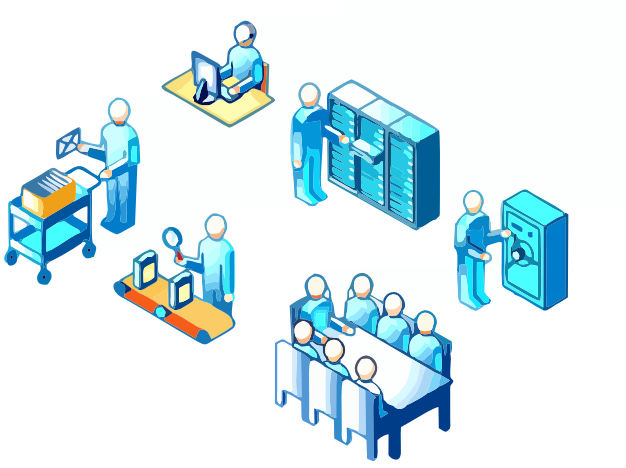
УНИВЕРСИТЕТ ИТМО

И.С. Лебедев, В.Ю. Петров

ИНФОРМАТИКА. ПРОГРАММИРОВАНИЕ

Часть 2



Санкт-Петербург

2017

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

И.С. Лебедев, В.Ю. Петров

ИНФОРМАТИКА. ПРОГРАММИРОВАНИЕ

Часть 2

УНИВЕРСИТЕТ ИТМО

Санкт-Петербург

2017

Лебедев Илья Сергеевич, Петров Вадим Юрьевич. Информатика. Программирование. Учебно-методическое пособие. Часть 2. – СПб: Университет ИТМО, 2017. – 71с.

Учебное пособие представляет собой дальнейшее продолжение курса по программированию на универсальном языке программирования C++, начало которого изложено в первой части пособия. В данной работе уделено внимание компилятору VC++. Подробно рассмотрены такие теоретические аспекты как: свойства языков объектно-ориентированного программирования, работа с объединениями, структурами, классами и др. Изложение материала сопровождается большим количеством примеров.

Пособие предназначено для студентов, обучающихся по направлению подготовки 38.03.01. - «Экономика» по общеобразовательным программам «Экономика предприятий и организаций», «Макроэкономическое планирование и прогнозирование»; и направлению подготовки 38.03.02. «Менеджмент» по общеобразовательным программам «Производственный менеджмент», «Управление проектом», «Финансовый менеджмент».

Рекомендовано к печати на заседании ученого совета факультета технологического менеджмента и инноваций, протокол № 10 от 31.10.16.

УНИВЕРСИТЕТ ИТМО

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

ОГЛАВЛЕНИЕ

1. ФУНКЦИИ
СТРУКТУРА ПРОГРАММЫ НА С++

Создавая сложные программы, важно заботиться о том, чтобы с ними было удобно работать. Одним из основных способов обеспечения этого требования является разделение программы на отдельные самостоятельные части. Функция обеспечивает удобный способ отдельно оформить некоторое вычисление и пользоваться им далее, не заботясь о том, как оно реализовано.

Рассмотрим некоторые полезные утверждения и определения.

А). *Функция*– это логически самостоятельная именованная часть программы, в которую может передаваться любое количество значений переменных аргументов, а возвращаемое функцией значение – только одно.

Б). Вместе с тем, есть функции, в которые не передаются аргументы, и есть функции, которые не возвращают значений.

В). *Значение функции* – величина переменная и, как всякая переменная, она должна быть объявлена.

Г). Объявление функции производится до ее использования и вне тела любой другой функции. При объявлении функции указывается:
тип_результатаимя_функции (список типов параметров);
Естественно, что имя функции задает программист. Желательно, чтобы из имени функции было ясно ее назначение. Такую запись называют *прототипом функции.*

Примеры

- ✓ *voidfun1 (floatb, charch);* - объявлена функция *fun1* с двумя параметрами. Функция не возвращает значения.
- ✓ *floatfun2(void);* - функция без параметров, возвращающая вещественное значение.

Д). Объявляются все функции программы, кроме main( ).
Таким образом, если мы создаем в программе функцию, она должна быть объявлена до главной функции *main().* Но это не все. Необходимо указать, что же эта функция будет выполнять, то есть необходимо определить функцию.

Е). *Определение функции* – это описание операций, которые выполняются в теле этой функции. Оно имеет вид:

Ж). Если функция должна возвращать значение, но *return* отсутствует, то она выдает “мусор”.

З). Функцию можно определять либо непосредственно при объявлении, либо после окончания главной функции *main().* Т.е. структура программы может иметь вид 1-А и 1-Б, представленный на рис.1.1. Функция *intsq (inta)* возвращает квадрат целого числа.

И). Если определение функции не соответствует объявлению - это ошибка.

Определив функцию, мы указали какие операции эта функция должна выполнять и какие переменные использовать. Осталось указать, в каком именно месте программы эта функция должна использоваться, т.е. где необходимо передать этой функции управление программой. Этот процесс называют *вызовом функции.*

К). Любая программа на *С* начинает выполняться с главной функции *main( ).* Обычно именно из нее вызываются другие функции.

Но в общем случае любая функция может быть вызвана из любой. Одна из функций становится *вызывающей* и временно передает управление *вызываемой* функции, которая, выполнив определенные операции, возвращает управление вызывающей функции.

При вызове функции указывается ее имя и в скобках – список аргументов.

1-А

```
#include<iostream.h>
int sq (int a);          // объявление функции
        void main ( ) //начало описания гл. функции
        {
        ….
        z = sq(x); //Вызов функции
        }
              int sq (int a)  //определение функции
              {
              int b;
              b = a*a;
              return b;       // оператор возврата управления
               }
```

1-Б.

```
#include<iostream.h>
      int sq (int a)          // объявление и определение функции
      {
            Int b;
            b = a*a;
            return b;       // оператор возврата управления
      }
void main ( )                 //начало описания гл. функции
{
……..
z = sq(x);                    //Вызов функции
…….. }
```

Рис.1.1 -Объявления и определения функций в программах

«Вызов функции» часто называют *«обращением к функции»;* это равнозначные термины. Обращение к функции следует рассматривать как выражение. Оно может использоваться всюду, где допускаются выражения.

