Федотов С.Н.

Программирование в физических исследованиях
(Часть 1. Основы C++)

УДК 539.107(076.5)

ББК 22.38я7

Ф 34

Федотов С.Н. Программирование в физических исследованиях (Часть 1. Основы С++): Учебное пособие. М.:НИЯУ МИФИ,2016.- 70 с.

Настоящее пособие посвящено основам объектно - ориентированного программирования на языке С++, как наиболее распространенном языке программирования при разработке широкого спектра различных приложений, в том числе и при проведении научных исследований. Кроме того, язык С++ используется в большинстве широко известных программ для обработки результатов экспериментов и моделирования физических процессов (Root, Geant 4) и служит базой для освоения этих программ.

Пособие может быть использовано для самостоятельного изучения основ программирования на языке С++. Кроме теоретического материала данное пособие содержит тексты учебных программ и задания для практических занятий. Также пособие может быть использовано в качестве основы для компьютерного лабораторного практикума.

Пособие предназначено для студентов, изучающих основы программирования при освоении курсов “Современные языки и методы программирования” , “Программирование на языке С++”, “Компьютерный практикум”, “Компьютерные технологии”, “Основы программирования в Windows”.

Рецензент

ISBN 978-5-7262-2244-8

Оглавление

Раздел 1. Элементы программирования на языке С 5
Простые типы данных. 5
Перечисляемые типы. 6
Массивы и структуры. 6
Стандартный ввод/вывод в языке С. 8
Указатели. 11
Операции инкремента ++ (декремента - -). 15
Оператор безусловного цикла for. 16
Циклы while и do/while 18
Условные операторы if и if/else. 19
Условный оператор switch/case. 20
Функции, определяемые пользователем. 23
Три способа передачи параметров в функцию 24
Массивы и указатели. Указатель на функцию. 27
Работа с файлами в языке С. 30
Раздел 2. Основы программирования на языке С++. 35
Особенности языка С++, не связанные с объектной ориентированностью. 35
Определение класса. Состав класса, создание экземпляров класса 36
Конструктор класса, его назначение. 37
Конструктор класса с аргументами, задаваемыми по умолчанию.
Конструктор копирования. 39
Наследование. 43
Множественное наследование. Виртуальные базовые классы. 44
Передача аргументов в базовый класс. 46
Конструкторы с инициализацией по умолчанию в иерархии классов. 47
Виртуальные функции 49
Темплейты. 53
Дружественные функции. Указатель this. 56
Перегрузка операций 58
Работа с файлами в языке С++. 60
Управление исключениями. 63
Спецификация исключений 66

Конструкторы и исключения. 67
Литература ……..70

Раздел 1

Элементы программирования на языке С

Простые типы данных

В языке С действует жесткое правило: все переменные перед тем, как их использовать, должны быть **объявлены.** Объявление переменной состоит из ее **типа**, произвольного **имени** и завершающего символа – точки с запятой (**;**). При объявлении нескольких переменных одного типа их можно перечислить в одном операторе объявления, разделяя запятыми. Например, объявление

int j;

определяет целочисленную (типа **int**) переменную с именем **j**, занимающую 2 байта. Объявление

char sign;

определяет переменную **sign**, занимающую 1 байт и предназначенную для хранения кода того или иного символа (буквы, цифры, знака препинания и др.). Объявление

float x, y, z;

определяет три переменные **x**, **y** и **z** с плавающей точкой, занимающие по 4 байта.

Объявление

double a, b;

определяет две переменные **a** и **b** с плавающей точкой (двойная точность), занимающие по 8 байт.

Переменную можно сначала объявить, как показано выше, а затем по мере необходимости присваивать ей те или иные значения. Допустимо также присвоение переменной инициализирующего значения одновременно с ее объявлением:

double p = 2.71, s;

Следует иметь в виду, что компилятор языка С различает строчные и прописные буквы. Таким образом, обозначения **b** и **B** относятся к разным переменным (которые вполне могут сосуществовать в программе).

Перечисляемые типы

Перечисляемый тип представляет собой набор целых чисел, определенный с помощью ключевого слова **enum**. Каждому числу приписывается имя. Объявление перечисляемого типа имеет форму:

enum имя_типа **{** имя_константы (= целое значение), …, …**};**

Указывать значение не обязательно. По умолчанию первой константе в списке приписывается значение 0. Последующие константы по умолчанию принимают значение, на единицу большее предыдущей константы. Например:

enum Seasons { Winter, Spring, Summer, Fall };

или

enum Drives { HardDrive = 2, DVD = 4, Print = 8 };

После объявления перечисляемого типа переменные перечисляемого типа (в нашем примере **s1, s2, Dr1, Dr2**) объявляются стандартным образом:

Seasons s1, s2;

Drives Dr1, Dr2;

Теперь объявленные переменные могут принимать соответствующие "говорящие" значения:

S2 = Summer; Dr1 = DVD;

Следует отметить, что "говорящими" эти значения будут только для программиста, а для компьютера они являются целыми числами, поэтому при выводе значений переменных перечисляемого типа необходимо использовать форматы, соответствующие целым числам.

Массивы и структуры

Массив представляет собой последовательность некоторого количества данных *одного типа*. Соответственно в программе могут быть массивы целых чисел, массивы чисел с плавающей запятой, символьные массивы и т. д. Весь массив в целом характеризуется именем, а обращение к элементам массива выполняется по их ин-

дексам. Например, в программе объявляется массив по имени **Arr** из 10 целых чисел:

int Arr[10];

Индексация массивов всегда начинается с нулевого индекса. Таким образом, первый элемент массива получает обозначение **Arr[0]**, следующий – **Arr[1]** и т. д. до последнего элемента **Arr[9].**
Аналогично объявляются массивы данных любых других типов:
char letter[20];//Массив из 20 символов
float width[16];//Массив из 16 чисел с плавающей запятой
Если в массиве не слишком много элементов и их значения известны заранее, массив можно инициализировать вместе с его объявлением, заключив перечень значений в фигурные скобки:

int num[]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

В данном случае количество элементов массива указывать необязательно. Компилятор выделит объем памяти по числу конкретных элементов.

Двумерные массивы определяются с указанием числа строк и столбцов. Например, объявление двумерного массива целых чисел, содержащего 5 строк и 6 столбцов, будет выглядеть следующим образом:

int Matrix[5][6];

В случае, когда элементы двумерного массива заранее известны, массив может быть объявлен следующим образом:

int Vec[ ][3] = {{1, 3, 2}, {3, 6, 1}, {5, 4, 9}};

Структура объявляется с помощью ключевого слова **struct**, за которым обычно следует имя структуры, выступающее в качестве нового имени конкретного структурного типа, а затем в фигурных скобках описание ее членов. Завершается описание нового структурного типа символом точки с запятой (**;**):

```
struct Line{       //Созданный пользователем новый тип
  char color;     //Символ для обозначения цвета линии
  int thickness;  //Номер толщины линии
  };              //Конец описания структуры
```

Существуют два способа объявления в программе структурных переменных: по имени и по адресу (с помощью указателя). Объявление в программе структурных переменных по адресу будет рассмотрено ниже.

Объявление в программе структурных переменных: по имени производится также как и для простых переменных:

int i; //Объявляется целочисленная переменная i (типа int)
Line S1, S2; //Объявляются две структурные переменные
//S1 и S2 типа Line

При таком объявлении компилятор выделяет для каждой из этих переменных столько места в памяти, сколько должны занимать все элементы структуры. Для обращения к членам структурной переменной используется оператор точка:

S1.color = 'R'; //Инициализация первого элемента
структурной переменной
S1.thickness = 2; //Инициализация второго элемента
S2.color = 'G'; //То же для второй
S2.thickness = 1; //структурной переменной

Стандартный ввод/вывод в языке С

Во многих системах стандартный ввод/вывод может быть перенаправлен, однако мы будем для простоты предполагать, что стандартный ввод – это ввод с клавиатуры, а стандартный вывод – это вывод на экран. Простейшая функция ввода **getche( ),** которая читает символы с клавиатуры. Функция ожидает, пока не будет нажата клавиша, и возвращает код, соответствующий символу. Одновременно происходит отображение введенного символа на экран. Другой функцией для ввода символа с клавиатуры является функция **getch( ),** которая действует также как и функция **getche( ),** но не выводит символ на экран. Эта функция часто используется для остановки действия программы до нажатия какой-либо клавиши именно потому, что она не выдает эхо на экран.

Прототипы функций **getche( )** и **getch( )** находятся в библиотечном файле CONIO.H. Аналогом функций **getche ()** является функция **getchar( ).** Функция **getchar( )** производит ввод, но требует нажатия клавиши Enter.

Простейшая функция вывода – **putchar( ).** Она выводит символ, который является ее аргументом на экран в текущую позицию курсора.

Функция **gets( )** реализует ввод с клавиатуры строки символов. Окончание ввода осуществляется нажатием клавиши Enter. При этом в конец строки записывается символ '\0'. Функция **puts( )** выводит на экран строку, которая является аргументом функции.

Прототипы функций **getchar( ), putchar( ), gets( )** и **puts( )** находятся в библиотечном файле STDIO.H.

Функции **printf( )** и **scanf( )** осуществляют форматированный ввод и вывод на консоль. Форматированный ввод и вывод означает, что функции могут читать и выводить данные в разном формате, которым вы можете управлять.

Функция **printf( )** имеет прототип в файле STDIO.H:

int printf(char* управляющая_строка**,**список параметров**);**

Управляющая строка содержит:

-символы, управляющие выводом,

-символы, которые непосредственно выводятся на экран,

- команды формата (спецификаторы формата), определяющие, как выводить аргументы. Команда формата начинается с символа %, за которым следует код формата. Основные команды формата следующие:

%c – символ,

%d – целое десятичное число,

%i – целое десятичное число,

%e – десятичное число в виде x.xx e+xx,

%E – десятичное число в виде x.xxE+xx,

%f – десятичное число с плавающей запятой xx.xxxx,

%o – восьмеричное число,

%s – строка символов,

%X – шестнадцатеричное число 5A7F,

%% - символ %,

%p – указатель (адрес).

В управляющей строке перед символами, управляющими выводом, должен находится символ “\”. На практике наиболее часто используемым символом, управляющими выводом, является символ n – переход на начало новой строки.

Между знаком % и форматом команды может стоять целое число. Оно указывает на наименьшее поле, отводимое для печати. Если строка или число больше этого поля, то строка или число печатается полностью, игнорируя ширину поля. Нуль, поставленный перед целым числом, указывает на необходимость заполнить неиспользованные места поля нулями. Вывод **printf(«%04d», 378);** даст результат 0378.

При выводе десятичное число с плавающей запятой для того, чтобы указать число десятичных знаков после целой части числа,

сначала указывается общее число символов для вывода этого числа (включая десятичную точку), затем ставиться точка и целое число, указывающее на количество десятичных знаков. Например:
printf("\n Float =%7.2f\n Integer =%5d\n", a, k):

Выравнивание выдачи производится по правому краю поля. Если мы хотим выравнивать по левому знаку поля, то сразу за знаком % следует поставить знак минус.

