ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ РФ
ПЕРМСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
Кафедра Информационных технологий и автоматизированных систем

Викентьева О.Л., Гусин А.Н., Полякова О.А.

# Проектирование программ и программирование на C++: структурное программирование (часть I)

Пермь 2007

Составители: Викентьева О.Л., Гусин А.Н., Полякова О.А.

УДК 681.3

Учебное пособие по изучению курсов «Алгоритмические языки и программирование», «Программирование на языках высокого уровня» для студентов направления 230100 «Информатика и вычислительная техника» специальностей 230101 «Вычислительные машины, комплексы, системы и сети», 230101 «Автоматизированные системы обработки информации и управления» для дневной и заочной форм обучения. Сост. Викентьева О.Л., Гусин А.Н., Полякова О.А.; Перм. гос. техн. ун-т. –Пермь, 2007,100с.

Рецензенты: доцент каф. Высшей математики и информатики Пермского филиала Государственного университета Высшая школа экономики, кандидат физико-математических наук Шестакова Л. В., доцент каф. Информационных технологий и автоматизированных систем Пермского государственного технического университета Ноткин А. М.

**Оглавление**

## Введение

В процессе создания промышленных программных продуктов людям приходится сталкиваться с проблемой преодоления сложности, лежащей в самой природе таких систем. От этой сложности невозможно избавиться, пренебрегая отдельными проявлениями поведения системы, как в физике. С этой сложностью можно только справиться.

Уровень сложности – это существенная черта промышленной программы. Один разработчик практически не в состоянии охватить все аспекты такой системы, поскольку это, в среднем, превышает возможности человеческого интеллекта.

Сложность программного обеспечения определяется:

- сложностью реальной предметной области, из которой исходит заказ на разработку;
- трудностью управления процессом разработки;
- необходимостью обеспечить достаточную гибкость программы;
- неудовлетворительными способами описания поведения больших дискретных систем.

Неумение создавать сложные программные систем проявляется в проектах, которые выходят за рамки установленных сроков и бюджетов и к тому же не соответствуют начальным требованиям. Подобный кризис приводит к растрате человеческих ресурсов и к существенному ограничению возможностей создания новых продуктов.

Цель проектирования – выявление ясной и относительно простой внутренней структуры системы, иногда называемой архитектурой. Результатами процесса проектирования являются модели, позволяющие понять структуру будущей системы, определить требования и наметить способы реализации.

Главным принципом проектирования любой сложной системы является разделение последней на все меньшие и меньшие подсистемы, каждую из которых можно проектировать и совершенствовать независимо. В этом случае для понимания любого уровня системы потребуется одновременно держать в уме информацию лишь о немногих ее частях.

Следующим по важности принципом является принцип иерархического упорядочивания, предполагающий организацию составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

На данный момент существуют два основных подхода к анализу предметной области и проектированию системы:

- Разделение по алгоритмам концентрирует внимание на порядке происходящих событий: каждый модуль системы выполняет один из этапов общего процесса.
- Разделение по объектам представляет предметную область совокупностью автономных действующих лиц, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню.

Эти концепции представлены методологиями структурного (1) и объектно-ориентированного (2) анализа соответственно.

## 1. Структурное программирование

Традиционная технология программирования складывалась в условиях, когда основными потребителями программ были научные учреждения, вычислительные ресурсы были ограничены, а проблемы сопровождения по существу неизвестны. Основными критериями качества программы считались ее узко понимаемая эффективность и компактность. Со временем сложность программ возросла настолько, что на их разработку уходили годы труда большого коллектива, а в результате системы появлялись с большим опозданием и содержали большое количество ошибок.

Для преодоления этой проблемы была разработана технология, которая снижала общие затраты на протяжении всего жизненного цикла программы: от замысла, до эксплуатации. Такая технология появилась в начале 70-х годов и была названа структурным программированием.

Главное требование, которому должна удовлетворять программа – работать в соответствии со своей спецификацией и адекватно реагировать на любые действия пользователя. Кроме этого, программа должна быть выпущена к заданному сроку, и допускать оперативное внесение изменений и дополнений. Таким образом, современные критерии качества программы – это ее надежность, а также возможность планировать производство программы и ее сопровождение. Для достижения этих целей программа должна иметь простую структуру, быть хорошо читаемой и легко модифицируемой.

Структурное программирование – это технология создания программ, позволяющая путем соблюдения определенных правил, уменьшить время разработки, количество ошибок, а также облегчить возможность модификации программы.

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур: линейной, разветвляющейся и циклической. Эти структуры называются базовыми конструкциями структурного программирования.

- Линейной называется конструкция, представляющая собой последовательное соединение двух или более операторов.
- Ветвление – задает выполнение одного из двух операторов, в зависимости от выполнения какого либо условия.
- Цикл – задает многократное выполнение оператора.

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения. Структурное программирование также называют программированием без goto, т. к. частое использование операторов перехода затрудняет понимание логики работы программы.

При структурном программировании также используется программирование сверху вниз – процесс пошагового разбиения алгоритма программы на все более мелкие части до тех пор, пока не получатся такие элементы, которые легко запрограммировать.

# 2. Среда программирования Visual C++ 6.0

## 2.1. Общий вид окна

Проект (project) – это набор файлов, которые совместно используются для создания одной программы.

Рабочее пространство (workspace) может включать в себя несколько проектов.

После запуска VC++ 6.0 на экране появится окно (рис. 1).

**Рис. 1 Окно VC++ 6.0.**

Окно содержит:

- Главное меню (1) – список основных команд VC++;
- Панель инструментов (2) - панель с кнопками команд Visual C++;
- Панель рабочего пространства Workspace (3) - содержит две вкладки:
  - ClassView – отображает список классов в проекте,
  - FileView – отображает список файлов, входящих в проект.
- Окно для редактирования кодов (4) – окно с текстом программы;
- Выходную панель результатов компиляции (5) - окно для вывода сообщений в процессе компиляции или отладки, показывает текущую стадию компиляции, список ошибок и предупреждений и их количество.

## 2.2. Создание консольного приложения и работа с ним

Консольное приложение – это приложение, которое с точки зрения программиста является программой DOS, но может использовать всю доступную оперативную память (если каждый элемент данных программы не будет превышать 1 Мб). Этот тип приложения запускается в особом окне, которое называется “Окно MS-DOS”. На примере

консольных приложений прослеживаются этапы развития VC++ при переходе от одной версии к другой.

Каждое приложение, разрабатываемое как отдельный проект в среде VC++6.0, нуждается в том, чтобы ему соответствовало свое собственное рабочее пространство. Рабочее пространство включает в себя те папки, в которых будут храниться файлы, содержащие информацию о конфигурации проекта. Для того чтобы создать новое пространство для проекта, надо выполнить следующие действия:

1. В линейке меню нажать на меню **File.**
2. Выбрать пункт **New** или нажать **Ctrl+N.**
3. Появится окно **New.** В нем содержится четыре вкладки: Files, Projects, Workspaces, Other Documents. Выбрать вкладку Projects.
4. Из списка возможных проектов выбрать **Win32 Console Application** для создания приложения DOS.
5. В поле **Project name** ввести имя проекта.
6. В поле **Location** ввести путь для размещения каталога проекта, или, нажав на кнопку справа **[…]**, выбрать нужную директорию.
7. Должен быть установлен флажок **Create New Workspace**. Тогда будет создано новое рабочее окно. Нажать кнопку **OK**
8. Установить один из флажков:

   **- An empty project** – создается пустой проект, не содержащий заготовок для файлов;

   **- A simple application** – создается простейшая заготовка, состоящая из заголовочного файла StdAfx.h, файла StdAfx.cpp и файла реализации;

   **- A "Hello World" application** и **An application that supports MFC** являются демонстрационными и разными способами демонстрируют вывод на экран строки символов.

Нажать кнопку **Finish.** Появится информация о созданном проекте содержащая: тип проекта, некоторые особенности и директорию.

После создания проекта в него необходимо записать код программы. При этом можно создать новый файл или добавить в проект существующий файл.

Для создания нового файла надо выполнить следующие действия:

1. Выбрать меню **File > New** или **Project > Add to Project > New**.
2. Открыть вкладку **Files.**
3. Выбрать **C++ Source File**.
4. Чтобы создаваемый файл был автоматически присоединен к проекту, необходимо установить флаг **Add to project**.
5. В поле **Filename** ввести имя файла.
6. В поле **Location** указать путь для создания файла.
7. Нажать **OK.**

Для добавления существующего файла надо:

1. Выбрать в меню **File > Add to Project > Files**
2. Указать полное имя файла, который нужно присоединить

Для открытия существующего проекта надо:

1. Выбрать меню **File > Open Workspace**
2. Указать файл с расширением **.dsw**

Для сохранения текущего проекта надо выбрать в главном меню **File > Save Workspace.**

Для закрытия текущего проекта надо выбрать в главном меню **File > Close Workspace.**

После создания или открытия проекта в окне **Workspace** появится или список классов, или список файлов входящих в проект. В зависимости от типа проекта, он будет или пустой, или содержать изначально некоторые файлы, присущие данному типу. Проект

приложения для DOS изначально пустой. В него можно добавить новые файлы или присоединить уже существующие.

### 2.3. Компиляция и запуск проекта

Для компиляции проекта надо выбрать в главном меню **Build > Build <имя проекта>** или нажать клавишу F7.

Visual C++ 6.0 откомпилирует исходные файлы и создаст соответствующие файлы с расширением .obj. Затем эти файлы соединяются в исполняемый файл. Весь процесс компиляции и создания исполняемого файла отображается в окне Output, вкладка Build. После компиляции файла его можно запустить.

Для запуска исполняемого файла надо выбрать в главном меню **Build > Execute <имя файла>.exe** или нажмите клавиши **Ctrl+F5** . Если файл был создан, то он запустится. Для повторного запуска файла не нужно его снова компилировать. Но если в программу были внесены изменения, то перед запуском необходимо выполнить компиляцию. Выполняется именно файл с расширением .exe, а не текущий проект, т.е. в процессе запуска компиляции не происходит.

### 2.4. Отладка программы

Для отладки программы используется команда главного меню **Build>Start Debug> Step Into** – отладка с заходом в функции, которая начинается с первой строки функции main или **Build>Start Debug> Run to Cursor** – выполнение программы до курсора, т. е. отладка начинается с той строки, в которой установлен курсор. После выполнения этой команды выполнение программы происходит в режиме отладчика. Переход к следующей строке программы можно выполнять с помощью команды **Step Into (F11)** (с заходом во все вызываемые функции) или с помощью команды **Step over (F10**) (без захода в вызываемые функции). Выход из функции нижнего уровня выполняется командой **Step Out (Shift+F11)**. Текущие значения переменных можно просматривать:

1) в специальных окнах **Watch** (отображает значения всех используемых переменных) и **Value (**отображает значения заданных пользователем переменных);
2) при наведении курсора мышки на переменную отображается текущее значение этой переменной.

### 2.5. Создание рабочего пространства для нескольких проектов

Несколько проектов можно объединить в одно рабочее пространство с помощью команды **Project/Insert Project into Workspace**. Активный проект, т. е. тот, который будет выполняться, устанавливается с помощью команды **Project/Set Active Project.** Активный процесс надо отметить галочкой.

## 3.Структура программы на языке C/C++

Логически программа на C++ представляет собой набор функций, каждая функция должна реализовывать какое-то логически законченное действие. Функции вызываются либо из других функций, либо из главной функции с именем `main()`.

Физически программа на C++ представляет собой один или несколько файлов. Главная функция `main()`находится в файле с расширением `.cpp` и произвольными именем (желательно, чтобы имя файла каким-то образом отражало ту задачу, которая решается этой программой). Другие файлы обычно содержат функции, вызываемые в `main()`, они оформляются в виде специальных заголовочных файлов и имеют расширение `.h`.

Рассмотрим пример программы:

| | |
|---|---|
| `//Дана последовательность //целых чисел из n //элементов. Найти среднее //арифметическое этой // последовательности.` | комментарии |
| `#include <iostream.h>` | директивы препроцессора |
| `#include <math.h>` | |
| `void main()` | заголовок функции, с которой начинается выполнение программы |
| `{` | начало функции |
| `int a,n,i,k=0;` | определения переменных, используемых в функции |
| `double s=0;` | |
| `cout<<"\nEnter n";` | |
| `cin>>n;` | ввод-вывод данных |
| `for(i=1;i<=n;i++)` | цикл с параметром |
| `{` | начало цикла |
| `cout<<"\nEnter a";` | тело цикла |
| `cin>>a;` | |
| `s+=a;` | |
| `k++;` | |
| `}` | конец цикла |
| `s=s/k;` | оператор присваивания |
| `cout<<"\nSr. arifm="<<s<<"\n";` | вывод результата |
| `}` | конец программы |

Директивы препроцессора управляют преобразованием текста программы до ее компиляции. Исходная программа, подготовленная на С++ в виде текстового файла, проходит 3 этапа обработки:

1) препроцессорное преобразование текста;
2) компиляция;
3) компоновка (редактирование связей или сборка).

**Рис. 2 Обработка C++ программы**

После этих трех этапов формируется исполняемый код программы. Задача препроцессора – преобразование текста программы до ее компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора. Директива начинается с #.

`#define` – указывает правила замены в тексте.

```
#define ZERO 0.0
```

означает, что каждое использование в программе имени `ZERO` будет заменяться на `0.0`.

`#include<имя заголовочного файла>` – директива предназначена для включения в текст программы текста из каталога заголовочных файлов, поставляемых вместе со стандартными библиотеками. Каждая библиотечная функция С имеет соответствующее описание в одном из заголовочных файлов. Список заголовочных файлов определен стандартом языка. Употребление директивы `include` не подключает соответствующую стандартную библиотеку, а только позволяют вставить в текст программы описания из указанного заголовочного файла. Если используется заголовочный файл из стандартной библиотеки, то его имя заключают в угловые скобки. Если используется заголовочный файл, который находится в текущем каталоге проекта (он может быть создан разработчиком программы), то его имя заключается в кавычки. Подключение кодов библиотеки осуществляется на этапе компоновки, т. е. после компиляции. Хотя в заголовочных файлах содержатся все описания стандартных функций, в код программы включаются только те функции, которые используются в программе.

После выполнения препроцессорной обработки в тексте программы не остается ни одной препроцессорной директивы.

Программа представляет собой набор описаний и определений, и состоит из набора функций. Среди этих функций всегда должна быть функция с именем `main`. Без нее программа не может быть выполнена. Перед именем функции помещаются сведения о типе возвращаемого функцией значения (тип результата). Если функция ничего не возвращает, то указывается тип `void`: `void main()`. Каждая функция, в том числе и `main`, должна иметь список параметров. Список может быть пустым, тогда он указывается как `(void)` (слово `void` может быть опущено: `()`).

За заголовком функции размещается тело функции. Тело функции – это последовательность определений, описаний и исполняемых операторов, заключенных в фигурные скобки. Каждое определение, описание или оператор заканчивается точкой с запятой.

Определения – вводят объекты (объект – это именованная область памяти, частный случай объекта – переменная), необходимые для представления в программе обрабатываемых данных. Примерами являются

```
const int y = 10 ; //именованная константа
float x ;  //переменная
```

Описания – уведомляют компилятор о свойствах и именах объектов и функций, описанных в других частях программы.

Операторы – определяют действия программы на каждом шаге ее исполнения.

## 4. Элементы языка C/C++

1) Алфавит языка который включает
   - прописные и строчные латинские буквы и знак подчеркивания;
   - арабские цифры от 0 до 9;
   - специальные знаки "{},| []()+-/%*.\':;&?<>=!#^
   - пробельные символы (пробел, символ табуляции, символы перехода на новую строку).
2) Из символов формируются лексемы языка:
   - *Идентификаторы* – имена объектов C/C++-программ. В идентификаторе могут быть использованы латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, `PROG1`, `prog1` и `Prog1` – три различных идентификатора. Первым символом должна быть буква или знак подчеркивания (но не цифра). Пробелы в идентификаторах не допускаются.
   - *Ключевые (зарезервированные) слова* – это слова, которые имеют специальное значение для компилятора. Их нельзя использовать в качестве идентификаторов.
   - *Знаки операций* – это один или несколько символов, определяющих действие над операндами. Операции делятся на унарные, бинарные и тернарную по количеству участвующих в этой операции операндов.
   - *Константы* – это неизменяемые величины. Существуют целые, вещественные, символьные и строковые константы. Компилятор выделяет константу в качестве лексемы (элементарной конструкции) и относит ее к одному из типов по ее внешнему виду.
   - *Разделители* – скобки, точка, запятая пробельные символы.

## 5. Константы в C/C++

Константа – это лексема, представляющая изображение фиксированного числового, строкового или символьного значения. Константы делятся на 5 групп:

- целые;
- вещественные (с плавающей точкой);
- перечислимые;
- символьные;
- строковые.

Компилятор выделяет лексему и относит ее к той или другой группе, а затем внутри группы к определенному типу по ее форме записи в тексте программы и по числовому значению.

*Целые константы* могут быть десятичными, восьмеричными и шестнадцатеричными. Десятичная константа определяется как последовательность десятичных цифр, начинающаяся не с 0, если это число не 0 (примеры: 8, 0, 192345). Восьмеричная константа – это константа , которая всегда начинается с 0. За 0 следуют восьмеричные цифры (примеры: 016 – десятичное значение 14, 01). Шестнадцатеричные константы – последовательность шестнадцатеричных цифр, которым предшествуют символы 0x или 0X (примеры: 0xA, 0X00F).

В зависимости от значения целой константы компилятор по-разному представит ее в памяти компьютера (т. е. компилятор припишет константе соответствующий тип данных).

*Вещественные константы* имеют другую форму внутреннего представления в памяти компьютера. Компилятор распознает такие константы по их виду. Вещественные константы могут иметь две формы представления: с фиксированной точкой и с плавающей точкой. Вид константы с фиксированной точкой: [цифры].[цифры] (примеры: 5.7, .0001, 41.). Вид константы с плавающей точкой: [цифры][.][цифры]E|e[+|-][цифры] (примеры:0.5e5, .11e-5, 5E3). В записи вещественных констант может опускаться либо целая, либо дробная части, либо десятичная точка, либо признак экспоненты с показателем степени.

*Перечислимые константы* вводятся с помощью ключевого слова enum. Это обычные целые константы, которым приписаны уникальные и удобные для использования обозначения.

```
enum {one=1, two=2, three=3, four=4};
enum {zero,one,two,three};
```

Если в определении перечислимых констант опустить знаки = и числовые значения, то значения будут приписываться по умолчанию. При этом самый левый идентификатор получит значение 0, а каждый последующий будет увеличиваться на 1.

```
enum {ten=10, three=3, four, five, six};
enum {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday};
```

*Символьные константы* – это один или два символа, заключенные в апострофы. Символьные константы, состоящие из одного символа, имеют тип char и занимают в памяти один байт, символьные константы, состоящие из двух символов, имеют тип `int` и занимают два байта. Последовательности, начинающиеся со знака \, называются управляющими, они используются:

- для представления символов, не имеющих графического отображения, например:

`\a` – звуковой сигнал,

`\b` – возврат на один шаг,

`\n` – перевод строки,

`\t` – горизонтальная табуляция;

- для представления символов: \ , ’ , ? , ” ( \\, \’ ,\? ,\” );
- для представления символов с помощью шестнадцатеричных или восьмеричных кодов (`\073, \0xF5`);

*Строковая константа* – это последовательность символов, заключенная в кавычки. Внутри строк также могут использоваться управляющие символы. Например:

```
"\nНовая строка",
"\n\"Алгоритмические языки программирования"".
```

## 6. Типы данных в C/C++

Типы C/C++ можно разделить на простые и составные. К простым типам относят типы, которые характеризуются одним значением. В языке C определено 4, а в C++ – 6 простых типов данных:

- `int` (целый)
- `char` (символьный)
- `wchar_t` (расширенный символьный) (C++)
- `bool` (логический) (C++)

} целочисленные

- `float` (вещественный)
- `double` (вещественный с двойной точностью)

} с плавающей точкой

Существует 4 спецификатора типа, уточняющих внутреннее представление и диапазон стандартных типов

`short` (короткий)
`long` (длинный)
`signed` (знаковый)
`unsigned` (беззнаковый)

### 6.1. Тип `int`

Значениями этого типа являются целые числа.

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

Если перед `int` стоит спецификатор `short`, то под число отводится 2 байта, а если спецификатор `long`, то 4 байта. От количества отводимой под объект памяти зависит множество допустимых значений, которые может принимать объект:

`short int` – занимает 2 байта, следовательно, имеет диапазон –32768 … +32767;

`long int` – занимает 4 байта, следовательно, имеет диапазон –2 147 483 648 … +2 147 483 647.

Тип `int` совпадает с типом `short int` на 16-разрядных ПК и с типом `long int` на 32-разрядных ПК.

Модификаторы `signed` и `unsigned` также влияют на множество допустимых значений, которые может принимать объект:

`unsigned short int` – занимает 2 байта, следовательно, имеет диапазон 0 … 65536;

`unsigned long int` – занимает 4 байта, следовательно, имеет диапазон 0 … +4 294 967 295.