Обратите внимание, что при объявлении и определении функции *square* в качестве аргумент, а мы указали целочисленную переменную *a*. А в программе при вызове функции в нее фактически передали тоже целочисленную переменную, но с именем *x*.

То есть необходимо, чтобы при передаче аргументов обязательно учитывались типы передаваемых и возвращаемых переменных.

Л). Переменную, указанную при объявлении функции, называют *формальным параметром***,** а переменную, передаваемую непосредственно при вызове функции – *фактическим параметром.*

В нашем примере *int a* – формальный параметр, *int x* – фактический параметр.

Задание 1.1. Написать программу для определения квадрата числа $z = x^2$. Возведение числа в квадрат оформить в виде отдельной функции. Доказать ее работоспособность.

2. ВНЕШНИЕ ПЕРЕМЕННЫЕ И ОБЛАСТИ ВИДИМОСТИ ПЕРЕМЕННЫХ

Переменные, которые объявлены внутри функции, называются *внутренними переменными или локальными.* Эти переменные могут использоваться только в тех функциях, где они объявлены, и никакие другие функции доступа к ним не имеют.

Каждая локальная переменная функции возникает только в момент обращения к этой функции и исчезает после выхода из нее. Такие переменные называются *автоматическими***.** Они образуются и исчезают одновременно с входом в функцию и выходом из нее; они не сохраняют своих значений от вызова к вызову.

Т.е., обратиться к переменной *b* в теле главной функции нельзя, ее там не «видно». После выхода из функции *square* ячейка оперативной памяти, которая была выделена для этой переменной, освобождается и при следующем вызове под переменную *b* будет отведена любая другая ячейка.

Т. о., локальные переменные по умолчанию относятся к классу памяти «автоматические» (auto).

Чтобы локальная переменная могла сохранять свои значения от вызова к вызову функции, необходимо переменную отнести к классу памяти *«static»* ***(****статические****).*** *Такие переменные не исчезают при выходе из функции.*

Рассмотрим пример. На рис.2.1 представлена программа и результат ее выполнения, который располагается на темных прямоугольниках. Верхний - для статического *k*, нижний -для автоматической переменной.

Статическая переменная *k* инициализируется один раз при первом вызове функции *qqq( )*.

В *"k"* записывается число *2*, которое посредством операции *k+x* увеличивается на *1* и передается в переменную *j*, которая выводится на экран. При втором вызове функции *qqq( )*в нее передается *x=2*, которое добавляется к сохраненному *k=3***.** Функция вернет *5*.

Чтобы переменная не только сохраняла свои значения в течение работы программы, но и ею могли пользоваться другие функции, она должна быть не внутренней, а внешней переменной.

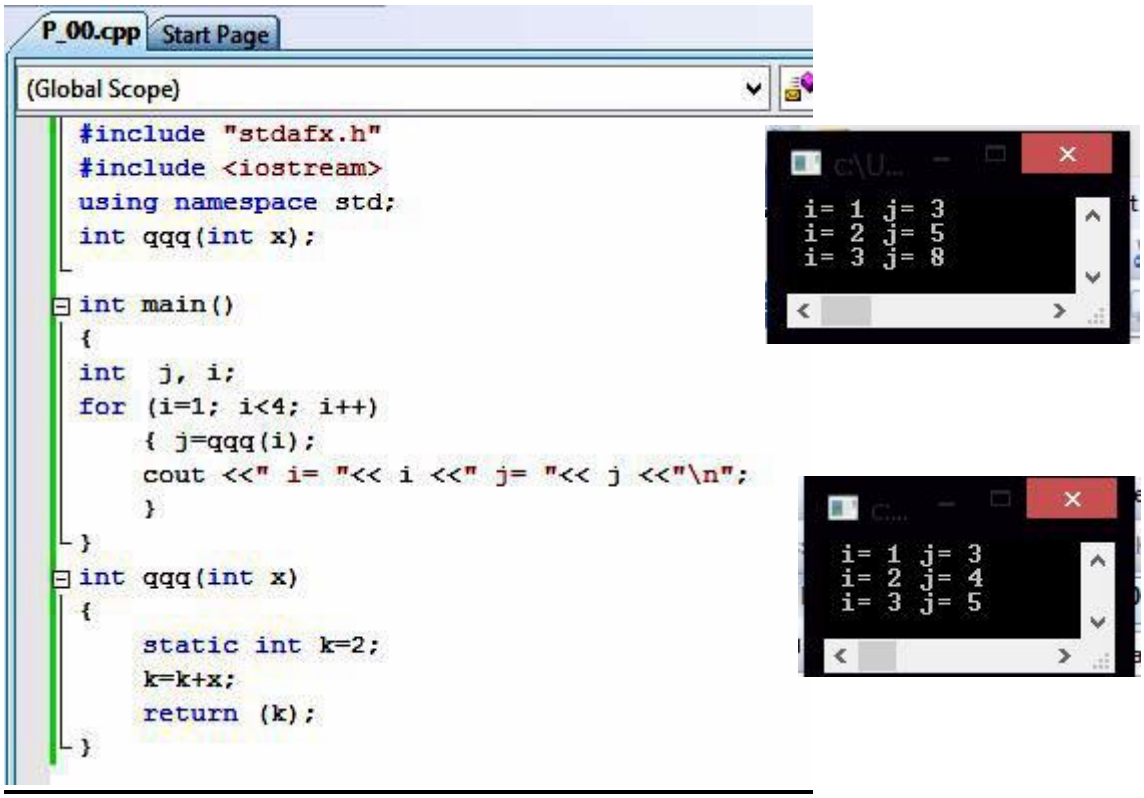


Рис.2.1 - Использование статической переменной

Внешние переменные доступны повсеместно. Их можно использовать в любом месте программы в любой функции. Кроме того, поскольку внешние переменные существуют постоянно, а не возникают и не исчезают на период выполнения функции, свои значения они сохраняют и после возврата из функций, их установивших.