В прототипе функции в списке параметров (переменных или констант) параметры следуют через запятую и подлежат выдаче в соответствующем формате. Порядок следования форматов находится в строгом соответствии с порядком следования параметров.

scanf( )- основная функция ввода с клавиатуры. Она предназначена для ввода данных любого встроенного типа и автоматически преобразует введенное число в заданный формат. Функция **scanf( )-** имеет прототип в файле STDIO.H:

int scanf(char* управляющая_строка**,** список адресов параметров**);**

Управляющая строка имеет то же назначение, что и для функции **printf( )**.При работе с функцией **scanf( )** от пользователя требуется знание последовательности вводимых параметров и знание строгого соответствия формата для каждого вводимого параметра, что часто создает значительные неудобства. На практике для ввода данных, как правило, используют так называемый "безопасный ввод", суть которого состоит в следующем. Сначала данные вводятся с клавиатуры в виде строки символов с помощью функции gets(), а затем выполняется преобразование строки символов в числовой тип данных. Функция **atoi(s)** преобразует строку **s** в целое число, функция **atof(s) –** в вещественное, а функция **atol(s) –** в длинное целое (тип **long**). Например:

```
char msg;
int i;
...
puts("Input integer");
gets(msg);
i = atoi(msg);
```

Функции **atoi( ), atof( ), atol( )** имеют прототипы в файле STDLIB.H.

Задание 1

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <stdio.h>
enum color{black, white};
enum Figure{King, Qween, Rook, Bishop, Knight, Pawn};
struct Chess_Fig{color col;Figure fig;};
void main( )
{
Chess_Fig Field[8][8], Chess;
Chess.co l= black;
Chess.fig = Knight;
Field[0][1] = Chess;
 printf("%d %d",Field[0][1].col, Field[0][1].fig );
}
```

Правильный ответ: **0 4 .**

Указатели

Указатель является переменной, которая содержит адрес другой переменной или функции. Описание указателя определяет тип данных, на которые указатель ссылается. Описание указателя имеет следующий вид:

тип указываемых данных *** имя** указателя **;**

Примеры возможных описаний указателей:

int *ip; // указатель **ip** на целое и целое **i**

double *dptr; // указатель **dptr** на число двойной точности

char *sn; // указатель **sn** на символ

С указателями связаны три специальные операции **&**, * и , которые имеют наивысший приоритет. Эти операции являются унарными, т.е. имеют один операнд, перед которыми они ставятся. Операция **&** означает «взять адрес». Результатом операции * является значение переменной, на которую ссылается указатель. Например:

int *ipoin, i = 1, j;

ipoin = &i; // указатель **ipoin** получает значение адреса целой переменной **i**

j = *ipoin; целая переменная **j** получает значение **1**

Нельзя создать переменную типа **void**, но можно создать указатель на тип **void**.

Так как указателям типа **void** не соответствует никакой тип данных, к ним нельзя применять арифметические операции.

Указателю на **void** можно присвоить указатель любого другого типа. Однако при обратном присваивании необходимо использовать явное преобразование указателя на **void** :

```
void *pv;
int a, *b;
b = &a;
pv = b;
b = (int)pv;
```

В объявлении переменной, являющейся указателем, очень важен базовый тип. По этому типу компилятор определяет сколько байт памяти занимает переменная, на которую ссылается (указывает) данный указатель. Над указателями можно производить сложение и вычитание целых чисел (операции ++ и - - являются частными случаями операций сложения и вычитания). Так например, если переменная **P** является указателем на какой-либо тип, то при каждой операции **P++** значение указателя будет увеличиваться на количество байт, занимаемых переменной данного типа. Пусть **к** – какое-либо целое число. Тогда для вычисления значения указателя после выполнения операции **P = P + к** будет справедливо выражение:

P = P + к *(количество байт памяти типа указателя).

Использование указателя в выражениях не соответствующих его типу приведет либо к ошибке при компиляции, либо к получению неверного результата в вычислениях. В языке существует предопределенная адресная константа **NULL** (никуда), значение которой может быть присвоено указателю любого типа.

Указатели можно сравнивать. Применимы следующие операции:

<, >, <=, >=, =, ==, != .

Сравнение двух указателей **P<B** означает, что адрес, находящийся в **P**, меньше адреса, находящегося в **B**.

В том случае, когда необходимо иметь адрес для хранения другого адреса, можно объявить указатель на указатель. Например:

```
float dat;
float* pd = &dat;
float** Pp = &pd;
```

Однако язык С не позволяет использовать операцию двойной адресации. Следующая запись будет восприниматься компилятором как ошибка:

```
float** Pp = &&dat;
```

Другие операции над указателями запрещены, например нельзя сложить два указателя, умножить указатель на число и т.д.

Указатель на символьную переменную часто используется при работе с символьными строками. Символьная строка представляет собой массив байтов, заполненный кодами символов, и завершающийся нулевым байтом (символом ‘\0’):

```
char Str[] = "Press any key";
```

Компилятор, выделяя память под этот массив, сам определит его длину, которая в данном случае будет равна 13 байтам (12 байт под собственно данные и 1 байт под завершающий нулевой байт). В данном примере фиксированный набор заранее известных символов представляет собой строку-константу. Такие строковые константы (фиксированные массивы символов) могут быть объявлены как указатели на символ, т.к. имя массива является указателем на массив:

```
char* Str = " Press any key";
```

В обоих случаях в конец строки автоматически добавляется нулевой байт.

Если содержимое символьной строки в момент ее объявления неизвестно, можно объявить пустую строку без инициализации (19 байт под собственно данные и 1 байт под завершающий нулевой байт):

```
char Message[20];
```

В дальнейшем в этот массив можно поместить содержательную строку.

Набор стандартных строк объявляется следующим образом:

```
char* Error[ ] = {"Can’t open file", "System error"};
```

При этом обращение к каждой строке производится как обращение к каждому элементу массива:

```
for( i = 0; i < 2; i++)
```

```
printf("\n %s", Error[i]);
```

Операция используется для доступа к полям структуры при использовании указателя на структуру. Например:

```
struct Person{   //Созданный пользователем новый тип
  char* name;   //Указатель на имя индивидуума
  int age;      //Его возраст
  };            //Конец описания структуры
Person S1, S2;  //Объявляются две структурные переменные
                  S1 и S2 типа Person
```

Объявление указателя на структуру:

```
Person * Sp;    //Объявляется указатель на тип Person
```

Однако в этом случае компилятор выделяет место только под сам указатель, а не под всю структурную переменную. Для использования объявленного указателя для работы со структурными переменными необходимо присвоить ему адрес переменной:

```
Sp = & S1;
```

Если структурная переменная объявляется с помощью указателя на структуру, то для обращения к членам структуры оператор "**.**" заменяется на " ":

```
int a;
Sp  name  =  "Ivan";   //Инициализация первого элементаструктурной переменной
Sp  age = 28;   //Инициализация второго элемента
a = Sp  age;
```

В языке C++ для объявления указателя на структурные переменные можно использовать оператор **new**, который служит для динамического выделения памяти:

```
Person * Sp = new Person;  //Объявление указателя на структурную
                             переменную с выделением под нее памяти
```

Пример использования указателя на структуру, объявленного с помощью оператора **new**:

```
struct comp{      //Объявление структуры
int R;
float M;}
comp *S = new comp;   //Объявление указателя на структурную
                      // переменную с выделением под нее памяти
int n = 2;
float f = 3.14;
```

S R = n; //Присваивание полям структуры соответствующих значений

S M = f;

Для того чтобы корректно объявлять указатели и правильно понимать объявления указателей необходимо соблюдать следующее правило: читать информацию при объявлении указателя следует не слева – направо, а наоборот: справа – налево относительно символа *****. Например, следующую запись

const *double dp

надо интерпретировать так: **dp** – это постоянный указатель (указатель – константа) на тип **double**.

Следующее объявление указателя

float *Ramp[10]

следует понимать как массив **Ramp[10],** состоящий из десяти указателей на тип **float.**

Операции инкремента ++ (декремента - -)

Операции инкремента ++ (декремента - -) используются в языке С для увеличения (уменьшения) значения целой переменной на единицу. Например, для целой переменной **i** выражения **i** ++ и **i** = **i** +1 будут эквивалентны. Операции инкремента ++ (декремента - -) обладают более высоким приоритетом, чем арифметические операции. Различаются операции пред-инкремента ++ (декремента - -) (например ++ **i**) и пост-инкремента ++ (декремента - -) (например **i** ++). В том случае, когда операция инкремента какой-либо целой переменной реализуется вне арифметического выражения (например ++ **i** ; или **i** ++ ;) пред-инкремент и пост-инкремент приводят к одному и тому же результату – увеличению переменной на единицу. Если же целая переменная принимает участие в арифметическом выражении (например **j** = ++ **i** **;** или **j = i ++ ;**), в случае пред-инкремента значение переменной сначала увеличивается на единицу, а затем обновленная переменная участвует в вычислениях. В случае пост-инкремента текущее значение переменной сначала участвует в вычислении арифметического выражения и только после вычислений увеличивает свое значение на единицу. Например:

int i = 2,j;

j = ++i; // после выполнения оператора присваивания значения **i** и **j** будут равны 3.

int i = 2, j;

j = i++; // после выполнения оператора присваивания значение **i** будет равно 3, а значение **j** будет равно 2.

Оператор безусловного цикла for

Оператор **for** позволяет выполнить некоторую последовательность операторов С (тело цикла) заданное число раз. Цикл **for** имеет следующую структуру:

for(начальное значениесчетчика; условие выполнения; новое значение счетчика**)**

{

Блок из одного или нескольких предложений языка С

}

Пример цикла **for** для вычисления суммы квадратов целых чисел от 0 до 19:

```
int i, j = 0;                //Объявляются  целые переменные i и j
for( i = 0; i < 20; i++)     //Цикл из 20 шагов для i от 0 до 19
  j = j + i*i;               //В j заносятся квадраты целых чисел
```

Начальное значение счетчика цикла вычисляется один раз при входе в цикл. Обычно это операция присваивания вроде **i=0**. Условие выполнения представляет собой операцию отношения, например **i < 20**. Пока указанное условие выполняется, цикл продолжается.

Новое значение счетчика вычисляется перед каждым очередным шагом цикла. Часто (хотя совсем не обязательно) здесь используется операция инкремента **i++.**

Если тело цикла состоит лишь из одного оператора, то фигурные скобки, ограничивающие цикл, можно опустить. Если же цикл состоит из нескольких операторов, то их следует заключить в фигурные скобки:

```
int i, k = 0, m = 0;   // Объявляются  целые переменные i , k и m
for(i = 1; i < 21; i++)  // Цикл из 20 шагов для i от 1 до 20
{                        // Начало тела цикла
  j = j + i*i;           // В j заносятся квадраты целых чисел
 m = m + i ;             // В m заносятся целые числа
}                        // Конец тела цикла
```

Счетчик цикла может быть инициализирован любым числом и получать любые приращения, не обязательно 1:
for(k = 10; k < 20; k+ = 2) //5 шагов, k изменяется от 10 до 18 через 2
Счетчик цикла может работать в обратном направлении:
for(j = 50; j >= 0; j--) // 51 шаг, j изменяется от 50 до 0 через 1
Счетчик цикла не обязательно должен быть целым числом:

```
float f, Tf[400];
int n = 0;
for( f = 0.01; f < 3.14; f+ = 0.01)
{
Tf[n] = sin(f)/f;   // Массив заполняется значениями sin(f)/f
  n++;              // Инкремент индекса массива
  }
```

В этом примере счетчик цикла **f**, являющийся дробным числом, не может служить индексом элемента массива. Для индексации элементов пришлось ввести отдельную целочисленную переменную **n** и инкрементировать ее значение на каждом шаге в теле цикла.

Задание 2

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <stdio.h>
void main( )
{ int i, j = 0, k = 0;
 for(i = 0; i < 3; i++)
 {  j = ++i;
    k = i++;
 printf("\n %d %d %d", j, k, i);
 };
}
```

