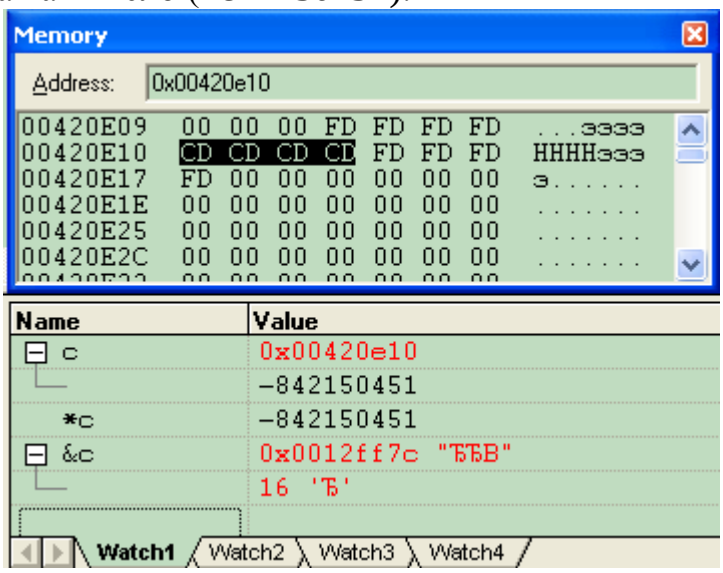


Рис. 3. Состояние памяти программы после выполнения 6-й строки

Выполнение строки 7 приводит к выводу на экран требуемых значений: адреса выделенного участка в шестнадцатеричном виде и его содержимого (интерпретируемого как целое) – в десятичном (рис. 4).

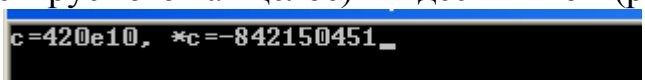


Рис. 4. Результат выполнения 7-й строки

Результатом 8 строки будет занесение значения в выделенный участок памяти (рис. 5) (напомним, что в соответствии с принятой системой кодирования осуществляется внутрибайтная перестановка разрядов числа).

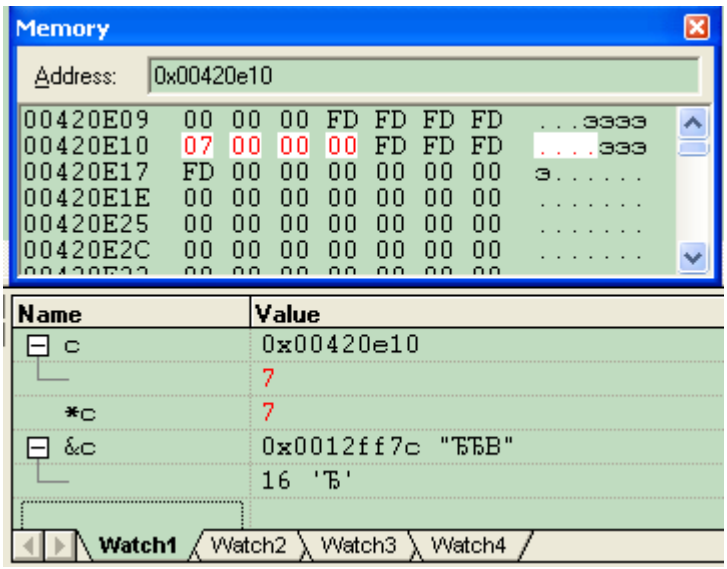


Рис. 5. Занесение значения в выделенную область памяти.

Что, естественно, приведет к изменению выдаваемой информации (рис.6).

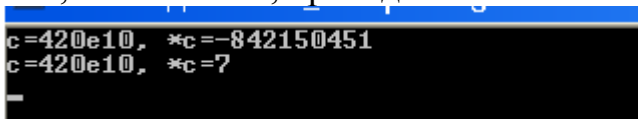


Рис. 6. Результат выполнения 9-й строки

Выполнение последней операции (рис. 7) приводит к высвобождению занятой памяти, заполнению её символами «*DD*», но, ***не изменяет состояние указателей, которые ссылались на эту область***, что может приводить к ошибкам типа «Нарушения защиты данных» или неправильным результатам вычислений.

Например, если после десятой строки сделать еще один вывод на экран, то полученный результат не приведет к останову программы, но результат будет непредсказуем (рис. 8).

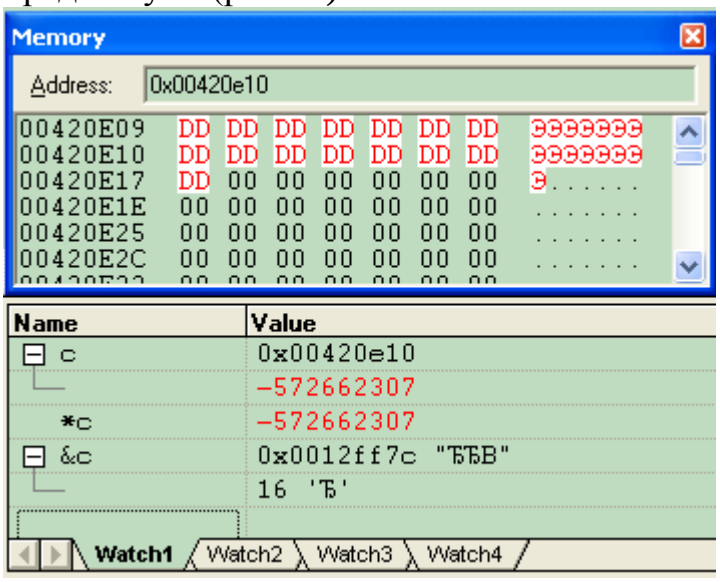


Рис. 7. Результат высвобождения памяти.

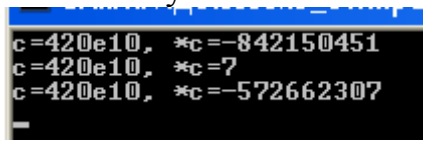


Рис. 8. Результат некорректной операции вывода.

2.15.4.5 Динамическое управление памятью

В C++ используется два способа работы с динамической памятью. Первый способ достался от C и использует семейство функций *malloc* в заголовочном файле <*malloc.h*>. Второй способ использует операции *new* и *delete*.

Первый способ. Функции заголовочного файла <*malloc.h*>

Имя функции	Синтаксис	Описание
calloc	*void *calloc(unsigned n, unsigned m)*	Возвращает указатель на начало области оперативной памяти для размещения *n* элементов по *m* байт каждый. При неудачном завершении возвращает значение *NULL*.
coreleft	*unsigned coreleft(void)* – для моделей памяти *tiny, small, medium*; *unsigned long coreleft(void)* – для других моделей	Возвращает значение объема неиспользуемой памяти.
free	*void free(void *bl)*	Освобождает ранее выделенный блок оперативной памяти с адресом первого байта *bl*.
malloc	*void *malloc(unsigned s)*	Возвращает указатель на блок памяти длиной в *s* байт. При неудачном завершении возвращает значение *NULL*.
realloc	*void *realloc(void *bl, unsigned ns)*	Изменяет размер ранее выделенной памяти с адресом начала *bl* на *ns* байт.

Второй способ. Память для величины какого-либо типа можно выделить, выполнив операцию ***new***. В качестве операнда выступает название типа, а результатом является адрес выделенной памяти. Например:

```
long *x;
x=new long;            //создать новое целое число
Object *ob;
ob=new Object;         //создать новый объект типа Object
int *m=new int (10);   //инициализация выделенной памяти числом 10
int *q=new int [10];   //выделение памяти для массива из 10 элементов
                       и запись начального адреса участка памяти в переменную q.
```

Созданные таким образом объекты существуют в памяти до тех пор, пока для их уничтожения не будет явно применена операция *delete*, операндом которой должен быть адрес, возвращаемый операцией new.

delete x;
delete ob;
delete m;
delete [ ] q;

Динамическое распределение памяти используется в тех случаях, когда заранее неизвестно, сколько объектов потребуется в программе. С помощью этого механизма можно гибко управлять временем жизни переменных, не объявляя их вначале программы, как глобальные, но, тем не менее, сохранять и использовать нужные данные в памяти до конца программы.