### 6.2. Тип `char`

Значениями этого типа являются элементы конечного упорядоченного множества символов. Каждому символу ставится в соответствие число, которое называется кодом символа. Под величину символьного типа отводится 1 байт. Тип `char` может

использоваться со спецификаторами `signed` и `unsigned`. В данных типа `signed char` можно хранить значения в диапазоне от –128 до 127. При использовании типа `unsigned char` значения могут находиться в диапазоне от 0 до 255. Для кодировки используется код ASCII (American Standard Code for International Interchange). Символы с кодами от 0 до 31 относятся к служебным и имеют самостоятельное значение только в операторах ввода-вывода.

Величины типа char также применяются для хранения чисел из указанных диапазонов.

### 6.3. Тип wchar_t

Предназначается для работы с набором символов, для кодировки которых недостаточно 1 байта, например Unicode. Размер этого типа, как правило, соответствует типу `short`. Строковые константы такого типа записываются с префиксом `L`: `L"String #1"`.

### 6.4. Тип bool

Тип `bool` называется логическим. Его величины могут принимать значения `true` и `false`. Внутренняя форма представления `false` – 0, любое другое значение интерпретируется как `true`.

### 6.5. Типы с плавающей точкой

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

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

Если перед именем типа `double` стоит спецификатор `long`, то под величину отводится байтов.

### 6.6. Тип void

К основным типам также относится тип `void`. Множество значений этого типа – пусто.

## 7. Переменные

Переменная в C++ – именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение. Перед использованием любая переменная должна быть описана.

```
int a; float x;
```

Общий вид оператора описания:

[класс памяти][const]тип имя [инициализатор];

Класс памяти определяет время жизни и область видимости переменной. Время жизни может быть постоянным – в течение выполнения программы или временным – в течение блока. Область видимости - это та часть программы, из которой можно обратиться к переменной обычным образом. В зависимости от области действия переменная может быть локальной и глобальной. Локальная переменная определена внутри блока (т. е. внутри оператора {…}), область ее действия от точки описания до конца блока. Глобальная переменная определена вне блока, область ее действия от точки описания до конца файла. Класс памяти может принимать значения: `auto, extern, static, register`. Если класс памяти не указан явно, то компилятор определяет его исходя из контекста объявления.

Классы памяти:

`auto` – автоматическая локальная переменная. Спецификатор `auto` может быть задан только при определении объектов блока, например, в теле функции. Этим переменным память выделяется при входе в блок и освобождается при выходе из него. Вне блока такие переменные не существуют.

`extern` – глобальная переменная, она находится в другом месте программы (в другом файле или долее по тексту). Используется для создания переменных, которые доступны во всех файлах программы.

`static` – статическая переменная, она существует только в пределах того файла, где определена переменная.

`register` – аналогичны `auto`, но память под них выделяется в регистрах процессора. Если такой возможности нет, то переменные обрабатываются как `auto`.

```
int a;          //глобальная переменная
void main()
{
     int b;     //локальная переменная
     extern int x;  //переменная   x   определена   в   другом
месте,
                    // память не выделяется
     static int c;  //локальная статическая переменная
     a=1;       //присваивание глобальной переменной
     int a;     //локальная переменная a
     a=2;       //присваивание локальной переменной
     ::a=3;     //присваивание глобальной переменной
}
int x=4;        //определение и инициализация x
```

В примере переменная `a` определена вне всех блоков. Областью действия переменной `a` является вся программа, кроме тех строк, где используется локальная переменная `a`. Переменные `b` и `c` – локальные, область их видимости – блок. Время жизни различно: память под `b` выделяется при входе в блок (т. к. по умолчанию класс памяти `auto`), освобождается при выходе из него. Переменная `c` (`static`) существует, пока работает программа.

Если при определении начальное значение переменным не задается явным образом, то компилятор обнуляет глобальные и статические переменные. Автоматические переменные не инициализируются.

Имя переменной должно быть уникальным в своей области действия.

Описание переменной может быть выполнено или как объявление, или как определение. Объявление содержит информацию о классе памяти и типе переменной,

определение вместе с этой информацией дает указание выделить память. В примере `extern int x;` – объявление, а остальные – определения.

`const` – показывает, что эту переменную нельзя изменять (именованная константа).

При описании можно присвоить переменной начальное значение (инициализация).

```
const num=10;
```

## 8. Выражения

Из констант, переменных, разделителей и знаков операций можно конструировать выражения. Каждое выражение представляет собой правило вычисления нового значения. Каждое выражение состоит из одного или нескольких операндов, символов операций и ограничителей. Если выражение формирует целое или вещественное число, то оно называется арифметическим. Пара арифметических выражений, объединенная операцией сравнения, называется отношением. Если отношение имеет ненулевое значение, то оно – истинно, иначе – ложно.

## 9. Ввод и вывод данных

В языке C/C++ нет встроенных средств ввода и вывода – он осуществляется с помощью функций, типов и объектов, которые находятся в стандартных библиотеках. Существует два основных способа: функции C и объекты C++.

Для ввода/вывода данных в стиле C используются функции, которые описываются в библиотечном файле `stdio.h`.

- `printf` (форматная строка, список аргументов);

форматная строка – строка символов, заключенных в кавычки, которая показывает, как должны быть напечатаны аргументы. Например:

```
printf (”Значение числа Пи равно %f\n”, pi);
```

Форматная строка может содержать:

- символы печатаемые текстуально;
- спецификации преобразования;
- управляющие символы.

Каждому аргументу соответствует своя спецификация преобразования:

`%d, %i` – десятичное целое число;

`%f` – число с плавающей точкой;

`%e,%E` – число с плавающей точкой в экспоненциальной форме;

`%u` – десятичное число в беззнаковой форме;

`%c` – символ;

`%s` – строка.

В форматную строку также могут входить управляющие символы:

`\n` – управляющий символ новая строка;

`\t` – табуляция;

`\a` – звуковой сигнал и др.

Также в форматной строке могут использоваться модификаторы формата, которые управляют шириной поля, отводимого для размещения выводимого значения. Модификаторы – это числа, которые указывают минимальное количество позиций для вывода значения и количество позиций ля вывода дробной части числа:

%[-]m[.p]C, где

– – задает выравнивание по левому краю,

m – минимальная ширина поля,

p – количество цифр после запятой для чисел с плавающей точкой и минимальное количество выводимых цифр для целых чисел (если цифр в числе меньше, чем значение p, то выводятся начальные нули),

C – спецификация формата вывода.

```
printf("\nСпецификации формата:\n%10.5d - целое, \n \\
%10.5f - с плавающей точкой\\n %10.5e - \\
в экспоненциальной форме\n%10s - строка", 10, 10.0, 10.0, "10");
```

Будет выведено:

```
Спецификации формата:
     00010 – целое
  10.00000 – с плавающей точкой
1.00000e+001 - в экспоненциальной форме
        10 – строка.
```

- scanf (форматная строка, список аргументов);

в качестве аргументов используются адреса переменных. Например:

```
scanf(" %d%f ", &x,&y);
```

При использовании библиотеки классов C++, используется библиотечный файл iostream.h, в котором определены стандартные потоки ввода данных от клавиатуры cin и вывода данных на экран cout, а также соответствующие операции

<< – операция записи данных в поток;

>> – операция чтения данных из потока.

```
#include <iostream.h>;
…
cout << "\nВведите количество элементов: ";
cin >> n;
```

## 10. Операторы C/C++

### 10.1. Базовые конструкции структурного программирования

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур: линейной, разветвляющейся и циклической. Эти структуры называются базовыми конструкциями структурного программирования.

Линейной называется конструкция, представляющая собой последовательное соединение двух или более операторов.

Ветвление – задает выполнение одного из двух операторов, в зависимости от выполнения какого либо условия.

Цикл – задает многократное выполнение оператора.

| Следование | Ветвление | Цикл |
|---|---|---|

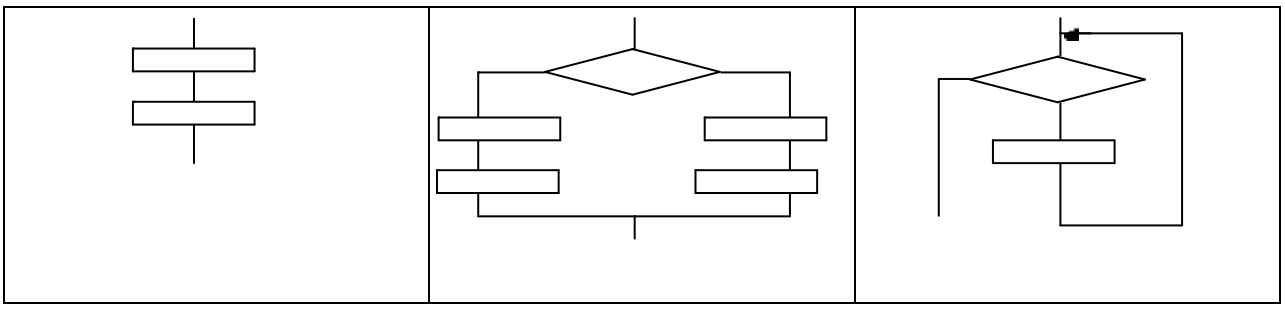


**Рис. 3 Базовые конструкции структурного программироваия**

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения. Структурное программирование также называют программированием без goto, т. к. частое использование операторов перехода затрудняет понимание логики работы программы. Но иногда встречаются ситуации, в которых применение операторов перехода, наоборот, упрощает структуру программы.

Операторы управления работой программы называют управляющими конструкциями программы. К ним относят:

- составные операторы;
- операторы выбора;
- операторы циклов;
- операторы перехода.

### 10.2. Оператор «выражение»

Любое выражение, заканчивающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении этого выражения. Частным случаем выражения является пустой оператор `;` (точка с запятой).

```
i++;
a+=2;
x=a+b;
```

### 10.2. Составные операторы

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

```
{
n++;                    //это составной оператор
summa+=n;
}

{
int n=0;
n++;                    //это блок
summa+=n;
}
```

### 10.3. Операторы выбора

Операторы выбора – это условный оператор и переключатель.

1. Условный оператор имеет полную и сокращенную форму.

```
if (выражение-условие) оператор;    //сокращенная форма
```

В качестве выражения-условия могут использоваться арифметическое выражение, отношение и логическое выражение. Если значение выражения-условия отлично от нуля (т. е. истинно), то выполняется оператор.

```
if (x<y&&x<z)min=x;
if (выражение-условие) оператор1;  //полная форма
else оператор2;
```

Если значение выражения-условия отлично от нуля, то выполняется оператор1, при нулевом значении выражения-условия выполняется оператор2.

```
if (d>=0)
{
x1=(-b-sqrt(d))/(2*a);
x2=(-b+sqrt(d))/(2*a);
cout<< "\nx1="<<x1<<"x2="<<x2;
}
else cout<<"\nРешения нет";
```

2. Переключатель определяет множественный выбор.

```
switch (выражение)
{
case константа1 : оператор1 ;
case константа2 : оператор2 ;
. . . . . .
[default: операторы;]
}
```

При выполнении оператора `switch`, вычисляется выражение, записанное после `switch`, оно должно быть целочисленным. Полученное значение последовательно сравнивается с константами, которые записаны следом за `case`. При первом же совпадении выполняются операторы, помеченные данной меткой. Если выполненные операторы не содержат оператора перехода, то далее выполняются операторы всех следующих вариантов, пока не появится оператор перехода или не закончится переключатель. Если значение выражения, записанного после `switch`, не совпало ни с одной константой, то выполняются операторы, которые следуют за меткой `default`. Метка `default` может отсутствовать.

```
#include <iostream.h>
void main()
{
     int i;
     cout<<"\nEnter the number";
     cin>>i;
     switch(i)
```

```
     {
     case 1:cout<<"\nthe number is one";
     case 2:cout<<"\n2*2="<<i*i;
     case 3: cout<<"\n3*3="<<i*i;break;
     case 4: cout<<"\n"<<i<<" is very beautiful!";
     default:cout<<"\nThe end of work";
     }
}
```

Результаты работы программы:

1. При вводе 1 будет выведено:

```
The number is one
2*2=1
3*3=1
```

2. При вводе 2 будет выведено:

```
2*2=4
3*3=4
```

3. При вводе 3 будет выведено:

```
3*3=9
```

4. При вводе 4 будет выведено:

```
4 is very beautiful!
```

5. При вводе всех остальных чисел будет выведено:

```
The end of work
```

### 10.4. Операторы циклов

- Цикл с предусловием:

```
while (выражение-условие)
оператор;
```

В качестве <выражения-условия> чаще всего используется отношение или логическое выражение. Если оно истинно, т. е. не равно 0, то тело цикла выполняется до тех пор, пока выражение-условие не станет ложным.

```
while (a!=0)
{
cin>>a;
s+=a;
}
```

- Цикл с постусловием:

```
do
оператор
while (выражение-условие);
```

Тело цикла выполняется до тех пор, пока выражение-условие истинно.

```
do
{
cin>>a;
```

```
s+=a;
}
while(a!=0);
```

- Цикл с параметром:

```
for (выражение_1;выражение-условие;выражение_3)
оператор;
```

выражение_1 и выражение_3 могут состоять из нескольких выражений, разделенных запятыми. Выражение_1 – задает начальные условия для цикла (инициализация). Выражение-условие определяет условие выполнения цикла, если оно не равно 0, цикл выполняется, а затем вычисляется значение выражения_3. Выражение_3 – задает изменение параметра цикла или других переменных (коррекция). Цикл продолжается до тех пор, пока выражение-условие не станет равно 0. Любое выражение может отсутствовать, но разделяющие их « ; » должны быть обязательно.

```
1.
for ( n=10; n>0; n--)// Уменьшение параметра
{
оператор;
}
```

```
2.
for ( n=2; n>60; n+=13)// Изменение шага корректировки
{
оператор;
}
```

```
3.
for ( num=1;num*num*num<216; num++)//проверка условия отличного от
//того, которое налагается на число итераций
{
оператор;
}
```

```
4.
for ( d=100.0; d<150.0;d*=1.1)//коррекция с помощью
//умножения
{
оператор;
 }
```

```
5.
for (x=1;y<=75;y=5*(x++)+10)//коррекция с помощью
//арифметического выражения
{
оператор;
}
```

```
6.
for (x=1, y=0; x<10;x++;y+=x);//использование нескольких
корректирующих выражений, тело цикла отсутствует
```

### 10.5. Операторы перехода

Операторы перехода выполняют безусловную передачу управления.

- `break` – оператор прерывания цикла.

```
{
оператор;
if (<выражение_условие>) break;
оператор;
}
```

Т. е. оператор `break` целесообразно использовать, когда условие продолжения итераций надо проверять в середине цикла.

```
     // Найти сумму чисел, числа вводятся с клавиатуры до тех
пор, пока не будет //введено 100 чисел или 0.
     for(s=0, i=1; i<100;i++)
     {
     cin>>x;
     if( x==0) break; // если ввели 0, то суммирование
заканчивается
     s+=x;
     }
```

- `continue` – переход к следующей итерации цикла. Он используется, когда тело цикла содержит ветвления.

```
     //Найти количество и сумму положительных чисел
     for( k=0,s=0,x=1;x!=0;)
     {
     cin>>x;
     if (x<=0) continue;
     k++; s+=x;
     }
```

- `goto <метка>` – передает управление оператору, который содержит метку.

В теле той же функции должна присутствовать конструкция: `<метка>:оператор;`

Метка – это обычный идентификатор, областью видимости которого является функция. Оператор `goto` передает управления оператору, стоящему после метки. Использование оператора `goto` оправдано, если необходимо выполнить переход из нескольких вложенных циклов или переключателей вниз по тексту программы или перейти в одно место функции после выполнения различных действий.

Применение `goto` нарушает принципы структурного и модульного программирования, по которым все блоки, из которых состоит программа, должны иметь только один вход и только один выход.

Нельзя передавать управление внутрь операторов `if, switch` и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию, на операторы, которые стоят после инициализации.

```
int k;
goto m;
```

```
. . .
{
int a=3,b=4;
k=a+b;
m:   int c=k+1;
. . .
}
```

В этом примере при переходе на метку m не будет выполняться инициализация переменных a , b и k.

- `return` – оператор возврата из функции. Он всегда завершает выполнение функции и передает управление в точку ее вызова. Вид оператора:

```
return [выражение];
```

### 11. Примеры решения задач с использованием основных операторов C++

Решение задач по программированию предполагает выполнение ряда этапов:

1. Разработка математической модели. На этом этапе определяются исходные данные и результаты решения задачи, а также математические формулы, с помощью которых можно перейти от исходных данных к конечному результату.
2. Разработка алгоритма. Определяются действия, выполняя которые можно будет от исходных данных придти к требуемому результату.
3. Запись программы на некотором языке программирования. На этом этапе каждому шагу алгоритма ставится в соответствие конструкция выбранного алгоритмического языка.
4. Выполнение программы (исходный модуль ->компилятор ->объектный модуль -> компоновщик -> исполняемый модуль)
5. Тестирование и отладка программы. При выполнении программы могут возникнуть ошибки 3 типов:

- синтаксические – исправляются на этапе компиляции;
- ошибки исполнения программы (деление на 0, логарифм от отрицательного числа и т. п.) – исправляются при выполнении программы;
- семантические (логические) ошибки – появляются из-за неправильно понятой задачи, неправильно составленного алгоритма.

Чтобы устранить эти ошибки программа должна быть выполнена на некотором наборе тестов. Цель процесса тестирования – определение наличия ошибки, нахождение места ошибки, ее причины и соответствующие изменения программы – исправление. Тест – это набор исходных данных, для которых заранее известен результат. Тест выявивший ошибку считается успешным. Отладка программы заканчивается, когда достаточное количество тестов выполнилось неуспешно, т. е. программа на них выдала правильные результаты.

Для определения достаточного количества тестов существует два подхода. При первом подходе программа рассматривается как «черный ящик», в который передают исходные данные и получают результаты. Устройство самого ящика неизвестно. При этом подходе, чтобы осуществить полное тестирование, надо проверить программу на всех входных данных, что практически невозможно. Поэтому вводят специальные критерии, которые должны показать, какое конечное множество тестов является достаточным для программы. При первом подходе чаще всего используются следующие критерии:

1) тестирование классов входных данных, т. е. набор тестов должен содержать по одному представителю каждого класса данных:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| X | 0 | 1 | 0 | 1 | -1 | 1 | -1 |
| Y | 0 | 1 | 1 | 0 | 1 | -1 | -1 |

2) тестирование классов выходных данных, набор тестов должен содержать данные достаточные для получения по одному представителю из каждого класса выходных данных.

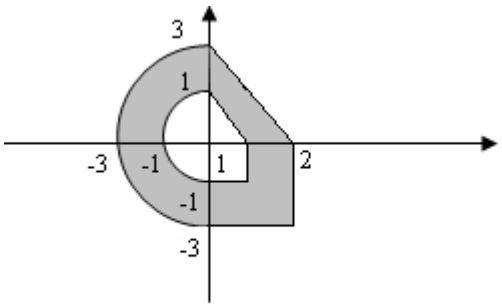
При втором подходе программа рассматривается как «белый ящик», для которого полностью известно устройство. Полное тестирование при этом подходе заканчивается после проверки всех путей, ведущих от начала программы к ее концу. Однако и при таком подходе полное тестирование программы невозможно, т. к. путей в программе с циклами бесконечное множество. При таком подходе используются следующие критерии:

1) Тестирование команд. Набор тестов должен обеспечивать прохождение каждой команды не менее одного раза.

2) Тестирование ветвей. Набор тестов в совокупности должен обеспечивать прохождение каждой ветви не менее одного раза. Это самый распространенный критерий в практике программирования.

### 11.1. Программирование ветвлений

**Задача №1**. Определить, попадет ли точка с координатами (x, y ) в заштрихованную область.




*Исходные данные: x,y*

*Результат:* да или нет

*Математическая модель:*

Ok=I || II || III || VI, где I, II, III, IV – условия попадания точки в заштрихованную область для каждого квадранта.

Квадрант I: Область формируется прямыми 0X и 0У, прямой, проходящей через точки (0,1)и(1,0) и прямой, проходящей через точки (0,3) и (2,0).

Необходимо определить уравнения прямых y=ax+b. Решаем две системы уравнений:

1) 1=a*0+b; 0=a*1+b;
2) 2=a*0+b; 0=a*3+b;

Из этих систем получаем следующие уравнения прямых:

y=-1x+1;

y=-2/3x+1;

Тогда условие попадания точки в I квадрант будет выглядеть следующим образом:

y>=-x+1&&y<=-2/3x+2&&y>=0&&x>=0.

Квадранты II и III: Область формируется прямыми 0X и 0У и двумя окружностями, описываемыми формулами $x^2+y^2=1$, $x^2+y^2=9$.

Тогда условие попадания точки во II и III квадранты будет выглядеть следующим образом:

$x^2+y^2>=1$&& $x^2+y^2<=9$&&&&x<=0.

Квадрант IV:

Область формируется двумя прямоугольниками. Точка может попадать либо в первый прямоугольник, либо во второй.

Условие попадания точки в IV квадрант будет выглядеть следующим образом:
(x>=0&&x<=1&&y<=-1&&y>=-3)|| (x>=1&&x<=3&&y<=0&&y>=-3) .