Внешняя переменная должна быть объявлена *вне текста любой функции*и только один раз. В этом случае ей будет выделена память. Обычно внешние переменные объявляются в начале программы после директив препроцессору и перед главной функцией *main( ).*

Поскольку внешние переменные доступны всюду, их можно использовать в качестве связующих данных между функциями как альтернативу связей через аргументы и возвращаемыми значениями функций.

Пример. Вычисление квадрата с использованием функции, производящей возведение числа в квадрат. Листинг представлен на рис.2.2.

Связь между функциями осуществляется через внешние переменные. Поэтому в *main( )* присваивается значение переменной *i* и вызывается функция *kvadrat ( ),* которая вычисляет i^2 и присваивает его внешней переменной *j* . Поскольку *main( )* имеет свободный доступ к переменной *j* , можно написать функцию *kvadrat ( )* так, чтобы она не возвращала никакого значения.

Внешние переменные в определенных случаях удобны и эффективны, но пользоваться ими необходимо в меру. Многократное, бездумное использование внешних переменных ухудшает структуру программы и приводит к слишком большому числу связей между функциями, которые не всегда предсказуемы.

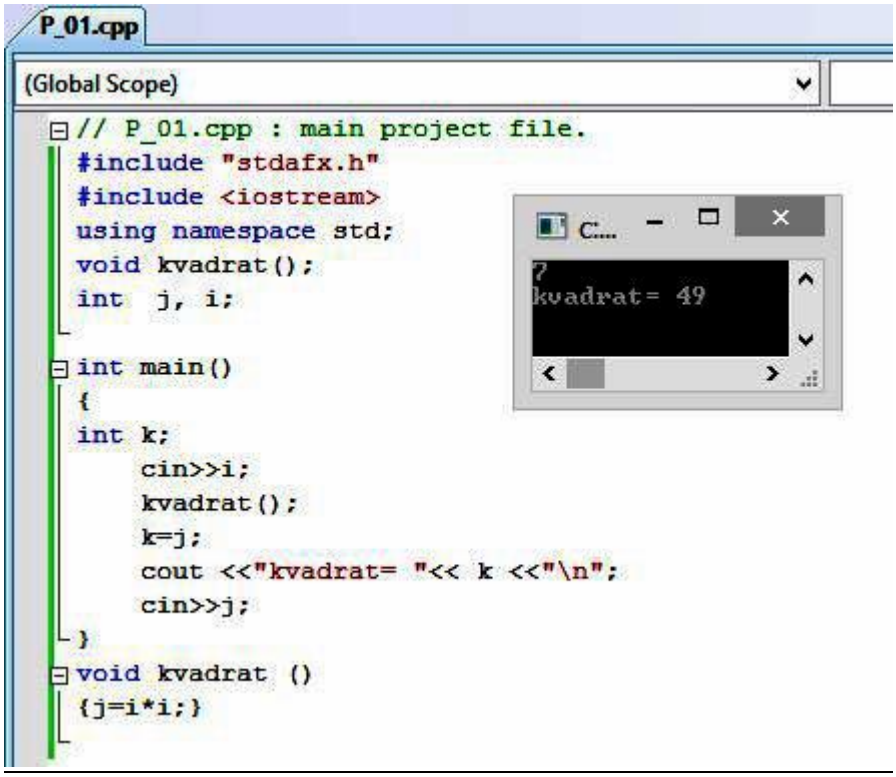


Рис. 2.2 - Листинг программы с внешними переменными

Внешние переменные могут задаваться и не в начале программы. Если переменная используется в функции раньше, чем была объявлена, то она должна быть заявлена в этой функции со словом extern. Пример для этого варианта приведен на рис. 2.3 ниже.

Важно заметить,

1. *объявление* внешней переменной приводит к выделению для нее памяти;

2. *заявление* указывает тип переменной, которая будет использоваться в функции.

3. *область видимости переменной (функции)* – это часть программы, в которой этой переменной можно пользоваться. Для локальных переменных – это функция, где они объявлены. *Область действия внешней* переменной простирается от точки программы, где она объявлена, до конца файла.

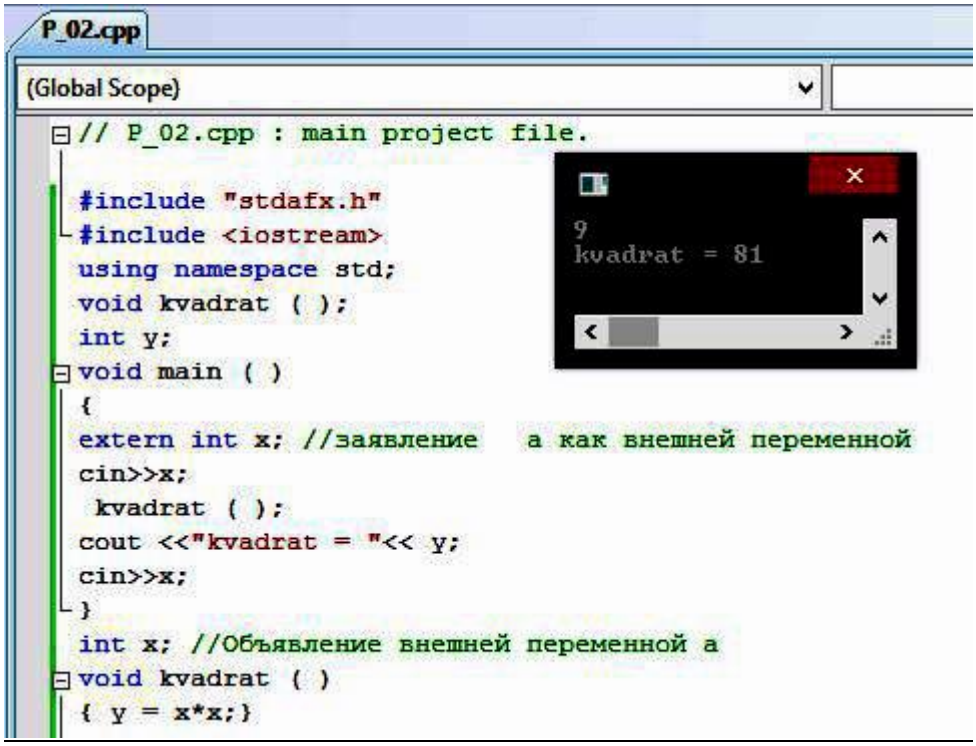


Рис.2.3-Листинг программы с внешними переменными

Задание 2.1

1. Создайте программу. Главная функция этой программы должна вызывать одну из приведенных функций. В программе использовать внешние переменные.