2.15.5 Задачи для самопроверки по доступу и области видимости переменных

ПРИЛОЖЕНИЯ
ПРИЛОЖЕНИЕ 1. ПРЕПРОЦЕССОР
ПРИЛОЖЕНИЕ 2. КЛЮЧЕВЫЕ СЛОВА ЯЗЫКА СИ
ПРИЛОЖЕНИЕ 3. ASCII-КОДЫ

Например: «?»=>$30_{(16)}+F_{(16)} = 3F_{(16)} = 63_{(10)}$

		0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	ull		pace													
	sp															
	ab															
		sk														
																lank

ПРИЛОЖЕНИЕ 4. БИБЛИОТЕЧНЫЕ ФУНКЦИИ

ЧАСТЬ 2. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ НА C/C++. ЭФФЕКТИВНЫЕ АЛГОРИТМЫ.

1 ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

1.1 КЛАССЫ И ОБЪЕКТЫ

Основные понятия

Практика использования языка C++ показала, что он хорошо подходит для разработки программ средней сложности. С увеличением длины программы она становилась трудно воспринимаемой и трудно управляемой. Необходимы были новые подходы к программированию. И они были разработаны. В 1979 году появился язык Си с классами, а в 1997 году был принят международный стандарт C++ (ISO/IES 14882). Разработчиком C++ является Бьерн Страуструп. C++ развивался на платформе языка Си и поэтому полностью включил его в себя. Язык Си реализовывал концепцию процедурного программирования, согласно которой каждой задаче приводился в соответствие свой блок данных и своя прикладная программа (функция). Вся программа представляла собой набор таких процедур. Язык C++ создан для поддержки концепции объектно-ориентированного программирования (ООП).

Ключевым понятием ООП является объект. Под объектом понимается переменная особого типа. Этот особый тип в C++ создан на базе типа данных структура (*struct*) и называется класс (*class*). Класс отличается от структуры языка Си тем, что в него помимо собственно данных различных типов входят также функции, обрабатывающие эти данные. Объявленная переменная типа *class* есть объект. Такое объединение данных и обрабатывающих их функций называется ***инкапсуляцией***. Она защищает данные от внешнего вмешательства или неправильного использования. В то же время объект может оснащаться средствами программной связи с другими объектами.

Данные и функции, входящие в класс, называются *представителями* или *членами* этого класса. Члены класса разделяются на закрытые (*private*) и открытые (*public*). К закрытым членам обращение возможно только внутри класса и из других частей программы они недоступны. Обычно, таким образом защищаются данные от внешнего вмешательства. К открытым членам класса может производиться обращение извне.

По умолчанию все члены класса являются закрытыми. Но такая конструкция не представляет практического интереса, т. к. члены класса не могут быть использованы в программе. Для открытия доступа к ним обязательно необходимы *public*-члены, в качестве которых используют функции, оперирующие закрытыми членами класса. Таким образом, *public*-функции являются посредниками между *private*-членами класса и другими частями программы.

Объявление классов и создание объектов

Класс объявляется с помощью ключевого слова *class*, за которым указывается имя класса. Затем в фигурных скобках перечисляются члены класса. Синтаксис объявления класса похож на синтаксис объявления структуры. Рассмотрим пример объявления.

```
class Myclass                 // класс под именем myclass
{
        int a;                // закрытая переменная
public:                       // ключевое слово (для открытых членов)
        void Set_a(int n);    // функции – открытые члены класса -
         int Get_a( );        // для доступа к закрытой переменной
};
```

В данном примере объявлен класс *Myclass*, который содержит одну закрытую переменную и две функции, которые являются открытыми членами класса. Перед закрытой переменной ключевое слово *private* может не ставиться. Оно предполагается по умолчанию. Перед открытыми членами записывается ключевое слово *public* с двоеточием. В конце описания класса, после закрывающейся фигурной скобки, ставится точка с запятой.

Функции *Set_a( )* и *Get_a( )* только объявлены внутри класса, но еще не определены (не описаны). Эти функции, являясь открытыми членами класса, должны обеспечивать доступ к закрытым членам. Доступ к закрытой переменной предполагает две операции:

- присваивание переменной определенного значения;
- получение значения переменной.

Таким образом, для «обслуживания» одной закрытой переменной требуются две функции - открытые члены класса. При определении функций необходимо указать не только функциональное их назначение, но и принадлежность к классу, в котором они объявлены. Рассмотрим на примере, как это делается.

```
void Myclass::Set_a(int n)
{
        a=n;
}
int Myclass::Get_a( )
{
        return a;
}
```

Для обозначения принадлежности функции к классу достаточно лишь перед её именем указать имя класса и два двоеточия, которые называются оператором расширения области видимости.

Функция *Set_a( )* предназначена для присваивания целочисленных значений закрытой переменной a. Поэтому она имеет один параметр, через который и передаются эти значения; функция, в свою очередь, ничего не возвращает. Вторая функция – *Get_a( )* – служит для получения значения закрытой переменной. Поэтому она имеет возвращаемое значение, но при её вызове аргументы ей не передаются.

Описание функций – членов можно производить и внутри класса. Тогда они называются встраиваемыми. Такой способ удобен, когда функции не объёмные и за счёт этого не затеняют структуру класса. Ниже приведён пример описания того же класса, но уже со встраиваемыми функциями.

```
class Myclass
{
        int a;
public:
        void Set_a(int n) {a=n;}
         int Get_a( ) {return a;}
};
```

Как можно видеть, запись получилась более компактной.

Объявление класса *Myclass* не задает ни одного объекта типа *Myclass*. Оно определяет только тип объекта (шаблон). Чтобы создать объект, необходимо указать имя класса, как спецификатор типа данных, после которого записать имя объекта. Например, в следующей строке заданы два объекта типа *Myclass* .

```
Myclass obj1, obj2;
```

Создание объекта связано с выделением под него памяти. При объявлении же класса память не выделяется.

После того, как объекты созданы, можно обращаться к открытым членам класса. Как и при работе со структурами, обращение к открытым членам класса производится через точку (.). В следующем фрагменте показано обращение к функциям- членам класса *Myclass*, который используется в данном параграфе.

```
obj1.Set_a(10);              // обращение к функции Set_a( ) первого объекта
obj2.Set_a(22);              // обращение к функции Set_a( ) второго объекта
cout<<obj1.Get_a( )<<«\t»; //обращение к функции Get_a( ) первого объекта
cout <<obj2.Get_a( )<<«\n»;//обращение к функции Get_a( ) второго объекта
```

Функции *Set_a( )* первого объекта в качестве аргумента передаётся численное значение 10, которое присваивается переменной a этого объекта. Аналогично, переменной *a* второго объекта присваивается значение *22*. Каждый объект имеет свою переменную *a*, свою функцию *Set_a( )* и функцию *Get_a( )*. По-

этому объекты еще называют экземплярами классов. В третьей строке фрагмента на экран выводится значение функции *Get_a( )* первого объекта, а в четвертой – второго объекта. После выполнения указанных операций на экране будет: *10 22*.

Массивы объектов

Объекты – это переменные, и они имеют те же возможности и признаки, что и переменные любых других типов. Поэтому объекты могут объединяться в массивы. Синтаксис объявления массива объектов совершенно аналогичен тому, который используется для объявления массива переменных любого другого типа. То же касается и доступа к элементам массива.

Пример с массивом объектов.

```
#include<iostream.h>
class one                       //  имя класса
{
        int a;                  // закрытая переменная
public:
        void set_a(int n) {a=n;} // функция доступа к закрытой переменной
         int get_a( ) {return a;} // функция доступа к закрытой переменной
};

void main( )                    // главная функция
{
        one obj[4];             // создание массива из 4-х объектов
        int i;
        for(i=0;i<4;i++)        // инициализация переменных объектов
                obj[i].set_a(i);
        for(i=0;i<4;i++)        // вывод значений переменных объектов
           cout<<obj[i].get_a( )<<" ";
}
```

В данном примере объявлен класс *one*, в котором имеется закрытая переменная и две функции доступа к ней. В главной функции создан массив из 4-х объектов, в котором циклической операцией производится инициализация закрытой переменной каждого элемента массива с помощью функции *set_a( )*. Во второй циклической операции производится вывод на экран значения переменной каждого объекта путем использования функции *get_a( )*.

Указатели на объекты

На объекты можно создавать указатели. В этом случае доступ к члену объекта производится с помощью оператора стрелка (->), как и в случае работы со структурой. Объявляется указатель на объект так же, как и указатель на переменную любого другого типа. Для этого следует задать имя класса этого объекта, а затем имя переменной со звездочкой перед ним. Для получения ад-

реса объекта перед ним необходим оператор **&**, точно так же, как это делается для получения адреса переменной другого типа. Если для указателя, например, производится операция инкремент, то он после этого будет указывать на объект такого же типа.