Правильный ответ: **1 1 2 .**

Циклы while и do/while

Конструкция **while** служит для повторения цикла все время, пока выполняется заданное условие. Цикл **while** имеет следующую структуру (если операторный блок состоит из одного оператора, фигурные скобки можно опустить):

```
while( условие )
  {
   Блок из одного или нескольких операторов С
  }
```

Блок операторов в фигурных скобках будет выполняться многократно, пока **условие** при операторе **while** остается истинным. Как только выражение станет ложным, управление передается на оператор, следующий за циклом **while**. Очевидно, что тело цикла должно как-то влиять на выражение, определяющее условие выполнения, и в конце концов, сделать его ложным, иначе цикл никогда не завершится. Следует отметить, что в языке С на месте условия может быть любое вычисляемое выражение. Если значение выражения не равно нулю, условие считается истинным. Таким образом, бесконечный цикл может записан в следующим виде:

```
while(1)
  {
   Блок из одного или нескольких операторов С
  }
```

В отличие от цикла **for**, с помощью которого обычно обрабатываются массивы с известным числом элементов, цикл **while** удобно использовать в тех случаях, когда размер массива неизвестен. Пусть, например, в символьном массиве **st** была программно сформирована некоторая фраза и нам надо определить ее длину. Учитывая, что любая символьная строка завершается символом **\0** , достаточно перебирать все элементы массива до нахождения элемента со значением **'\0'**:

```
char*  st = "qwerghknc";
 int  i = 0, length;
 while( *(st++) != '\0' )  i++;
length = i;
```

Здесь в тело цикла входит единственное предложение **i++**, т. е. в цикле нет ни одного "содержательного" предложения. Действи-

тельно, нам ничего не надо делать с символами строки, а только перебирать их до нахождения требуемого. С другой стороны конструкция **while** (в отличие от конструкции **for**) не наращивает значение индекса. Поэтому инкремент индекса, если он используется, необходимо выполнять явным образом.

Как только сравнение ***(st++)** и **' \0'** даст положительный результат, действие **i++** будет пропущено и управление передастся за пределы цикла **while**, т. е. на предложение **length=i**. Таким образом, в переменную **length** будет занесено значение индекса **i**, соответствующее завершающему нолю. Это значение совпадает с длиной фразы.

В цикле do/while проверка условия производится после исполнения тела цикла. Это означает, что оператор или блок операторов, связанный с циклом, будет обязательно исполняться хотя бы один раз. Синтаксис оператора имеет следующий вид:

do { оператор или блок операторов **}**
while (условие **)**

Цикл do/while весьма полезен, когда нужно спросить у пользователя следует ли повторять определенную операцию. Например:

```
char answer;
 int  i = 0;
 do{
     // Блок операторов
 i++;
 printf ("Хотите завершить исполнение ?");
 answer = getche( );
 }  while ( answer != 'Y' && answer! = 'y');
  printf ("Число операций =  %d", i );
```

Условные операторы if и if/else

Оператор if осуществляет условное ветвление программы, проверяя истинность условия (логического выражения или комбинации выражений). Он имеет следующий вид:

if (условие**){**
оператор или блок операторов, исполняемый если условие истинно
};

Если условие неистинно, выполняется следующий за **if** оператор.

Следующий код иллюстрирует применение оператора if с простым исполняемым оператором:

```
int a=2, b=5, c, d;
if(a<b) c = a + b;
d=a*b;
```

При необходимости в комбинации с **if** можно использовать ключевое слово **else,** позволяющее выполнить альтернативный оператор, если выражение в условии неистинно. Ниже следует пример применения комбинации **if / else :**

```
int a=2, b=5, c;
if(a<b) c =a +b;
else  c= a -b;
d=a*b;
```

Операторы **if** и **if / else** могут быть вложенными. Если такая конструкция является двусмысленной, компилятор ставит каждое **else** в соответствие ближайшему **if.** Чтобы избежать двусмысленности, следует использовать операторные скобки **{ }.** Например:

```
if(a<b)
{ c = a + b;
  if(a == 0) c = 0;
}
else c = a -b;
d=a*b;
```

Условный оператор switch/case

В качестве альтернативы оператора **if**, для сложного условного ветвления язык С предоставляет конструкцию с ключевыми словами **switch** и **case**, синтаксис которой имеет следующий вид:

switch (выражение**)**
{
case константное выражение**:** оператор или группа операторов
case константное выражение**:** оператор или группа операторов
...
default: оператор или группа операторов
}

Результат вычисления *выражения* сравнивается с каждым из *константных выражений.* Если находится совпадение, то управление передается оператору или операторам, связанным с данным

case.Исполнение продолжается до конца тела оператора **switch** или пока не встретится оператор **break**, который передает управление из тела **switch** вовне. Операторы, связанные с ключевым словом **default**, выполняются, если выражение не соответствует ни одному из константных выражений в **case.** Например:

```
int a, b;
char s;
...
    switch(s)
{case '+': printf("\n a + b = %d", a + b);break;
  case  '-': printf("\n a – b = %d", a - b);break;
  case '*': printf("\n a*b = %d", a*b);break;
  case '/': if(b!=0)printf("\n a/b = %d", a/b);
     else printf("Деление на нуль!");
default: printf("Операция не определена!");
}
```

Задание 3

Объясните, как работает нижеприведенная программа. Ввести, отладить и запустить программу.

```
/*
==================  КАЛЬКУЛЯТОР
=================
==================  ( + , - , * , / )  ================
*/
//..подключение библиотек
#include <conio.h>  //функции clrscr(),gotoxy(),getche()
#include <stdlib.h> //функции exit(),atof()
#include <stdio.h>  //функции puts(),printf()
#include <math.h>   //функция pow()

//..начало функции main()
void main(void)
{
 char st[50];              //описание символьных переменных
 char *s = st;
 int i, j, k;
```

```
int sign[10];
double a, b;
clrscr( );                    //очистка экрана
puts("Калькулятор выполняет операции: +, -, *, /, ^ ,$.($ - вы-
ход)");
puts("\n Введите A, знак операции, B, знак""="" \n");
for( ; ; )                    // while(1)  // начало бесконечного цикла
 {
  i = 0; j = 0;
  while ((*s = getche()) != '=')
   {
         if(*s == '$')
          {
           puts(" - Good bye!");
           exit(0);
          };
         switch (*s)        //выбор варианта операции
          {
           case '+': ;       //переход на выполнение при sign='+'
           case '-': ;
           case '*': ;
           case '/': ;
           case '^': sign[j++] = i; break;
    };
          i++; s++;
        };                     // конец ввода строки
          s = st;
          a = atof(s);
          for(k = 0; k < j; k++)
          {
                B = atof(s + sign[k] + 1);
           switch (*(s + sign[k]))      //выбор варианта операции
            {
             case '+': a+ = b; break;
             case '-': a- = b; break;
             case '*': a* = b; break;
             case '/': if(b! = 0) a/ = b;
                       else printf("Деление на нуль!\n"); break;
             case '^': a = pow(a,b); break;
```

```
            default: printf("\n Знак операции неопределен\n");
          }                          //конец switch
        }                            //конец for
      printf("%f\n", a);
  };                  //конец бесконечного цикла for(;;), (while(1))
}                     //конец функции main()
```

Функции, определяемые пользователем

Использование в программе прикладной функции требует, вообще говоря, выполнения трех действий:
- объявления прототипа функции;
- определения самой функции, т. е. выполняемых ею действий;
- вызова этой функции в одной или нескольких точках программы.

Если в программе используется некоторая прикладная функция, то в начале программы должен быть указан **прототип** этой функции, чтобы компилятор заранее получил информацию об ее свойствах.

Прототип функции имеет следующий формат:

тип_возвращаемого_значения имя_функции (тип_аргумента_1, тип_аргумента_2,...);

Примеры прототипов:

```
int Search(char*, float);     // функция требует 2 аргумента
                              // типов char* и float и возвращает int
void Print(int);              // функция требует 1 аргумент типа int
                              // и ничего не возвращает
char* Letter(void);           // функция не требует аргументов
                              // и возвращает char*
```

В программе должно содержаться **определение** функции. Это определение может находиться в любом месте программы – как до главной функции **main()**, так и после нее.

Если определение функции находится до главной функции **main()**, прототип этой функции указывать необязательно. Если тип функции не указывается, по умолчанию тип функции определяется как **int.**

Определение начинается с заголовка, фактически повторяющего прототип, но с указанием формальных имен аргументов функции; за заголовком в фигурных скобках перечисляются предложения

языка, составляющие тело функции. Если функция возвращает какое-либо значение, ее имени должен быть присвоен тип возвращаемого значения, а ее текст должен заканчиваться оператором **return**:

```
float Add(float a1, float a2)
 {float b;
  b=a1+a2;
  return b;
 }
```

Если функция не возвращает никакого значения, ее текст не должен заканчиваться оператором **return**, а сама функция должна возвращать тип **void**.

Вызов функции производится в теле главной функции **main()** или в теле других функций, при этом место формальных аргументов функции занимают фактические значения переменных. Например:

```
  float x=3.14, y=8.87, z;
  z=Sum(x, y);
```

Три способа передачи параметров в функцию

В языке С++ предусмотрены три способа передачи параметров в вызываемую функцию:

по значению, когда в функцию передается числовое значение параметра;

по адресу, когда в функцию передается не значение параметра, а его адрес, что особенно удобно для передачи в качестве параметров массивов и структур;

по ссылке, когда в функцию также передается адрес параметра, однако действуют особые правила использования этого адреса в вызываемой функции.