*Программа:*

```
#include <iostream.h>
#include <math.h>

void main()
{
     float x,y;
     cout<<"\nEnter x,y";
     cin>>x>>y;
     bool Ok=(y>=-x+1&&y<=2/3*x+2&&x>=0&&y>=0)||
(pow(x,2)+pow(y,2)>=1&&pow(x,2)+pow(y,2)<=9&&x<=0)||
     (x>=0&&x<=1&&y<=-1&&y>=-3)||(x>=1&&x<=2&&y<=0&&y>=-3);
     cout<<"\n"<<Ok;
}
```

*Тесты:*

| Квадрант | Исходные данные (X,Y) | Результат (Ok) |
|---|---|---|
| I | 0.2,0.2 | 0 |
| I | 0.7,0.5 | 1 |
| II | -0.5, 0.5 | 0 |
| II | -2,0 | 1 |
| III | -0.5,-0,5 | 0 |
| III | -2,-1 | 1 |
| IV | 0,5,-0.5 | 0 |
| IV | 1.5, -1 | 1 |
| Центр системы координат | 0,0 | 0 |

### 11.2. Программирование арифметических циклов

Для арифметического цикла заранее известно сколько раз выполняется тело цикла.

**Задача №2.** Дана последовательность целых чисел из n элементов. Найти среднее арифметическое этой последовательности.

```
#include <iostream.h>
#include <math.h>
void main()
{
     int a,n,i;
     double  s=0;  //инициализируем переменную начальным
значением
     cout<<"\nEnter n";
     cin>>n;    //    вводим    количество    элементов    в
последовательности
          for(i=1;i<=n;i++)// цикл выполняется n раз
          {
```

```
                cout<<"\nEnter a";
                cin>>a; // вводим переменную
                s+=a;//добавляем значение переменной к сумме
            }
            s=s/n;//находим среднее арифметическое
            cout<<"\nсреднее арифметическое равно="<<s<<"\n";
}
```

**Задача №3.** Найти значение S=1+2+3+4+. . . +N

```
#include <iostream.h>
#include <math.h>
void main()
{
//описываем переменные и инициализируем s начальным
значением
      int n,i,s=0;
      cout<<"\nEnter n";
      cin>>n;    //    вводим    количество    элементов    в
последовательности
      for(i=1; i<=n; i++)++)// цикл выполняется n раз
   s+=i; //добавляем значение переменной к сумме
      cout<<"\nS="<<s<<"\n";
}
```

### 11.3. Программирование итерационных циклов

Для итерационного цикла должно быть известно условие выполнения цикла. При использовании цикла с предусловием (while) тело цикла может не выполняться ни разу, если сразу же не выполняется условие цикла. При использовании цикла с постусловием тело цикла будет выполнено хотя бы один раз. И в том, и в другом случае используется условие выполнения цикла.

**Задача №5.** Дана последовательность целых чисел, за которой следует 0. Найти минимальный элемент этой последовательности.

Для решения этой задачи используем цикл с предусловием, т. к. ноль не входит в последовательность и его не надо обрабатывать при поиске минимального значения.

```
#include <iostream.h>
#include <math.h>
void main()
{
int a,min;
cout<<"\nEnter a";
cin>>a; //вводим первое число
min=a;//присваиваем переменной min начальное значение
while(a!=0)//цикл с предусловием
{
cout<<"\nEnter a";
cin>>a;//вводим следующее число
/*сравниваем с нулем, т. к. 0 не входит в последовательность
 и не может быть минимальным и с минимальным значением*/
if (a!=0&&a<min)
```

```
min=a; //запоминаем в min новое значение
}
cout<<"\nmin="<<min<<"\n";//вывод результата
}
```

Тесты:

| a | 2 | 55 | -3 | -10 | 0 |
|---|---|---|---|---|---|
| min | -10 | | | | |

| a | 12 | 55 | 4 | 27 | 0 |
|---|---|---|---|---|---|
| min | 4 | | | | |

| a | -6 | -43 | -15 | -10 | 0 |
|---|---|---|---|---|---|
| min | -10 | | | | |

Для решения этой же задачи можно написать программу с использованием цикла с постусловием.

```
#include <iostream.h>
#include <math.h>
void main()
{
int a,min;
cout<<"\nEnter a";
cin>>a; //вводим первое число
min=a;//присваиваем переменной min начальное значение
do //цикл с постусловием
{
cout<<"\nEnter a";
cin>>a;//вводим следующее число
if (a==0)break;// выход из цикла, если ввели 0
if (a<min)//сравниваем a с текущим min
min=a; //запоминаем в min новое значение
}
//бесконечный цикл, т. к. выход осуществляется с помощью
break
while(1);
cout<<"\nmin="<<min<<"\n";//вывод результата
}
```

Эту же задачу можно решить с помощью цикла for:

```
#include <iostream.h>
#include <math.h>
void main()
{
int a,min;
cout<<"\nEnter a";
cin>>a; //вводим первое число
min=a;//присваиваем переменной min начальное значение
for(;a!=0;)//цикл используется как цикл с предусловием
{
cout<<"\nEnter a";
```

```
cin>>a;//вводим следующее число
/*сравниваем с нулем, т. к. 0 не входит в последовательность
 и не может быть минимальным и с минимальным значением*/
if (a!=0&&a<min)
min=a; //запоминаем в min новое значение
}
cout<<"\nmin="<<min<<"\n";//вывод результата
}
```

**Задача №6**. Найти сумму чисел Фибоначчи, меньших заданного числа Q. Числа Фибоначчи – это последовательность чисел: 1, 1, 2, 3, 5, 8, 13, …., т. е. каждое следующее число – это сумма двух предыдущих.

```
#include<iostream.h>
void main()
{
     int a=1,//первое число
b=1,//второе число
s=2,//сумма чисел Фибоначчи
Q,
c;//следующее число
     cout<<"\nEnter Q";
     cin>>Q;//вводим число Q
     if(Q<=0)cout<<"Error in Q";
     else
     //если Q=1, то сумма тоже будет 1 (первое число)
     if(Q==1)s=1;
     else
     {
c=a+b;//вычисляем следующее число
while(c<Q)
     {
          s+=c;//вычисляем сумму
          a=b;//меняем первое число на второе
          b=c;//меняем второе число на текущее
          c=a+b;//вычисляем текущее число Фибоначчи
     }

}
cout<<"\nS="<<s<<"\n";//выводим результат
}
```

Тесты:

| Q | S |
|---|---|
| -1 | Error in Q |
| 0 | Error in Q |
| 1 | 1 |
| 2 | 2 |
| 10 | 20 |

### 11.4. Программирование вложенных циклов

Тело цикла может содержать любые операторы, в том числе и другие циклы. Оператор цикла, который содержится в теле другого цикла, называется вложенным.

**Задача №7**: Напечатать N простых чисел.

Разделим эту задачу на две подзадачи:

1. определить является ли число простым;
2. напечатать n чисел, удовлетворяющих заданному условию.

Простым называется число, которое делится только само на себя и на единицу (сама единица простым числом не является). Тогда, чтобы определить является число простым или нет, нужно проверить есть ли у него другие делители. Для этого будем делить число К на все числа от 2 и до К, используя цикл с постусловием. Если К разделится на d без остатка только, когда d станет равно К, значит, число простое:

```
          d=1;//начальное значение делителя
     do
          {
               d++;//увеличиваем делитель
          }
     /*цикл выполняется пока остаток от деления К на d не равен 0
(не делится)*/
          while(K%d!=0);
     if(K==d)//делитель равен К, т. е. число простое
               cout<<a<<" ";
```

Для решения второй подзадачи, нам нужно перебирать все числа, начиная с 2, и печатать только те, которые являются простыми. При этом, нужно подсчитывать напечатанные числа. Выполнение цикла закончится, когда будет напечатано n чисел.

```
K=1;//присваиваем начальное значение числу
//выполняем цикл пока не будет напечатано n чисел
for(int i=0;i<n;)
{
K++;//берем следующее число
if (K простое число)
i++;//увеличиваем счетчик простых чисел
}
```

Объединив эти два фрагмента вместе, получим решение поставленной задачи

```
#include<iostream.h>
void main()
{
     int K=1,n,d;
     cout<<"\nEnter N";
     cin>>n;
//неправильно задано число n
if(n<1) {
cout<<”\nerror in data”;
return;//завершение программы
}
     for(int i=0;i<n;)//внешний цикл
     {
     K++;d=1;
     do //внутренний цикл
```

```
        {
             d++;
        }
        while(K%d!=0);//конец внутреннего цикла
        if(K==d){
             cout<<K<<" ";
             i++;}

        }//конец внешнего цикла
}
```

Тесты

| | |
|---|---|
| n=0 | Error in data |
| n=1 | 2 |
| n=5 | 10 |

## 12. Массивы

В языке C/C++, кроме базовых типов, разрешено вводить и использовать производные типы, полученные на основе базовых. Стандарт языка определяет три способа получения производных типов:

- массив элементов заданного типа;
- указатель на объект заданного типа;
- функция, возвращающая значение заданного типа.

Массив – это упорядоченная последовательность переменных одного типа. Каждому элементу массива отводится одна ячейка памяти. Элементы одного массива занимают последовательно расположенные ячейки памяти. Все элементы имеют одно имя – имя массива и отличаются индексами – порядковыми номерами в массиве. Количество элементов в массиве называется его размером. Чтобы отвести в памяти нужное количество ячеек для размещения массива, надо заранее знать его размер. Резервирование памяти для массива выполняется на этапе компиляции программы.

### 12.1. Определение массива в C/C++

Массивы определяются следующим образом:

```
int a[100];//массив из 100 элементов целого типа
```

Операция `sizeof(a)` даст результат 400, т. е. 100 элементов по 4 байта.
Элементы массива всегда нумеруются с 0.

| 45 | 352 | 63 | | 124 | значения элементов массива |
|---|---|---|---|---|---|
| 0 | 1 | 2 | ….. | 99 | индексы элементов массива |

**Рис. 4. Размещение массива в памяти**

Чтобы обратиться к элементу массива, надо указать имя массива и номер элемента в массиве (индекс):

`a[0]` – индекс задается как константа,
`a[55]` – индекс задается как константа,
`a[i]` – индекс задается как переменная,
`a[2*i]` – индекс задается как выражение.

Элементы массива можно задавать при его определении:

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

Операция `sizeof(a)` даст результат 40, т. е. 10 элементов по 4 байта.

```
int a[10]={1,2,3,4,5};
```

Операция `sizeof(a)` даст результат 40, т. е. 10 элементов по 4 байта. Если количество начальных значений меньше, чем объявленная длина массива, то начальные элементы массива получат только первые элементы.

```
int a[]={1,2,3,4,5};
```

Операция `sizeof(a)` даст результат 20, т. е. 5 элементов по 4 байта. Длина массива вычисляется компилятором по количеству значений, перечисленных при инициализации.

### 12.2. Примеры решения задач и использованием массивов

**Задача 1.** Формирование массива с помощью датчика случайных чисел.

Для того чтобы сформировать массив необходимо организовать цикл для перебора элементов массива. Для формирования каждого элемента вызывается функция `rand()`, которая возвращает псевдослучайное число из диапазона от 0 до 32767. Чтобы получить значения из диапазона от 0 до 100 используется операция получения остатка от деления %. Если нужно получить не только положительные, но и отрицательные числа вычитается число k равное половине от требуемого диапазона.

```
a[I]=rand()%100-50;//числа из диапазона -50..49.
```

Чтобы использовать функцию `rand()`, надо подключить библиотечный файл `<stdlib.h>`

```
#include <iostream.h>
#include <stdlib.h> //файл с описанием функции rand()
void main()
{
int a[100];
int n;
cout<<"\nEnter the size of array:";cin>>n;
for(int I=0;I<n;I++)
{
     a[I]=rand()%100;// числа из диапазона 0..100
     cout<<a[I]<<" ";    //вывод элемента массива
}
}
```

**Задача 3.** Найти максимальный элемент массива.

Для решения этой задачи нужно сформировать массив, присвоить начальное значение в переменную max, в качестве начального значения выступает первый элемент массива, а затем сравнить каждый элемент массива с переменной max. Если значение элемента будет больше max, то его значение записывается в max и последующие элементы сравниваются уже с этим значением.

```
#include <iostream.h>
#include <stdlib.h>
void main()
{
      int a[100];
      int n;
      cout<<”\nEnter the size of array:”;
      cin>>n;
      for(int I=0;I<n;I++)          //заполнение массива
      {
      a[I]=rand()%100-50;
      cout<<a[I]<<” ”;
 }

int max=a[0];  //задаем начальное значение max
for(I=1;I<n;I++)
//сравниваем элементы массива с max
     if(a[I]>max)max=a[I];

cout<<”\nMax=”<<max;
}
```

**Задача 3.** Найти сумму элементов массива с четными индексами.
Для решения этой задачи можно предложить несколько способов.

1. Массив перебирается с шагом 2 и все элементы суммируются.
2. Массив перебирается с шагом 1 и суммируются только элементы, имеющие четные индексы. Для проверки на четность используется операция получения остатка от деления на 2.

```
//Первый способ
#include <iostream.h>
#include <stdlib.h>
void main()
{
int a[100];
int n;
cout<<”\nEnter the size of array:”;
cin>>n;
for(int I=0;I<n;I++)
{
     //заполнение массива случайными числами
     a[I]=rand()%100-50;
     cout<<a[I]<<” ”;
}
int Sum=0;
for(I=0; I<n; I+=2)
//суммируются элементы с индексами 0, 2, 4, и т. д.
                  Sum+=a[I];

cout<<”\nSum=”<<Sum”;
}
```

```
//Второй способ
#include <iostream.h>
#include <stdlib.h>
void main()
{
int a[100];
int n;
cout<<”\nEnter the size of array:”;
cin>>n;
for(int I=0;I<n;I++)
{
     //заполнение массива случайными числами
     a[I]=rand()%100-50;
     cout<<a[I]<<” ”;
}
int Sum=0;
for(I=0; I<n; I++)
                  if(I%2==0)
//суммируются элементы с индексами 0, 2, 4, и т. д.
                  Sum+=a[I];

cout<<”\nSum=”<<Sum”;
}
```

## 13. Указатели

### 13.1. Понятие указателя

Указатели являются специальными объектами в программах на C/C++. Указатели предназначены для хранения адресов памяти.

Когда компилятор обрабатывает оператор определения переменной, например, `int i=10;`, то в памяти выделяется участок памяти в соответствии с типом переменной (для `int` размер участка памяти составит 4 байта) и записывает в этот участок указанное значение. Все обращения к этой переменной компилятор заменит адресом области памяти, в которой хранится эта переменная.

**Рис. 5 Значение переменной и ее адрес**

Программист может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями. Указатель не является самостоятельным типом, он всегда связан с каким-то другим типом.

Указатели делятся на две категории: указатели на объекты и указатели на функции. Рассмотрим указатели на объекты, которые хранят адрес области памяти, содержащей данные определенного типа.

В простейшем случае объявление указателя имеет вид:

```
тип* имя;
```

Знак *, обозначает указатель и относится к типу переменной, поэтому его рекомендуется ставить рядом с типом, а от имени переменной отделять пробелом, за исключением тех случаев, когда описываются несколько указателей. При описании нескольких указателей знак * ставится перед именем переменной-указателя, т. к. иначе будет не понятно, что эта переменная также является указателем.

```
int* i;
double *f, *ff;//два указателя
char* c;
```

Тип может быть любым, кроме ссылки.

Размер указателя зависит от модели памяти. Можно определить указатель на указатель: `int** a;`

Указатель может быть константой или переменной, а также указывать на константу или переменную.

```
int i;              //целая переменная
const int ci=1;     //целая константа
int* pi;       //указатель на целую переменную
const int* pci;     //указатель на целую константу
```

Указатель можно сразу проинициализировать:

```
//указатель на целую переменную
int* pi=&i;
//указатель на целую константу
const int* pci=&ci;

//указатель-константа на переменную целого типа
int* const cpi=&i;

//указатель-константа на целую константу
const int* const cpc=&ci;
```

Если модификатор `const` относится к указателю (т. е. находится между именем указателя и *), то он запрещает изменение указателя, а если он находится слева от типа (т. е. слева от *), то он запрещает изменение значения, на которое указывает указатель.

Для инициализации указателя существуют следующие способы:

- с помощью операции получения адреса

```
int a=5;
int* p=&a; или int p(&a);
```

- с помощью проинициализированного указателя

```
int* r=p;
```

- адрес присваивается в явном виде

```
char* cp=(char*)0x B800 0000;
```

где `0x B800 0000` – шестнадцатеричная константа,
`(char*)` – операция приведения типа.

- присваивание пустого значения:

```
int* N=NULL;
int* R=0;
```

### 13.2. Динамическая память

Все переменные, объявленные в программе размещаются в одной непрерывной области памяти, которую называют сегментом данных (64 Кб). Такие переменные не меняют своего размера в ходе выполнения программы и называются статическими. Размера сегмента данных может быть недостаточно для размещения больших массивов информации. Выходом из этой ситуации является использование динамической памяти. Динамическая память – это память, выделяемая программе для ее работы за вычетом сегмента данных, стека, в котором размещаются локальные переменные подпрограмм и собственно тела программы.

Для работы с динамической памятью используют указатели. С их помощью осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными. Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программ, либо до тех пор, пока не будут уничтожены с помощью специальных функций или операций.

Для создания динамических переменных используют операцию new, определенную в C++:

```
указатель = new имя_типа[инициализатор];
```

где инициализатор – выражение в круглых скобках.

Операция `new` позволяет выделить и сделать доступным участок динамической памяти, который соответствует заданному типу данных. Если задан инициализатор, то в этот участок будет занесено значение, указанное в инициализаторе.

```
int* x=new int(5);
```

Для удаления динамических переменных используется операция delete, определенная в C++:

```
delete указатель;
```

где указатель содержит адрес участка памяти, ранее выделенный с помощью операции `new`.

```
delete x;
```

### 13.3. Операции с указателями

С указателями можно выполнять следующие операции:

- разыменование (`*`);
- присваивание;
- арифметические операции (сложение с константой, вычитание, инкремент ++, декремент `--`);
- сравнение;
- приведение типов.

Операция разыменования предназначена для получения значения переменной или константы, адрес которой хранится в указателе. Если указатель указывает на переменную, то это значение можно изменять, также используя операцию разыменования.

```
int a; //переменная типа int
     int* pa=new int; //указатель и выделение памяти под
//динамическую переменную
     *pa=10;//присвоили значение динамической
//переменной, на которую указывает указатель
a=*pa;//присвоили значение переменной a
```

Присваивать значение указателям-константам запрещено.

Приведение типов. На одну и ту же область памяти могут ссылаться указатели разного типа. Если применить к ним операцию разыменования, то получатся разные результаты.

```
int a=123;
int* pi=&a;
char* pc=(char*)&a;
float* pf=(float*)&a;
printf("\n%x\t%i",pi,*pi);
printf("\n%x\t%c",pc,*pc);
printf("\n%x\t%f",pf,*pf);
```

При выполнении этой программы получатся следующие результаты:

```
66fd9c  123
66fd9c  {
66fd9c  0.000000
```

Т. е. адрес у трех указателей один и тот же, но при разыменовании получаются разные значения в зависимости от типа указателя.

В примере при инициализации указателя была использована операция приведения типов. При использовании в выражении указателей разных типов, явное преобразование требуется для всех типов, кроме `void*`. Указатель может неявно преобразовываться в значения типа `bool`, при этом ненулевой указатель преобразуется в `true`, а нулевой в `false`.

Арифметические операции применимы только к указателям одного типа.

- Инкремент увеличивает значение указателя на величину `sizeof(тип)`.

```
char* pc;
int* pi;
double* pd;
. . . . .
pc++;       //значение увеличится на 1
pi++;       //значение увеличится на 4
pd++;       //значение увеличится на 8
```

- Декремент уменьшает значение указателя на величину `sizeof(тип)`.

- Разность двух указателей – это разность их значений, деленная на размер типа в байтах.

```
int a=123, b=456, c=789;
int* pi1=&a;
int* pi2=&b;
int* pi3=&c;
printf("\n%x",pi1-pi2);
printf("\n%x",pi1-pi3);
```

Результат выполнения программы:

```
1
2
```

Суммирование двух указателей не допускается.

- Можно суммировать указатель и константу:

```
pi3=pi3+2;
pi2=pi2+1;
printf("\n%x\t%d",pi1,*pi1);
printf("\n%x\t%d",pi2,*pi2);
printf("\n%x\t%d",pi3,*pi3);
```

Результат выполнения программы:

```
66fd9c  123
66fd9c  123
66fd9c  123
```

**14. Ссылки**

Ссылка – это синоним имени объекта, указанного при инициализации ссылки.
Формат объявления ссылки

```
тип & имя =имя_объекта;
```

```
int  x;        // определение переменной
int&  sx=x;    // определение ссылки на переменную x
const char& CR='\n';     //определение ссылки на константу
```

Правила работы со ссылками:

- Переменная-ссылка должна явно инициализироваться при ее описании, если она не является параметром функции, не описана как extern или не ссылается на поле класса.
- После инициализации ссылке не может быть присвоено другое значение.
- Не существует указателей на ссылки, массивов ссылок и ссылок на ссылки.
- Операция над ссылкой приводит к изменению величины, на которую она ссылается.