Рассмотрим пример использования указателя на объект.

```
#include<iostream.h>
class two                    // имя класса
{
        int a;               // закрытая переменная
public:
        void set_a(int n) {a=n;} // функция доступа к закрытой переменной
         int get_a( ) {return a;}   // функция доступа к закрытой переменной
};

void main( )
{
        two ob;                      // создание объекта
        ob.set_a(25);                // инициализация переменной объекта
        two *p;                      // создание указателя на объект
        p=&ob;                       // передача адреса ob в p
        cout<<"First rezult: "<<ob.get_a( )<<«\n»;
        cout<<"Second rezult: "<<p->get_a( )<<«\n»;
        cout<<"Adress: "<<p<<«\n»;
}
```

В примере класс *two* имеет закрытую переменную и две функции доступа к ней. После создание объекта, в нём инициализируется закрытая переменная. Затем объявляется указатель на объект класса *two*. Здесь важно понимать, что объявление указателя не создает объект, а только создает указатель на него. В следующей строке адрес объекта передается переменной *p* (указателю). В заключении программы показано обращение к членам объекта через имя объекта и через указатель.

Конструкторы и деструкторы

Среди членов класса могут быть две особенные функции. Их особенность заключается в том, что обращение к ним производится по умолчанию. Первая функция называется ***конструктор***. Она вызывается автоматически при создании объекта. Её назначение состоит в том, чтобы присвоить начальные значения (инициализировать) переменным объекта. Вторая функция называется ***деструктор***. Она служит для ликвидации последствий использования объекта (например, для освобождения памяти). Поэтому она автоматически вызывается при удалении объекта. Конструктор имеет то же имя, что и класс, частью которого он является. Имя деструктора также совпадает с именем класса, но перед ним ставится символ ~ (тильда). Конструктор и деструктор не возвращают ни-

каких значений. Рассмотрим пример использования кострукторов и деструкторов в программах.

```
#include<iostream.h>
class abc                    // объявление класса abc
{
        int a;               // закрытая переменная
public:
        abc( );              // конструктор
       ~abc( );              // деструктор
  void show( );
};

abc::abc( )                  // описание конструктора
{
        cout<<"I am constructor \n";
         a=10;
}
abc::~abc( )                       // описание деструктора
{
        cout<<"I am destructor \n";
}
void abc::show( )          // описание функции- члена класса
{
        cout<<a<<"\n";
}

void main( )
{
        abc obj;             // создание объекта
        obj.show( );
}
```

В данном примере класс *abc* содержит одну переменную и три функции, две из которых являются конструктором и деструктором. Следует отметить, что перед конструктором и деструктором слово *void* не ставится (оно предполагается по умолчанию). Определение функций производится вне класса. В главной функции создается объект *obj*. В момент создания объекта вызывается конструктор. Затем вызывается функция *show( )* объекта и программа завершается. При окончании программы объект перестает существовать, что автоматически приводит к вызову деструктора.

После выполнения программы на экран будет выведено:

I am constructor
10
I am destructor.

В этом примере конструктор не содержал параметров. Но обычно параметры присутствуют, т.к. основное назначение конструкторов – производить инициализацию переменных объекта. В следующем примере используется конструктор с параметрами.

```
#include<iostream.h>
class black                    // объявление класса black
{
        int a,b;               // закрытые переменные
public:
        black(int x, int y)    // конструктор с параметрами
        {
                a=x;
                b=y;
        }
        void show( )           // функция- член класса
        {
                cout<<a<<«\t»<<b;
        }
};

void main( )
{
        black ob(20,100);      // создание объекта с инициализацией пе -
        ob.show( );            // ременных
}
```

Класс *black* содержит две закрытые переменные и две функции, одна из которых конструктор с параметрами. Значения, которые передаются конструктору, будут присвоены закрытым переменным. При создании объекта, в этом случае, после его имени в скобках через запятую указываются аргументы – численные значения, которые будут переданы параметрам *x* и *y*. На экран будет выведено: *20 100*.

Объекты в качестве аргументов и значений функций

Объекты можно передавать функциям в качестве аргументов точно так же, как передаются данные других типов. Для этого параметр функции объявляется типом *class*, а в качестве аргумента при вызове функции используется объект этого класса. Рассмотрим пример.

```
#include<iostream.h>
class simple                   // имя класса
{ int i;                       // закрытая переменная
  public:
        simple(int n)          // конструктор
```

```
        {i=n;}
  int get( )
  { return i;}
};
int sqr(simple obj)                // функция – не член класса, которой пе -
{ return obj.get( )*obj.get( );}  // редается объект

void main( )
{
        simple ob1(2),ob2(10);   // создание двух объектов
        cout<<sqr(ob1)<<"  "<<sqr(ob2);
}
```

В этом примере объявляется класс *simple*, который содержит одну переменную, конструктор и функцию, возвращающую значение переменной. Функция *sqr( )* в качестве параметра имеет объект типа *simple*, получает из него значение переменной *i* и возвращает её квадрат. Здесь необходимо отметить, что функция *sqr( )* не является членом класса. В главной функции создаются два объекта с инициализацией. На экран выводится значение квадрата переменной каждого объекта: *4 100*.

Возвращение объектов функциям производится так же, как и значений других типов данных. Для этого необходимо объявить функцию так, чтобы её возвращаемое значение имело имя типа класса, а в операторе *return* указать имя возвращаемого объекта. Возвращенный объект должен быть скопирован в месте вызова функции в объект такого же класса; другими словами, операция присваивания может производиться только между объектами одного класса. Рассмотрим пример, полагая, что класс *simple* уже объявлен.

```
simple input( )     // функция возвращает объект типа simple
{
        simple locob(25);  // создание объекта типа simple
        return locob;        // возвращение объекта функцией
}
void main( )
{
        simple obj(10);      // создание объекта типа simple
        cout<<obj.get( )<<«\n»;
        obj=input( );          // возвращенный объект копируется в obj
        cout<<"New i = "<<obj.get( );
}
```

В классе имеется конструктор с параметром *int n*. При создании объекта *obj(10)* в функции *main( )* указанному конструктору передается число *10*, которое затем выводится на экран. Затем объекту *obj* присваивается результат рабо-

ты функции *input( ).* При вызове этой функции, в ней создается локальный объект *locob*, который сопровождается работой конструктора, в результате чего переменной *i* присваивается значение *25*. Функция возвращает объект *locob*, который копируется в объект *obj*, и переменная *i* в *obj* принимает значение, равное *25*. Результат работы программы будет:

10

New i =25.

Дружественные функции

При программировании возникают ситуации, когда требуется функция, которая имела бы доступ к закрытым членам класса, но сама не являлась бы членом этого класса. Для этого введены дружественные функции.

Дружественная функция определяется в программе как обычная функция. Чтобы показать, к какому классу она дружественна, её объявляют в этом классе с ключевым словом *friend*. Особенность дружественной функции состоит в следующем. Дружественная функция не является членом класса. К закрытым членам класса она не может обращаться непосредственно. Она это может сделать только через объекты этого класса. Поэтому в качестве параметра у дружественной функции должен указываться объект со спецификатором дружественного класса.

Рассмотрим использование дружественной функций на примере.

```
#include<iostream.h>
class data
{ int a,b;
public:
        data(int x,int y)          // конструктор с параметрами x и y
        { a=x; b=y;}
        friend int diff(data ob);  // дружесвенная функция
};

int diff(data ob)       // описание дружественной функции
{ return ob.a-ob.b;}

void main( )
{ int d;
  data obj(8,5);
  d=diff(obj);    // вызов дружественной функции
  cout<<"Difference = "<<d;
}
```

В классе *data* объявлены две переменные *a* и *b*, а также конструктор с параметрами. Кроме того, объявлена дружественная функция *diff(data ob).* Необходимо обратить внимание на порядок её объявления: сначала записывается ключевое слово *friend*, затем указывается тип возвращаемого значения функ-

ции, после чего – имя функции и в скобках – класс и имя объекта. В главной функции создается объект; при этом запускается конструктор и инициализируются переменные *a* и *b*. Созданный объект передается в функцию *diff( )*, которая вычисляет разность *a* и *b*. Разность возвращается переменной *d* и затем выводится на экран.