Пример передачи в функцию скалярного значения и возврата скалярного результата. Пусть у нас определена функция **AddF (float, float),** осуществляющая сложение двух вещественных чисел, причем переменной **AddF** присваивается значение суммы (классическое определение функции):

```
float AddF(float a, float b)
   {
```

```
  return a + b;
  }
```

Программный фрагмент с вызовом этой функции может иметь, например, такой вид:

```
float x = 3.14;     //Определим первое слагаемое
float y = 2.71;     //Определим второе слагаемое
float z;            //Объявим переменную для результата
z = Sum(x, y);      //Собственно вызов функции
```

Рассмотрим теперь передачу в функцию адреса, при этом в качестве параметра в функцию могут передаваться как скалярные величины, так и массивы. Определим функцию **AddA (float, float, float*),** осуществляющую сложение двух вещественных чисел, причем значение суммы будет передаваться по адресу третьего параметра:

```
/*   Определение функции AddA(float, float, float*) */
void AddA(float A, float B, float* Z)  // Заголовок функции
  { *z = a+b;              // Значение суммы передается по адресу z
  }
/*   Фрагмент программы с вызовом функции AddA()  */
float x = 3.14;           //Определим первое слагаемое
float y = 2.71;           //Определим второе слагаемое
float z;                  //Объявим переменную для результата
z = AddA(x, y, &Z);       //Собственно вызов функции
```

Определим функцию **Sub(int*, int*, int*)**осуществляющую вычисление минимального элемента **min** массива **A** и вычисление элементов массива **B = A - min:**

```
void Sub(int A[10], int B[10], int* min) // Имя массива A – это
                          //указатель (адрес) на первый элемент массива
{ int i;
  *min =A[0];
  for(i = 1; i < 10; i++)
  if(A[i] < *min)
  *min=A[i];
  for(i = 1; i < 10; i++)
  B[i] = A[i] - *min;
}
/*  Фрагмент программы с вызовом функции Sub(int*, int*, int*), */
int H[10], m;             //Массив с данными
int G[10];                //Вычисляемый массив
```

Sub(H, G, &m); //Вызов функции

При передаче параметра по ссылке в функцию также передается адрес параметра, однако в дальнейшем все действия производятся не как с адресом, а как со значением параметра. Это снижает вероятность "описок " при создании кодов большого объема.
Определим функцию **SumS(int, int, int&),** осуществляющую сложение двух целых чисел, причем значение суммы будет передаваться по ссылке на третий параметр:

```
/*  Определение функции SumS(int , int , int& ) */
void SumS(int x, int y, int& z)          //Заголовок функции
  { z = x + y;          // Значение суммы передается по ссылке на z
  }
/*  Фрагмент программы с вызовом функции SumS()  */
int i = 1;          // Определим первое слагаемое
int j = 10;         // Определим второе слагаемое
int k;              // Объявим переменную для результата
Sum(i, j, k);       // Вызов функции
```

Задание 4

Какая информация появится на экране дисплея после выполнения данной
программы:

```
#include<stdio.h>
void min(double*, double&);
void main( )
{
 double a=100, b[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
 int i;
 printf("%f \n", a);
 min(b, a);
 printf("%f \n", a);
 }
 void min(double mas[10], double &num)
 { num = mas[0]; int i;
 for(i = 0; i < 10; i++)
 if(mas[i] < num)
```

```
num = mas[i];
}
```

Правильный ответ **100 1**.

Массивы и указатели. Указатель на функцию

В языке С существует важная связь между массивами и указателями. Имя массива соответствует адресу его первого элемента. Поэтому можно присваивать указателю адрес первого элемента, используя имя массива:

```
int Mas[10];
int* ip = Mas;                    // То же, что и ip = &Mas[0];
```

Компилятор преобразует индексацию в арифметику указателей. Например:

Mas[5] = 20; представляется как ***(Mas+5) = 20;**

Если массив этот используется как параметр для передачи в функцию, то для описания типа параметра удобно использовать указатель **int*** .

Для работы с двумерными массивами может быть использован указатель на указатель:

```
int** b;
int* a[3], i, j, k;
int aa[ ][3] = {{1,2,3}, {4,5,6}, {7,8,9}};          //  aa[3][3]
for(i = 0; i < 3; i++)
a[i]=&aa[i][0];
b = a;
k = *(*(b+2)+1);
j = aa[2][1];
```

Две последние строки этого кода выполняют одну и ту же операцию - обращение к элементу третьей строки второго столбца (число 8). Здесь запись **int* a[3]** означает объявление массива указателей на целое.

Строчная константа в языке С ассоциируется с адресом начала строки в памяти, тип строки получается **char*.** Поэтому активно используется следующее присваивание:

```
char* Str;
Str="Hello, friends!"
```

Часто массив указателей используется, если надо иметь стандартный набор строк:

char* Season[]={“Winter”, Spring”, ”Summer”, ”Fall”};

При таком объявлении строчные константы будут занесены в раздел констант в памяти, массив указателей будет состоять из четырех элементов, под которые будет выделена память, и эти элементы будут инициализированы адресами, указывающими на начало этих строчных констант.

Объявление указателя **Pf** на функцию, которая имеет тип **T1** и параметры с типами **T2** и **T3** выглядит следующим образом:

T1 (*Pf)(T2, T3);

Любые операции над указателями на функцию запрещены. Данный пример показывает, как объявляется указатель на функцию и как производится вызов функции с помощью указателя на функцию:

```
   double Qb(double g)
{ return g*g*g;
}
void main()
{ double (*p)( double);          // Объявление указателя на функцию
double t =1.78;
...
p = Qb;              // Инициализация  указателя p на функцию Qb()
...
(*p)(t);                    // Вызов функции p(t);
...
 }
```

Указатель на функцию используется для передачи функции как параметра другой функции:

```
    float F(double (*w)(double z))
  {z = 1;
   return w(z);}
   ...
   float a, b;
   ...
   double (*pt)(double );
   pt=tan;
   a=F(pt);
```

```
   b=FF(sin);
   …
```

Задание 5

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <stdio.h>
void preob(int* );
void main( )
{  int i, mas[3];
          for( i = 0; i < 3; i++)
          {mas[i] = 0;
           printf("%d \n", mas[i]);
          }
 preob(mas);
  for(i = 0; i < 3; i++)
   printf("%d \n",*(mas+i));
  return 0;
 }
  void preob(m[3])
  { int k;
    for(k = 0; k < 3; k++)
     m[k] = k + 1;
  }
```

Правильный ответ **0 0 0 1 2 3** .

Задание 6

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <stdio.h>
 f(int [ ][3]);
void main()
{ int* a[3], i, j;
  int aa[ ][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
```

```
 int** b;
 j = f(aa);
 b = a;
 for(i = 0; i < 3; i++)
 a[i] = &aa[i][0];
  i = *(*(b+2)+1);
 printf("%d\n", i);
 i = aa[2][1];
 printf("%d\n", i);
 printf("%d\n", j);
}
f(int s[3][3])
{ return s[2][1];
}
```

Правильный ответ: **8 8 8** .

Работа с файлами в языке С

Для того чтобы работать с файлом, в программе, прежде всего, следует объявить указатель на стандартную структуру с именем **FILE**, которая описана в библиотеке **stdio.h** и в которой задаются характеристики файла (размер, тип, дата и т.д.). Например, **FILE* fpnt.** Затем необходимо связать файловую переменную **fpnt** с конкретным файлом. Эта связь выполняется с помощью функции **fopen()** (открыть файл), которая также описана в библиотеке **stdio.h** и возвращает указатель на структуру **FILE** [1]. Операция открытия файла записывается следующим образом:

fpnt=fopen(“имя файла**”,”**режим открытия файла**”);**

Функция **fopen()** имеет два параметра (строковые константы): имя открываемого файла и режим открытия файла. Режим открытия файла определяет, как будет пользователь работать с файлом: читать его, записывать в него информацию или проделывать другие манипуляции с файлом. Основные режимы открытия файла:

r – файл открывается только для чтения;

w - файл создается только для записи информации в него (существующая информация стирается);

a - файл открывается для дозаписи информации в конец файла. Если файл не существует, он создается для записи в него;

r+ – файл открывается для обновления: чтения и записи;
w+ - файл создается для работы в режиме обновления: такой файл можно будет читать и записывать в него;
a+ - файл открывается для дозаписи информации в конец файла. Если файл не существует, он создается.

Если по какой-либо причине файл не открылся, функция **fopen()** возвращает значение **NULL.** Процедуру открытия файла рекомендуется производить следующим образом:

if((fpnt = fopen("C:\Student\Petrov\MyFile.txt", "w")) == NULL)
{puts("Can't open file"); exit(1);}

После того как программа с данным файлом отработала, нужно закрыть файл – разорвать связь структуры **FILE** с отработавшим файлом. Эта процедура реализуется с помощью функции **fclose(fpnt).** Только после закрытия с файлом можно выполнять какие-либо действия – удалить или заново открыть в другом режиме.

После того как файл открыт, для чтения и записи обычно используются следующие наиболее часто используемые специальные функции:

fputc(c, fp) – запись символа в файл, **c** – символ, **fp** – указатель файла;
fputs(s, fp) – запись строки в файл, **s**– строка, **fp** – указатель файла;
fgetc(fp) - чтение символа из файла с указателем **fp;**
fgets(s, m, fp) - чтение строки **s** из файла с указателем **fp, m** - максимальное число символов в строке;
fread(buf ,m ,n, fp) – чтение **n** элементов данных из файла с указателем **fp.**

Каждый элемент имеет длину **m** байт. Чтение производится в буфер, на который указывает указатель **buf**.

fwrite(buf ,m, n, fp) – запись **n** элементов данных в файл с указателем **fp.**

Каждый элемент имеет длину **m** байт. Запись производится из буфера, на который указывает указатель **buf**;
fseek(fp ,n, m) - указатель в файле **fp** устанавливается в позицию, отстоящую на **n** байт от точки отсчета, которая задается одним из значений (0,1,2) параметра **m**:

0 – отсчет от начала файла;
1 - отсчет от текущей позиции файла;
2 - отсчет от конца файла**;**

fprintf(fp, Str, a1, a2, …, aN)- запись данных **a1 ,a2 ,…, aN** в файл с указателем **fp** в форматах, которые указаны в управляющей строке **Str;**
fscanf(fp, Str, &a1, &a2 ,…, &aN)- чтение данных **a1, a2, …, aN** из файла с указателем **fp** в форматах, которые указаны в управляющей строке **Str;**
putc(symb, fp) – запись символа **symb** в файл с указателем **fp**, в случае ошибки функция возвращает константу **EOF;**
getc(fp) – чтение символа из файла с указателем **fp**. Функция возвращает константу **EOF**, если достигнут конец файла или произошла ошибка при чтении;
feof(fp) – определение конца файла файла с указателем **fp.**Если конец файла достигнут функция возвращает значение "1", в противном случае – "0".

При запуске любой программы автоматически открываются сразу три файла:
- файл стандартного ввода (его указатель называется **stdin**),
- файл стандартного вывода (его указатель называется **stdout**),
- файл стандартного ввода ошибок (его указатель называется **stderr**).

При работе с файлами мы можем использовать эти указатели, чтобы направлять данные в стандартные потоки, в которых по умолчанию ввод идет с клавиатуры, а вывод - на экран.

Например, чтобы ввести строку с клавиатуры можно использовать функцию **fgets(s, m, fp)** в виде **fgets(s, m, stdin)**, а для вывода строки на экран – функцию **fputs(s, fp)** в виде
fputs(s, stdout).

Задание 7

Объясните, как работает нижеприведенная программа. Ввести, отладить и запустить программу.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main( )
{ int k, k1;
  char str[50], str1[50], ch, ch1;
```

```
 FILE* fp;
 if((fp = fopen("my_file.txt","w")) == NULL)
 {puts("Can't open file\n");
 exit(1);
 }
 puts("Input integer");
 gets(str);
 k = atoi(str);
 fprintf(fp,"%d\n",k);
 puts("Input char");
 ch = getche( );
 putc(ch,fp);
 puts("\nInput string");
 gets(str);
 fputs(str,fp);
 fclose(fp);
        if((fp = fopen("my_file.txt","r")) == NULL)
 {puts("Can't open file\n");
 exit(1);
 }
 fscanf(fp,"%d\n",&k1);
 printf("%d\n",k1);
 ch1=getc(fp);
 putchar(ch1);
 puts(" ");
 fgets(str1, 50, fp);
 puts(str1);
 fclose(fp);
}
```

Задание 8

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main( )
{int k, i;
```

```
int const N = 10;
double a[N], b[N], c[N], d[N];
FILE* fp;
for(i = 0; i < N; i++)
 {a[i]=i;
  b[i]=N-i;
 }
if((fp = fopen("mult.txt","w")) == NULL)
 {puts("Can't open file \n");
 exit(1);
 }
fwrite(a, sizeof(double), N, fp);
fwrite(b, sizeof(double), N, fp);
fclose(fp);
if((fp = fopen("mult.txt", "r")) == NULL)
  {puts("Can't open file\n");
   exit(1);
  }
fread(c, sizeof(double), N, fp);
fread(d, sizeof(double), N, fp);
for(i = 0; i < N; i++)
   {printf("%f", c[i]);
   }
puts("");
for(i = 0; i < N; i++)
 { printf("%f", d[i]);
 }
fclose(fp);
}
```

Правильный ответ: **0 1 2 3 4 5 6 7 8 9**
10 9 8 7 6 5 4 3 2 1 .