Ссылка не занимает дополнительного пространства в памяти, она является просто другим именем объекта.

```
#include<iostream.h>
void main()
{
     int i=123;
     int& si=i;
     cout<<"\ni="<<i<<" si="<<si;
     i=456;
     cout<<"\ni="<<i<<" si="<<si;
     i=0;
     cout<<"\ni="<<i<<" si="<<si;
}
```

Выведется:

```
i=123 si=123
i=456 si=456
i=0 si=0
```

## 15. Указатели и массивы

### 15.1. Одномерные массивы и указатели

При определении массива ему выделяется память. После этого имя массива воспринимается как константный указатель того типа, к которому относятся элементы массива. Исключением является использование операции `sizeof(имя_массива)` и операции `&имя_массива`.

```
     int a[100];

     /*определение количества занимаемой массивом памяти, в нашем
случае это 4*100=400 байт*/
     int k=sizeof(a);

     /*вычисление количества элементов массива*/
     int n=sizeof(a)/sizeof(a[0]);
```

Результатом операции `&` является адрес нулевого элемента массива:

```
имя_массива==&имя_массива=&имя_массива[0]
```

Имя массива является указателем-константой, значением которой служит адрес первого элемента массива, следовательно, к нему применимы все правила адресной арифметики, связанной с указателями. Запись `имя_массива[индекс]` это выражение с двумя операндами: имя массива и индекс. Имя_массива – это указатель-константа, а индекс определяет смещение от начала массива. Используя указатели, обращение по индексу можно записать следующим образом: `*(имя_массива+индекс)`.

```
for(int i=0;i<n;i++)     //печать массива
cout<<*(a+i)<<" ";
```

```
/*к имени (адресу) массива добавляется константа i и
полученное значение разыменовывается*/
```

Так как имя массива является константным указателем, то его невозможно изменить, следовательно, запись `*(a++)` будет ошибочной, а `*(a+1)` – нет.

Указатели можно использовать и при определении массивов:

```
int  a[100]={1,2,3,4,5,6,7,8,9,10};

//поставили указатель на уже определенный массив
int* na=a;

/*выделили в динамической памяти место под массив из 100
элементов*/
int b = new int[100];
```

## 15.2. Многомерные массивы и указатели

Многомерный массив это массив, элементами которого служат массивы. Например, массив с описанием `int a[4][5]` – это массив из 4 указателей типа `int*`, которые содержат адреса одномерных массивов из 5 целых элементов (см. рис.3).

**Рис. 6. Выделение памяти под массив, элементами которого являются массивы**

Инициализация многомерных массивов выполняется аналогично одномерным массивам.

```
//проинициализированы все элементы массива
int     a[3][4]     =     {{11,22,33,44},     {55,66,77,88},
{99,110,120,130}};

//проинициализированы первые элементы каждой строки
int b[3][4] = {{1},{2},{3}};

//проинициализированы все элементы массива
int c[3][2]={1,2,3,4,5,6};
```

Доступ к элементам многомерных массивов возможен и с помощью индексированных переменных и с помощью указателей:

`a[1][1]` – доступ с помощью индексированных переменных,

`*(*(a+1)+1)` – доступ к этому же элементу с помощью указателей (см. рис.3).

### 15.3. Динамические массивы

Операция new при использовании с массивами имеет следующий формат:
new тип_массива

Такая операция выделяет для размещения массива участок динамической памяти соответствующего размера, но не позволяет инициализировать элементы массива. Операция new возвращает указатель, значением которого служит адрес первого элемента массива. При выделении динамической памяти размеры массива должны быть полностью определены.

```
//выделение динамической памяти 100*sizeof(int) байт
int* a = new int[100];

//выделение динамической памяти 10*sizeof(double) байт
double* b = new double[10];

//указатель на массив из 4 элементов типа long
long(*la)[4];

//выделение динамической памяти 2*4*sizeof(long) байт
la = new[2][4];

//другой способ выделения памяти под двумерный массив
int** matr = (int**) new int[5][10];
```

Изменять значение указателя на динамический массив надо аккуратно, т. к. этот указатель затем используется при освобождении памяти с помощью операции delete.

```
/*освобождает память, выделенную под массив, если a адресует
его начало*/
delete[] a;
delete[] b;
delete[] la;
```

Если квадратных скобок нет, то сообщение об ошибке выдаваться не будет, но помечен как свободный будет только первый элемент массива, и остальные ячейки окажутся недоступными для дальнейшего использования («мусор»).

```
//Удалить из матрицы строку с номером K
#include <iostream.h>
#include <string.h>

#include <iostream.h>
#include <stdlib.h>
void main()
{
     int n,m;       //размерность матрицы
     int i,j;
     cout<<"\nEnter n";
     cin>>n;        //ввод количества строки
     cout<<"\nEnter m";
     cin>>m;        //ввод количества столбцов
```

```
//выделение памяти под массив указателей на строки
     int** matr = new int* [n];

     //выделение памяти под элементы матрицы
     for(i=0;i<n;i++)
          matr[i] = new int [m];

     //заполнение матрицы
     for(i=0;i<n;i++)
     for(j=0;j<m;j++)
          matr[i][j] = rand()%10);

     //печать сформированной матрицы
     for(i=0;i<n;i++)
     {
          for(j=0;j<m;j++)
               cout<<matr[i][j]<<" ";
          cout<<"\n";//переход на следующую строку
     }

     //удаление строки с номером k
     int k;
     cout<<"\nEnter k";
     cin>>k;

     //формирование новой матрицы
     int** temp = new int*[n-1];
     for(i=0;i<n;i++)
          temp[i]=new int[m];

     //заполнение новой матрицы
     int t;
     for(i=0,t=0;i<n;i++)
     /*если номер строки отличен от k, переписываем строку в
новую матрицу*/
          if(i!=k)
          {
               for(j=0;j<m;j++)
                    temp[t][j] = matr[i][j];
               t++;
          }

     //удаление старой матрицы
     for(i=0;i<n;i++)
          delete matr[i];
     delete[] matr;

n--;      //уменьшаем количество строк

     //печать новой матрицы
     for(i=0;i<n;i++)
     {
```

```
            for(j=0;j<m;j++)
                 cout<<temp[i][j]<<" ";
            cout<<"\n";
       }
}
```

## 16. Символьная информация и строки

### 16.1. Представление символьной информации

Для символьных данных в C++ введен тип `char`. Для представления символьной информации используются символы, символьные переменные и текстовые константы.

```
//символ занимает один байт, его значение не меняется
const char c='c';

/*символьные переменные, занимают по одному байту, значения
могут меняться*/
char a,b;

//текстовая константа
const char *s="Пример строки\n";
```

Строка в C++ – это массив символов, заканчивающийся нуль-символом – `'\0'` (нуль-терминатором). По положению нуль-терминатора определяется фактическая длина строки. Количество элементов в таком массиве на 1 больше, чем изображение строки.

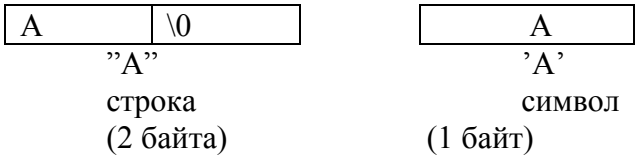



**Рис. 7. Представление строки и символа**

Присвоить значение строке с помощью оператора присваивания нельзя. Поместить строку в массив можно либо при вводе, либо с помощью инициализации.

```
void main()
{
     char s1[10]="string1";
     int k=sizeof(s1);
     cout<<s1<<"\t"<<k<<endl;
     char s2[]="string2";
     k=sizeof(s2);
     cout<<s2<<"\t"<<k<<endl;
     char s3[]={'s','t','r','i','n','g','3'}
     k=sizeof(s3);
     cout<<s3<<"\t"<<k<<endl;

//указатель на строку, ее нельзя изменить:
     char *s4="string4";
     k=sizeof(s4);
     cout<<s4<<"\t"<<k<<endl;
}
```

Результаты:
string1 10 – выделено 10 байтов, в том числе под \0
string2 8 – выделено 8 байтов (7+1байт под \0)
string3 8 – выделено 8 байтов (7+1байт под \0)
string4 4 – размер указателя

```
char *s="String5";  //выделяется 8 байтов для строки
char* ss;                //описан указатель

/*память не выделяется, поэтому программа может закончиться аварийно: */
ss="String6";

char *sss=new char[10];  //выделяем динамическую память
strcpy(sss,"String7");   //копируем строку в память
```

Для ввода и вывода символьных данных в библиотеке языка С определены следующие функции:

int getchar(void) – осуществляет вод одного символа их входного потока, при этом она возвращает один байт информации (символ) в виде значения типа int. Это сделано для распознавания ситуации, когда при чтении будет достигнут конец файла.

int putchar (int c) – помещает в стандартный выходной поток символ c.

char* gets(char*s) – считывает строку s из стандартного потока до появления символа '\n', сам символ '\n' в строку не заносится.

int puts(const char* s) записывает строку в стандартный поток, добавляя в конец строки символ '\n', в случае удачного завершения возвращает значение больше или равное 0 и отрицательное значение (EOF=-1) в случае ошибки.

```
char s[20];
cin>>s;        //ввод строки из стандартного потока
cout<<s;       //вывод строки в стандартный поток
```

При вводе строки "123 456 789", чтение байтов осуществляется до первого пробела, т. е. в строку s занесется только первое слово строки "123\0", следовательно, выведется: 123.

```
char s[20];
gets(s);       //ввод строки из стандартного потока
puts(s);       //вывод строки в стандартный поток
```

При вводе строки "123 456 789", чтение байтов осуществляется до символа '\n', т. е. в s занесется строка "123 456 789\n\0", при выводе строки функция puts возвращает еще один символ '\n', следовательно, будет выведена строка "123 456 789\n\n".

```
char s[20];
scanf("%s",s); //ввод строки из стандартного потока
printf("%s",s);     //вывод строки в стандартный поток
```

При вводе строки "123 456 789", чтение байтов осуществляется до первого пробела, т. е. в строку s занесется только первое слово строки "123/0", следовательно,

выведется: 123. Т. к. s – имя массива, т. е. адрес его первого элемента, операция & в функции scanf не используется.

### 16. 2. Библиотечные функции для работы со строками

Для работы со строками существуют специальные библиотечные функции, которые содержатся в заголовочном файле string.h.

**Таблица 1. Некоторые библиотечные функции для работы со строками**

| Прототип функции | Краткое описание | Примечание |
|---|---|---|
| `unsigned` **`strlen`**`(const char* s);` | Вычисляет длину строки s. | |
| `int` **`strcmp`**`(const char* s1, const char* s2);` | Сравнивает строки s1 и s2. | Если s1<s2, то результат отрицательный, если s1==s2, то результат равен 0, если s2>s1 – результат положительный. |
| `int` **`strcnmp`**`(const char* s1, const char* s2);` | Сравнивает первые n символов строк s1 и s2. | Если s1<s2, то результат отрицательный, если s1==s2, то результат равен 0, если s2>s1 – результат положительный. |
| `char*` **`strcpy`**`(char* s1, const char* s2);` | Копирует символы строки s1 в строку s2. | |
| `char*` **`strncpy`**`(char* s1, const char* s2, int n);` | Копирует n символов строки s1 в строку s2. | Конец строки отбрасывается или дополняется пробелами. |
| `char*` **`strcat`**`(char* s1, const char* s2);` | Приписывает строку s2 к строке s1 | |
| `char*` **`strncat`**`(char* s1, const char* s2);` | Приписывает первые n символов строки s2 к строке s1 | |
| `char*` **`strdup`**`(const char* s);` | Выделяет память и переносит в нее копию строки s | При выделении памяти используются функции |

### 16. 3. Примеры решения задач с использованием строк

**Задача 1**. Дана строка символов, состоящая из слов, слова разделены между собой пробелами. Удалить из строки все слова, начинающиеся с цифры

Для решения этой задачи сформируем массив слов исходной строки, а затем перенесём в строку из этого массива только слова, удовлетворяющие поставленному условию. Этот прием можно использовать для решения большинства задач, в которых нужно удалять или добавлять слова в исходную строку.

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

```
{
     char s[250], //исходная строка
     w[25],         //слово
     mas[10][25];   //массив слов
     puts(”\nвведите строку”);
     gets(s);
     int k=0,t=0,i,len,j;
     len=strlen(s);
     while(t<len)
          {
          for(j=0,i=t;s[i]!= ’ ’;i++,j++)
          w[j]=s[i];     //формируем слово до пробела
          w[j]=’/0’;     //формируем конец строки
          strcpy(mas[k],w);   //копируем слово в массив
          k++;      //увеличиваем счетчик слов
  //переходим к следующему слову в исходной строке
          t=i+1;
          }
     strcpy(s,””);//очищаем исходную строку
     for(t=0;t<k;t++)
     if(mas[t][0]<’0’ && mas[t][0]>’9’)
     //если первый символ не цифра
          {
          strcat(s,mas[t]);//копируем в строку слово
          strcat(s,” ”);//копируем в строку пробел
          }
     puts(s);//выводим результат
}
```

**Задача 2.** Сформировать динамический массив строк. Удалить из него строку с заданным номером.

В динамическом массиве строк каждая строка имеет свою длину, поэтому при формировании такого массива нужно выделять память под каждую строку отдельно.

```
#include <iostream.h>
#include <string.h>
void main()
{
     int n;
     cout<<”\nN=?”;
cin>>n;

     //вспомогательная переменная для ввода строки
          char s[100];

char** mas=new char*[n];
     for(int i=0;i<n;i++)
     {
          cout<<”\nS=?”;
          cin>>s;             //вводим строку
            /*выделяем столько памяти, сколько символов в
       строке + один байт для ’\0’ */
              mas[i]=new char[strlen(s)+1];
```

```
            //копируем строку в массив
                strcpy(mas[i],s);
        }

        for(i=0;i<n;i++)         //вывод массива
        {
            cout<<mas[i];
            cout<<"\n";
        }

        int k;
        cout<<"\nK=?";
        cin>>k;        //ввести номер удаляемой строки
        if(k>=n)
            {
            cout<<"There is not such string\n";
            return;
            }

        /*выделяем память для размещения нового массива строк
    (без строки с номером k)*/
        char** temp=new char*[n-1];
        int j=0;

      //переписываем в новый массив все строки, кроме k-ой

        for(i=0;i<n;i++)
        if(i!=k)
            {
            //выделяем память под строку
                temp[j]=new char[strlen(mas[i])];

        //копируем строку
            strcpy(temp[j],mas[i]);
            j++;
            }
        n--;//уменьшаем размер массива

        for(i=0;i<n;i++)         //вывод нового массива
        {
            cout<<temp[i];
            cout<<"\n";
        }
    }
```

Эту же задачу можно решить не выделяя память под элементы нового массива, а меняя указатели: temp[j]=mas[i];

```
        /*выделяем память для размещения нового массива строк (без
строки с номером k)*/
        char** temp=new char*[n-1];
            int j=0;
```

```
  //переписываем в новый массив все строки, кроме k-ой

for(i=0;i<n;i++)
     if(i!=k)
          {
     temp[j]=mas[i];// меняем указатели

          j++;
          }
     n--; //уменьшаем размер массива

     delete []mas;//удаляем старый массив
     mas=temp;//ставим указатель на измененный массив
     for(i=0;i<n;i++)
     {
          cout<<mas[i]; //вывод массива
          cout<<"\n";
     }


}
```

## 17. Функции в C++

С увеличением объема программы становится невозможно удерживать в памяти все детали. Чтобы уменьшить сложность программы, ее разбивают на части. В C++ задача может быть разделена на более простые подзадачи с помощью функций. Разделение задачи на функции также позволяет избежать избыточности кода, т. к. функцию записывают один раз, а вызывают многократно. Программу, которая содержит функции, легче отлаживать.

Часто используемые функции можно помещать в библиотеки. Таким образом, создаются более простые в отладке и сопровождении программы.

### 17.1. Объявление и определение функций

Функция – это именованная последовательность описаний и операторов, выполняющая законченное действие, например, формирование массива, печать массива и т. д.

Функция, во-первых, является одним из производных типов C++, а, во-вторых, минимальным исполняемым модулем программы.

**Рис. 8. Функция как минимальный исполняемый модуль программы**

Любая функция должна быть объявлена и определена.

Объявление функции (прототип, заголовок) задает имя функции, тип возвращаемого значения и список передаваемых параметров.

Определение функции содержит, кроме объявления, тело функции, которое представляет собой последовательность описаний и операторов.

```
тип имя_функции([список_формальных_параметров])
{
тело_функции
}
```

Тело_функции – это блок или составной оператор. Внутри функции нельзя определить другую функцию.

В теле функции должен быть оператор, который возвращает полученное значение функции в точку вызова. Он может иметь две формы:

1) `return выражение;`
2) `return;`

Первая форма используется для возврата результата, поэтому выражение должно иметь тот же тип, что и тип функции в определении. Вторая форма используется, если функция не возвращает значения, т. е. имеет тип `void`. Программист может не использовать этот оператор в теле функции явно, компилятор добавит его автоматически в конец функции перед `}`.

Тип возвращаемого значения может быть любым, кроме массива и функции, но может быть указателем на массив или функцию.

Список формальных параметров – это те величины, которые требуется передать в функцию. Элементы списка разделяются запятыми. Для каждого параметра указывается тип и имя. В объявлении имена можно не указывать.

Для того, чтобы выполнялись операторы, записанные в теле функции, функцию необходимо вызвать. При вызове указываются: имя функции и фактические параметры. Фактические параметры заменяют формальные параметры при выполнении операторов тела функции. Фактические и формальные параметры должны совпадать по количеству и типу.

Объявление функции должно находиться в тексте раньше вызова функции, чтобы компилятор мог осуществить проверку правильности вызова. Если функция имеет тип не `void`, то ее вызов может быть операндом выражения.

---

```
/*Заданы координаты сторон треугольника, если такой треугольник существует, то найти его площадь. */
#include <iostream.h>
#include <math.h>

/*функция возвращает длину отрезка, заданного координатами x1,y1 и x2,y2*/
double line(double x1,double y1,double x2,double y2)
{
return sqrt(pow(x1-x2,2)+pow(y1-y2,2));
}

/*функция возвращает площадь треугольника, заданного длинами сторон a,b,c*/
double square(double a, double b, double c)
{
double s, p=(a+b+c)/2;
```

```
return s=sqrt(p*(p-a)*(p-b)*(p-c));//формула Герона
}

//возвращает true, если треугольник существует
bool triangle(double a, double b, double c)
{
if(a+b>c&&a+c>b&&c+b>a) return true;
else return false;
}

void main()
{
double x1=1,y1,x2,y2,x3,y3;
double point1_2,point1_3,point2_3;
do
{
     cout<<"\nEnter koordinats of triangle:";
     cin>>x1>>y1>>x2>>y2>>x3>>y3;
     point1_2=line(x1,y1,x2,y2);
     point1_3=line(x1,y1,x3,y3);
     point2_3=line(x2,y2,x3,y3);
     if(triangle(point1_2,point1_3,point2_3)==true)

cout<<"S="<<square(point1_2,point2_3,point1_3)<<"\n";

     else cout<<"\nTriagle doesnt exist";
}
while(!(x1==0&&y1==0&&x2==0&&y2==0&&x3==0&&y3==0));
}
```

### 17.2. Прототип функции

Для того, чтобы к функции можно было обратиться, в том же файле должно находиться определение или описание функции (прототип).

```
double line(double x1,double y1,double x2,double y2);
double square(double a, double b, double c);
bool triangle(double a, double b, double c);
double line(double ,double ,double ,double);
double square(double , double , double );
bool triangle(double , double , double );
```

При наличии прототипов вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией, а могут оформляться в виде отдельных модулей и храниться в откомпилированном виде в библиотеке объектных модулей. Это относится и к функциям из стандартных модулей. В этом случае определения библиотечных функций уже оттранслированные и оформленные в виде объектных модулей, находятся в библиотеке компилятора, а описания функций необходимо включать в программу дополнительно. Это делают с помощью препроцессорных команд `include<имя_файла>`.

Имя_файла – определяет заголовочный файл, содержащий прототипы группы стандартных для данного компилятора функций. Например, почти во всех программах мы

использовали команду #include <iostream.h> для описания объектов потокового ввода-вывода и соответствующие им операции.

При разработке программ, которые состоят из большого количества функций, размещенных в разных модулях, прототипы функций и описания внешних объектов (констант, переменных, массивов) помещают в отдельный файл, который включают в начало каждого из модулей программы с помощью директивы include "имя_файла".

### 17.3. Параметры функции

Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм параметров. Существует два способа передачи параметров в функцию: по адресу и по значению.