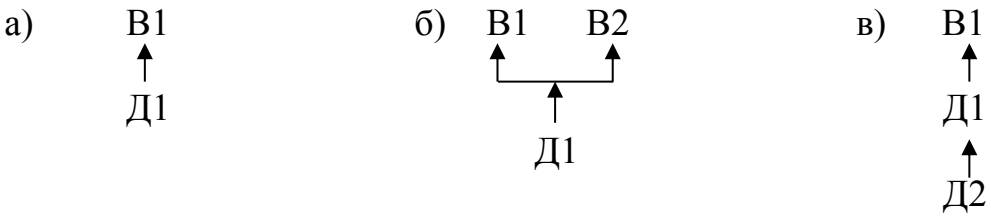
1.2 НАСЛЕДОВАНИЕ

Иерархия классов

Иерархия классов возникает в программе за счет механизма наследования, реализованного в С++.

Наследование – это процесс, посредством которого один объект может приобретать свойства другого. Так как объекты принадлежат к классам, то между классами в этом случае устанавливаются определенные взаимоотношения. Класс, который делегирует свойства называется базовым (*base*), а который наследует свойства – производственным (*derived*). У одного базового класса может быть несколько производных, и наоборот, один производный класс может наследовать характеристики нескольких базовых классов.

Схемные примеры наследования:



В примере *а*) показана простейшая схема подчиненности с одним базовым *В1* и одним производным *Д1* классом. В примере *б*) показан один производный *Д1* класс от двух базовых *В1* и *В2*. На схеме *в*) показано, что *В1* есть базовый класс для *Д1*, который в свою очередь является базовым для *Д2*. В этом последнем случае *В1* называется для *Д2* косвенно-базовым. Взаимосвязи классов в программе образуют иерархию классов.

При указании наследования соблюдаются следующие правила.

Базовый класс описывается обычным образом. Например:

```
class base
{
//содержание класса base
};
```

Для указания наследования после имени производного класса ставится двоеточие и указывается имя базового класса со спецификатором доступа (*private, public*). Для примера *а*) описание будет выглядеть следующим образом:

```
class derived: public base
{
//содержание класса derived
```

};

Спецификатор выбирается в зависимости от того, как предполагается использовать члены класса *base*.

Для примера *б*) описание производного класса будет выглядеть, например, так:

***class** derived: **public** B1, **public** B2*

{

//содержание класса derived

};

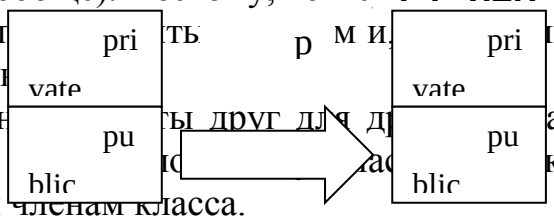
В примере *в*) для производного класса *Д2* базовым будет класс *Д1*, поэтому описание *Д2* соответствует схеме *а*).

Преимущества, получаемые от механизма наследования:

- повторное использование данных и функций, ранее созданных в базовом классе, без их дублирования в производном классе;
- получение более компактной и обозримой программы;
- повышение степени защищенности закрытых членов базового класса;
- адекватность программной модели предметам реального мира.

Взаимоотношения между членами классов и доступ к ним.

Базовый класс является первым в иера не знает» ск о будет производных классов (или не будет вообще). Поэтому, во-пе ***1.1. deri*** - ектируется самодостаточным в плане доступ ть р м и их, члены производных классов в нем не упоми



Как известно, внутри класса его член ты друг для др ая функция-член может обращаться к любой ко извне можно обратиться только к открытым членам класса.

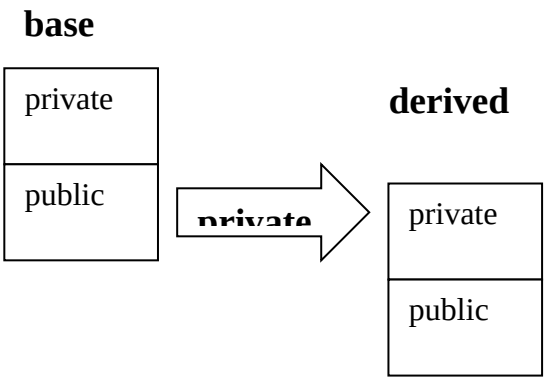
Важно отметить, что при наследовании производный класс полностью включает в себя базовый класс. Однако не все члены кл ыты друг для друга в этом объединении. Здесь необходимо следовать следующим правилам.

1. «Круг общения» членов производного класса расширяется только за счет *public*-членов базового класса. Это означает, что члены производного класса могут обращаться напрямую только к открытым членам класса *base*; *private*-члены базового класса можно использовать в описании функций производного класса только через открытые члены базового класса.

2. Если наследование производится со спецификатором доступа *public*, то открытые члены *base* становятся *public*-членами класса *derived*; если наследование производится по типу *private*, то *publik*-члены базового класса становятся *private*-членами *derived*. *Private*-члены базового класса безусловно наследуются производным классом, но только обращаться к ним члены исходного класса *derived* могут исключительно посредством *public*-членов класса *base*.

***#include** <iostream.h>*

```
class base
{int x;
public:
void set_x (int n) {x=n;}
void show_x ( ) {cout<<x<<«\n»;}
};
class derived: public base
{int y;
public:
void set_y (int k) {y=k;}
void show_y ( ) {cout<<y<<«\n»;}
};
void main ( )
{
derived obj;
ob.set_x(10);
ob.set_y(12);
ob.show_x( );
ob.show_y( );
}
```



В данном примере *base* наследуется как открытый. Поэтому в объекте *obj*, который является экземпляром производного класса, его *public* функции доступны как обычные открытые члены. В данном примере “добраться” до закрытого члена базового класса невозможно; и инструкция типа *cout <<obj.x* будет ошибкой.

В следующем примере наследование производится с типом *private*. В этом случае открытые члены базового класса в производном становятся закрытыми. Рассмотрим, как в этом случае получить доступ и к открытым и к закрытым членам базового класса через производный.

```
# include <iostream.h>
class chief
        { int a;
        public:
void set_a (int n) {a=n;}
                int get_a( ) {return a;}
        };
class clerk: private chief
        { int b;
        public:
void set_ab (int n, int m) {set_a(n);b=m;}
                void show_b( ) {cout <<b<<«\n»;}
                void show_a( ) {cout <<get_a( )<<«\n»;}
```

```
};
                void main( )
{clerk.obj;
obj.set_ab(2500,500);
obj.show_b( );
obj.show_a( );
}
```

Т.к. при наследовании с типом *private* получить доступ к “наследству” можно только через открытые члены “наследника”, то в классе *clerk* проектируется функция для ввода значения закрытой переменной “*a*” базового класса и вторая функция – для вывода этого значения на экран. Однако доступ к “*a*” возможен только через открытые члены базового класса, которые и выступают в этих функциях в качестве посредников.

Защищенные члены класса

Из вышеизложенного все больше проступает неудобство работы с закрытыми членами базового класса в производном классе. Чтобы допустить членов производного класса к общению с *private*-членами базового класса, введен спецификатор доступа *protected* (защищенный).

Теперь члены класса разделены на три группы: *private*, *protected* и *public*. Спецификатор *protected* может располагаться в любом месте описания класса, но обычно он располагается перед *public* . И это оправдано, потому что *protected*-члены – это часть закрытых членов базового класса, к которым напрямую могут обращаться члены производных классов.

Правила наследования:

1. Когда базовый класс наследуется производным как *public*, защищенные члены базового класса становятся защищенными членами производного класса.

2.Когда базовый класс наследуется как закрытый, то защищенные члены базового становятся *private*-членами производного (аналогично и *public*-члены).

3.Если базовый класс наследуется как защищенный, то открытые и защищенные члены базового класса наследуются как *protected*.

```
#include <iostream.h>
class base
{ int a;
protected:
int b;
public:
void set_ab (int n, int m) {a=n; b=m;}
};
void show_a( ) {cout<<a<<" ";}
class derived: protected base
{int c;
public:
void set_c (int k) {c=k}
void show_bc( ){cout<<b<<" "<<c<"\n";}
};
void main( )
{derived ob;
ob.set_ab(1,2); //ОШИБКА! set_ab( ) стала закрытым членом derived
ob.set_c(3);
ob.show_a( ); //ОШИБКА! (аналогичная)
ob.show_bc( )
}
```

В этом примере базовый класс наследуется как защищенный. В базовом классе недоступной останется переменная *a*. Чтобы вывести ее значение предусмотрена *public*-функция. Члены *base* при наследовании переходят в категорию *protected* (кроме переменной *a*), поэтому напрямую недоступны. Из-за этого ошибки в программе.