а) y1=| sin(a*x)/(a*x)|

б) $y2=x^3$

в) y3=a*Sin(a*x)*exp(-x)

г) y4=Tg(ax)* exp(-x)

2. Создайте программу, которая должна:

- записывать в файл 3 числа. Результат записи контролируйте с помощью *«Блокнота».*

- добавить в файл еще 3 числа.

- прочитать эти 6 чисел и вывести их на экран в виде матрицы 3x2

3. Создайте программу, которая позволяла вводить 10 символов с клавиатуры.

Функция mainдолжна обращаться к дополнительно созданной функции. Эта дополнительная функция с помощью статической переменной должна считать сколько раз при вводе символов нажали клавишу «к».

3. ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ И МАССИВОВ
3.1. УКАЗАТЕЛИ

В любой программе используется понятие «переменная». Что такое переменная? *Переменная – это некоторая область памяти компьютера, занимающая один, два или более байтов, которой присвоено имя (иначе называемое идентификатором).*

Так, например, если в программе объявлена переменная:

char Alfa=’s’**;**

то под нее будет выделена область памяти, которая может иметь адрес *0065FDE7*. Это случай изображен на рис. 3.1.

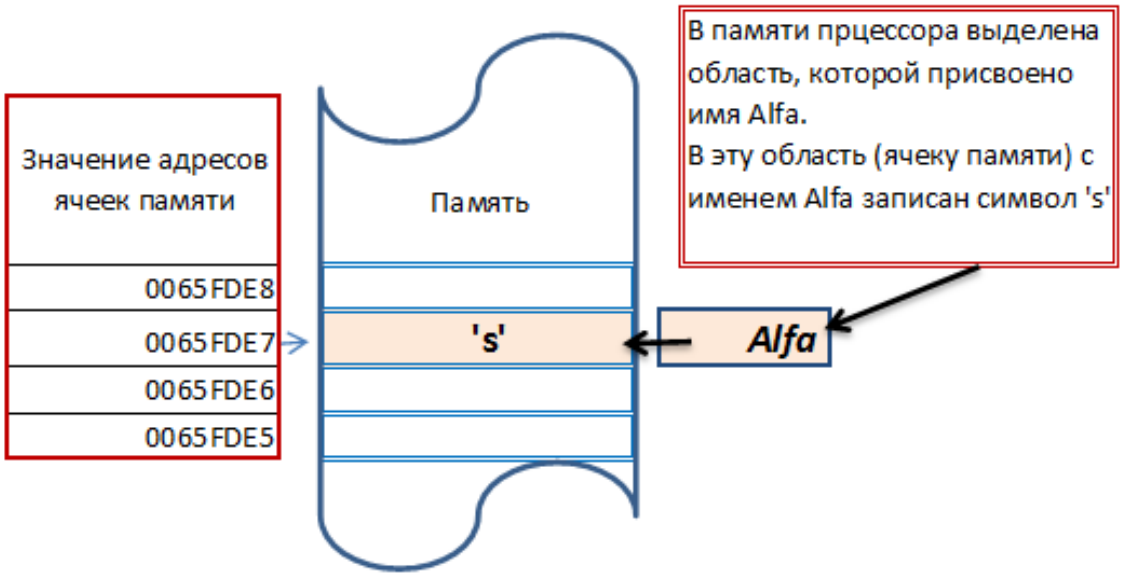


Рис. 3.1. –Расположение переменных в памяти ПК

Как известно, обращение к переменной в программе производится по ее имени. Например, если переменной присвоить другое значение (символ ‘f’):

Alfa=’f’; //В ячейку памяти по имени Alfa записано символ ‘f’

……

Именно так мы использовали переменные в программах.

*Указатель***,** как и обычная переменная, имеет имя (идентификатор), задаваемое программистом. Но, в отличие от просто переменной, мы не можем записать в переменную-указатель число или символ (компилятор выдаст ошибку). *В переменную-указатель можно записать только адрес какого-либо программного объекта (переменной, константы, функции и т.д.)***.** Это

обусловлено тем, что указатель имеет особый формат хранения информации, отличный от формата обычной переменной.

Указатель – это переменная, содержащая адрес другой переменой.

Для объявления указателя используется знак ***.**
Формат объявления указателя имеет вид:

*<тип указателя> * <идентификатор указателя>;*

Наличие или отсутствие пробелов между знаком ***** и идентификатором указателя и типом указателя значения не имеет.

Тип указателя — это тип того, что хранится в памяти по адресу на который указывает указатель.

Примеры объявления указателей:

Пример 3.1.

```
//
int * pBeta;          //Объявлен указатель pBeta на нечто типа int
float * pDelta;       //Объявлен указатель pDelta на нечто типа float
```

Пример 3.2.

```
//
intAlfa, y=1, *pAlfa;       // объявлены 2 переменные
                            // и указатель на int
Alfa = 5;
pAlfa = &Alfa;      //теперь указатель pAlfaуказывает на Alfa.
                    //Он содержит адрес переменной Alfa
y=*pAlfa            //теперь y=5
```

Резюмируем результаты примера.