Раздел 2

Основы программирования на языке С++

Особенности языка С++, не связанные с объектной ориентированностью

Первой особенностью является стандартный ввод-вывод, поддерживаемый стандартной библиотекой языка С++ - < **iostream.h** >, при этом используются новые операторы. Например, оператор
int k=24:
cout << ”Число итераций равно ” << k << ”\n”;
выведет на экран дисплея сообщение “Число итераций равно 24“ и курсор переместится в начало следующей строки. В языке С++ операция << имеет двоякое значение. В языке С она была определена как поразрядный сдвиг влево. Но в приведенном примере эта операция также является операцией вывода. Такая операция является перегружаемой. Ключевое слово **cout** связано со стандартным потоком вывода, аналогичным **stdout** языка С, по умолчанию связанного с экраном. Можно использовать **cout<<** для вывода на экран данных основных типов и символьных строк.
Ввод с клавиатуры целого числа, связанного с переменной целого типа **k** будет реализован оператором ввода
cin>>k;

Ключевое слово **cin** связано со стандартным потоком ввода, аналогичным **stdin** языка С, и по умолчанию связанного с клавиатурой. Используя **cin>>** можно вводить с клавиатуры значения переменных основных типов, а также символьные строки. Операция >> языка С++ также является перегружаемой.
Если мы пишем имена переменных, из которых выводятся или в которые вводятся данные, то по умолчанию для ввода/вывода используются определенные форматы. В записи мы не видим форматов, но при вводе значений этих переменных с клавиатуры (после каждого значения надо нажимать <Enter>) их форматы будут учтены.

Язык С++ имеет еще одну особенность. В языке С описание переменных должно заканчиваться до первого исполняемого оператора. Если в теле функций описание какой-либо переменной встре-

тится после хотя бы одного оператора, компилятор выдаст сообщение об ошибке. В языке С++ описание переменных может располагаться в любом месте программы. Например:

```
for(int i = 0; i < N; i++)
 c[i]= sin(i)/(i + 1);
```

Определение класса. Состав класса, создание экземпляров класса

Класс – это основной тип данных в языке С++, который является расширением понятия структуры в языке С. Определим класс **Student** с некоторыми данными (данные, входящие в состав класса, называют *данными-членами*) и базовым набором функций (*функций-членов*, или *методов*) для инициализации и чтения данных, принадлежащих этому классу.

```
class Student{
private:            // Закрытые данные:
  char* name;       // Имя
  int age;          // Возраст
public:             // Открытые данные и функции:
  void SetName(char* n){name = n;}  // Инициализация имени
  void SetAge(int a){age = a;}      // Инициализация возраста
  char* GetName( ){return name;}    // Чтение имени
  int GetAge( ){return age;}        // Чтение возраста
protected:          // Данные, доступные только наследникам класса:
  ...
 };          //  Завершающая скобка и точка с запятой
```

Данные, включенные в описание класса после описателя **private**, являются *закрытыми*. К ним можно обращаться только в функциях, принадлежащих этому же классу. Из программы или из другого класса к ним обратиться нельзя: компилятор сообщит, что данное с таким именем не существует. Закрытие (сокрытие) данных является важной концепцией объектно-ориентированного программирования (ООП), способствующей защите данных от несанкционированного доступа.

Функции объявлены и описаны в *открытой* части класса после описателя **public**. Это делает их доступными из программы, т. е. из главной функции **Main()**, а также и из прикладных функций, вызы-

ваемых по ходу выполнения программы. В классе обязательно должны быть открытые функции, иначе к элементам класса просто нельзя будет обратиться.

Как и в случае структур, *объявление* класса не создает реальных данных. Это лишь шаблон, описывающий созданный нами тип данных, в который входят не только собственно данные, но и функции для их обработки. Создание *объектов*, или *экземпляров* класса осуществляется точно так же, как и создание обычных переменных.

Создание экземпляра класса или объекта:

```
Student m1;
...
m1.SetName("Иванов");      // Вызов функций объекта m1
m1.SetAge(5)
```

Создание экземпляра класса с помощью указателя:

```
Student* pm2 = new Student;
...
pm2     SetName("Иванов");  // Вызов функций объекта m2
pm2     SetAge(23);
```

Перед именем вызываемой функции необходимо указывать объект, для которого она вызывается (имя объекта или указатель на объект). Нельзя выполнить предложение

```
SetName("Петров");
```

так как неизвестно, к какому объекту оно относится (в приведенном выше примере **m1** или **pm2**). Если объект класса создан по имени, то, как и для структур, имя объекта и имя функции разделяются точкой; если обращение к объекту выполняется с помощью указателя на объект, то указатель и имя функции разделяются оператором .

Конструктор класса, его назначение

Функция-конструктор формально отличается от всех остальных функций тем, что имеет имя, совпадающее с именем класса. Например, для класса **Student** конструктор должен имеет вид

Student(*параметры***){***тело конструктора***}**

Конструктор может иметь произвольное число параметров или не иметь их совсем, хотя часто в конструкторе предусматривается столько параметров, сколько данных-членов класса необходимо

инициализировать при создании нового объекта. Конструктор не может возвращать никакого значения (даже **void**).

Конструктор вызывается автоматически при создании объекта, и его назначение – выполнение необходимых инициализирующих действий. Прежде всего это, конечно, инициализация данных-членов. Часто в конструкторе выполняется создание и инициализация объектов вспомогательных классов, которые будут использоваться функциями данного класса, однако инициализирующие действия могут и не иметь прямого отношения к классу, например, это может быть открытие файлов с данными или вывод на экран какого-либо сообщения.

Поскольку назначение конструкторов – создание экземпляров класса (объектов), они, как правило, объявляются в открытой части класса, чтобы иметь к ним доступ из программы или из других классов.

Приведем пример класса **Student** с конструктором, который, как ему и положено, будет служить для инициализации данных-членов класса задаваемыми в программе значениями (именем и возрастом индивидуума):

```
class Student{
  char* name;
  int age;
public:
  Student(char*  n,  int  a){                    // Конструктор с двумя параметрами
    name = n;
    age = a;
    }
  char* GetName( ){return name;}  // Встроенные
  int GetAge( ){return age;}              // функции-члены
  };
```

Конструктор класса **Men** имеет два параметра и, следовательно, при создании объектов класса необходимо указывать два аргумента. Создать объект класса без указания начальных значений имени и возраста не удастся, так как в данном классе не предусмотрен конструктор без параметров.

Создание объектов класса Men при наличии конструктора будет выглядеть следующим образом:

```
  Student m1("Петров", 17);
```

Student* pm2 = new Student("Орлов", 23);

Конструктор класса с аргументами, задаваемыми по умолчанию. Конструктор копирования

Конструктор может иметь произвольное число параметров или не иметь их совсем, хотя часто в конструкторе предусматривается столько параметров, сколько данных-членов класса необходимо инициализировать при создании нового объекта. В том случае, когда за редким исключением объект инициализируется стандартным набором параметров, используется конструктор с параметрами, задаваемыми по умолчанию. Например:

```
class A{
int x, y, z;
public:
A(int z1, int x1 = 0, int y1 = 25){z = z1; x = x1; y = y1;}
int GetX1( ){return x;}
int GetX2( ){return y;}
};
```

При использовании такого конструктора инициализация объектов может быть произведена следующим образом:

```
A Ob(1, 2, 3);
A Ob(1, 2);
A Ob(1);
```

В первом случае при инициализации переменные получат следующие значения:

z = 1; x = 2; y = 3;

Во втором случае при инициализации переменные получат следующие значения:

z = 1; x = 2; y = 25;

В третьем случае при инициализации переменные получат следующие значения:

z = 1; x = 0; y = 25;

Параметры, задаваемые по умолчанию, должны находится в конце списка аргументов конструктора. Так, например, нельзя объявить конструктор с параметрами, задаваемыми по умолчанию, в следующем виде:

A(int z1 = 10, int x1 = 0, int y1){z = z1; x = x1; y = y1;}

Для копирования объектов и для передачи объектов как параметров в функции используются конструкторы копирования. Конструктор копирования имеет следующий формат:
A(const A& ob). Например:

```
class A{
private:
int* p;
public:
A(const A& ob)
{p=new int;
*p=*ob.p;}
```

При копировании объектов без использования конструкторов копирования создаются побитные копии объектов, что приводит к ошибкам при использовании динамической памяти.

Задание 1

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <iostream.h>
class B{
int x1, x2;
public:
B(int ax1, int ax2){x1 = ax1; x2 = ax2;}
int GetX1( ){return x1;}
int GetX2( ){return x2;}
};
void main( )
{B b1(5, 10, 20);
 cout<<b1.GetX1()<<" "<<b1.GetX2()<<" "<<b1.GetY()<<"\n";
 B b2(5,10);
 cout<<b2.GetX1()<<" "<<b2.GetX2()<<" "<<b2.GetY()<<"\n";
 B b3(5);
 cout<<b3.GetX1()<<" "<<b3.GetX2()<<" "<<b3.GetY()<<"\n";
 B b4;
 cout<<b4.GetX1()<<" "<<b4.GetX2()<<" "<<b4.GetY()<<"\n";
 }
```

Правильный ответ: **5 10 20**
5 10 3

5 2 3
1 2 3.

Задание 2

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <iostream.h>
#include <conio.h>
class A{
private:
int x, y;
public:
A(int ax, int ay){x = ax; y = ay;}
A(const A& ob)
 {
x = ob.x + 5;
y = ob.y + 5;
 }
int GetX( ){return x;}
int GetY( ){return y;}
A Sum(A od)
{
 od.x+ = x;
 od.y+ = y;
 return od;
 }
};
//void Print(A& oc)
void Print(A oc)
 {
Cout << "\n x = "<<oc.GetX() << "  y = " << oc.GetY();
 }
 void main( )
{
A a1(1, 2);
A a2(10, 20);
A asum=a1.Sum(a2);
Print(asum);
```

```
}
```
Правильный ответ: **x = 26 y = 37**

Задание 3

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <iostream.h>
#include <conio.h>
class A{
private:
int* p;
public:
A(int var){
p = new int;
*p = var;   }
A(const A& ob)
//A(A& ob)
{p = new int;
*p = *ob.p + 1; }
int Get (){return *p;}
~A( ){delete p;}
};
void Print(A oc){
cout << "\n var = "<< oc.Get();
}
void main( )
{
 A a1(1);
 //Print(a1);
 A a2 = a1;
 //A a2(10);
 Print(a1);
 Print(a2);
// a1.~A();
// Print(a1);
```