Конструкторы и деструкторы в наследовании

Если у базового и производного классов имеются конструкторы и деструкторы, то конструктор базового класса выполняется раньше конструктора производного; для деструкторов соблюдается обратный порядок. Если конструкторы имеют параметры, то:

а) все аргументы для конструктора базового и конструктора производного классов передаются конструктору производного класса;

б) затем, используя расширенную форму объявления конструктора производного класса, соответствующие аргументы передаются в базовый класс.

```
# include <iostream.h>
class base
{int i;
public:
base(int n) {cout <<"Base constructor"; i=n;}
~base ( ) {cout<<"Base destructor";}
```

```
void show_i( ) {cout <<i<<"\n";}
};
class derived : public base
{ int j;
public:
derived(int n, int m}: base(m)
{cout<<"Derived constructor"; j=n;}
~derived( ) {cout<<"Derived destructor";}
void show_j( ) {cout<<j<<«\h»;}
};
void main( )
derived obj(10,20);
obj.show_i( );
obj.show_j( );
}
```

Оба класса в примере имеют конструкторы и деструкторы. Вначале на экран выведется: *Base constructor*, затем – *Derived constructor*. Конструктор производного класса объявлен так, что требует два аргумента; один он использует сам, другой – передает в базовый класс. В результате инициализируются переменные *i* и *j*. Затем на экран будет выведено: *20* и *10*.

После этого выводится сообщение:

Derived destructor

Base destructor.

1.3 ПОЛИМОРФИЗМ

Понятие полиморфизма в С++

Слово «полиморфизм» происходит от греческого *πολιμορφία* – многообразие. Оно характеризует чего-либо реализовываться в различных форматах. Применительно к языку С++ под полиморфизмом понимается использование одних и тех же имен (или знаков) для указания ЭВМ на выполнение различных, но схожих по типу действий. При этом ЭВМ выбирает тот или иной вариант в зависимости от используемых типов данных.

Полиморфизм в скрытом виде присутствует и в языке Си. Например, при операции сложения ЭВМ реализует различные последовательности действий, если мы складываем целые числа или числа с плавающей запятой. Но и в том и в другом случае пишется знак «+». В С++ полиморфные конструкции проектирует сам программист.

Если одно имя функции используется для различных последовательностей действий, то это называется перегрузкой функции. Если один оператор используется для обозначения различных действий, то такой тип полиморфизма называется перегрузкой операторов.

В более общем смысле полиморфизм позволяет реализовывать идею: один интерфейс – множество методов. Это означает, что можно создать один интерфейс для группы близких по смыслу действий. Его преимущество состоит

в том, что он помогает снижать сложность программ, разрешая использовать один интерфейс для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор.

Компилятор связывает имя функции с конкретным вариантом ее реализации до выполнения программы. Этот процесс называется ранним связыванием.

В рамках полиморфизма в С++ применяются виртуальные функции, которые реализуют динамический полиморфизм. Его особенностью является то, что на этапе компиляции не ясно, какой вариант функции следует выполнить. Ситуация проясняется только в ходе выполнения программы. Этот процесс называется поздним связыванием.

Преимуществом раннего связывания является высокое быстродействие программы. Его главный недостаток – потеря гибкости. Программы, реализующие позднее связывание, менее быстродейственны, но более гибки.

Перегрузка функций

Если две или более функций имеют одинаковое имя, то говорят, что они перегружены. В этом случае функции должны отличаться типом и (или) числом своих аргументов. Перегрузить функцию не сложно – просто следует объявить и определить все требуемые варианты. На компилятор возлагается задача выбора соответствующей конкретной версии вызываемой функции (а значит и метода обработки данных). Рассмотрим пример перегрузки функции.

```
#include <iostream.h>
void date (char date) {cout <<date<< "\n";}
void date (int mouth, int day, int year)
{cont <<day<< "/" <<mouth<<"/" <<year<<"/";}
void main ( )
{
  date ("11/12/2002");
  date (11, 12, 2002);
};
```

В данном примере функция *date( )* перегружается для получения даты либо в виде строки, либо в виде трех целых чисел. Данный пример показывает, что перегрузка функции может производиться и вне объектно-ориентированного подхода.

Компилятор различает перегруженные функции и в случае, когда параметры одного типа, но их разное количество.

Перегрузка конструкторов

Перегрузка конструкторов в программах на С++ – обычное дело. Она осуществляется для обеспечения гибкости и выполняет ряд других функций. Рассмотрим пример.

```
#include <iostream.h>
class number
{ int x;
```

```
  public:
  number( ) {x=0;}
  number(int n) {x=n;}
  int get_x( ) {return x;}
};
void main ( )
{
number 0b1(5);
number 0b2;
cout <<"0b1:" <<0b1.get_x<<( )<<"\n";
cout <<"0b2:" <<0b2.get_x<<( )<<"\n";
}
```

В данном примере в классе *number* описаны два конструктора: один производит инициализацию, второй – нет.

Затем, в главной функции создается два объекта различными способами. У первого объекта *x=5*, у второго *x=0*.

Рассмотрим использование перегруженных конструкторов для инициализации массивов объектов.

Массив объектов задается так же, как и массив любого другого типа данных. Например для класса *number*, массив из трех объектов задается так:

```
number obj[3];
```

Количество элементов массива указывается в прямоугольных скобках. Приведем пример инициализации массива объектов.

```
#include <iostream.h>
class mass
{
 int x;
 public:
 mass( ) {x=0;}
 mass(int n) {x=n;}
 int get_x( ) {return x;}
};
void main ( )
{
 mass ob1[10];
 mass ob2[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
 int i;
{cout <<"Elements of ob1:\n";
  for (i=0; i<10; i++)
  cout <<ob1[i].get_x( ) <<"\t";
 }
 { cout <<"Elements of ob2:\n";
   for (i=0; i<10; i++)
```

```
    cout <<ob2[i].get_x( ) <<"\t";
  }
 }
```

В этом примере используются два конструктора: один – с инициализацией, другой – без.

В функции *main ( )* создаются два массива по десять объектов. Первый создается так, что у всех объектов *x=0*. Для второго массива применяется инициализация с использованием конструктора с параметром.

В заключении на экран выводятся значения переменной *x* всех объектов.

Перегрузка операторов

Перегрузка операторов позволяет использовать знакомые операторы для работы с новыми типами данных, такими, например, как класс. Когда оператор перегружается, то старое его значение сохраняется; и дополнительно к этому он приобретает новое значение, связанное с классом, для которого он определен.

Для перегрузки оператора создается оператор-функция, которая является членом класса, для которого она определена. Имя оператор-функции –

operator#( ),

где вместо знака # следует подставить перегружаемый оператор (например: *operator**). В скобках указывается список аргументов (параметров).

В C++ допускается перегружать большинство операторов, кроме: точки (.),двойного двоеточия (::), вопросительного знака (?).

Оператор-функция (за исключением оператора =) наследуется производным классом. Тем не менее, для производного класса тоже можно перегружать любой оператор (в том числе уже перегруженный в базовом классе).

Перегрузка бинарных операторов

C++ содержит специальный указатель *this*. Это указатель, который автоматически передается любой функции-члену при ее вызове и указывает на объект, генерирующий вызов. Например, имеется запись:

y=obj.get_x( );

Функции *get_x( )* автоматически передается указатель на объект *obj*. Этот указатель и называется *this*.

Если в классе перегружается бинарный оператор, то необходимо указать, где брать оба операнда. Однако в операторе-функции, которая перегружает этот оператор, указывается только один параметр. Этим параметром будет объект, стоящий справа от оператора; объект, стоящий слева генерирует вызов оператора-функции и передается неявно, с помощью указателя *this*. Рассмотрим пример.

```
#include <iostream.h>
class coord
{int x, y;
 public:
 coord( ) {x=0; y=0;}
 cord(int i, int j) {x=i; y=j;}
 int get_x( ) {return x;}
```

```
  int get_y( ) {return y;}
  coord operator+(coord ob2);
};
coord coord:: operator+(coord ob2)
{
  coord temp;
  temp.x=x+ob2.x;
  temp.y=y+ob2.y;
  return temp;
}
void main ( )
{
coord obj1(10, 5),
        obj2(5, 3),
        obj3;
obj3=obj1+obj2;
cout <<"obj1+obj2: \n" <<"x=" <<obj3.get_x( )<<"t";
cout <<"y=" <<obj3.get_y( ) <<"\n";
}
```

В описанном в примере классе две закрытые переменные, перегруженный конструктор, функции получения значений закрытых переменных и оператор-функция.

Оператор-функция возвращает объект типа *coord*; чтобы его иметь, он создается внутри функции. У оператора-функции – один параметр, вторым является *this* и его переменные *x* и *y* внутри оператора-функции указываются открыто.