Так как указатель – это переменная, которая создается для хранения адреса программного объекта, то можно получить этот адрес.

Для получения значения адреса какого-либо программного объекта, используется операция взятия адреса - &.

Выражение *&<идентификатор объекта>* возвращает адрес, по которому расположен программный объект в оперативной памяти. Инициализировать указатель (т.е. присвоить ему адрес какого-либо программного объекта) можно сразу, при его объявлении или в теле программы. При этом тип инициализирующего объекта должен быть тем же, что и тип указателя.

Рассмотрим, что происходит с оперативной памятью в этом случае.

На рис. 3.2 покажем указатель *pAlfa*, содержащий адрес переменной *Alfa* и сама переменная *Alfa*. Говорят, что указатель *pAlfa* указывает или ссылается на переменную *Alfa*.

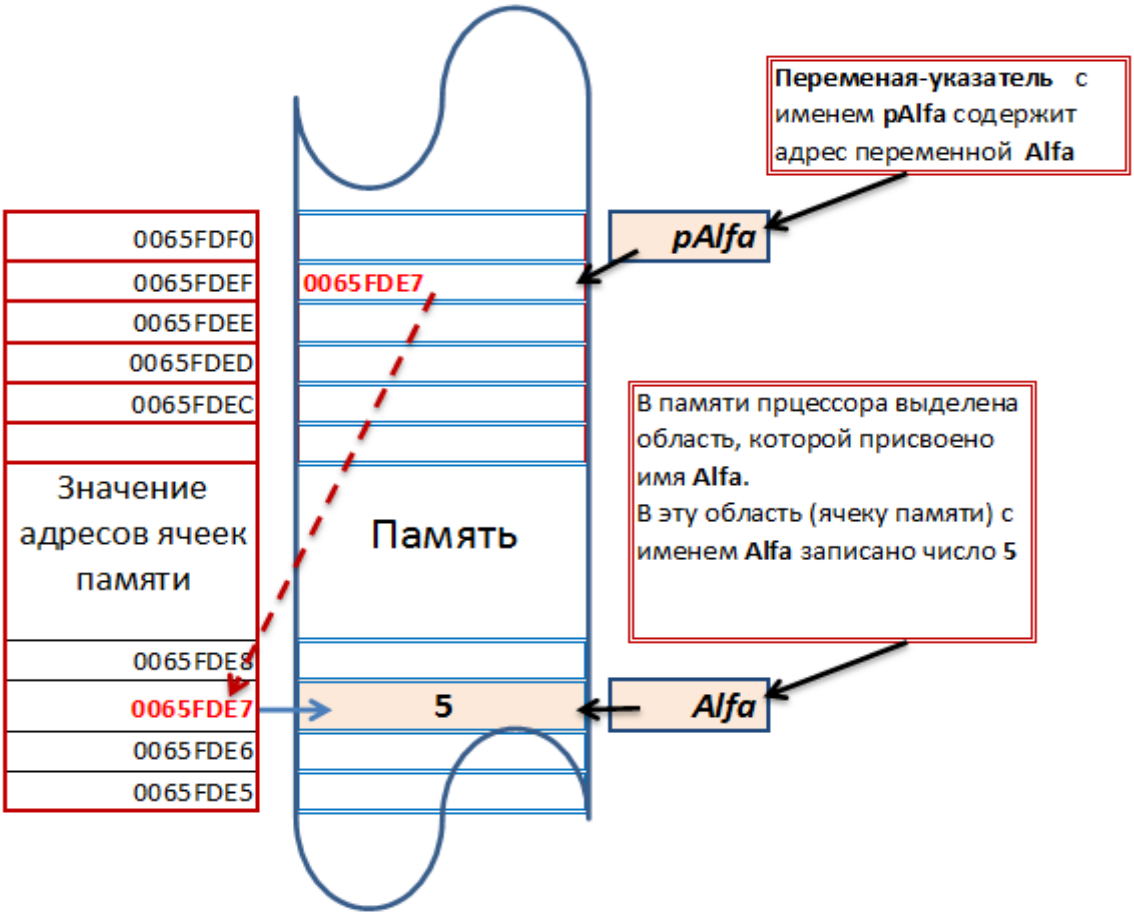


Рисунок 3.2 -Расположение переменных и указателей в памяти ПК

Тип указателя - это тип объекта, на который указывает указатель.

Тип указателя может быть одним из стандартных типов языка C++ или произвольного типа, объявленного программистом.

Указание типа необходимо для того, чтобы компилятору был известен объем памяти, занимаемый объектом, на который ссылается указатель. Эта информация позволит программе, при обращении к объекту, выбирать из памяти точно то количество байтов, которое занимает объект.

Так как указатель – это адрес некоторого объекта, то через него можно обращаться к этому объекту.

Пример 3.3. Листинг с примером объявления и использования указателей приведен на рис. 3.3. Здесь же представлен результат выполнения программы. Около каждого оператора проставлены комментарии.

```
// VU_01.cpp : mainprojectfile.

#include "stdafx.h"
#include<iostream>              // Дляоператорacout
using namespace std;            // Дляоператора cout
using namespace System;         // Дляоператора Console

int main()
{
        int Alfa=15, Beta=5, Gamma;  // Объявлены перем-е Alfa, Beta и Gamma
                                     // Alfa и Beta инициализированы
        int *pAlfa=&Alfa;            // Объявлен указатель pAlfa типа int
                                     // и инициализирован адресом переменной
Alfa
        int *pBeta;          // Объявлен указатель pBeta типа int
        pBeta=&Beta;         // Указателю pBeta присвоен адрес переменной Beta
        Gamma=*pAlfa + *pBeta;       /* В переменную Gamma записывается
                                     сумма содержимого переменных Alfa и Beta,
                                     извлекаемого с помощью указателей pAlfa и
                                     pBeta. Т.е. Gamma =20 */
        *pAlfa=Gamma;                /* По адресу, на который указывает указатель
                                     pAlfa, т.е. переменной Alfa, присваивается
                                     значение переменной Gamma */
        // Вывод информации
        cout <<"Gamma= "<<Gamma<<endl;
        cout<<"*pAlfa= "<<*pAlfa<<endl;
        cout<<"Alfa="<<Alfa<<endl;
        Console::WriteLine(L"Hello World");
        return 0;
}
```