```
// Print(a2);
 }
```

Правильный ответ: **var = 2**
var = 3

Наследование

Язык C++ позволяет наследовать данные и функции одного или нескольких классов. Новый класс может получать атрибуты и поведение от уже существующего класса. Новый класс называют производным классом. Класс, элементы которого наследуются производным классом, называется базовым классом. В свою очередь производный класс может служить базовым для другого класса. Наследование дает возможность заключить некоторое общее поведение различных объектов в одном базовом классе. Несколько классов будут затем наследовать это поведение, являясь производными от базового. Наследование также позволяет немного изменить поведение существующего класса. Производный класс может переопределить некоторые функции базового, наследуя, тем не менее, основной объем свойств и атрибутов базового класса. Синтаксис наследования имеет следующий вид:

class Base{
...
};
class Derived: ключ доступа **Base{**
...
};

Ключ доступа не является обязательным и может быть private, protected или public. Если не указан, доступ принимается по умолчанию private для классов и public для структур. В следующем примере класс **B1** является открытым базовым классом для класса **Derived, B2** является защищенным базовым классом для класса **Derived, B3** является закрытым базовым классом для класса **Derived:**

class B1{
...
};

```
class B2{
 ...
};
class B3{
 ...
};
 class Derived: public B1, protected B2, private B3{
...
};
```

Если базовый класс наследуется как private, его открытые элементы будут являться private и в производном классе. Однако можно выборочно сделать некоторые из элементов базового класса открытыми в производном классе, указав их в секции public производного класса:

```
class Base{
            public: void f1();
                    void f2();
           };
class Derived: private Base{
public: void f1( );    // Делает void Base:: f1( ) доступной как public
```

Множественное наследование. Виртуальные базовые классы

Наследование, при котором производный класс, наследующий возможности не одного, а нескольких классов, называется множественным. Синтаксис множественного наследования выглядит следующим образом:

```
class A{
 ...
};
class B{
 ...
};
class C{
 ...
};
 class D: public A, public B, public C{
...
```

};

При объявлении производного класса с множественным наследованием все базовые классы перечисляются через запятую вместе с описателем public. В этом случае объект класса **D** будет включать в себя все члены классов **A, B** и **C**. Как и при одиночном наследовании из производного класса D можно обратится к любым public и protected членам базовых классов, private члены базовых классов производному классу недоступны. При создании объекта производного класса сначала конструируются (вызываются конструкторы) базовых классов в порядке их перечисления в объявлении производного класса и лишь после этого составляется объект производного класса. При завершении программы деструкторы производного и базовых классов вызываются в обратном порядке.

Виртуальные базовые классы используются в тех случаях, когда один и тот же класс служит базовым для нескольких производных классов, а те, в свою очередь, служат базовыми для некоторого производного класса следующего уровня. Рассмотрим следующий пример:

```
class A{
protected:
int x;
 ...
};
class B: public A{
 ...
};
class C: public A{
 ...
};
 class D: public B, public C{
...
};
```

Здесь переменная **x**, являющаяся членом класса **A**, будет присутствовать в объекте класса **D** в двух экземплярах. При обращении к переменной **x** из производных классов придется ее однозначно специфицировать (использовать оператор разрешения видимости):

```
class D: public B, public C{
public:
```

```
int BackX (){return x;}   // Недопустимо. Неясно какое из двух x выбрать
int BackX( ){return B:: x;} // Допустимо. x берется из B
int SetX(int a) {C:: x = a;}   // Допустимо. x берется из C
...
};
```

Чтобы не допустить появления нескольких экземпляров одной и той же переменной, надо базовый класс объявить виртуальным:

```
class A{
protected:
int x;
 ...
};
class B: public virtual A{
 ...
};
class C: public virtual A{
 ...
};
 class D: public B, public C{
...
};
```

Теперь допустимо обращаться к переменной **x** просто по имени, т.к. в объекте класса **D** она присутствует в единственном экземпляре.

Передача аргументов в базовый класс

При создании объектов производного класса требуется инициализировать данные не только производного, но и базового классов. Для этого в конструкторе производного класса нужно вызвать конструктор базового класса, для чего существует специальная конструкция. Например:

```
class A{
int x1, x2;
public:
A(int ax1, int ax2){x1 = ax1; x2 = ax2;}
...
};
```

```
class B: public A{
int y;
public:
B(int ax1 ,int ax2, int ay):A(ax1, ax2){y = ay;}
...
};
```

В конструкторе производного класса перечисляются в качестве параметров все переменные как производного, так и базового классов. Все параметры предваряются их типами. После двоеточия указывается имя конструктора базового типа с его параметрами. Здесь типы переменных уже не указываются. В теле конструктора производного класса осуществляется обычная инициализация его данных. В нашем примере при вызове конструктора производного класса необходимо передать ему все три параметра. В конструкторе класса **B** параметры класса **A** следует перечислять в том порядке, в каком они указаны в конструкторе (уже имеющемся) класса **A**. В нашем примере при создании объекта класса **B**:

```
B objt(1, 2, 3);
```

значение 1 будет передано в переменную **x1** класса **A** , значение 2 будет передано в переменную **x2** класса **A** , значение 3 будет передано в переменную **y** класса **B**.

У базового и производного классов может быть любое число инициализируемых параметров и, следовательно, число аргументов у их конструкторов.

Конструкторы с инициализацией по умолчанию в иерархии классов

Конструкторы базовых и производных классов могут включать в себя переменные, назначаемые по умолчанию. Рассмотрим пример:

```
#include <iostream.h>
class A{
        int x1, x2;
         public:
        A(int ax1, int ax2){x1 = ax1; x2 = ax2;}
         int GetX1( ){return x1;}
         int GetX2( ){return x2;}
```

```
        };
class B: public A{
         int y;
         public:
         B(int ax 1= 1 ,int ax2 = 2, int ay = 3) :A(ax1, ax2){y = ay;}
         int GetY(){return y;}
                  };
void main( )
{B b1(5, 10, 20);
 cout<<b1.GetX1()<<" "<<b1.GetX2()<<" "<<b1.GetY()<<"\n";
 B b2(5, 10);
 cout<<b2.GetX1()<<" "<<b2.GetX2()<<" "<<b2.GetY()<<"\n";
 B b3(5);
 cout<<b3.GetX1()<<" "<<b3.GetX2()<<" "<<b3.GetY()<<"\n";
 B b4;
 cout<<b4.GetX1()<<" "<<b4.GetX2()<<" "<<b4.GetY()<<"\n";
 }
```

При создании объекта класса **B** тем его переменным, которые принадлежат классу **A** (данные x1 и x2) будут присвоены либо значения, указанные при вызове, либо значения по умолчанию из конструктора производного класса. Таким образом, результат выполнения данной программы будет следующим:

```
5  10  20
5  10   3
5   2   3
1   2   3.
```

Конструктор класса **A** также может иметь переменные по умолчанию, однако эти значения можно реализовать только при создании объекта класса **A:**

```
A(int ax1 = 10, int ax2 = 20){x1 = ax1; x2 = ax2;}
...
 A a;
cout << a.GetX1() << " " << a.GetX2() << "\n";
```

Если инициализация данных базового класса по умолчанию устраивает пользователя, то в конструкторе класса **B** можно не вызывать конструктор класса **A:**

```
class B: public A{
int y;
public:
```

B(int ay = 0){y = ay;}

Если в классе **B** отсутствуют собственные данные, то, возможно, конструктор **B** вообще не нужен. Однако умолчание в классе **A** может не устраивать пользователя. Тогда в конструкторе **B** необходимо вызвать конструктор **A,** чтобы передать ему нужные значения аргументов:

```
class B: public A{
int y;
public:
B(int ax1, int ax2, int ax3):A(ax1, ax2, ax3){ };
```

Конструкторы базовых классов составляются таким образом, что в них сначала перечисляются наиболее важные параметры, изменение которых потребуется с большей вероятностью, а ближе к концу списка располагаются параметры, для которых во многих случаях годится инициализация по умолчанию.

Виртуальные функции

Виртуальная функция объявляется в базовом классе с помощью ключевого слова virtual и затем обязательно замещается хотя бы в одном производном классе. При этом не только имя, но и типы и число параметров, а также тип возвращаемого значения для замещаемых функций должны быть такими же, как и для виртуальной функции, объявленной в базовом классе. Если функция объявлена виртуальной в базовом классе, то все замещающие ее функции в производных классах автоматически становятся виртуальными:

```
class A{
public:
virtual void Func1();          //Прототип виртуальной функции Func1
virtual int Func2(char*);   // Прототип виртуальной функции Func2
};
class B: public A{
public:
virtual void Func1();        // Замещение виртуальной функции Func1
virtual int Func2(char*); // Замещение виртуальной функции Func2
 };                                  //  слово virtual можно опустить
```

Виртуальные функции проявляют свойства виртуальности только в том случае, когда они вызываются через указатель на базовый

класс. Указатель на базовый класс может принимать конкретное значение указателя на производный класс. Если этот указатель к моменту вызова функции фактически указывает на объект базового класса, вызывается вариант функции из базового класса. Если же указатель фактически указывает на объект производного класса, вызывается вариант функции из этого производного класса:

```
void main()
{ A* ap=new A;                    // Создается объект класса A
                                  // с указателем ap на него
  B* bp=new B;                    // Создается объект класса  B
                                  // с указателем bp на него
  C* cp=new C;                    // Создается объект класса  C
                                  // с указателем cp на него
  ap     Func1();                 //Вызывается Func1 из класса A
  ap=bp;
  ap     Func1();                 //Вызывается Func1 из класса B
  ap=cp;
  ap     Func1();                 //Вызывается Func1 из класса C
}
```

Таким образом, вызов виртуальной функции через указатель на базовый класс позволяет, в зависимости от конкретного значения этого указателя, вызывать варианты функций из разных классов.

Виртуальная функция базового класса часто нужна лишь для того, чтобы в производных классах было что замещать. В этом случае содержимое исходной функции не имеет значения и может быть пустым:

```
class A{
public:
  virtual void Func( ){ };
};
```

Если же у функции есть возвращаемое значение, пустой ее сделать нельзя. В этом случае лучше объявить в базовом классе чистую виртуальную функцию, для чего достаточно приравнять нулю прототип функции:

```
virtual float Func(int, double)=0;
```

Такую функцию надо обязательно замещать в производных классах, где она и наполнится разумным содержимым. Объявление ее в базовом классе носит формальный характер и требуется лишь для придания ей свойств виртуальности. Класс, в котором имеется

хотя бы одна виртуальная функция, называется абстрактным классом. Для него нельзя создать объект. Поэтому абстрактные классы применяются лишь в иерархических структурах, в которых содержательными являются только производные классы.

Задание 4

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <iostream.h>
class Base{
public:
void PrintMe( );
virtual void PrintMeV( );
};
void Base :: PrintMe( )
{cou t<< " BasePrintMe\n ";}
void Base :: PrintMeV( )
{cout << " BasePrintMeV\n ";}
class Derived:public Base{
public:
void PrintMe( );
void PrintMeV( );
};
void Derived :: PrintMe()
{cout << " DerivedPrintMe \n ";}
void Derived :: PrintMeV( )
{cou t<< " DerivedPrintMeV \n ";}
void main( )
{Base* Bp;
 Bp = new Base;
 Bp     PrintMe( );
 Bp     PrintMeV( );
 delete Bp;
 Bp = new Derived;
 Bp     PrintMe( );
 Bp     PrintMeV( );
 delete Bp;
 Derived Static;
```

```
Static.PrintMe( );
Static.PrintMeV( );
}
```