При передаче по значению выполняются следующие действия:

- вычисляются значения выражений, стоящие на месте фактических параметров;
- в стеке выделяется память под формальные параметры функции;
- каждому фактическому параметру присваивается значение формального параметра, при этом проверяются соответствия типов и при необходимости выполняются их преобразования.

```
/*функция возвращает площадь треугольника, заданного длинами сторон a,b,c */
double square(double a, double b, double c)
{
double s, p=(a+b+c)/2;
return s=sqrt(p*(p-a)*(p-b)*(p-c));//формула Герона
}
```

Вызовы этой функции могут быть следующими:

```
1) double s1=square(2.5,2,1);
```

Будет сформирован стек:

| | |
|---|---|
| A | 2.5 |
| B | 2 |
| C | 1 |
| S | |
| P | |

где P и S – локальные переменные.

```
2) double a=2.5,b=2,c=1;
double s2=square(a,b,c);

3) double x1=1,y1=1,x2=3,y2=2,x3=3,y3=1;

double s3=square(

//расстояние между 1 и 2:
sqrt(pow(x1-x2,2)+pow(y1-y2,2)),
```

```
//расстояние между 1 и 3:
sqrt(pow(x1-x3,2)+pow(y1-y3,2))

//расстояние между 2 и3:
sqrt(pow(x3-x2,2)+pow(y3-y2,2))
);
```

Для 2) и 3) будет сформирован такой же стек, как и для 1), т. к. выражения, стоящие на месте фактических параметров сначала вычисляются, а потом передаются в стек. Таким образом, в стек заносятся копии фактических параметров, и операторы функции работают с этими копиями. Доступа к самим фактическим параметрам у функции нет, следовательно, нет возможности их изменить.

При передаче по адресу в стек заносятся копии адресов параметров, следовательно, у функции появляется доступ к ячейке памяти, в которой находится фактический параметр и она может его изменить.

```
void Change(int a,int b) //передача по значению
{
     int r=a;
     a=b;
     b=r;
}
```

Для вызова функции

```
int x=1,y=5;
Change(x,y);
```

будет сформирован стек:

| | | |
|---|---|---|
| A | 1 | 5 |
| B | 5 | 1 |
| r | | 1 |

и при выполнении оператора

```
cout<<"x="<<x<<" y="<<y;
```

в функции `main` выведется:

```
x=1 y=5
```

Таким образом, эта функция не решает поставленную задачу.

```
void Change(int* a, int* b)   //передача по адресу
{
int r=*a;
*a=*b;
*b=r;
}
```

Для вызова функции

```
int x=1,y=5;
Change(&x,&y);
```

будет сформирован стек:

| | | |
|---|---|---|
| A | &x | 5 |
| B | &y | 1 |
| r | | 1 |

и при выполнении оператора

```
cout<<"x="<<x<<" y="<<y;
```

в функции `main` выведется:

```
x=5 y=1,
```

т. е. выполнится обмен переменных.

Для передачи данных в функцию также могут использоваться ссылки. При передаче по ссылке в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются.

```
void Change(int& a,int& b)
{
int r=a;
a=b;
b=r;
}
```

Для вызова функции

```
int x=1,y=5;
Change(x,y);
```

будет сформирован стек:

| | | |
|---|---|---|
| A | &x | 5 |
| B | &y | 1 |
| r | | 1 |

и при выполнении оператора

```
cout<<"x="<<x<<" y="<<y;
```

в функции `main` выведется:

```
x=5 y=1,
```

т. е. выполнится обмен переменных.

Использование ссылок вместо указателей улучшает читаемость программы, т. к. не надо применять операцию разыменовывания. Использование ссылок вместо передачи по значению также более эффективно, т. к. не требует копирования параметров. Если требуется запретить изменение параметра внутри функции, используется модификатор `const`. Рекомендуется ставить `const` перед всеми параметрами, изменение которых в функции не предусмотрено (по заголовку будет понятно, какие параметры в ней будут изменяться, а какие нет).

### 17.4. Локальные и глобальные переменные

Переменные, которые используются внутри данной функции, называются локальными. Память для них выделяется в стеке, поэтому после окончания работы функции они удаляются из памяти. Нельзя возвращать указатель на локальную переменную, т. к. память, выделенная такой переменной, будет освобождаться.

```
int* f()
{
int a;
…
return &a;// ОШИБКА!
}
```

Глобальные переменные – это переменные, описанные вне функций. Они видны во всех функциях, где нет локальных переменных с такими именами.

```
int a,b;        //глобальные переменные
void change()
{
     int r;     //локальная переменная
     r=a;
     a=b;
     b=r;
}

void main()
{
     cin>>a,b;
     change();
     cout<<”a=”<<a<<”b=”<<b;
}
```

Глобальные переменные также можно использовать для передачи данных между функциями, но этого не рекомендуется делать, т. к. это затрудняет отладку программы и препятствует помещению функций в библиотеки. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

## 17.5 Функции и массивы

### 13.5.1. Передача одномерных массивов как параметров функции

При использовании массива как параметра функции, в функцию передается указатель на его первый элемент, т. е. массив всегда передается по адресу. При этом теряется информация о количестве элементов в массиве, поэтому размерность массива следует передавать как отдельный параметр. Так как в функцию передается указатель на начало массива (передача по адресу), то массив может быть изменен за счет операторов тела функции.

```
//Удалить из массива все четные элементы
#include <iostream.h>
#include <stdlib.h>

//формирование массива
//массив передается в функцию по адресу
int form(int *a)
{
int n;
     cout<<"\nEnter n";
     cin>>n;

     for(int i=0;i<n;i++)
          a[i]=rand()%100;

//возвращается количество элементов в массиве
     return n;
}
```

```
//печать массива
void print(int *a,int n)
{
     for(int i=0;i<n;i++)
          cout<<a[i]<<" ";
     cout<<"\n";
}

/*удаление четных элементов из массива, массив передается по
адресу, количество элементов передается по адресу как ссылка */
void del(int *a, int& n)
{
     int j=0,i,b[100];
     for(i=0;i<n;i++)
          if(a[i]%2!=0)
          {
               b[j]=a[i];
          j++;
          }
          n=j;
          for(i=0;i<n;i++)
          a[i]=b[i];

}

void main()
{
     int a[100];
     int n;
     n=form(a);     //вызов функции формирования массива
     print(a,n);    //вызов функции печати массива
     del(a,n); //вызов функции удаления элементов массива
     print(a,n);    //вызов функции печати массива
}
```

При использовании динамических массивов, в функцию передается указатель на область динамической памяти, в которой размещаются элементы массива. Функция также может возвращать указатель на область динамической памяти, в которой размещаются элементы массива.

```
//Удалить из массива все элементы, совпадающие с первым
//элементом, используя динамическое выделение памяти.
#include <iostream.h>
#include <stdlib.h>

/*формирование массива, количество элементов массива
передается по адресу как ссылка, функция возвращает указатель на
динамический массив*/
int* form(int& n)
{
     cout<<"\nEnter n";
     cin>>n;
```

```
//выделить динамическую область памяти под массив
     int* a=new int[n];
     for(int i=0;i<n;i++)
          a[i]=rand()%100;         //заполнить массив

//вернуть указатель на динамическую область памяти
     return a;

}
void print(int*a,int n)//печать массива
//в функцию передается указатель на массив и количество
//элементов в массиве
{
     for(int i=0;i<n;i++)
          cout<<a[i]<<" ";
     cout<<"\n";
}

/*удаление из массива в функцию передается указатель на
массив и количество элементов в массиве по адресу, как ссылка */
int* del(int* a,int& n)
{
     int k,j,i;
     //считаем количество элементов в новом массиве
     for(k=0,i=0;i<n;i++)
          if(a[i]!=a[0])k++;
          int* b;
          //выделяем память под вспомогательный массив
          b=new int[k];
          //переписываем элементы из старого массива во
     //вспомогательный
          for(j=0,i=0;i<n;i++)
          if(a[i]!=a[0])
          {
               b[j]=a[i];
          j++;
          }
          n=k;           //меняем количество элементов
     //возвращаем новый массив как результат функции
     return b;
}

void main()
{
     int* a;   //указатель на динамический массив
     int n;    //количество элементов в массиве
     a=form(n);     //формируем массив
     print(a,n);    //печатаем массив
     a=del(a,n);    //удаляем элементы
     print(a,n);    //печатаем массив
}
```

### 13.5.2. Передача строк в качестве параметров функций

Строки при передаче в функции могут передаваться как одномерные массивы типа char или как указатели типа `char*`. В отличие от обычных массивов в функции не указывается длина строки, т. к. в конце строки есть признак конца строки `'\0'`.

```
//Найти количество гласных букв в строке
//Функция поиска заданного символа в строке
int find(char* s, char c)
{
     for (int I=0;I<strlen(s);I++)
     if(s[I]==c) return I;
     return -1
}

void main()
{
     char s[255];
     gets(s)
     char gl="aouiey";//строка, которая содержит гласные
     for(int I=0,k=0;I<strlen(s);I++)
     if(find(gl,s[I])>0)k++;
     printf("%d",k);
}
```

### 13.5.3. Передача многомерных массивов в функцию

При передаче многомерных массивов в функцию все размерности должны передаваться в качестве параметров. По определению многомерные массивы в С и С++ не существуют. Если мы описываем массив с несколькими индексами, например, массив `int mas[3][4]`, то это означает, что мы описали одномерный массив `mas`, элементами которого являются указатели на одномерные массивы `int[4]`.

Рассмотрим функцию, которая решает задачу транспонирования квадратной матрицы (поворот матрицы на 90 градусов). Если определить заголовок функции как:

void transp(int a[][],int n){….},

то получится, что мы хотим передать массив с неизвестными размерами. По определению, массив должен быть одномерным и его элементы должны иметь одинаковую длину. При передаче массива о размере элементов ничего не сказано, поэтому компилятор выдаст ошибку.

Самый простой вариант решения этой проблемы, определить функцию следующим образом:

```
//Транспонирование квадратной матрицы
#include<iostream.h>
/*глобальная переменная, задает количество строк и столбцов
в  матрице*/
const int N=10;
void transp(int a[][N],int n)
{
     int r;
     for(int I=0;I<n;I++)
```

```
        for(int j=0;j<n;j++)
        if(I<j)
        {
             r=a[I][j];
             a[I][j]=a[j][I];
             a[j][I]=r;
        }
}

void main()
{
        int mas[N][N];
        for(int I=0;I<N;I++)
        for(int j=0;j<Nlj++)
        cin>>mas[I][j];
        for(I=0;I<N;I++)
        {
             for(j=0;j<N;j++)
             cout<<mas[I][j]
             cout<<"\n";
        }
        transp(N,mas);
        for(I=0;I<N;I++)
        {
             for(j=0;j<N;j++)
             cout<<mas[I][j]
             cout<<"\n";
        }
}
```

Также при передаче многомерных массивов можно интерпретировать их как одномерные и внутри функции пересчитывать индексы (рис.9 )

```
//передача матрицы как одномерного массива
        #include <iosream.h>
        //передаем указатель на первый элемент матрицы
        void form_matr(int *matr, int n)
        {
        for (int i=0;i<n; int n)
        for( int j=0;j<n;j++)
        matr[i*n+j]=rand()%10;//пересчитываем индекс элемента
        }
        void print_matr(int *matr, int n)
        {
        for (int i=0;i<n; int n)
        {
             for( int j=0;j<n;j++)
             cout<<matr[i*n+j]<<" ";
        cout<<"\n";
        }
        }
        void main()
        {
        int matr[5][5];
```

```
        /*передаем адрес первого элемента матрицы как
фактический параметр*/
     form_matr (&matr[0][0],5);
     print_matr (&matr[0][0],5);
     }
```

---

Вызов такой функции можно записать еще и следующим образом:

```
     //выполняем приведение типов
     form_matr ((int*)matr,5);
     print_matr ((int*)matr,5);
```

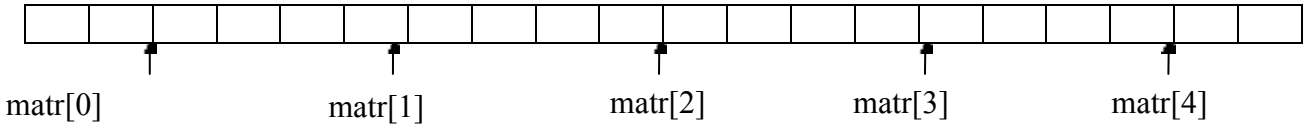



**Рис. 9 Размещение двумерного массива в памяти**

### 17.6 Функции с начальными значениями параметров (по-умолчанию)

В определении функции может содержаться начальное (умалчиваемое) значение параметра. Это значение используется, если при вызове функции соответствующий параметр опущен. Все параметры, описанные справа от такого параметра, также должны быть умалчиваемыми.

```
void print(char* name="Номер дома: ", int value=1)
{
     cout<<"\n"<<name<<value;
}
```

Вызовы:
1. `print();`
Вывод: `Номер дома: 1`
2. `print("Номер квартиры",15);`
Вывод: `Номер квартиры: 15`
3. `print(,15);` - ошибка, т. к. параметры можно опускать только с конца.
Поэтому функцию лучше переписать так:

```
void print(int value=1, char* name="Номер дома: ")
{
     cout<<"\n"<<name<<value;
}
```

Вызовы:
1. `print();`
Вывод: `Номер дома: 1`
2. `print(15);`
Вывод: `Номер дома: 15`
3. `print(6, "Размерность пространства");`
Вывод: `Размерность пространства: 6`

### 17.7. Подставляемые (inline) функции

Некоторые функции в C++ можно определить с использованием служебного слова `inline`. Такая функция называется подставляемой или встраиваемой.

```
     /* функция возвращает расстояние от точки с координатами
(x1,y1) (по умолчанию центр координат) до точки с координатами
(x2,y2)*/
     inline float Line(float x1,float y1,float x2=0,float y2=0)
     {
```

```
return sqrt(pow(x1-x2)+pow(y1-y2,2));
}
```

Обрабатывая каждый вызов подставляемой функции, компилятор пытается подставить в текст программы код операторов ее тела. Спецификатор inline определяет для функции так называемое внутреннее связывание, которое заключается в том, что компилятор вместо вызова функции подставляет команды ее кода. При этом может увеличиваться размер программы, но исключаются затраты на передачу управления к вызываемой функции и возврата из нее. Подставляемые функции используют, если тело функции состоит из нескольких операторов.

Подставляемыми не могут быть:

- рекурсивные функции;
- функции, у которых вызов размещается до ее определения;
- функции, которые вызываются более одного раза в выражении;
- функции, содержащие циклы, переключатели и операторы переходов;
- функции, которые имеют слишком большой размер, чтобы сделать подстановку.

### 17.8. Функции с переменным числом параметров

В C++ допустимы функции, у которых при компиляции не фиксируется число параметров, и, кроме того, может быть неизвестен тип этих параметров. Количество и тип параметров становится известным только в момент вызова, когда явно задан список фактических параметров. Каждая функция с переменным числом параметров должна иметь хотя бы один обязательный параметр. Определение функции с переменным числом параметров:

```
тип имя (явные параметры,. . . )
{
тело функции
}
```

После списка обязательных параметров ставится запятая, а затем многоточие, которое показывает, что дальнейший контроль соответствия количества и типов параметров при обработке вызова функции производить не нужно. При обращении к функции все параметры и обязательные, и необязательные будут размещаться в памяти друг за другом. Следовательно, определив адрес обязательного параметра как `p=&k`, где `p` - указатель, а `k` – обязательный параметр, можно получить адреса и всех остальных параметров: оператор `k++`; выполняет переход к следующему параметру списка. Еще одна сложность заключается в определении конца списка параметров, поэтому каждая функция с переменным числом параметров должна иметь механизм определения количества и типов параметров. Существует два подхода:

1) известно количество параметров, которое передается как обязательный параметр;
2) известен признак конца списка параметров.

```
//Найти среднее арифметическое последовательности
//чисел, если известно количество чисел
#include <iostream.h>
float sum(int k, . . .)
```

```
//явный параметр k задает количество чисел
{
     int *p=&k;//настроили указатель на параметр k
     int s=0;
     for(;k!=0;k--)
     s+=*(++p);
     return s/k;
}
void main()
{
     //среднее арифметическое 4+6
     cout<<"\n4+6="<<sum(2,4,6);
     //среднее арифметическое 1+2+3+4
     cout<<"\n1+2++3+4="<<sum(4,1,2,3,4);
}
```

Для доступа к списку параметров используется указатель `*p` типа `int`. Он устанавливается на начало списка параметров в памяти, а затем перемещается по адресам фактических параметров (`++p`).

```
     /*Найти среднее арифметическое последовательности чисел,
если известен признак конца списка параметров */
     #include<iostream.h>
     int sum(int k, …)
     {
          int *p = &k;   //настроили указатель на параметр k
          int s = *p;         //значение    первого    параметра
     присвоили s
          for(int i=1;p!=0;i++)    //пока нет конца списка
          s += *(++p);
          return s/(i-1);
     }

     void main()
     {
          //находит среднее арифметическое 4+6
          cout<<"\n4+6="<<sum(4,6,0);
          //находит среднее арифметическое 1+2+3+4
          cout<<"\n1+2++3+4="<<sum(1,2,3,4,0);
     }
```

### 17.9. Рекурсия

Рекурсией называется ситуация, когда какой-то алгоритм вызывает себя прямо (прямая рекурсия) или через другие алгоритмы (косвенная рекурсия) в качестве вспомогательного. Сам алгоритм называется рекурсивным.

Рекурсивное решение задачи состоит из двух этапов:

1. исходная задача сводится к новой задаче, похожей на исходную, но несколько проще;

2. подобная замена продолжается до тех пор, пока задача не станет тривиальной, т. е. очень простой.

**Задача 1**. Вычислить факториал (n!), используя рекурсию.

*Исходные данные: n*

*Результат: n!*

Рассмотрим эту задачу на примере вычисления факториала для n=5. Более простой задачей является вычисление факториала для n=4. Тогда вычисление факториала для n=5 можно записать следующим образом:

5!=4!*5.

Аналогично:

4!=3!*4;

3!=2!*3;

2!=1!*2 ;

1!=0!*1

Тривиальная (простая) задача:

0!=1.

Можно построить следующую *математическую модель*:

$$f(n) = \begin{cases} 1, n = 0 \\ f(n-1) * n, n \geq 1 \end{cases}$$

---

```
#include <iostream.h>
int fact(int n)
{
     if (n==0)return 1;//тривиальная задача
     return (n*fact(n-1));
}
}
void main()
{
     cout<<"n?";
     int k;
     cin>>k;//вводим число для вычисления факториала
     cout<<k<<"!="<<fact(k);//вычисление и вывод
результата
}
```

---

**Задача 2**. Вычислить степень, используя рекурсию.

*Исходные данные: x,n*

*Результат:* $x^n$

*Математическая модель:*

$$pow(x, y) = \begin{cases} 1, n = 0 \\ pow(x, n-1) * x, n \geq 1 \end{cases}$$

---

```
#include <iostream.h>
int pow( int x,int n)
{
     if(n==0)return 1;//тривиальная задача
     return(x*pow(x,n-1));
}
void main()
{
     cout<<"n?";
     int x;
```

```
          cin>>x; //вводим число
     int k;
          cin>>k; //вводим степень
     //вычисление и вывод результата
cout<<x<<"^"<<k<<"="<<pow(x,k);
     }
```

---

### 17.10 Перегрузка функций

Цель перегрузки состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с различными типами и различным числом фактических параметров. Для обеспечения перегрузки необходимо для каждой перегруженной функции определить возвращаемые значения и передаваемые параметры так, чтобы каждая перегруженная функция отличалась от другой функции с тем же именем. Компилятор определяет, какую функцию выбрать по типу фактических параметров.

---

```
#include <iostream.h>
#include <string.h>

//сравнение двух целых чисел
int max(int a, int b)
{
     if (a>b) return a;
     else return b;
}

//сравнение двух вещественных чисел
float max(float a, float b)
{
     if(a>b)return a;
     else return b;
}

//сравнение двух строк
char* max(char* a, char* b)
{
     if (strcmp(a,b)>0) return a;
     else return b;
}

void main()
{
     int a1,b1;
     float a2, b2;
     char s1[20];
     char s2[20];

     cout<<"\nfor int:\n";
     cout<<"a=?";cin>>a1;
     cout<<"b=?";cin>>b1;
     cout<<"\nMAX="<<max(a1,b1)<<"\n";
```

```
    cout<<"\nfor float:\n";
    cout<<"a=?";cin>>a2;
    cout<<"b=?";cin>>b2;
    cout<<"\nMAX="<<max(a2,b2)<<"\n";

    cout<<"\nfor char*:\n";
    cout<<"a=?";cin>>s1;
    cout<<"b=?";cin>>s2;
    cout<<"\nMAX="<<max(s1,s2)<<"\n";
}
```

Правила описания перегруженных функций:

- Перегруженные функции должны находиться в одной области видимости.
- Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В разных вариантах перегруженных функций может быть разное количество умалчиваемых параметров.
- Функции не могут быть перегружены, если описание их параметров отличается только модификатором `const` или наличием ссылки: функции `int& f1(int&, const int&){…}` и `int f1(int, int){…}` – не являются перегруженными, т. к. компилятор не сможет узнать какая из функций вызывается, потому что нет синтаксических отличий между вызовом функции, которая передает параметр по значению и функции, которая передает параметр по ссылке.