Локальный объект *temp* позволяет хранить результат сложения, оставляя неизменными слагаемые и обеспечивая присваивание результата сложения.

Виртуальные функции

Виртуальные функции являются инструментом для поддержки динамического полиморфизма. Основой для использования виртуальных функций выступают указатели на производные классы.

Указатели на производные классы

Указатель, объявленный как указатель на базовый класс, может использоваться как указатель на любой класс, производный от этого базового. Следующие инструкции являются правильными:

***base** *p;* //указатель базового класса

***derived** ob;* //указатель производного класса

p=&ob; //указатель *p* для объекта производного класса.

При использовании указателя базового класса в производном, доступ может быть обеспечен только к тем членам производного класса, которые были унаследованы от базового. Рассмотрим пример.

```
#include <iostream.h>
```

```
class base
{int x;
 public:
 void set_x(int i) {x=i;}
 int get_x( ) {rerurn x;}
};
class derived: public base
{int y;
 public:
 void set_y(int i) {y=i;}
 int get_y( ) {rerurn y;}
};
void main( )
{base *p;
base base_ob;
derived der_ob;
p=&base_ob;
p→set_x(10); //доступ к члену базового класса
cout <<"x of base=" <<p→get_x( ) <<"\n";
p=&der_ob;
p→set_x(99);
der_ob.set_y(88); //y через указатель установить нельзя
cout <<"x of derived="<<p→get_x( ) <<"\n";
cout <<"y of derived="<<der_ob.get_y( ) <<"\n";
}
```

В данном примере спроектированы базовый и производный от него классы. Объявляется указатель *p* базового класса и устанавливается на объект базового класса. Через него производится запись и чтение переменной *x*. Затем указатель ставится на объект производного класса и через него осуществляется доступ к переменной *x* этого объекта. К переменной *y* так обратиться нельзя, так как она является особенной (ненаследуемой) переменной производного класса. К ней обращаться можно только через имя объекта производного класса.

Особенности виртуальных функций

Виртуальная функция является членом класса. Она объявляется внутри базового класса и переопределяется в производном. Только в базовом классе перед ней ставится слово *virtual*. Динамический полиморфизм поддерживается только в том случае, когда вызов виртуальной функции производного класса производится через указатель базового. Какая версия виртуальной функции будет вызвана, определяется типом объекта, на который ссылается указатель.

```
#include <iostream.h>
class base
{public:
 int i;
 base (int x) {i=x;}
```

```
 virtual void func( )
 {cout <<"virtual of base" <<i <<"\n;}
};
class derived1: public base
{public:
 derived1(int x): base(x) { }
 void func( )
 {cout <<"virtual of derived1" <<i*i <<"\n;}
};
class derived2: public base
{public:
 derived2(int x): base(x) { }
 void func( )
 {cout <<"virtual of derived2" <<i+i <<"\n;}
};
void main( )
{base*p; base ob(10);
 derived1 d_ob1(10), derived2 d_ob1(10);
 p=&ob; p→func( );
 p=&d_ob1; p→func( );
 p=&d_ob2; p→func( );
}
```

Отличия виртуальной функции от перегрузки функции:

Перегружаемая функция должна отличаться типом и (или) числом параметров; у виртуальной функции эти характеристики должны совпадать;

- виртуальная функция должна быть членом класса;
- выбор варианта перегружаемой функции производится при компиляции (раннее связывание); конкретная реализация виртуальной функции производится при выполнении программы (позднее связывание).

Перегрузка функций

Иногда бывает удобно иметь несколько функций, выполняющих различные действия, но имеющие одно и то же имя. Такого рода результат можно достичь с помощью, так называемой, перегрузки функций. Перегрузка функций – это такой метод написания программы, когда несколько функций имеют одно и то же имя, но заголовки их отличаются друг от друга типами и количеством параметров. Компилятор производит анализ типа и количества фактических параметров в операторе вызова такой функции, в результате чего происходит вызов функции, имеющей наиболее подходящий набор формальных параметров. Для объяснения сути процесса перегрузки функций рассмотрим фрагмент программы приведенной ниже. В приведенном примере описаны 4 функций под одним и тем же именем *FunX*( ), но имеющие формальные параметры разных типов.

Обращаем внимание на то, что в описаниях функций отсутствуют имена формальных параметров, указаны только их типы. Такое упрощение в данном примере допустимо, т.к. параметр в теле функции не используется. Типы формальных параметров здесь используются только для выбора одной из перегружаемых функций при ее вызове.

Одним из допустимых вариантов перегружаемой функции является функция без параметров, что демонстрируется на примере последней из приведенных функций:

```
#include<iostream.h>

void FunX(char)
{ cout<<"Вызвана функция с char-параметром"; }
void FunX(int)
{ cout<<"Вызвана функция с int-параметром"; }
void FunX(double)
{ cout<<"Вызвана функция с double-параметром"; }
void FunX( )
{ cout<<"Вызвана функция без параметров"; }

void main( )
{
   char Alfa;
   int Beta;
   double Gamma;
```

/*Далее следуют вызовы функции FunX( ) с фактическими параметрами разных типов. В результате каждый раз вызывается функция, имеющая соответствующий тип формального параметра. В данном примере значения фактических параметров не существенны, поскольку они в теле программы не используются, поэтому они имеют случайные значения, получаемые при объявлении. Функции выдают сообщения на экран о своих вызовах.*/

```
   FunX(Alfa);          //На экран выводится сообщение
                        //   "Вызвана функция с char-параметром"
   FunX(Beta);          //На экран выводится сообщение
                        //   "Вызвана функция с int-параметром"
   FunX(Gamma);             //На экран выводится сообщение
                        //  "Вызвана функция с double-параметром"
   FunX( );             //На экран выводится сообщение
                        //   "Вызвана функция без параметров"
}
```

Признаком различия функций при их перегрузке является также количество параметров. Так, например, при вызове будут считаться разными две следующие функции, имеющие одно и то же имя:

```
    void FunY(int)          //Пример описания двух перегружаемых
    {                       //функций. Функции имеют одно имя,
...Тело функции...          //но разное число формальных параметров
     }

    void FunY(int, int)
    {
  ...Тело функции...
    }
```

Тип возвращаемого функцией значения в перегрузке функций не участвует. Это значит, что компилятор выдаст сообщение об ошибке, если в тексте программы будут приведены следующие два описания функции *FunZ( )*, имеющие разные типы возвращаемых значений, но одинаковые наборы формальных параметров:

```
int FunZ(int, int)      //Пример описания двух функций,
{                       //с одним именем, но имеющих
...Тело функции...      //различные возвращаемые значения.
 }                      //Эти функции не могут быть перегружены,
                        //т.к. имеют одинаковые наборы формальных
double FunZ(int, int)   //параметров
{...Тело функции...}
```

Не различаются также при перегрузке функций обычный параметр от параметра типа ссылка того же типа. В следующем примере приведены описания функций, при котором компилятор выдаст сообщение об отсутствии различия в описаниях двух функций:

```
void FunV(int)      //Описание функции с параметром типа int
{
...Тело функции…
}
void FunV(int&)     //Описание функции с параметром типа ссылка на int
{
...Тело функции…    //Эти две функции не могут быть
}                   //перегружены, т.к. типы int и int& не
                    //различаются при перегрузке функций
```

Функции не могут быть перегружены и в том случае, когда их параметры отличаются только спецификатором const, например:

```
void FunV(const int)          //Эти две функции не могут быть
{                             //перегружены, т.к. типы int и const int не
  ...Тело функции...          //различимы при перегрузке функций
 }

void FunV(int)
{
  ...Тело функции...
}
```

В то же время могут быть перегруженными две функции, одна из которых имеет параметр обычного типа, а другая имеет параметр – указатель того же типа, например:

```
void FunW(int)          //Эти две функции могут быть перегружены,
{                       //т.к. при перегрузке функций типы
...Тело функции...      //парметров int и int* различаются между
}                       //собой

void FunW(int*)
{
...Тело функции...
}
```

Перегружаемые функции могут иметь параметры со значениями по умолчанию. Параметры, имеющие значения по умолчанию не участвуют в процессе выбора экземпляра вызываемой функции. В этом случае выбор экземпляра перегружаемой функции при вызове функции производится только по параметрам, не имеющим значений по умолчанию. Количество параметров по умолчанию в функциях для выбора перегружаемой функции при этом не имеет значения. Покажем это на примере.

```
#include<iostream.h>

void FunX(int,int Ro=5)                 //Описание перегружаемой функции
{                                       //с параметром типа int, который
  cout<<"Вызвана функция с              //используется для выбора функции,
  параметром типа int " << endl;        //и одним параметром со значением
                                        //по умолчанию
... другие операторы функции ...
}

void FunX(float,int Pi=3,int Ro=4)  //Описание перегружаемой функции
```

```
{                                       //с параметром типа float, который
  cout<<"Вызвана функция с              //используется для выбора функции,
параметром типа float" << endl; //и двумя параметрами со значениями
                                        //по умолчанию
… другие операторы функции ...
}

void main( )
{
  int Alfa;                  //Описания переменных Alfa, Beta, Gamma
  float Beta;
  int Gamma=10;

  FunX(Alfa);      //Вызов перегружаемой функции, имеющей один
                   //параметр со значением по умолчанию. На экран
                   //выводится сообщение:
                   //   "Вызвана функция с параметром типа int"
  FunX(Beta);      //Вызов перегружаемой функции, имеющей два
                   //параметра со значениями по умолчанию. На экран
                   //выводятся сообщение:
}                  //   "Вызвана функция с параметром типа float"
```

Как видно из приведенного примера, параметры по умолчанию не принимают участие в процессе выбора экземпляра одной из перегружаемых функций. Экземпляр вызываемой функции определяется типом только одного параметра, не имеющего значения по умолчанию.