Результат выполнения программы

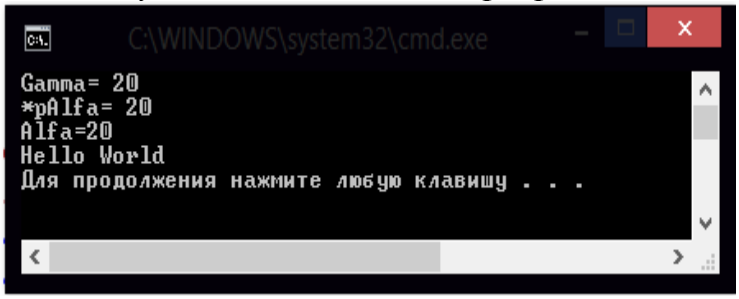


Рис. 3.3 – Объявление и использование указателей

Пример 3.4. Использование указателей как аргументов функций.

Весьма полезный и наглядный пример в этом плане представлен в учебнике Керригана и Ритчи [3].

Используя функции с аргументами, поменяем местами значения у двух целых переменных, **a** и **b**. Значение переменной **a** нужно присвоить переменной **b,** а значение переменной **b** соответственно переменной **a.** Причем попытаемся сделать это обычным образом, используя в качестве формальных параметров переменные целого типа, и вторым с использованием указателей. Пример программы представлен ниже.

```
// VU_02.cpp : main project file.
#include "stdafx.h"
#include <iostream>
using namespace std;
void swap(int x, int y);
void swap1(int *px, int *py);

void main()
{
int a=111, b=222;
cout <<"Ishod.dannye: a= "<<a<<"\t b= "<<b<<endl;
        swap(a, b);
        cout <<endl<<"swap(int x, int y)  \t  a= "<<a<<"\t b= "<<b<<endl;
swap1(&a, &b);
cout <<"swap1(int *px, int *py)   a= "<<a<<"\t b= "<<b<<endl<<endl;
}
        void swap(int x, int y)
        {
        int temp;
        temp=x;
        x=y;
        y=temp;
        }
void swap1(int *px, int *py)
{
int temp;
temp=*px;
*px=*py;
*py=temp;
}
```

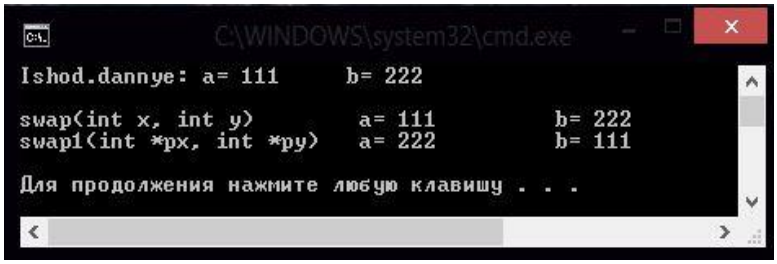


Рис. 3.4. -Пример использования указателей в качестве аргументов

Использование функции *voidswap(intx, inty*) не приводит к желаемому результату.

Это происходит потому, что она получает копии значений переменных *a* и *b* (переменные передаются по значению). Работая с копиями переменных, программа *swap* не может повлиять на значения переменных *a* и *b* из основной программы *main,* которые ей передали.

Использование в программе функции *void swap1(int *px, int *py)* позволяет достичь цель.

Вызываемой программе передаются указатели на те значения, которые должны быть изменены (значения передаются по ссылке).

swap1(&a,&b);

В этом случае п/программа работает с ячейками памяти и с их непосредственным содержимым.

3.2. МАССИВЫ

3.2.1. Понятие о массивах данных

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

Массив – переменная хранящая сразу несколько значений.

Основные *свойства* массива:

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

Признаком массива является наличие в его описании после имени массива *квадратных скобок*. В квадратных скобках при объявлении массива указывается размер массива, т.е. количество элементов в массиве. Перед именем массива, в его объявлении записывается тип массива (тип элементов, составляющих массив).

Рассмотрим несколько примеров.

int Mass[8]; // одномерный массив (вектор) с именем *Mass* из восьми целочисленных элементов

double b[10]; // одномерный массив с именем *b* из 10 элементов, имеющих тип *double*

int a[2][3]; // двухмерный массив, представлен в виде матрицы с двумя строками и тремя столбцами

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

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

3.2.2. Связь указателей и массивов

2.2.2.1.Одномерные массивы

В языке СИ между указателями и массивами существует тесная связь.

Например, когда объявляется массив в виде «*int array[25]»*, то этим определяется не только выделение памяти для двадцати пяти элементов массива, но и определяется указатель с именем *«array». Его значение равно адресу первого по счету (нулевого) элемента массива*.

Поскольку имя массива является указателем, допустимо, например, такое присваивание:

```
int array[25];
int *ptr;
ptr=array;
```

Здесь указатель *ptr* устанавливается *на адрес первого элемента массива,* причем присваивание *ptr=array* можно записать в эквивалентной форме *ptr=&array[0]*.

Для доступа к начальному элементу массива (т.е. к элементу с нулевым индексом) можно использовать просто значение указателя *array* или *ptr*. Любое из присваиваний

```
*array = 2;
array[0] = 2;// традиционное присваивание. Самое медленное.
*ptr = 2;
```

присваивает начальному элементу массива значение 2, но быстрее всего выполнятся присваивания **array=2* и **ptr=2*.