Правильный ответ:

```
Base 1
Now for the Derived
Derived  7
Base 7
```

Задание 5

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <iostream.h>
class F{
protected:
double a, b;
public:
F(double x, double y = 0){a = x; b = y;}
virtual void GetS(){cout << "???\n";}
};
class t : public F{
public:
void GetS( ){cou t<< "S = " << 0.5*a*b << "\n";}
};
class r : public F{
public:
void GetS(){cout << "S=" << a*b << "\n";}
};
class c : public F{
public:
void GetS(){cout << "S = " << 3.14*a*a << "\n";}
};
void main( )
{F f(0);
 F *p;
 t tt(3,3);
 r rr(4,4);
 circle c(5);
 p=&f;
```

```
p     GetS( );
p=&tt;
p     GetS( );
p=&rr;
p     GetS( );
p=&cc;
p     GetS();
}
```

Правильный ответ:

```
???
S=4.5
S=16
S=78.5
```

Темплейты

Идея шаблона состоит в создании класса, который оперирует некоторым неопределенным типом данных. Тип объекта определяется позже, когда создается экземпляр класса. Определим связный список. Для этого используем класс для целых данных:

```
class List{
public:                    // Для данных с плавающей точкой
int GetData( );            // float
void SetData(int val);     // float
List* GetNext( );
private:
int Data;                  // float
List* Next;
};
```

Версия шаблона для этого класса (вместо символа **T** может быть любой символ):

```
template <class T>
class List{
public:
T GetData( );
void SetData(T val);
List* GetNext( );
private:
T Data;
```

```
List* Next;
};
```
Создание объекта **InList** для целого типа будет выглядеть следующим образом:
```
List <int>  InList;
```
или динамически:
```
List <int>*  Fp;
Fp = new List <int>;
```
Создание объекта **FList** для типа float будет выглядеть следующим образом:
```
List <float>  FList;
```
Если внутри шаблона используется более одного родового типа, шаблон объявляется так:
```
template <class T1, class T2, ...>
```
Определение шаблона для функций этого класса:
```
template <class T>
T List<T> :: GetData(){return Data;}
template <class T>
void List<T> :: SetData(T val){Data = val;}
template <class T>
List<T>* List<T> :: GetNext(){return Next;}
```
Пример определения и использования функции, определенной вне класса:
```
template <class H>
H sqrT(H x){return x;}
void main()
{int i=10;
 float g=3.14;
 double f=1.71e+12;
 cout << "int" << sqrT(i) << "\n";
 cout << "float" << sqrT(g) << "\n";
 cout << "double" << sqrT(f) << "\n";
```
При объявлении шаблона класса может использоваться не только тип, но и другие параметры. При создании объекта эти параметры должны быть константами:
```
template <class R, int Size>
class Array{
...
};
```

```
void main()
Array<int, 5> a;
Array<float, 100> b;
```

Часто определяют новый тип, чтобы не использовать угловые скобки и всю атрибутику шаблонов:

```
typedef List<int>  InList;
```

а далее используют новые типы:

```
InList *IL, Fill;
```

Задание 6.

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <iostream.h>
template<class H>
class List{
public:
H GetData( );
void SetData(H val);
List* GetNext( );
void Add ();
private:
H Data;
List* Next;
};
template <class H>
H List <H> :: GetData(){
return Data;}
template <class H>
void List<H> :: SetData(H val){Data=val;}
template <class H>
List<H>* List<H> :: GetNext(){return Next;}
template<class H>
void List<H> :: Add(){
H Temp;
List* NewOne;
Temp = 2;
NewOne = new List<H>;
NewOne      Data = Temp;
```

```
NewOne     Next = 0;
Next = NewOne;
                        };
void main( )
{ List<int> intList;
  cout << "Integer List\n";
  intList.SetData(1);
  intList.Add();
  cout << intList.GetData() << "\n";
  cout << (intList.GetNext())     GetData() << "\n";
  List<float> floatList;
  Cout << "Floating List\n";
  floatList.SetData(1.43);
  floatList.Add();
  cout << floatList.GetData() << "\n";
  cout << (floatList.GetNext())     GetData() <<"\n";
  }
```

Правильный ответ: **Integer List 2 1 2**
Floating List 2 1.43 2

Дружественные функции. Указатель this

Дружественность в языке С++ - это способ разрешить функциям полный доступ к элементам класса с помощью описания **friend**.

Такими функциями могут быть как функции, не принадлежащие какому-либо классу (например **char* Setme(double)**), так и функции-члены другого класса В (**void Ff(int)**), которым разрешается полный доступ к элементам класса А, объявленным как private или protected, с помощью описания **friend** в определении класса А:

```
class A{
friend void B :: Ff(int);
friend char* Setme(double);
...
};
```

У функций-членов класса существует неявный параметр - указатель **this** (указатель на объект, который вызывает функцию). У функций, объявленных вне класса, указатель **this** отсутствует.

Указатель **this** используется как для возврата указателя на подразумеваемый объект (**return this**), так идля возврата самого объекта (**return *this**). Например:

```
class A{
int x,y;
public:
A(int xx, int yy){x = xx; y = yy;}
A Even( ){if(x%2) x++; return *this;}
};
```

Задание 7.

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <conio.h>
#include <iostream.h>
class A{
int x, y;
public:
A(int xx = 0, int yy = 0){x = xx; y = yy;}
A Even( );
void GetA( ){cout << "x = " << x << " y = " << y << "\n";}
        };
A A::Even()
{ if(x%2) x++;
  return *this;
}
void main()
{A a1(33, 55);
 a1.GetA();
 A a2=a1.Even();
 a1.GetA();
 a2.GetA();
}
```

Правильный ответ: **x=33 y=55**
x=34 y=55
x=34 y=55

Перегрузка операций

Большинству операций языка С++ может быть придано специальное значение относительно вновь определенных классов. Для встроенных типов данных значение операции изменить нельзя. Просто вводится новая операция относительно нового конкретного класса. Чтобы перегрузить операцию, надо определить, что эта операция значит относительно класса, к которому она будет применяться. Для этого создается специальная функция операции (operator function),которая определяет действие этой операции.

Основная форма функции-операции, являющейся членом класса:

тип имя_класса **:: operator** знак операции (список аргументов)
{
... //тело функции
}

Здесь **тип** – это тип возвращаемого значения. Обычно тип возвращаемого значения – класс, хотя возможен и другой тип.

Функция-операция должна быть или членом класса или дружественной функцией. Рассмотрим на примере программы, как реализуется перегрузка операций ” +” и ” =” относительно класса vector, который определяет трехмерный вектор:

```
#include <iostream.h>
class R{
float x, y;
public:
 R(float , float)
 R operator + (R);
 R operator = (R);
 void Get( );
};

 R R::operator + (R r)    // перегрузка операции  “+”
{ R mp;
        mp.x = x + r.x; mp.y = y + r.y;
        return mp;
}
 R R::operator = (R r)   // перегрузка операции “ =”
{ x = r.x;  y = r.y;
```

```
        return *this;                    // использование указателя this
}
void R::Get()
{cout << x << " " << y << "\n";}
R :: R(float m, float n)
{x = m; y = n;
void main( )
{ R a(3, 5), b(4,6) ,c;
a.Get( );
b.Get( );
c  =a + b;    // операции "+ "и "=", т. е.   c.operator=(a.operator+(b))
c.Get();
}
```

В этом примере выражение **a+b** означает вызов из объекта **a** функции-оператора **operator+(b)**, аргументом которой является объект **b**. Полученный результат в свою очередь является аргументом функции-оператора **operator=(a.operator+(b))**, которая вызывается из объекта **c**.

Во всех случаях, когда используются функции-операторы как члены классов, для реализации бинарных операций требуется один параметр, а для унарных операций параметры не нужны.

Задание 8

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <iostream.h>
class W{
int x, y, z;
public:
W(int, int, int);
 W operator+(W s);
 W operator = (W);
 int operator * (W);
 void Show( );
 };
 W W :: operator+(W s)
 { W tmp(0,0,0);
```

```
        tmp.x = x + s.x; tmp.y = y + s.y; tmp.z = z + s.z;
        return tmp;
}
W W :: operator=(W s)
{ x = s.x; y = s.y; z = s.z;
        return *this;
}
void W :: Show( )
{cout << x << " " << y << " " << z << "\n";}
W :: W(int m=0, int n=0, int k=0)
{x = m; y = n; z = k;}
int W:: operator * (W s)
{int i =  x*s.x+ y*s.y+ z*s.z;
return i;}
void main()
{ W a(3,5,7), b(4, 6, 8), c;
int j;
c = a + b;
c.Show( );
c = a + b + c;
c.Show( );
c = b = a;
c.Show( );
j=a*c;
cout<<j;
}
```

Правильный ответ:

```
7  11  15
14  22  30
3  5  7
 83
```

Работа с файлами в языке C++

Чтобы работать с файлами в программе на языке C++, необходимо подключить стандартную библиотеку **<fstream.h>**. Класс **fstream** является потомком классов **istream** и **ostream**. Эти же классы являются родителями классов **ifstream** и **ofstream**. Класс

fstream используется для ввода-вывода в один и тот же файл. Классы **ifstream** и **ofstream** – соответственно для ввода (чтения) файла и вывода (записи в файл). Экземплярами классов **istream** и **ostream** являются **cin**, **cout**, **cerr**, с помощью которых реализуется ввод со стандартного вводного устройства (клавиатура) и вывод на стандартное выводное устройство (экран). Файловый ввод-вывод выполняется с помощью операций записи в файл << и чтения файлов >>, переопределенных в поточных классах. Для работы с файлом, его сначала следует открыть, т.е. связать со стандартной структурой, в которой заданы характеристики файла. Эта связь реализуется функцией open(), входящей в класс, который определяет ввод-.вывод (**fstream, ifstream, ofstream**). Чтобы получить доступ к этой функции, сначала необходимо создать объект соответствующего класса. Например, чтобы выполнить вывод в файл нужно создать объект класса **ofstream** (например: **ofstream Sub**;), после чего выполнить **Sub.open( ).** Соответственно для чтения файла нужно создать объект класса **ifstream** (например: **ifstream Bart**), после чего выполнить **Bart.open ().** В скобках необходимо указать имя открываемого файла. Если файл для записи заранее не существует, его можно создать и открыть двумя способами – либо при объявлении объекта класса **ofstream** (например **ofstream Sub("new.txt");**), либо с использованием функции open() (например **Sub.open("new.txt");).** После того как файл открыт для чтения или записи, используют операции записи в файл << и чтения файлов >>. Для того, чтобы записать в наш файл (**new.txt**) строку текста и две переменные **int k** и **float f** , можно записать **Sub << "Hi, friends!" << k << f << endl;** , где **endl** – конец вывода и переход на новую строку. Когда работа с файлом закончена, файл следует закрыть, чтобы разорвать связь со структурой, с которой файл связывался при его открытии. Это выполняется с помощью функции **close( )** того же объекта, который создавался, чтобы выполнить функцию **open( )** (в нашем примере **Sub.close( )).**

Кроме функций **open( )** и **close( )** классы **ifstream** и **ofstream** обладают рядом других методов.

В классе **ofstream,** предназначенном для записи в файл, наиболее широко используемыми методами являются:

is_open( ) -- возвращает TRUE, если файл открыт, и FALSE в противном случае;

put( ) – записывает в файл один символ;

write( ) -- записывает в файл заданное число символов.