### 17.11. Шаблоны функций

Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные. Например, алгоритм сортировки можно использовать для массивов различных типов. При перегрузке функции для каждого используемого типа определяется своя функция. Шаблон функции определяется один раз, но определение параметризируется, т. е. тип данных передается как параметр шаблона.

```
template <class имя_типа [,class имя_типа]>
заголовок_функции
{
тело функции
}
```

Таким образом, шаблон семейства функций состоит из 2 частей – заголовка шаблона: `template<список параметров шаблона>` и обыкновенного определения функции, в котором вместо типа возвращаемого значения и/или типа параметров, записывается имя типа, определенное в заголовке шаблона.

```
    /*шаблон функции, которая находит абсолютное значение числа
любого типа */
    template<class type>     //type имя параметризируемого типа
    type abs(type x)
    {
        if(x<0)return -x;
        else return x;
```

```
}
```

Шаблон служит для автоматического формирования конкретных описаний функций по тем вызовам, которые компилятор обнаруживает в программе. Например, если в программе вызов функции осуществляется как `abs(-1.5)`, то компилятор сформирует определение функции `double abs(double x){…}`.

```
//шаблон функции, которая меняет местами две переменных
template <class T>  //T – имя параметризируемого типа
void change(T* x, T* y)
{
     T z=*x;
     *x=*y;
     *y=z;
}
```

Вызов этой функции может быть:

```
long k=10, l=5;
change(&k, &l);
```

Тогда, компилятор сформирует определение:

```
void change(long* x, long* y)
{
long z=*x;
*x=*y;
*y=z;
}
```

```
/*шаблон функции, которая находит номер максимального элемента в массиве*/
#include<iostream.h>
template<class Data>
Data& rmax(int n, Data a[])
{
     int im=0;
     for(int i=0;i<n;i++)
     if(a[im]<a[im]) im=i;
     return d[im];       //возвращает ссылку
}

void main()
{
     int n=5;
     int x[]={10,20,30,15};
//найти номер максимального элемента в массиве целых чисел
     cout<<"\nrmax(n,x)="<<rmax(n,x)<<"\n";
     rmax(n,x)=0;   //заменить максимальный элемент на 0
     for(int i=0;i<n;i++)          //вывод массива
     cout<<x[i]<<" ";
     cout<<"\n";
```

```
     //следующий массив
     float y[]={10.4,20.2,30.6,15.5};
/*найти номер максимального элемента в массиве вещественных
чисел*/
     cout<<"\nrmax(n,y)="<<rmax(n,y)<<"\n";
     rmax(4,y)=0;   //заменить максимальный элемент на 0
     for(in i=0;i<n;i++)      //вывод массива
     cout<<y[i]<<" ";
     cout<<"\n";
}
```

Результаты работы программы:

```
rmax(n,x)=30
10 20 0 15
rmax(n,y)=30.6
10.4 20.2 0 15.5
```

---

Основные свойства параметров шаблона функций

- Имена параметров должны быть уникальными во всем определении шаблона.
- Список параметров шаблона не может быть пустым.
- В списке параметров шаблона может быть несколько параметров, каждый из них начинается со слова class.

### 17.12. Указатель на функцию

Каждая функция характеризуется типом возвращаемого значения, именем и списком типов ее параметров. Если имя функции использовать без последующих скобок и параметров, то он будет выступать в качестве указателя на эту функцию, и его значением будет выступать адрес размещения функции в памяти. Это значение можно будет присвоить другому указателю. Тогда этот новый указатель можно будет использовать для вызова функции.

Указатель на функцию определяется следующим образом:

```
тип_функции(*имя_указателя)(спецификация параметров)
```

---

```
int f1(char c){…}   //определение функции
int(*ptrf1)(char);  //определение указателя на функцию f1
char* f2(int k,char c){…}          //определение функции
char* ptrf2(int,char);   //определение указателя на функцию
f2
```

---

В определении указателя количество и тип параметров должны совпадать с соответствующими типами в определении функции, на которую ставится указатель.

Вызов функции с помощью указателя имеет вид:

```
(*имя_указателя)(список фактических параметров);
```

---

```
#include <iostream.h>
void f1()
{
     cout<<"\nfunction f1";
}
```

```
void f2()
{
     cout<<”\nfunction f2”;
}

void main()
{
     void(*ptr)();  //указатель на функцию
     ptr=f2;        //указателю присваивается адрес функции
f2
     (*ptr)();      //вызов функции f2
     ptr=f1;        //указателю присваивается адрес функции
f1
     (*ptr)();      //вызов функции f1с помощью указателя
}
```

---

При определении указатель на функцию может быть сразу проинициализирован.

```
void (*ptr)()=f1;
```

Указатели н функции могут быть объединены в массивы. Например, `float(*ptrMas[4])(char)` – описание массива, который содержит 4 указателя на функции. Каждая функция имеет параметр типа `char` и возвращает значение типа `float`.

Обратиться к такой функции можно следующим образом:

```
float a=(*ptrMas[1])('f');    //обращение ко второй функции
```

---

```
#include <iostream.h>
#include <stdlib.h>

void f1()
{
cout<<"\nThe end of work";exit(0);
}

void f2()
{
cout<<"\nThe work #1";
}

void f3()
{
cout<<"\nThe  work #2";
}

void main()
{
     void(*fptr[])()={f1,f2,f3};//массив  и  указателей  на
функции
     int n;
     while(1)                  //бесконечный цикл
     {
          cout<<"\n Enter the number";
          cin>>n;
          fptr[n]();//вызов функции с номером n
```

```
    }
}
```

Указатели на функции удобно использовать в тех случаях, когда функцию надо передать в другую функцию как параметр.

```
#include <iostream.h>
#include <math.h>

typedef   float(*fptr)(float);//тип-указатель   на   функцию
уравнения

/*решение уравнения методом половинного деления, уравнение
передается с помощью указателя на функцию */
float root(fptr f, float a, float b, float e)
{
float x;
do
{
     x=(a+b)/2;                //находим середину отрезка
     if ((*f)(a)*f(x)<0)       //выбираем отрезок
     b=x;
     else a=x;
}
while((*f)(x)>e && fabs(a-b)>e);
return x;
}

//функция, для которой ищется корень
float testf(float x)
{
return x*x-1;
}

void main()
{
     /*в   функцию   root   передается   указатель   на   функцию,
координаты отрезка и точность */
float res=root(testf,0,2,0.0001);
cout<<"\nX="<<res;
}
```

### 17.13. Ссылки на функцию

Подобно указателю на функцию определяется и ссылка на функцию:

```
тип_функции(&имя_ссылки)(параметры)
инициализирующее_выражение;
```

```
int f(float a,int b){…}       //определение функции
int (&fref)(float,int)=f;     //определение ссылки
```

Использование имени функции без параметров и скобок будет восприниматься как адрес функции. Ссылка на функцию является синонимом имени функции. Изменить значение ссылки на функцию нельзя, поэтому более широко используются указатели на функции, а не ссылки.

```
#include <iostream.h>

void f(char c)
{
     cout<<"\n"<<c;
}

void main()
{
     void (*pf)(char);   //указатель на функцию
     void (&rf)(char);   //ссылка на функцию
     f('A');   //вызов по имени
     pf=f;     //указатель ставится на функцию
     (*pf)( 'B');   //вызов с помощью указателя
     rf('C');  //вызов по ссылке
}
```

## 18. Типы данных, определяемые пользователем

### 18.1. Переименование типов

Типу можно задавать имя с помощью ключевого слова `typedef`:

```
typedef тип имя_типа [размерность];

typedef unsigned int UNIT;
typedef char Msg[100];
```

Такое имя можно затем использовать также как и стандартное имя типа:

```
UNIT a,b,c;//переменные типа unsigned int
Msg str[10];// массив из 10 строк по 100 символов
```

### 18.2. Перечисления

Если надо определить несколько именованных констант таким образом, чтобы все они имели разные значения, можно воспользоваться перечисляемым типом:

```
enum [имя_типа] {список констант};
```

Константы должны быть целочисленными и могут инициализироваться обычным образом. Если инициализатор отсутствует, то первая константа обнуляется, а остальным присваиваются значение на единицу большее, чем предыдущее.

```
Enum Err{ErrRead, ErrWrite, ErrConvert);
Err error;
…
switch(error)
{
```

```
     case ErrRead: …
     case ErrWrite: …
     case ErrConvert: …
}
```

### 18.3. Структуры

Структура – это объединенное в единое целое множество поименованных элементов данных. Элементы структуры (поля) могут быть различного типа, они все должны иметь различные имена.

Форматы определения структурного типа следующие:

```
struct имя_типа  //способ  1
     {
     тип1 элемент1;
     тип2 элемент2;
     …
     };
```

```
struct Date          //определение структуры
{
int day;
int month;
int year;
};

Date birthday;       //переменная типа Date
```

```
struct               //способ 2
     {
     тип1 элемент1;
     тип2 элемент2;
     …
     } список идентификаторов;
```

```
struct
{
     int min;
     int sec;
     int msec;
} time_beg, time_end;
```

В первом случае описание структур определяет новый тип, имя которого можно использовать наряду со стандартными типами.

Во втором случае описание структуры служит определением переменных.

Структурный тип можно также задать с помощью ключевого слова `typedef`:

```
typedef struct       //способ 3
{
     floar re;
     float im;
} Complex;
```

```
Complex a[100];     //массив из 100 комплексных чисел.
```

### 18.3.1. Работа со структурами

**Инициализация структур.**
Для инициализации структур значения ее полей перечисляют в фигурных скобках.

```
//пример 1
struct Student
{
     char name[20];
     int kurs;
     float rating;
};

Student s={”Иванов”,1,3.5};

//пример 2
struct
{
     char name[20];
     char title[30];
     float rate;
} employee={”Петров”, ”программист”,10000};
```

**Присваивание структур.**
Для переменных одного и того же структурного типа определена операция присваивания. При этом происходит поэлементное копирование.

```
Student t=s;
```

**Доступ к элементам структур.**
Доступ к элементам структур обеспечивается с помощью уточненных имен:

`имя_структуры.имя_элемента`
`employee.name` – указатель на строку «Петров»;
`employee.rate` – переменная целого типа со значением 10000

```
#include <iostream.h>

void main()
{
struct Student
{
     char name[30];
     char group[10];
     float rating;
};

Student mas[35];    //массив структур
```

```
//ввод значений массива
for(int i=0;i<35;i++)
{
     cout<<"\nEnter name:";cin>>mas[i].name;
     cout<<"\nEnter group:";cin>>mas[i].group;
     cout<<"\nEnter rating:";cin>>mas[i].rating;
}
//вывод студентов, у которых рейтинг меньше 3
cout<<"Raitng <3:";
for(i=0; i<35; i++)
if(mas[i].name<3)
cout<<"\n"<<mas[i].name;
}
```

**Указатели на структуры.**

Указатели на структуры определяются также как и указатели на другие типы.

```
Student* ps;
```

Можно ввести указатель для типа `struct`, не имеющего имени (способ 2):

```
struct
{
char *name;
int age;
} *person;        //указатель на структуру
```

При определении указатель на структуру может быть сразу же проинициализирован.

```
Student* ps = &mas[0];
```

Указатель на структуру обеспечивает доступ к ее элементам 2 способами:

```
1.(*указатель).имя_элемента
2. указатель->имя_элемента
```

```
cin>>(*ps).name;
cin>>ps->title;
```

### 18.3.2. Битовые поля

Битовые поля – это особый вид полей структуры. При описании битового поля указывается его длина в битах (целая положительная константа).

```
struct
{
   int a:10;    //длина поля 10 бит
   int b:14;    //длина поля 14 бит
} xx, *pxx;
…
xx.a = 1;
pxx = &xx;
pxx->b = 8;
```

Битовые поля могут быть любого целого типа. Они используются для плотной упаковки данных. Например, с их помощью удобно реализовать флажки типа «да» / «нет».

Особенностью битовых полей является то, что нельзя получить их адрес. Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

### 18.3.3. Объединения

Объединение (union) – это частный случай структуры. Все поля объединения располагаются по одному и тому же адресу. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в такой переменной может храниться только одно значение. Объединения применяют для экономии памяти, если известно, что более одного поля не потребуется. Также объединение обеспечивает доступ к одному участку памяти с помощью переменных разного типа.

```
union
{
     char s[10];
     int x;
} u1;
```

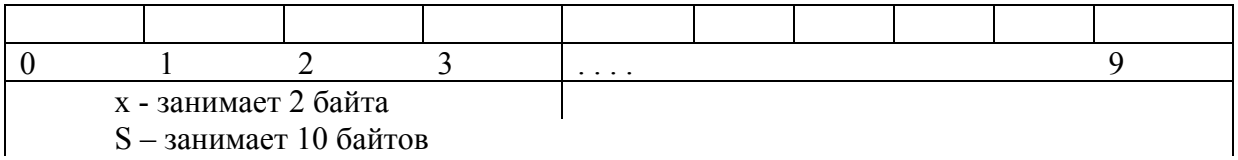



**Рис. 10. Расположение объединения в памяти**

И s, и x располагаются на одном участке памяти. Размер такого объединения будет равен 10 байтам.

```
//использование объединений
enum paytype{CARD,CHECK};     //тип оплаты
struct
{
     /*поле, которое определяет, с каким полем объединения
будет выполняться работа*/
paytype ptype;

union
{
          char card[25];
          long check;
};

} info;

switch (info.ptype)
{
case CARD:cout<<"\nОплата по карте:"<<info.card; break;
```

```
case CHECK:cout<<"\nОплата чеком:"<<info.check; break;
}
```

## 19. Динамические структуры данных

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

- линейные списки;
- стеки;
- очереди;
- бинарные деревья;

Они отличаются способом связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки динамической памяти.

Наиболее простой динамической структурой является линейный однонаправленный список, элементами которого служат объекты структурного типа (рис.7).

**Рис. 11. Линейный однонаправленный список**

Описание простейшего элемента такого списка выглядит следующим образом:

```
struct имя_типа
{
информационное поле;
адресное поле;
};
```

- информационное поле – это поле любого, ранее объявленного или стандартного, типа;
- адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.

Информационных полей может быть несколько.

```
//пример 1
struct point
{
  int key;     //информационное поле
  point* next; //адресное поле
```

```
};
```

```
//пример 2
struct person
{
   char* name;  //информационное поле
   int age;     //информационное поле
   person* next;     //адресное поле
};
```

Каждый элемент списка содержит ключ, который идентифицирует этот элемент. Ключ обычно бывает либо целым числом (пример 1), либо строкой (пример 2).

Над списками можно выполнять следующие операции:

- начальное формирование списка (создание первого элемента);
- добавление элемента в конец списка;
- добавление элемента в начало списка;
- поиск элемента с заданным ключом;
- удаление элемента с заданным ключом;
- удаление элемента с заданным номером;
- добавление элемента с заданным номером;
- и т. д.

Рассмотрим основные операции над списком из примера 1.

### 19.1. Создание элемента списка

Каждый элемент списка содержит как минимум два поля: информационное и адресное. Для создания одного элемента необходимо выделить под него память и заполнить поля элемента. Значение адресного поля можно вести с клавиатуры, в адресное поле записать нулевое значение.

```
#include <iostream.h>
//описание структуры
struct point
{
   int key;          //информационное поле
   point* next;      //адресное поле
};

//создание элемента
point* make_point()
{
     point*p=new(point);//выделить память под элемент списка
   cout<<"\nEnter the key";
     cin>>p->key;//заполнить информационное поле
     p->next=0;//сформировать адресное поле
return p;//вернуть указатель на созданный элемент
}
```

### 19.2. Создание списка из n элементов

Для того чтобы создать список из n элементов нужно создать первый элемент с помощью рассмотренной выше функции `make_point()`, а затем добавить в список оставшиеся (n-1) элементы. Добавление может осуществляться либо в начало списка, либо в конец списка. При добавлении элемента в начало списка элементы списка будут располагаться в порядке обратном тому, в котором элементы вводились, т. к. тот элемент, который был введен первым, в списке окажется последним.

```
     /*формирование списка из n элементов путем добавления
элементов в начало списка*/
     point* make_list(int n)
     {
     point* beg=make_point();//сформировать первый элемент
     point*r;//вспомогательная переменная для добавления
     for(int i=1;i<n;i++)
     {
          r=make_point();     //сформировать следующий элемент
               //добавление в начало списка
               r->next=beg;   //сформировать адресное поле
               beg=r;    //изменить адрес первого элемента списка
     }
          return beg;    //вернуть адрес начала списка
     }
```

Можно добавлять элементы не в начало, а вконец списка, тогда порядок элементов не изменится.

```
     /*формирование списка из n элементов путем добавления
элементов в конец списка*/
     point* make_list(int n)
     {
          If(n==0) return 0;//список пустой

     point* beg=make_point();//сформировать первый элемент
     if (n==1) return beg;//список из одного элемента

     point*r,//новый элемент списка
     //указатель для хранения адреса последнего элемента списка
     *q;
     q=beg;//поставили указатель q на первый элемент
     for(int i=1;i<n;i++)
     {
          r=make_point();     //сформировали следующий элемент
               //добавление в конец списка
               q->next=r;     //связали список с новым элементом
               q=r; //изменить адрес последнего элемента списка
     }
          return beg;    //вернуть адрес начала списка
     }
```

### 19. 3. Перебор элементов списка

Чтобы перебрать элементы списка нужно встать на первый элемент `p=beg` и переходить от одного элемента к следующему, используя адресное поле next: `p=p->next`, до тех пор, пока список не закончится, т. е. p не станет равно нулевому значению.

При обходе выполняется обработка ключевого поля списка. Рассмотрим перебор элементов списка на примере функции печати.

```
void print_list(point*beg)//печать списка
{

     point* p=beg;//встали на начало списка
//проверка наличия элементов в списке
     if (!p) {cout<<"\nThe list is empty!"; return;}
     while(p)//пока значение p не станет равно нулю
     {
          cout<<p->key<<" ";//вывод ключевого поля
          p=p->next;//переход к следующему элементу списка
     }
}
```

### 19. 4. Удаление элемента с заданным номером

Чтобы удалить элемент с заданным номером (например, с номером k) нужно поставить указатель на элемент с номером k-1 и изменить его адресное поле, присвоив ему значение адреса элемента с номером k+1. Затем элемент с номером k удаляется с помощью функции delete (рис.).

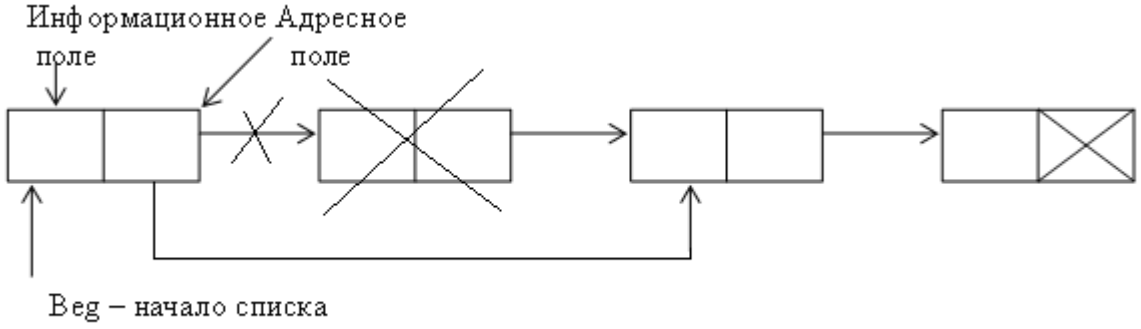



**Рис. 12. Удаление элемента из списка**

```
//Удаление из однонаправленного списка элемента с номером k
point*del_point(point*beg,int k)
{
     //поставить вспомогательную переменную на начало списка
     point*p=beg;
point *r;      //вспомогательная переменная для удаления
int i=0;       //счетчик элементов в списке

     if(k==0)  //удалить первый элемент
          {
               beg=p->next;
               delete p;      //удалить элемент из списка
               return beg;    //вернуть     адрес     первого
элемента
          }

     while(p)//пока нет конца списка
     {
```

```
            /*дошли до элемента с номером k-1, чтобы поменять
        его поле next*
        if(i==k-1)
            {
            /*поставить r на удаляемый элемент
            r=p->next;

            if(r)     //если p не последний элемент
                {
                    p->next=r->next;      //исключить  r  из
списка
                    delete r;    //удалить элемент из списка
                }

                /*если p -последний элемент, то в поле next
            присвоить 0*/
                else p->next=0;
            }

        p=p->next;     //переход к следующему элементу
    i++;           //увеличить счетчик элементов
        }

    return beg;//вернуть адрес первого элемента
    }
```

Удаление элемента с заданным ключом осуществляется аналогично, но вместо условия

`if(i==k-1) //проверка номера`

нужно использовать условие:

`if(p->next->key==KEY)//проверка ключа,`

где KEY заданный ключ. Ключ KEY передается как параметр функции удаления.

### 19. 5. Добавление элемента с заданным номером

Для добавления в список элемента с номером k нужно поставить указатель на элемент с номером k-1. Затем нужно создать новый элемент и поменять значения адресных полей таким образом, чтобы адресное поле нового элемента содержало адрес элемента с номером k, а адресное поле k-1 элемента – адрес нового элемента (см. рис.).