Для доступа к произвольному элементу массива (т.е. к элементу с НЕнулевым индексом) следует учитывать то, что, если указатель *prt* указывает на некоторый элемент массива (нулевой-*a[0]*), то указатель *prt+1* будет указывать на следующий элемент –первый *a[1]*, т.е.:

** prt* - *это содержимое a[0]*
** (prt+1)* - *это содержимое a[1]*
** (prt+i)* - *это содержимое a[i]*

Важно! Между указателем и именем массива существует одно различие:
- указатель это переменная, поэтому можно написать *prt=a* или *prt++*,
- имя массива переменной не является, поэтому *array++ Недопустимо.*

3.2.2.2. *Многомерные массивы*

Рассмотрим двумерный массив целых чисел, в котором 3 строки и 5 столбцов

```
int a[3][5]={
        {  0,  1,  2,  3,  4  },
         { 10, 11, 12, 13, 14, },
         { 20, 21, 22, 23, 24  } };
```

Хранение. Хотя строки никак не отделены между собой, в памяти этот массив будет разбит на 3 отдельных одномерных массива. Каждый из них будет представлять только одну из трех строк двухмерного массива. Первый массив *a[0]* представляет собой нулевую строку, затем массив *a[1]* –вторую строку и *a[2]* - третью.

Доступ. В этих одномерных массивах хранятся элементы строки. Для доступа к третьему элементу второй строки следует написать *a[2] [3]. Это традиционный подход.*

Подход с помощью указателей.

Сначала обращаемся ко второй строке массива – т.е. одномерному массиву *a[2]*.

Для этого нужно прибавить к адресу начала массива *int a[3][5]* смещение, равное номеру строки и выполнить разадресацию (*помним, что имя массива –это указатель*):

**(a + 2)* *(!)*

На самом деле при сложении указателя с константой учитывается длина адресуемых элементов. Поэтому в этом случае, т.к. длина строки - 5 элементов типа *int*, место начала второй строки в памяти ПК определим так:

2(5 *sizeof(int)) (!!)*

Далее требуется обратится к третьему элементу полученного массива.

Для получения его адреса применяют сложение указателя *(!)* с константой 3 и произвести снова разадресацию.

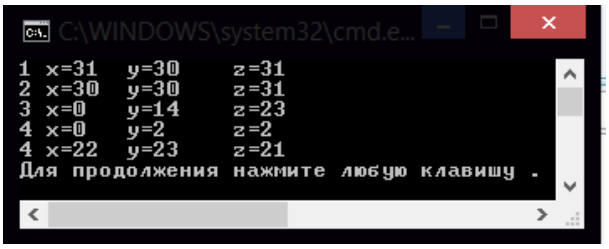
**(*(a + 2) + 3) (!!!)*

На самом деле происходит так:

2(5 *sizeof(int)) +3 *sizeof(int))*

Обращение к массивам и работа с ними демонстрирует рис.3.5.

```
// VU-3.cpp : main project file.
#include "stdafx.h"
#include <iostream>
using namespace std;

void main()
{
	int x, y, z, i=2, j=3;
//Одномерные массивы
	int arr[3]={30, 31, 32};
	int *ppa;
	ppa=arr;
		x=arr[1]; y= *ppa; z= *ppa+1;
		cout <<"1 x="<<x<<"\ty="<<y<<"\tz="<<z<<endl;
			x=*arr; y= *ppa; z= *arr+1;
			cout <<"2 x="<<x<<"\ty="<<y<<"\tz="<<z<<endl;
//Дмухмерныемассивы
int vuarr[3][5]=
		{{0, 1, 2, 3, 4},
		{10,11,12,13,14},
		{20,21,22,23,24}};
```




```
x=vuarr[0][0];  y=vuarr[1][4];  z=vuarr[i][j];
cout <<"3 x="<<x<<"\ty="<<y<<"\tz="<<z<<endl;
	x=*(*vuarr);  y=*(*vuarr+2);  z=*(*vuarr+2);
	cout <<"4 x="<<x<<"\ty="<<y<<"\tz="<<z<<endl;
x=*(vuarr[2]+2);  y=*(*(vuarr+i)+j);  z=*(*(vuarr+2)+1);
cout <<"4 x="<<x<<"\ty="<<y<<"\tz="<<z<<endl;
}
```

Рис. 3.5 - Пример работы с массивами

3.2.2.3 Динамические массивы

Массивы, размерность которых может быть переменной, называются динамическими.

Они создаются с помощью операции *new*, например,

```
int n=10;
float *pa=new int[n]
```

В последней строке создается указатель *pa* на вещественную величину. Причем, указателю присваивается адрес начала непрерывной области в динамической памяти, достаточной для размещения в ней *n* величин типа *float*.

Заметим:

- *n* может быть переменной**.**
- Динамические массивы нельзя инициализировать.
- Динамические массивы не обнуляются.
- Если динамический массив перестает быть нужным необходимо освободить память, которую он занимает. Для этого следует написать инструкцию:

```
delete [ ] a;
```

Пример использования динамических массивов представлен на рисунке 3.6.

Сначала вводим количество строк и столбцов матрицы – задаем размеры массива. Это 3 строки и 5 столбцов.

Инструкцией

*int **a = new int *[nrow]*

объявляем переменную типа указатель *на указатель на int*. При этом выделяется память под массив указателей на строки массива (количество строк *nrow=3)*

Поскольку массив еще не создан, строка *(1)* выдаст мусор (см. результат).

Цикл *for* организован для выделения памяти под каждую строку массива.

В строке *(2)*

a[i] = new int [ncol]

каждому элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку двухмерного массива. Каждая строка состоит из *ncol=5* элементов типа *int.*

В строке *(3)* задаем значения произвольным элементам массива.

В строке *(4)* задаем значения первым элементам (нулевым) строк массива.

Обратите внимание, что в цикле выводится значение указателя *a[i]* и значение переменной на которую он показывает **a[i]*.