Еще один пример демонстрирующий то, что параметры, имеющие значения по умолчанию, не участвуют в процессе выбора перегружаемых функций:

```
void FunY(float,int Pi=10 )          //Эти две функции не могут быть
{                                    //перегружены, т.к. они отличаются
  ...Тело функции…                   //только количеством параметров,
 }                                   //имеющих значения по умолчанию

void FunY(float,int Fi=20, int Ro=30)
{
  ...Тело функции…
}
```

Возможна такая ситуация, когда в операторе вызова функции указан тип фактического параметра, которого нет ни у одной из перегружаемых функций. В таких случаях компилятор ищет среди описаний перегружаемых функций функцию, имеющую ближайший аналогичный, но более старший тип параметра. Так, например, если вызывается функция с фактическим параметром *char*, а

такого типа параметра нет ни у одной из перегружаемых функций, то компилятор ищет перегружаемую функцию с параметром типа *short*, если не находит такой функции, то ищет функцию с параметром типа *int*, если нет такой функции, то ищет функцию с параметром типа *long*. Если нет и такой функции, то компилятор выдает сообщение об ошибке. Аналогичный алгоритм поиска подходящей перегружаемой функции реализуется и для фактического параметра типа *float*. В этом случае компилятор последовательно ищет вначале функцию с параметром типа *float*, затем *double* и затем *long double* и если не находит ни одного из этих типов, то выдает сообщение об ошибке. Сказанное иллюстрируется примером программы

```
#include<iostream.h>

void FunV(int)
{ cout<<"Вызвана функция с int-параметром"<<endl; }          //

void FunV(long int)
{ cout<<"Вызвана функция с long int-параметром"<<endl; }

void FunV(long double)
{ cout<<"Вызвана функция с long double-параметром"<<endl; }

void main( )
{
char Alfa;                    //Объявление переменных Alfa, Beta
float Beta;

FunV(Alfa);          //Вызов перегружаемой функции с фактическим
//параметром типа char.
/* Ближайший, имеющийся среди описаний перегружаемых функций старший тип для целочисленного типа char – это тип int, поэтому происходит вызов перегружаемой функции с формальным параметром типа int. На экран выводится сообщение
"Вызвана функция с int-параметром" */
FunV(Beta);          //Здесь вызывается функция с формальным
                     //параметром типа long double. На экран выводится
                     //сообщение
}                    //"Вызвана функция с long double-параметром"
```

Так же, как и для обычных функций, возможны два варианта размещения описаний перегружаемых функций:

- описания перегружаемых функций приводятся в одном файле с операторами вызова функций и по тексту программы предшествуют им;

- описания перегружаемых функций приведены в тексте программы после операторов их вызова или приведены в отдельном файле.

При втором варианте размещения описаний перегружаемых функций операторам вызова функций должны предшествовать их прототипы. Сказанное проиллюстрируем примером программы.

```
#include<iostream.h>

void main( )
{
      int Alfa;                  //Объявление переменных
      float Beta;

      void FunS(int);            //Прототипы перегружаемых функций с
      void FunS(float);          //разными параметрами

      FunS(Alfa);          //Оператор вызова функции с int-параметром.
                           //На экран выводится сообщение:
                           // ”Вызвана функция с int-параметром”
      FunS(Beta);          //Оператор вызова функции с
                           //float-параметром.
                           //На экран выводится сообщение:
}                          //”Вызвана функция с float-параметром”

void FunS(int)             //Описания перегружаемых функций
{ cout<<"Вызвана функция с  //приведены после операторов вызова
   int-параметром"<<endl; }//этих функций

void FunS(float)
{ cout<<"Вызвана функция с float-параметром"<<endl; }
```

1.4 Шаблоны функций

Еще одной разновидностью универсальных функций, наряду с перегружаемыми функциями и с функциями с переменным числом параметров, являются так называемые шаблоны функций. По сути дела шаблон функции – это есть описание такой функции, которая, в отличие от обычной функции, может при ее вызове принимать фактические параметры любого типа и возвращать значение любого типа.

Для того чтобы это было возможным, описание шаблона функции включает в себя идентификаторы, которые мы назовем формальными типами. При вызове функции эти формальные типы заменяются конкретными фактическими типами. Причем эти фактические типы определяются типами передаваемых функции фактических параметров. Компилятор определяет типы фактических

параметров в операторе вызова функции и использует эти типы вместо формальных типов, имеющихся в описании шаблона.

Таким образом, шаблон функции – это описание функции, которая имеет фиксированный набор операторов в своем теле, но при этом эти операторы могут обрабатывать данные любых типов, задаваемых при вызове такой функции, а сама функция может возвратить значение одного из заданных типов.

Рассмотрим на примере формат описания шаблона функции:

```
template<class T1,class T2>   //Заголовок шаблона функции состоит из
T1 FunS(T1 Ksi, T2 Psi)       //предзаголовка и основной части
                              //заголовка.
{                             //T1 и T2 – формальные типы,
                              //Ksi и Psi – формальные параметры
Тело шаблона функции.....     //шаблона функции

}
```

Здесь приведен пример описания шаблона функции, которому присвоено имя *FunS( )*. Как видно из приведенного примера, заголовок шаблона функции состоит из двух частей: предзаголовка, в данном случае он имеет вид

***template*<*class T1*,*class T2*>,**

и основной части заголовка

***T1 FunS(T1** Ksi, **T2** Psi).*

Предзаголовок шаблона функции всегда начинается с ключевого слова *template*, за которым следуют угловые скобки. В угловых скобках приводится через запятую перечень формальных типов, каждый из которых предваряется ключевым словом «*class*». Аналогично тому, как формальные параметры в описании обычной функции заменяются фактическими параметрами при вызове функции, формальные типы в описании шаблона функции заменяются фактическими типами в операторе вызова функции. Имена формальных типов могут быть произвольными, в данном примере им присвоены имена *T1* и *T2*. Далее эти формальные типы могут использоваться в основной части заголовка шаблона функции и в теле шаблона для описания параметров, переменных и, как в данном примере, для указания типа возвращаемого функцией значения.

Основная часть заголовка шаблона функции представляет собой обычный заголовок функции с тем отличием, что каждый из приведенных в предзаголовке формальных типов должен быть хотя бы один раз использован для описания формальных параметров функции в основной части заголовка шаблона. Тип возвращаемого функцией значения в основной части заголовка шаблона может быть стандартным типом языка C++ или любым из приведенных в предзаго-

ловке формальных типов. Количество формальных параметров может быть произвольным, но не меньшим числа приведенных в предзаголовке формальных типов, т.к., как было уже сказано, каждый из формальных типов должен быть использован для описания хотя бы одного формального параметра. Кроме обязательных параметров объявленных формальных типов шаблон функции может иметь параметры и стандартных типов языка Си, которые могут иметь значения по умолчанию.