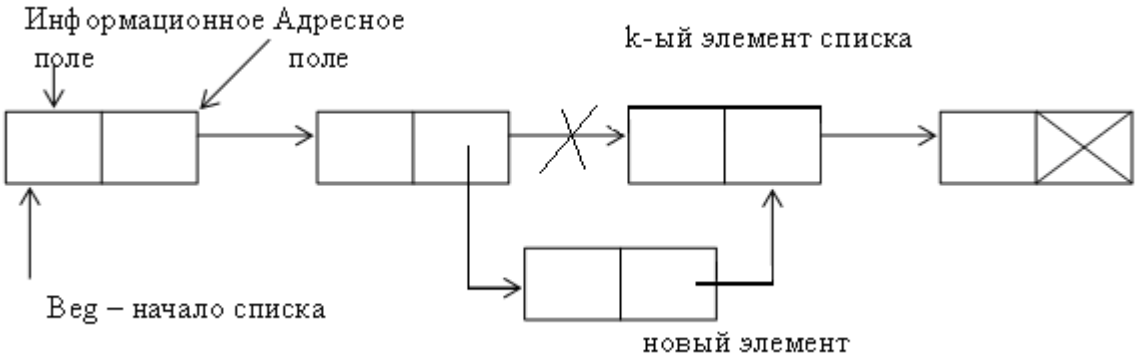



**Рис. 13. Добавление элемента номером k**

```
point* add_elem(point*beg, int NOM)
{
     point*p=make_point();//создание нового элемента
          if (NOM==0)//добавление первого элемента
     {
          p->next=beg;//связываем новый элемент со списком
          beg=p;//меняем адрес beg
          return beg;
     }
          point*r=beg;//указатель для перехода на нужный
номер

     /*проходим по списку до элемента с номером NOM-1 или до
конца списка, если такого элемента нет */
          for(int i=0; i<NOM-1&& r!=0;i++)
               r=r->next;
     //если элемента с указанным номером в списке нет
          if (!r)
          {
          cout<<"\nNot such NOM!"; //сообщение об ошибке
          return beg;
     }

          p->next=r->next;//связываем новый элемент со списком
      //связываем элемент с номером NOM-1 с новым элементом

     r->next=p;
          return beg;
     }
```

### 19.6. Двунаправленные списки

Двунаправленный список имеет два адресных поля, которые указывают на следующий элемент списка и на предыдущий. Поэтому двигаться по такому списку можно как слева направо, так и справа налево.

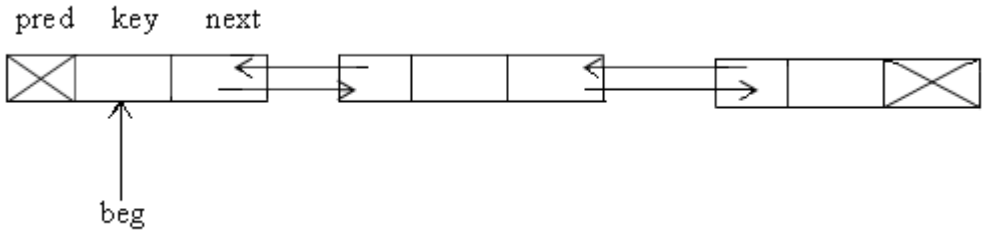



**Рис. 14. Двунаправленный список**

```
//пример описания двунаправленного списка
struct point   //описание структуры
{
     int key;  //ключевое поле
//адресные поля
     point* pred,   //адрес предыдущего элемента
```

```
     *next;    // адрес следующего элемента

};
```

Ниже рассматривается программа, которая создает двунаправленный список, выполняет удаление элемента с заданным номером, добавление элемента с заданным номером и печать полученных списков

```
#include <iostream.h>

struct point   //описание структуры
{
     int key;  //ключевое поле
     point* pred,*next;  //адресные поля
};

//формирование  списка
point*make_list()
{
     int n;    //количество элементов списка
     cout<<"n-?";
cin>>n;

     point* p, *r, *beg;
     p=new (point); //создать первый элемент
     /*запомнить адрес в переменную beg, в которой хранится
начало списка*/
     beg=p;
     cout<<"key-?";
cin>>p->key;   //заполнить ключевое поле
     p->pred=0;p->next=0;     //запомнить адресные поля
//добавить элементы в конец списка

     for(int i=1;i<n;i++)
{
          r=new(point);  //новый элемент
          cout<<"key-?";
     cin>>r->key;   //адресное поле
          p->next=r;     //связать начало списка с r
          r->pred=p;     //связать r с началом списка
          r->next=0;     //обнулить последнее адресное поле
          p=r; //передвинуть p на последний элемент списка
     }

     return beg;//вернуть первый элемент списка
}
//печать списка
void print_list(point *beg)
{
     if (beg==0)    //если список пустой
     {
     cout<<"The list is empty\n";
     return;
```

```
 }
     point*p=beg;
     while(p)  //пока не конец списка
     {
          cout<<p->key<<"\t";
          p=p->next;     //перейти на следующий элемент
     }
     cout<<"\n";
}

//удаление элемента с номером k
point* del_point(point*beg, int k)
{
     point *p=beg;
     if(k==0)//удалить первый элемент
     {
     //переставить начало списка на следующий элемент
          beg=beg->next;
     //если в списке только один элемент
          if(beg==0)return 0;
          //если в списке более одного элемента
          beg->pred=0;//обнулить адрес предыдущего элемента
          delete p;      //удалить первый
          return beg;    //вернуть начало списка
     }

     /*если удаляется элемент из середины списка, пройти по
     списку либо до элемента с предыдущим номером, либо до
     конца списка*/
     for(int i=0;i<k-1&&p!=0;i++,p=p->next);

     //если в списке нет элемента с номером k
     if(p==0||p->next==0)return beg;

     //если в списке есть элемент с номером k
     point*r=p->next;    //встать на удаляемый элемент
     p->next=r->next;    //изменить ссылку
     delete r;           //удалить r
     r=p->next;          //встать на следующий
     //если r существует, то связать элементы
     if(r!=0)r->pred=p;
     return beg;         //вернуть начало списка
}

//добавить элемент с номером k
point* add_point(point *beg,int k)
{
     point *p;

//создать новый элемент и заполнить ключевое поле
     p=new(point);
cout<<"key-?";
cin>>p->key;
```

```
     if(k==0)//если добавляется первый элемент
     {
          p->next=beg;   //добавить перед beg
          p->pred=0;     //обнулить адрес предыдущего
          //связать список с добавленным элементом
          beg->pred=p;
          beg=p;         //запомнить первый элемент в beg
          return beg;    //вернуть начало списка
     }

     //если добавляется элемент  середину или конец списка
     point*r=beg;   //встать на начало списка

/*пройти по списку либо до конца списка, либо до элемента с
номером k-1*/
     for(int i=0;i<k-1&&r->next!=0;i++,r=r->next);

     p->next=r->next;    //связать p с концом списка
//если элемент не последний, то связать конец списка с p
     if(r->next!=0)r->next->pred=p;
     p->pred=r;     //связать p и r
     r->next=p;
     return beg;    //вернуть начало списка
}

void main()
{
     point*beg;
int i,k;
do
{
          //печать меню
          cout<<"1.Make list\n";
          cout<<"2.Print list\n";
          cout<<"3.Add point\n";
          cout<<"4.Delete point\n";
          cout<<"5.Exit\n";
          cin>>i;   //ввод выбранного пункта меню
          switch(i)
          {
          case 1://формирование списка
          {
          beg=make_list();
          break;
     }
          case 2://печать списка
          {
          print_list(beg);
               break;
     }
          case 3://добавление элемента
          {
```

```
                cout<<"\nk-?";
            cin>>k;
                beg=add_point(beg,k);
                break;
            }
            case 4://удаление элемента
            {
                cout<<"\nk-?";
                cin>>k;
                beg=del_point(beg,k);
                break;
            }
        }
}
while(i!=5);   //выход из программы
}
```

### 19. 7. Очереди и стеки

Очередь и стек – это частные случаи однонаправленного списка.

В стеке добавление и удаление элементов осуществляются с одного конца, который называется вершиной стека. Поэтому для стека можно определить функции:

- top() – доступ к вершине стека (вывод значения последнего элемента
- pop() – удаление элемента из вершины (удаление последнего элемента);
- push() – добавление элемента в вершину (добавление в конец).

Такой принцип обслуживания называют LIFO (last in – first out, последний пришел, первый ушел).

```
point* pop(point*beg)
{
     point*p=beg;
     if (beg->next==0)//один элемент
      {
          delete beg;
          return 0;
     }
     //ищем предпоследний элемент
     while (p->next->next!=0)
     p=p->next;
     // ставим указатель r на последний элемент
     point*r=p->next;
     //обнуляем адресное поле предпоследнего элемента
     p->next=0;
     delete r; //удаляем последний элемент
     return beg;
}
point* push(point* beg)
{
     point*p=beg;// ставим указатель p на начало списка
     point*q=new(point);// создаем новый элемент
     cout<<"Key?";
     cin>>q->data;
     q->next=0;
```

```
        if (beg==0)//список пустой
        {
             beg=q;//q становится первым элементом
             return beg;
        }
        while (p->next!=0)//проходим до конца списка
        p=p->next;
        p->next=q; //связываем последний элемент с q
        return beg;
}
```

В очереди добавление осуществляется в один конец, а удаление из другого конца. Такой принцип обслуживания называют FIFO (first in – first out, первый пришел, первый ушел). Для очереди также можно определить функции:

- front() – доступ к первому элементу;
- back() – доступ к последнему элементу;
- pop() – удаление элемента из конца;
- push() – добавление элемента в начало.

### 19. 8. Бинарные деревья

Бинарное дерево - это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных не более двух ссылок на другие бинарные деревья (поддеревья). На каждое поддерево имеется ровно одна ссылка (рис. )

Описать такую структуру можно следующим образом:

```
struct point
{
int data;//информационное поле
point* left;//адрес левого поддерева
point* right;//адрес правого поддерева
};
```

Начальный узел называется конем дерева. Узел, не имеющий поддеревьев, называется листом дерева. Исходящие листы называются предками, входящие – потомками. Высота дерева определяется количеством уровней, на которых располагаются узлы.

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева - больше называется ***деревом поиска***. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения в каждом узле. Такой поиск является более эффективным, чем поиск по линейному списку, т. к. время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму количества узлов (аналогично бинарному поиску).

В ***идеально сбалансированном дереве*** количество узлов в справа и слева отличается не более, чем на единицу.

Линейный список можно представить как вырожденное бинарное дерево, в котором каждый узел имеет не более одной ссылки. Для списка среднее время поиска равно половине длины списка.

Деревья и списки являются рекурсивными структурами данных, т. к. каждое поддерево также является деревом. Дерево можно определить как рекурсивную структуру, в которой

каждый элемент является:

- либо пустой структурой;

- либо элементом, с которым связано конечное число поддеревьев.

Действия с рекурсивными структурами удобнее всего описываются с помощью рекурсивных алгоритмов.

## 19.9.Обход дерева

Для того, чтобы выполнить определенную операцию над всем узлами дерева, надо обойти все узлы. Такая задача называется обходом дерева. При обходе узлы должны посещаться в определенном порядке. Существуют три принципа упорядочивания, которые естественно вытекают из структуры дерева.

Рассмотрим следующее дерево.

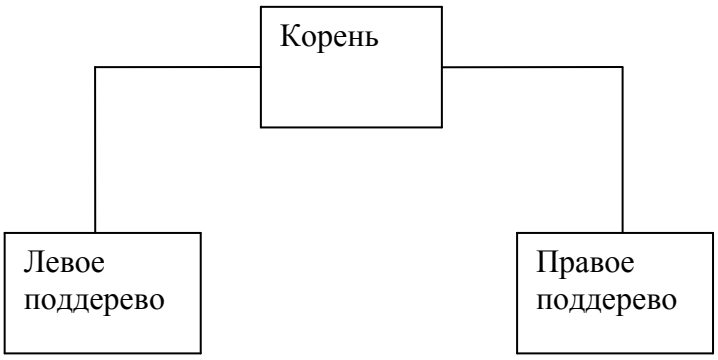



**Рис. 15. Структура бинарного дерева**

На этом дереве можно определить три метода упорядочения:

- Слева направо: Левое поддерево – Корень – Правое поддерево;
- Сверху вниз: Корень – Левое поддерево – Правое поддерево;
- Снизу вверх: Левое поддерево – Правое поддерево – Корень.

Эти три метода можно сформулировать в виде рекурсивных алгоритмов. Эти алгоритмы служат примером того, что действия с рекурсивными структурами лучше всего описываются рекурсивными алгоритмами.

```
//Обход слева направо:
void Run(point*p)
{
if(p)
{
Run(p->left); //переход к левому п/д
<обработка p->data>
Run(p->right);//переход к правому п/д
}
}
```

Если в качестве операции обработки узла поставить операцию вывода информационного поля узла, то мы получим функцию для печати дерева.

Для дерева поиска на рис. обход слева направо даст следующие результаты:

1 3 5 7 8 9 12

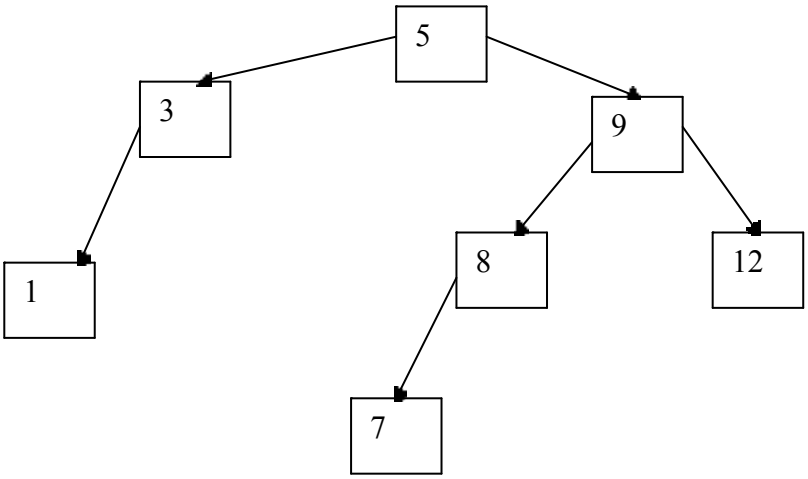



**Рис. 16. Пример дерева поиска**

```
//Обход сверху вниз:
void Run(point*p)
{
if(p)
{
 <обработка p->data>
Run(p->left); //переход к левому п/д
Run(p->right);//переход к правому п/д
}
}
```

Результаты обхода слева направо для дерева поиска на рис. :
5 3 1 9 8 7 12

```
//Обход снизу вверх
void Run(point*p)
{
if(p)
{
Run(p->left); //переход к левому п/д
Run(p->right);//переход к правому п/д
<обработка p->data>
}
}
```

Результаты обхода сверху вниз для дерева поиска на рис. :
1 3 7 8 12 9 5

### 19.10. Формирование дерева

Рассмотрим построение идеально сбалансированного дерева. При построении такого дерева надо распределять узлы таким образом, чтобы количество узлов в левом и правом поддеревьях отличалось не более чем на единицу.

```
# include <iostream.h>
struct point
{
```

```
        int data;
        point*left,*right;
    };

    point* Tree(int n,point* p)
    {
        point*r;
        int nl,nr;
        if(n==0){p=NULL;return p;}

        nl=n/2;//считаем количество узлов в левом поддереве
        nr=n-nl-1;   //считаем   количество   узлов   в   правом
поддереве

            r=new point;//создаем новый узел
            cout<<"?";
            cin>>r->data;//заполняем информационное поле
            r->left=Tree(nl,r->left);//формируем         левое
поддерево
            r->right=Tree(nr,r->right);//формируем      правое
поддерево
            p=r;//связываем
            return p;
    }
    //печать дерева
    void Print(point*p, int l)
    {
    //обход слева направо
        if(p)
        {
            Print(p->left,l+5);
            for(int i=0;i<l;i++)cout<<" ";
            cout<<p->data<<"\n";
            Print(p->right,l+5);
        }
    }

    void main()
    {
        point*root;
        root=Tree(10,root);
        Print(root,1);
    }
```

Функция печати, которая используется в данной программе, печатает дерево по уровням слева направо. Переменная `l` считает количество пробелов, которые надо отступить для печати следующего уровня. Результат работы программы представлен на рис.17.

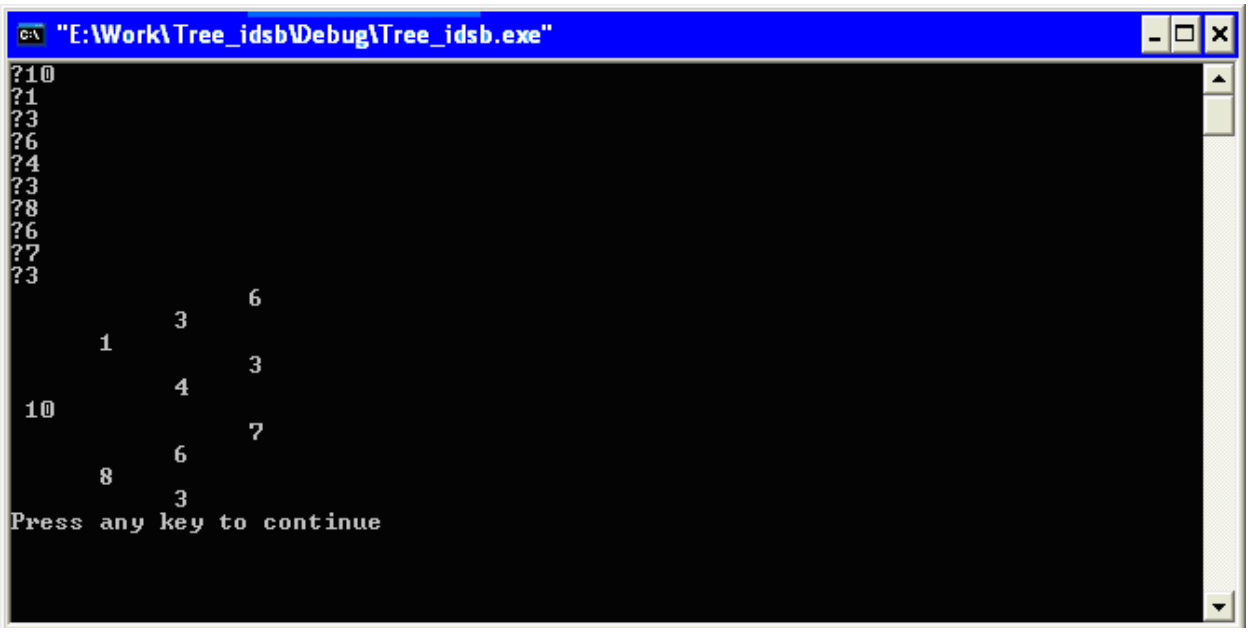



**Рис. 17. Результат работы программы формирования и печати дерева поиска**

При формировании дерева поиска нужно учитывать упорядоченность элементов в таком дереве, т. е. добавление элемента с ключом больше текущего осуществляется в правое поддерево, а меньше текущего – в левое. Ключи не дублируется, поэтому необходимо проверить существует ли элемент с заданным ключом в дереве и если существует, то завершить функцию добавления элемента.

```
point* first(int d)//формирование первого элемента дерева
{
     point* p=new point;
     p->key=d;
     p->left=0;
     p->right=0;
     return p;
}
//добавление элемента d в дерево поиска
Point* Add(point*root,int d)
{
     Point*p=root,*r;
// флаг для проверки существования элемента d
     bool ok=false;
     while(p&&!ok)
     {
          r=p;
          if(d==p->key)ok=true;
          else
          if(d<p->key)p=p->left;//пойти в левое поддерево
          else p=p->right;//пойти в правое поддерево
     }
     if(ok) return p;//найдено, не добавляем
     //создаем узел
     point* q=new point();//выделили память
     q->key=d;
     q->left=0;
     q->right=0;
```

```
          //добавляем в левое поддерево
          if(d<r->key)r->left=q;
          //добавляем в правое поддерево
          else r->right =q;
          return q;
     }
```

### 19.11. Удаление элемента из дерева

Рассмотрим удаление элемента из дерева поиска (рис. 16). Следует учитывать следующие случаи:

1. Узла с требуемым ключом в дереве нет.
2. Узел с требуемым ключом является листом, т. е. не имеет потомков.
3. Узел с требуемым ключом имеет одного потомка.
4. Узел с требуемым ключом имеет двух потомков.

В первом случае необходимо выполнить обход дерева и сравнить ключи элементов с заданным значением.

Во втором случае нужно заменить адрес удаляемого элемента нулевым. Для этого нужно выполнить обход дерева и заменить адрес удаляемого элемента нулевым. Например, при удалении элемента с ключом 7 мы меняем левое адресное поле элемента с ключом 8 на 0.

Третий случай похож на второй, т. к. мы должны заменить адресное поле удаляемого элемента адресом его потомка. Например, при удалении элемента с ключом 8 мы меняем левое адресное поле элемента с ключом 9 на адрес элемента с ключом 7.