В классе **ifstream,** предназначенном для чтения из файла, наиболее широко используемыми методами являются:

is_open( ) -- возвращает TRUE, если файл открыт, и FALSE в противном случае;

get( ) – читает из файла один символ;

read( ) -- читает из файла заданное число символов.

eof( ) – возвращает ненулевое значение, когда указатель позиционирования в файле достигает конца файла.

Следующий пример иллюстрирует работу с классами **ifstream** и **ofstream.** В данном примере создаются два файла - один (**new.txt**) для записи текстовой информации, другой - (**numbers.txt**) для записи числовой информации. После чего, производится запись в файлы соответствующих данных, и файлы закрываются. Далее файлы открываются для чтения, считывается информация (чтение текстовой информации производится по словам), эта информация выводится на экран, и файлы закрываются.

```
#include <iostream.h>
#include <fstream.h>
void main( )
{ofstream Outfile("new.txt");
 if(!Outfile) cout << "Can't open file. \n";
 Outfile << "Hello friends!";
 Outfile.close( );
 ofstream file;
 file.open("numbers.txt");//Outfile.open("numbers.txt");
 file << 15 << " "<< 42 <<" "<<1;//Outfile<<15<<" "<<42<<" "<<1;
 file.close( );//Outfile.close( );
 ifstream Infile;
 Infile.open("new.txt");
 char p[50];
 Infile >> p;
 cout << p <<"\n";
 Infile >> p;
 cout << p <<"\n";
 Infile.close( );
 int Num;
 Infile.open("numbers.txt");
 while(!Infile.eof( ))
```

```
{Infile >> Num; if(!Infile.eof( ))cout << Num << "\n";};
Infile.close( );
Infile.open("new.txt");
while(!Infile.eof( )){Infile >> p; cout << p << "\n";};
}
```

Управление исключениями

Под управлениями исключениями понимается механизм для обнаружения и обработки необычных, непредвиденных (исключительных) ситуаций или событий. Когда программа встречает ненормальную ситуацию, на которую она не была рассчитана, можно передать управление другой части программы, способной справится с этой проблемой, и либо продолжить выполнение, либо завершить программу. Выброс исключения (**throwing**) позволяет собрать в точке выброса информацию, которая может оказаться полезной для диагностики причин, приведших к нарушению нормального хода выполнения программы. Можно задать обработчик исключений, выполняющий необходимые действия перед завершением программы. После того, как исключение зафиксировано, обрабатывающая процедура не может потребовать, чтобы исполнение было продолжено. Исключения C++ не поддерживают обслуживание асинхронных событий, таких как ошибки оборудования, или обработку прерываний. Обслуживаются только те исключения, которые инициализированы некоторой функцией. При управлении исключениями применяются три ключевых слова **try**, **catch**, **throw**. Блок кода, который может генерировать исключение, начинается ключевым словом **try** и заключается в фигурные скобки. Например:

```
try
 { cout<<"в try блоке" ;
…….
function( );          // function() может генерировать исключение
…….
 }
```

Если **try** - блок обнаруживает исключение внутри себя, происходит программное прерывание и выполняется следующая последовательность действий:

1. Ищется подходящий обработчик исключения.
2. Если обработчик найден, стек очищается и управление передается обработчику исключений.
3. Если обработчик не найден, вызывается функция **terminate()** для завершения программы.

Блок обработки исключения начинается ключевым словом **catch**. За ключевым словом следует описание исключения, заключенное в скобки и состоящее из имени типа и необязательной переменной. Имя типа определяет тип исключений, которые данный код обработчика может обслуживать. Описание исключения можно рассматривать как аналог описания параметра функции. Блок кода обработки исключения заключается в фигурные скобки. За **try** – блоком могут следовать несколько операторов **catch** со своими кодовыми блоками обработчиков исключений. Для каждого исключения, которое может сгенерировать программа, должен быть предусмотрен свой обработчик. Обработчики исключений просматриваются по порядку следования за **try** – блоком и выбирается тот обработчик, тип которого соответствует типу аргумента в операторе **catch**.

Например:

```
void Func( );
void main( )
{ try
  { cout<<"в try блоке" ;
  ……….
     Func ();          // Func () может генерировать исключение
  ……….
   }
 catch(int i)         //обрабатывает исключения типа int
    {
      ……          // код обработчика int
     }
 catch(char *)        //обрабатывает исключения типа char *
    {
      ……          // код обработчика char *
     }
```

```
catch(…)          //обрабатывает исключения любого типа
   {
    ……          // код обработчика …
   }
}
```

Оператор **catch** с многоточием **catch(…)** перехватывает исключения любого типа и должен быть последним из операторов **catch**, следующих за **try –** блоком.

Как упоминалось выше, какая-либо функция в **try –** блоке может генерировать (выбрасывать) исключение, используя служебное слово **throw**, после чего происходит передача управления к соответствующему обработчику. Выражение, следующее за **throw**, сводится к значению переменной определенного типа. Можно рассматривать операнд **throw** как аргумент вызова функции. Тип операнда определяет, который из обработчиков может перехватить исключение. Местоположение оператора **throw** обычно называют точкой выброса. Например:

```
void Func1( )        //выбрасывает исключение типа char*
{ if ( что-то не так )
  throw  "Обнаружена ошибка" ;
}
void Func2( )        //выбрасывает исключение типа NewClass
{ if ( что-то не так )
  {  NewClass Object( "Ой! …");
  throw  Object;
}
```

Можно выбрасывать как встроенные, так и определенные пользователем типы. Таким образом, можно передавать обработчику сложные, нагруженные информацией объекты.

Когда выполняется оператор **throw** , функции исполнительной библиотеки С++ производят следующие действия:

1. Создают копию выброшенного объекта (переменной).
2. Вызывают деструкторы созданных локальных объектов.
3. Передают управление ближайшему обработчику **catch**, принимающему параметр, совместимый по типу с выброшенным объектом. Копия объекта передается обработчику в качестве параметра.

После того, как исключение выброшено, процедуры исполнительной библиотеки C++ ищут подходящий обработчик. Обработчик считается найденным, если:

1. Тип выброшенного объекта тот же самый, что и тип, ожидаемый обработчиком. То есть, если выбрасывается тип **H**, ему будет соответствовать обработчик, который перехватывает **H**, **const H**, **H&** или **const H&.**
2. Тип обработчика является базовым классом выброшенного объекта.
3. Обработчик ожидает указатель, и выброшенный объект является указателем, который может быть преобразован к типу обработчика по стандартным правилам.

Очень важна последовательность, в которой располагаются обработчики исключений. Обработчик, ожидающий исключений базового класса, автоматически скрывает обработчик производного класса. А обработчик для пустого указателя (**void***) будет автоматически скрывать обработчик для указателя любого типа.

Если для выброшенного исключения не найдено подходящего обработчика, вызывается функция **terminate( )**. Она вызывает функцию **abort( )**, аварийно завершающую текущий процесс.

Спецификация исключений

Список исключений, которые функция может выбрасывать, присоединяется к заголовку функции. Спецификация имеет следующий формат:

throw(тип, тип,…**).**

Спецификация без типа предполагает, что функция не должна выбрасывать никаких исключений. Функции без спецификации, напротив, могут выбрасывать исключения любого типа. Например:

```
class X {
public:
int i;
…….
 …… };
void funcA( ) throw(int)
{……..                  // функция должна выбрасывать только
 ……..                  //  исключения типа int
```

```
}
void funcB( ) throw(char*, X*)
{……..                    // функция должна выбрасывать только
 ……..                     //  пременные типа char*, указатели на X или
}                          //  на типы,  производные от X
void funcC( ) throw( )
{……..                   // функция не должна выбрасывать исключения
 ……..
}
```

Спецификация ни к чему не обязывает. То есть функция может прямо или косвенно выбрасывать исключение, которое она обещала не использовать. К примеру, нижеприведенный код при компиляции не вызовет ошибки и даже предупреждения:

```
void func( ) throw(int)
{……..
  throw " Ой! …. " ;
 ……..
}
```

Нарушение списка допустимых значений обнаруживается только во время исполнения программы. Непредвиденные исключения приводят к вызову функции **unexpected( )**. По умолчанию **unexpected( )** реализует вызов функции **terminate( )**.

Однако с помощью функции **set_unexpected( )** можно определить свою собственную процедуру, которая будет выбрасываться, если функция выбрасывает не специфированное исключение.

Конструкторы и исключения

Рассмотрим случай, когда исключение выбрасывается конструктором класса. При возникновении исключения деструкторы вызываются только для полностью сконструированных локальных объектов. То есть если исключение происходит в конструкторе объекта, то соответствующие деструкторы будут вызваны только для уже сконструированных объектов данных и базовых классов. В приведенном примере вызываются только деструкторы классов **T** и **Base**:

```
# include < iostream.h >
# include < except.h >
class T {
            public:
            T( ) {cout << ” T :: T( )\n”;}
           ~ T( ) {cout <<” T :: ~T( )\n”;}
           };
class Base {
                public:
                Base( ) {cout << “ Base :: Base( )\n”;}
                ~Base( ) {cout << “ Base :: ~Base( )\n”;}
               };
class D : public Base{
                          T data;
                           public:
                           D( ) {cout <<” D :: D( )\n”;
                                 cout <<” Выбрасываем исключение \n”;
                                  throw “ Ой! Что-то случилось”;
                                 }
                                ~D( ) {cout << “ D :: ~D( )\n”;}
                                };
void main( )
{ try{
         T t;
         //……
        }
 catch( const char* msg)
      {cout << “Пойманное исключение:” << msg << “\n”;}
}
```

Вывод программы:

```
Base :: Base( )
T :: T( )
D :: D( )
```

Выбрасываем исключение
T :: ~T( )
Base :: ~Base( )
Пойманное исключение: Ой! Что-то случилось

Задание 9

Какая информация появится на экране дисплея после выполнения данной программы:

```
#include <fstream.h>
class Tale{
        public:
        Tale ( ){cout << " Tale: В некотором царстве... \n ";}
        ~Tale ( ){cout << " Tale: ...а кто слушал – молодец! \n ";}
             };
void F1( )
{ ifstream ifs(" invalid_file_name ");
  if(!ifs)
   {cout << " Выбрасываем исключение \n ";
     throw " Ошибка при открытии файла \n";
    };
}
void F2( )
{Tale tellme;
          Func1();
}
void main()
{try {cout << " Входим в try-блок \n ";
 Func2();
 cout << " Выходим из try-блока \n ";
}
 catch(int i)
 { cout << " Вызван обработчик 'int' c " << I << " \n ";}
 catch(const char*p)
 { cout << " Вызван обработчик 'char*' [" << p << ']' << "\n";}
 catch(...)
 { cout << " Вызван обработчик catch-all \n";}
 }
```

Правильный ответ:

Входим в try-блок
Tale: В некотором царстве...
Выбрасываем исключение
Tale: ...а кто слушал – молодец!
Вызван обработчик 'char*' [Ошибка при открытии файла]

Литература

1. 004 И83 Пахомов Б.И.
   C/C++ и MS Visual C++ 2010, СПб.: БХВ – Петербург, 2011.
2. 004 И97 C++ . Начала программирования : , Москва:
   Бином, 2011.
3. 004 П12 C# . Программирование на языке высокого уровня : ,
   Москва [и др.]: Питер, 2014.
4. 004 К53 Искусство программирования Т.1 Основные
   алгоритмы, Москва [и др.]: Вильямс, 2011.