Фрагмент программы демонстрирует применение шаблона функции:

```
#include<iostream.h>

template<class T1,class T2>    //Описание шаблона функции FunS( ).
T1 FunS(T1 Pi, T2 Fi)  //Формальные типы T1 и T2, объявленные в
{                      //предзаголовке шаблона использованы в
                       //основной части заголовка для описания
                       //формальных параметров Pi, Fi и для
                       //указания типа возвращаемого функцией
                       //значения.
   T1 Tau;             //Формальный тип T1 используется для
                       //объявления переменной Tau.
cout << Pi << " " << Fi; //На экран выводятся значения параметров
                         //Pi и Fi
   Tau=Pi + Fi;        //Переменной Tau присваивается значение,
                       //равное сумме значений параметров.
   return Tau;         //Функция FunS( ) возвращает значение,
}                      //равное значению переменной Tau

void main( )
{
char   Alfa=«B», Beta=«J»;          //Объявления переменных
double Gamma=2.22, Delta=3.33;      //различных типов
 int   Ksi=5, Psi=10;

 cout<<FunS(Alfa,Beta)<<endl;  //Вызов функции FunS( ) с передачей ей
                               //двух параметров типа char. Вначале
                               //функция выводит на экран символы
                               «B» и «J»,
                               //а затем суммирует значения кодов символов
                               //«B» и «J», т.е. шестнадцатеричные числа
//42 и 4A. Полученное шестнадцатеричное
//число 8C является кодом русской буквы M.
                               //Это значение возвращается функцией.
                               //В данном случае тип возвращаемого
//функцией значения – char, поэтому на экран
```

```
//выводится символ «M»

cout<<FunS(Ksi,Psi)<<endl;//Вызов функции FunS( ) с передачей ей
//двух параметров типа int. Функция выводит
//на экран значения параметров Ksi и Psi,
//т.е. числа 5 и 10 и возвращает их сумму.
//На экран выводится число 15.

cout<<FunS(Gamma,Delta)<<endl;//Вызов функции FunS( ) с передачей ей
//двух параметров типа double. Функция
                                        //выводит на экран значения параметров //Gamma и Delta, т.е.числа 2.22 и 3.33
//и возвращает их сумму.
//На экран выводится число 5.55
}
```

В приведенном фрагменте программы описание шаблона функции приведено по тексту выше главной функции *main( )*, т.е. раньше операторов вызова функции, которую этот шаблон описывает. Но описание шаблона функции может быть приведено и после главной функции *main( )*. В этом случае перед главной функцией *main( )* в тексте программы должен быть приведен прототип шаблона функции. Прототип шаблона функции можно получить аналогично тому, как получается прототип обычной функции, т.е. записывается полный заголовок шаблона функции, после которого ставится точка с запятой. Так же, как и в прототипе обычной функции, в прототипе шаблона функции можно опустить имена формальных параметров, указав только их типы. Особенностью применения прототипа шаблона функции является то, что его нельзя привести в теле функции. Как ранее говорилось, прототип обычной функции может быть приведен в любом месте программы, вне функций, в теле функции *main( )* или в теле какой-либо другой функции. При необходимости же применения прототипа шаблона функции, он может быть приведен в программе только вне тела какой либо функции, в том числе вне главной функции *main( )*. Следующий пример схематично демонстрирует то, как выглядела бы предыдущая программа, если бы описание шаблона было приведено после главной функции *main( )*:

```
#include<iostream.h>

template<class T1,class T2>    //Прототип шаблона функции FunS( )
T1 FunS(T1 Pi, T2 Fi);

void main( )
{
   ……..
```

```
  Тело главной функции
  ……..
}

template<class T1,class T2>    //Описание шаблона функции
T1 FunS(T1 Pi, T2 Fi)  //приведено после главной функции main( ),
{                      //поэтому перед функцией main( ) необходимо
  ……..//привести прототип шаблона функции.
  Тело шаблона функции
  ……
}
```

Наряду с формальными параметрами формальных типов шаблон функции может иметь также формальные параметры стандартных типов языка С++ или типов создаваемых пользователем, например:

```
template<class T>
void FunV(T Fi, int i)          //Первый формальный параметр шаблона
{                               //имеет формальный тип T,
    ……..                       //второй – стандартного типа int
    Тело шаблона функции
    ……..
}
```

Как было сказано, описание шаблона функции содержит формальные типы, которые при вызове функции заменяются фактическими типами. При этом эти фактические типы определяются типами фактических параметров, которые передаются функции при ее вызове. Это продемонстрировано на примере программы, описанной выше. В то же время существует возможность явного задания фактических типов в операторе вызова функции. Это достигается указанием конкретных фактических типов в угловых скобках в операторе вызова функции. Поясним сказанное примером программы.

```
#include<iostream.h>

template <class T1,class T2>  //Описание шаблона функции FunS( ).
void FunS(T1 P, T2 Q)         //T1 и T2 – формальные типы
{
      cout<<P<<endl;          //Функция выводит на экран значения
      cout<<Q<<endl;          //передаваемых ей параметров
}

void main( )
{
```

```
    double Alfa=87.5;

    FunS<int,char>(Alfa,Alfa);
}
```

Показан вызов функции *FunS( )*, описанной шаблоном. В качестве обоих фактических параметров функции использована одна и та же переменная *Alfa* типа *double*. В угловых скобках указаны фактические типы *int* и *char*. При такой записи формальный тип *T1* замещается в теле функции *FunS( )* типом *int*, а тип *T2* замещается типом *char*. Соответственно первый фактический параметр *Alfa* преобразуется к типу *int* , второй – к типу *char*. В результате функция вначале выводит на экран целое число 87, а затем латинскую букву *W* (код буквы *W* - десятичное число 87)

Особенностью применения шаблонов функций является то, что в общем случае все вызовы функций, описываемых шаблоном могут находиться только в том файле, в котором приведено описание шаблона. Если возникает необходимость разместить описание шаблона функции в отдельном файле, то описание шаблона можно разместить в файле с расширением *.cpp* или в файле с расширением *.h* , и затем включить этот файл директивой препроцессора *include* в файл, содержащий вызовы функций.

В начале этого параграфа было сказано, что шаблон функции можно представить как описание функции, которая может принимать фактические параметры произвольных типов. В то же время нужно понимать, что компилятор для каждого вызова этой функции, отличающегося от других вызовов набором фактических параметров, создает отдельный блок кода функции. То есть, в отличие от обычной функции, когда объектный код функции создается в единственном числе, к которому идет обращение при каждом вызове функции, для шаблона функции компилятор создает столько исполняемых блоков функции, сколько в программе есть отличающихся фактическими параметрами вызовов этой функции. Таким образом, использование шаблонов функций приводит к некоторому увеличению размера исполняемого кода программы.

Функции, описываемые шаблоном, могут быть перегружены или другими шаблонами, или обычными функциями, имеющими то же самое имя, но разное число и типы параметров. Поясним сказанное примером программы, представленном ниже.

В этом примере под одним и тем же именем *FunP( )* описаны:

- обычная функция, имеющая один параметр типа char,
- шаблон функции с одним формальным типом,
- шаблон функции с двумя формальными типами.

То, какая из этих функций будет вызвана, определяется числом и типом фактических параметров, указываемых в операторе вызова функции.

В целях упрощения примера описанные функция и шаблоны не имеют операторов обработки их параметров, поэтому имена формальных параметров опущены. Указаны только типы параметров. В описании функции это тип char, в описаниях шаблонов – это формальные типы *T, T1, T2*. Идентификаторы же

типов формальных параметров необходимы для выбора экземпляра функции при вызове функций:

```
#include<iostream.h>

void FunP(char)                 //Описание обычной функции FunP( )
{
  cout<<"Вызвана функция, описанная без шаблона";
}

template<class T>               //Описание шаблона FunP( ) с одним
void FunP(T)                    //формальным типом
{
                cout<<"Вызвана функция, описанная шаблоном с одним
        формальным типом";
}

template<class T1,class T2>             //Описание шаблона FunP( ) с двумя
void FunP(T1, T2)                       //формальными типами
{
                    cout<<"Вызвана функция, описанная шаблоном с двумя
            формальными типами";
}

void main( )
{
  char Pi;
  int Fi;
  FunP(Pi);         //Оператор вызова функции имеет один фактический
//параметр типа char, поэтому вызывается обычная
//функция FunP( ). На экран выводится сообщение
//"Вызвана функция, описанная без шаблона"
FunP(Fi);           //Оператор вызова функции имеет один фактический
//параметр типа int, поэтому вызывается функция
//FunP( ), описанная шаблоном с одним формальным
//типом. На экран выводится сообщение:
//   "Вызвана функция, описанная шаблоном с одним
//    формальным типом"
FunP(Pi,Fi);        //Оператор вызова функции имеет два фактических
//параметра типа int, поэтому вызывается функция
//FunP( ), описанная шаблоном с двумя формальными
//параметрами. На экран выводится сообщение
//    "Вызвана функция, описанная шаблоном с двумя
}                   //     формальными параметрами"
```