Самым сложным является четвертый случай, т .к. возникает вопрос каким элементом мы должны заменить удаляемый элемент. Этот элемент должен иметь два свойства. Во-первых, он должен иметь не более одного потомка, а во-вторых, мы должны сохранить упорядоченность ключей, т. е. он должен иметь ключ либо не меньший, чем любой ключ левого поддерева удаляемого узла, либо не больший, чем любой ключ правого поддерева удаляемого узла. Таким свойством обладают два узла: самый правый узел левого поддерева удаляемого узла и самый левый узел правого поддерева удаляемого узла. Любым из них и можно заменить удаляемый элемент. Например, при удалении узла 9 его можно заменить узлом 12 (самый левый узел правого поддерева удаляемого узла).

Для удаления будем использовать рекурсивную функцию.

```
/*вспомогательная переменная для удаления узла, имеющего двух
потомков*/
point *q;

Point* Del(Point* r)
{
/*удаляет узел, имеющий двух потомков, заменяя его правым узлом
левого поддерева*/
     if (r->right!=0) r=Del(r->right);//идем в правое поддерево
     else //дошли до самого правого узла
     {
          //заменяем этим узлом удаляемый
          q->key=r->key;
          q=r;
          r=r->left;
     }
     return r;
}
```

```
Point* Delete(Point*p, int KEY)
{
     //Point*q;
     if(p) //ищем узел
          if (KEY<p->key)//искать в левом поддереве
               p->left=Delete(p->left,KEY);
          else if (KEY>p->key)//искать в правом поддереве

               p->right=Delete(p->right,KEY);
          else //узел найден
          {
               //удаление
               q=p;//запомнили адрес удаляемого узла
               // узел имеет не более одного потомка слева
               if(q->right==0)
                    p=q->left;//меняем на потомка
               else
               //узел имеет не более одного потомка справа
                    if(q->left==0)
                    p=q->right;//меняем на потомка
                    else //узел имеет двух потомков
                         p->left=Del(q->left);
                    delete q;
          }
          return p;
}
```

### 19. 12. Обработка деревьев с помощью рекурсивного обхода

**Задача 1**. Найти количество четных элементов в дереве.

Для решения этой задачи необходимо перебрать все элементы дерева и проверить информационные поля на четность. Для перебора будем использовать обход дерева слева направо. Результат запоминаем в специальной переменной, которую передаем по ссылке.

```
//количество четных элементов в дереве
void kol(Point *p, int &rez)
{

     if(p)
     {
     kol(p->left,rez);
     if(p->key%2==0)rez++;
     kol(p->right, rez);
     }
}
```

**Задача 2**. Найти количество отрицательных элементов в дереве.

Решается аналогично предыдущей.

```
//количество отрицательных элементов в дереве
void quantity_otr(int a,int &k)
{
     if(a<0)k++;
}
```

```
void kol(Point *p,void (*ptr)(int,int&), int &rez)//итератор
{

     if(p)
     {
     kol(p->left,quantity_otr,rez);
     ptr(p->key,rez);
     kol(p->right, quantity_otr,rez);
     }
}
```

## 20. Препроцессорные средства

### 20.1. Стадии и команды препроцессорной обработки

Назначение препроцессора – обработка исходного текста программы до ее компиляции (можно назвать первой фазой компиляции). Инструкции препроцессора называют директивами. Они начинаются с символа #.

На стадии обработки директив препроцессора возможно выполнение следующих действий:

1. Замена идентификаторов заранее подготовленными последовательностями символов (#define).
2. Включение в программу текста из указанных файлов (#include).
3. Исключение из программы отдельных частей (условная компиляция).
4. Макроподстановка, т. е. замена обозначения параметризируемым текстом.

### 20.2. Директива #define

Директива #define имеет несколько модификаций. Они предусматривают определение макросов или препроцессорных идентификаторов, каждому из которых ставится в соответствие некоторая последовательность символов. В последующем тексте программы препроцессорные идентификаторы заменяются на заранее оговоренные последовательности символов.

```
#define идентификатор строка_замещения
```

**Таблица 2. Пример работы директивы define**

| До обработки | После обработки |
|---|---|
| `#define begin {`<br>`#define end }`<br>`void main()`<br>`begin`<br>`операторы`<br>`end` | `void main()`<br>`{`<br>`операторы`<br>`}` |

С помощью #define удобно определять размеры массивов

**Таблица 3. Пример работы директивы define**

| До обработки | После обработки |
|---|---|
| `#define N 10`<br>`#define M 100` | `void main()`<br>`{` |

| | |
|---|---|
| void main()<br>{<br>int matr[N][N];<br>double mas[M]<br>….<br>} | int matr[10][10];<br>double mas[100]<br><br>} |

Те же возможности в C++ обеспечивают константы, определенные в тексте программы. Поэтому в C++ по сравнению с классическим C `#define` используется реже.

```
void main() //использование констант в C++
{
const int N=10;
const int M=100;
int matr[N][N];
double mas[M]

}
```

### 20.3. Включение текстов из файлов

Для включения текста из файла используется команда `#include`. Она имеет две формы записи:

```
#include <имя_файла>
#include "имя_файла"
```

В первом случае препроцессор разыскивает файл в стандартных системных каталогах. Во втором случае препроцессор сначала обращается к текущему каталогу и только потом к системному.

По принятому соглашению к тем файлам, которые надо помещать в заголовке программы приписывается расширение h (заголовочные файлы).

Заголовочные файлы оказываются эффективным средством при модульной разработке крупных программ, в которых используются внешние объекты (переменные, массивы, структуры), глобальные для нескольких частей программы. Описание таких объектов помещается в одном файле, который с помощью директивы include включается во все модули, где необходимы эти объекты.

```
//файл tree. h
//дерево и функции для его формирования и печати
#include <iostream.h>
struct Point
{
   int key;
   Point*left,*right;
};

Point* first(int d)//формирование первого элемента дерева
{
   Point* p=new Point;
   p->key=d;
   p->left=0;
   p->right=0;
```

```
        return p;
      }
      Point*  Add(Point*root,int  d)//добавление  элемента  d  в
дерево поиска
      {
        Point*p=root,*r;
        bool ok=false;
        while(p&&!ok)
        {
             r=p;
             if(d==p->key)ok=true;
             else
                  if(d<p->key)p=p->left;//пойти     в     левое
поддерево
                  else p=p->right;//пойти в правое поддерево
        }
        if(ok) return p;//найдено, не добавляем
        //создаем узел
        Point* New_point=new Point();//выделили память
        New_point->key=d;
        New_point->left=0;
        New_point->right=0;
        if(d<r->key)r->left=New_point;
        else r->right =New_point;
        return New_point;
      }
      void Show(Point*p,int level)
      {
        if(p)
        {
             Show(p->left,level+5);
             for(int i=0;i<level;i++)cout<<" ";
             cout<<p->key<<"\n";
             Show(p->right,level+5);
        }
      }
```

---

```
      //Файл с основной программой
      #include <iostream.h>
      #include "tree.h"

      void main()
      {

        int n,k;
        cout<<"n?";
        cin>>n;
        Point *root=first(10);//первый элемент
        for(int i=0;i<n;i++)
        {
             cout<<"?";
             cin>>k;
             Add(root,k);
        }
```

```
    Show(root,0);
}
```

Препроцессор добавляет текст файла `tree.h` в файл, в котором расположена основная программа и как единое целое передает на компиляцию.

### 20.4. Условная компиляция

Для условной компиляции используются следующие команды:

| | |
|---|---|
| `#if константное_выражение`<br>`#ifdef препроцессорный идентификатор`<br>`#ifndef препроцессорный идентификатор` | позволяют выполнить проверку условий |
| `#else`<br>`#endif` | позволяют определить диапазон действия проверяемого условия. |

Общая структура применения директив условной компиляции следующая:

```
# if условие текст1
#else текст2
#endif
```

Конструкция `#else текст2` необязательна. `текст1` включается в компилируемый текст только при истинности проверяемого условия. Если условие ложно – то при наличии директивы `else` на компиляцию передается `текст2`, если эта директива отсутствует, то при ложном условии `текст1` просто опускается.

Различие между форматами команд `#if` следующее:

1. Директива `#if константное_выражение` проверяет значение константного выражения. Если оно отлично от нуля, то считается, что проверяемое условие истинно.
2. В директиве `#ifdef` препроцессорный идентификатор проверяется, определен ли с помощью директивы `#define идентификатор`, помещенный после `#ifdef`. Если идентификатор определен, то `текст1` используется компилятором.
3. В директиве `#ifndef` препроцессорный идентификатор проверяется обратное условие: истинным считается неопределенность идентификатора. Если идентификатор не определен, то `текст1` используется компилятором.

Файлы, которые предназначены для препроцессорного включения, обычно снабжают защитой от повторного включения. Такое повторное включение может произойти, если несколько модулей, в каждом из которых подключается один и тот же файл, объединяются в общий текст программы. Такими средствами защиты снабжены все заголовочные файлы стандартной библиотеки (например, `iostream.h`).

Схема защиты от повторного включения:

```
//Файл с именем filename включается в другой файл
#ifndef _FILE_NAME
….//включаемый текст файла filename
#define _FILE_NAME 1
#include <…>//заголовочные файлы
<текст модуля>
#endif
```

_FILE_NAME – зарезервированный для файла filename препроцессорный идентификатор, который не должен встречаться в других текстах программы.

### 20.5. Макроподстановки средствами препроцессора

Макрос, по определению, есть средство замены одной последовательности символов на другую. Для выполнения замен должны быть заданы соответствующие макроопределения.

С помощью директивы `#define` идентификатор `строка_замещения` можно вводить макроопределения, в которых строка замещения фиксирована. Большими возможностями обладает макроопределение с параметрами:

`#define имя (список_параметров) строка_замещения`

Здесь `имя` – это имя макроса (идентификатор), `список_параметров` – список разделенных запятыми идентификаторов.

```
//пример 1
#define max(a,b) (a<b?b:a)//макроопределение
…..
max(x,y)// заменяется выражением (x<y?y:x)
max(z,4)// заменяется выражением (z<4?4:z)
```

```
//пример 2
#define ABS(x) (x<0?-(x):x)//макроопределение
….
ABS(E-Z)// заменяется выражением (E-Z<0?-(E-Z):E-Z)
```

Отличия макроса от функции:

1. Определение функции присутствует в программе в одном экземпляре, коды формируемые макросом вставляются в программу столько раз, сколько используется макрос. В этом отношении макросы похожи на inline функции, но подстановка для макроса выполняется всегда.
2. Функция определена для данных того типа, который указан в спецификации и возвращает значения только одного конкретного типа. Макрос пригоден для обработки параметров любого типа. Тип полученного значения зависит от типов параметров и самих выражений. В примерах 1 и 2 в качестве параметров могут использоваться любые целочисленные или вещественные типы.

Механизм перегрузки и шаблоны функций позволяют решать те же проблемы, что и макросы, поэтому в программах на C++ макросредства используются реже, чем в программах на базовом C.

```
#define max(a,b) (a<b?b:a)
#define t(e) e*3
#define PRN(c) cout<<"\n"<<#c<<" равно ";cout<<c;
#define E x*x
#include<iostream.h>
void main()
{
int x=2;
PRN(max(++x,++x);
PRN(t(x));
```

```
PRN(t(x+x));
PRN(t(x+x)/3);
PRN(E);
}
```

При подстановке параметров макроса в строку замещения запрещены подстановки внутрь кавычек, апострофов или ограничителей комментариев. При необходимости параметр макроса можно заключить в строке замещения в кавычки, для этого используется операция #, которая записывается перед параметром.

**Таблица 4. Примеры подстановок параметров макроса**

| | |
|---|---|
| PRN(max(++x,++x)<br>; | 1-ая подстановка<br>cout<<"\n"<<"max(++x,++x)"<<" равно ";cout<< max(++x,++x);<br>2-ая подстановка<br>cout<<"\n"<<"max(++x,++x)"<<" равно ";cout<<(++x<++x?++x:++x);//т. е. x увеличится на 3 (=5) |
| PRN(t(x)); | 1-ая подстановка<br>cout<<"\n"<<"t(x)"<<" равно ";cout<< t(x)<br>2-ая подстановка<br>cout<<"\n"<<"t(x)"<<" равно ";cout<<x*3;//x=5, результат 15 |
| PRN(t(x+x)); | 1-ая подстановка<br>cout<<"\n"<<"t(x+x)"<<" равно ";cout<< t(x+x)<br>2-ая подстановка<br>cout<<"\n"<<"t(x+x)"<<" равно ";cout<<(x+x)*3;//x=5, результат 30 |
| PRN(t(x+x)/3); | 1-ая подстановка<br>cout<<"\n"<<"t(x+x)"<<" равно ";cout<< t(x+x)/3<br>2-ая подстановка<br>cout<<"\n"<<"t(x+x)"<<" равно ";cout<<(x+x)*3/3;//результат (5+5)*3/3=10 |
| PRN(E); | 1-ая подстановка<br>cout<<"\n"<<"E"<<" равно ";cout<<E;<br>2-ая подстановка<br>cout<<"\n"<<"E"<<" равно ";cout<<x*x;// результат25 |

Результаты:
max(++x,++x) равно 5
t(x) равно 15
t(x+x) равно 20
t(x+x)/3 равно 10
E равно 25

## 21. Технология создания программ

### 21.1. Проектирование программы

Структурный подход к программированию охватывает все стадии разработки проекта: спецификацию, проектирование, собственно программирование и тестирование. Задачи, которые при этом ставятся – это уменьшение числа возможных ошибок за счет применения только допустимых структур, как можно более раннее обнаружение ошибок и

упрощение процесса их исправления. Ключевыми идеями структурного подхода являются структурное программирование и нисходящее тестирование.

Этапы создания программы:

1**. Постановка задачи.** Изначально задача ставится в терминах предметной области, и необходимо перевести ее в термины более близкие к программированию. Это достаточно трудоемкий процесс, т. к. программист обычно плохо разбирается в предметной области, а заказчик не может правильно сформулировать свои требования. Постановка задачи завершается созданием технического задания и внешней спецификации программы. Спецификация программы должна включать в себя:

- описание исходных данных и результатов;
- описание задачи, реализуемой программой;
- способ обращения к программе;
- описание возможных аварийных ситуаций и ошибок пользователя.

2. Разработка внутренних структур данных. Большинство алгоритмов зависит от способа организации данных (статические или динамические, массивы, списки или деревья и т. п.).

3. Проектирование программы, которое заключается в определении общей структуры и способов взаимодействия модулей. На данном этапе может применяться технология нисходящего проектирования, при котором задачу разбивают на подзадачи меньшей сложности. На этом этапе очень важной является спецификация интерфейсов, т. е. способов взаимодействия подзадач. Для каждой подзадачи составляется внешняя спецификация, аналогичная п. 1. На этом же этапе решаются вопросы разбиения программы на модули, взаимодействие этих модулей должно быть минимальным. На более низкий уровень проектирования переходят только после окончания проектирования верхнего уровня. Алгоритмы для модулей записывают в обобщенной форме (словесная запись, блок-схемы). Проектировать программу надо таким образом, чтобы в нее достаточно легко можно было внести изменения. Процесс проектирования является итерационным, т. к. невозможно учесть все детали с первого раза.

4. Структурное программирование. Процесс программирования также должен быть организован сверху вниз: сначала кодируются модули самого верхнего уровня и составляются тестовые примеры для их отладки, на месте модулей, которые еще не написаны, ставятся, так называемые «заглушки». Заглушки выдают сообщение о том, что им передано управление, а затем снова возвращают управление в вызывающую программу. При программировании следует отделять интерфейс модуля от его реализации и ограничивать доступ к ненужной информации. Этапы проектирования и программирования совмещены во времени: сначала проектируется и кодируется верхний уровень, затем – следующий и т. д. Такая стратегия применяется, т. к. в процессе кодирования может возникнуть необходимость внести изменения, которые потом отразятся на модулях нижнего уровня.

5. Нисходящее тестирование. Проектирование и программирование сопровождаются тестированием. Цель процесса тестирования – определение наличия ошибки, нахождение места ошибки, ее причины и соответствующие изменения программы – исправление. Тест – это набор исходных данных, для которых заранее известен результат. Тест, выявивший ошибку, считается успешным. Процесс исправления ошибок в программе называется отладкой, исправляются ошибки обнаруженные при тестировании. Отладка программы заканчивается, когда достаточное количество тестов выполнилось неуспешно, т. е. программа на них выдала правильные результаты.

Цель тестирования показать, что программа работает правильно и удовлетворяет всем проектным спецификациям. Чем больше ошибок обнаружено на начальных стадиях тестирования, тем меньше их остается в программе. Чем меньше ошибок осталось в программе, тем сложнее искать каждую из этих ошибок.

Идея нисходящего тестирования заключается в том, что к тестированию программы надо приступать еще до того, как завершено ее проектирование. Только после того как проверен и отлажен один уровень программы, можно приступать к программированию и тестированию следующего уровня.

Для исчерпывающего тестирования рекомендуется проверить:

- каждую ветвь алгоритма;
- граничные условия;
- ошибочные исходные данные.

## 21.2. Кодирование и документирование программы

• Главная цель, к которой нужно стремиться при написании программы – это получение легко читаемой программы простой структуры. Для этого написание программы рекомендуется начинать с записи на естественном языке или в виде блок-схем ее укрупненного алгоритма (что и как должна делать программа). Алгоритм надо записать как последовательность законченных действий. Каждое законченное действие оформляется в виде функции. Каждая функция должна решать одну задачу. Тело функции не должно быть длинным (30-50 строк), т. к. сложно разбираться в длинной программе, которая содержит длинные функции. Если некоторые действия повторяются более одного раза, их тоже рекомендуется оформить как функцию. Короткие функции лучше оформить как подставляемые функции (inline).

• Имена переменных выбираются таким образом, чтобы можно было понять, что делает эта переменная, например, сумму обозначают `Sum`, `Summa` или `S`, массив – `Array` или `Arr` и т. п. Для счетчиков коротких циклов лучше использовать однобуквенные имена, например, `i` или `j`. Чем больше область видимости переменной, тем более длинное у нее имя. Не рекомендуется использовать имена, начинающиеся с символа подчеркивания, имена типов, идентификаторы, совпадающие с именами стандартной библиотеки C++.

• Переменные рекомендуется объявлять как можно ближе к месту их использования. Но можно и все объявления локальных переменных функции расположить в начале функции, чтобы их легко можно было найти. Переменные лучше инициализировать при их объявлении.

• Глобальные переменные лучше не использовать. Если использование глобальной переменной необходимо, то лучше сделать ее статической, тогда область ее видимости будет ограничена одним файлом.

• Информация, которая необходима для работы функции, должна передаваться ей в качестве параметров, а не глобальных переменных.

• Входные параметры, которые не должны изменяться в функции лучше передавать как ссылки со спецификатором `const`, а не по значению. Этот способ более эффективен, особенно при передаче сложных объектов.

• Выходные параметры лучше передавать по указателю, а не по ссылке, тогда из семантики вызова функции будет понятно, что этот параметр будет изменяться внутри функции.

• Нельзя возвращать из функции ссылку на локальную переменную, т. к. эта переменная будет автоматически уничтожаться при выходе из функции. Также не рекомендуется возвращать ссылку на динамическую локальную переменную, созданную с помощью операции `new` или функции `malloc()`.

• Если в программе используются числа, например, размеры массивов, то для них лучше использовать символические имена – константы или перечисления. Это делает программу более понятной и, кроме того, в такую программу легче будет вносить изменения, т. к. достаточно будет изменить константу в одном месте.

- Следует избегать лишних проверок условий, т. е. если для вычисления отношений надо вызывать одну и ту же функцию несколько раз, то вычисление функции лучше оформить в виде оператора присваивания, а в условном операторе использовать вычисленное значение. Не следует в условном операторе выполнять проверку на неравенство нулю, т. к. это не имеет смысла. Например, условие `if(ok!=0)….` лучше записать как `if(ok)….` Более короткую ветвь оператора `if` рекомендуют помещать сверху, иначе управляющая структура может не поместиться на экране.
- При использовании циклов надо объединять инициализацию, проверку условия выхода и приращения в одном месте. Если есть хотя бы два из инициализирующего, условного или корректирующего выражения, то лучше использовать цикл `for`. При использовании итеративных циклов необходимо предусматривать выход при достижении максимального количества итераций.
- Необходимо проверять коды возврата ошибок и предусматривать печать соответствующих сообщений. Сообщение об ошибке должно быть информативным и подсказывать пользователю как ее исправить. Например, при вводе неверного значения должен указываться допустимый диапазон.
- Операции выделения и освобождения динамической памяти следует помещать в одну функцию. Иначе может возникнуть ситуация, когда память выделили, а освободить забыли.
- Программа должна иметь комментарии. Комментарии должны представлять собой правильные предложения, но они не должны подтверждать очевидное (за исключением тех случаев, когда программа используется как пример для обучения). Комментарий, который занимает несколько строк, размещают до фрагмента программы. Для разделения функций и других логически законченных фрагментов можно использовать пустые строки или комментарии вида

//----------

- Вложенные блоки должны иметь отступы в 3-4 символа, причем блоки одного уровня вложенность должны быть выровнены по вертикали. Закрывающая фигурная скобка должна находиться под открывающей.