В строках (6), (7), (8) продемонстрирован еще один способ, как добраться до значений переменных в массиве, используя указатели.

```
#include <iostream>
using namespace std;
using namespace System;
void main()
{
        int i, nrow, nkol, x, y, z;

        cout<<" Введи количество строк и столбцов<<endl;
        cin>>nrow>>nkol;
        cout <<"nrow="<<nrow<<"    nkol="<<nkol<<endl;

        int **a=new int *[nkol];
        cout <<"a[0][1]="<<a[0][0]<<"\ta[1][2]="<<a[1][2];
        cout <<"a[2][3]="<<a[2][3]<<endl;                                   //(1)

        for (i=0; i<nrow; i++)  {
                a[i]=new int[nkol];                                         //(2)
                a[0][1]=1; a[1][2]=12; a[2][3]=23;                          //(3)
                a[0][0]=0; a[1][0]=10; a[2][0]=20;                          //(4)
                cout <<"i="<<i<<"\ta[i]="<<a[i]<<"\t*a[i]="<<*a[i]<<endl;  //(5)
                }
        x=(*a[0]+1);
        y=(*a[1]+2);                                                        //(6)
        y=(*a[1]+2);                                                        //(7)
        z=(*a[2]+3);                                                        //(8)
        cout <<"x="<<x<<"\ty="<<y<<"\tz="<<z<<endl;
}
```

```
Введи количество строк и столбцов
3
5
nrow=3 ncol=5
a[0][1]=7244    a[1][2]=10    a[2][3]=7264
i=0       a[i]=0x3294      *a[i]=0
i=1       a[i]=0x32a4      *a[i]=10
i=2       a[i]=0x32b4      *a[i]=20
x=1       y=12      z=23
```

Рис. 3.6. - Использование динамических массивов

3.3 СТРОКИ

3.3.1 Определение строк

В отличие от других языков программирования, в Си отсутствует строковый тип данных. Вместо него строка символов представляется в памяти ЭВМ как *массив элементов типа char*, в конце которого помещен *нуль-символ* ***'\0'***. Таким образом *строка – это последовательность символов, оканчивающаяся нуль-символом*.

Нуль-символ как шестнадцатеричная константа представляется – *0x00*.

Символы строки кодируются, например, ASCII-кодом, и в таком виде записываются в память ЭВМ.

При определении строк символы строки заключаются в двойные кавычки

Пример 3.5.

Допустим задана строка *Hello!*

В ЭВМ эта строка будет иметь вид, представленный на рисунке 3.7.

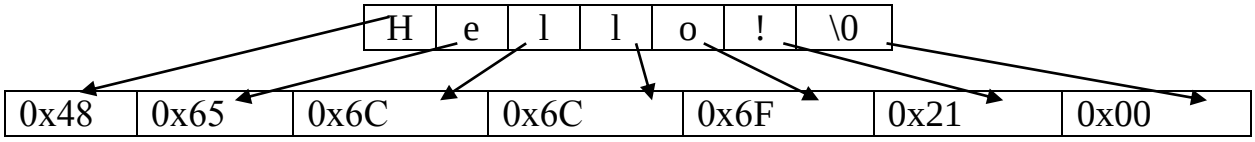


Рис. 3.7. – Представление текста в памяти ЭВМ

Существует следующие *основные способы определения строк:*

1) строковые константы;
2) массивы символов;
3) указатели на строки;

Рассмотрим эти способы.

1. *Строковые константы*. Как только компилятор встречает нечто, заключенное в двойные кавычки, он определяет это как строку и размещает ее символы последовательно в памяти, добавляя в конце ***'\0'***. Для определения констант используется директива *# define*.

Пример 3.6

```
# define STR "Обнаружена новая цель"
 . . . . . .
printf ("Пример: % s",STR);
```

2. *Массивы символов*. Для определения массива, необходимо сообщить компилятору требуемый размер памяти. Это можно сделать по-разному. Следующие три строки совершенно эквивалентны:

```
char str [ ] = "Слово";
char str [6] = "Слово";
char str [6] = {'С', 'л', 'о', 'в', 'о','\0'};
```

3. *Указатели на строки***.** В качестве напоминания следует отметить, что *имя массива есть указатель (константа) на нулевой элемент массива*:

Пример 3.7.

&str [0]< =>str, т.е. это – адрес нулевого элемента. Обращение по этому адресу дает букву ’с’.

*char *pst = “еще строка”;* Определение строки с помощью указателя.

Встретив такую запись, компилятор создает указанную строковую константу, а также переменную-указатель, которой присваивает адрес первого элемента строки.

3.3.2. Ввод-вывод строк.

Ввод-вывод символьных строк можно производить различными способами. Рассмотрим основные из них.

3.3.2.1. Ввод-вывод с помощью форматного ввода/вывода (средствами языка Си)

Для вывода строки символов на экран в функции *printf( )* используется спецификатор *% s* и указывается имя символьного массива.

Пример:

printf(“%s”, string);

Для ввода строки символов с клавиатуры в функции *scanf( )* также используется спецификатор *% s* и имя символьного массива.

Пример:

scanf (“%s”, string);

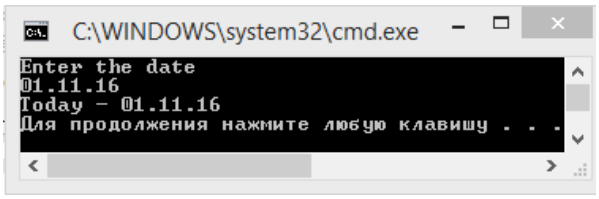
Знак **&** (амперсанд) перед именем массива не ставится, т.к. оно и так является указателем (т.е. адресом). При записи строки с клавиатуры необходимо позаботиться, чтобы заданный размер символьного массива был достаточным для записи в него строки. В противном случае может произойти затирание следующих за массивом ячеек памяти.

Следует отметить, что если при вводе символов ввести пробел, то после пробела информация в программу введена не будет, т. к. ввод осуществляется до первого разделителя, которым является пробел.

Пример 3.8.

```
// VU_05.cpp : main project file.

#include "stdafx.h"
```

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


void main()
{
	char data[10];
	printf("Enter the date\n");
	scanf("%s", data);
	printf("Today - %s\n", data);
}
```



3.3.2.2. Ввод-вывод с использованием неформатного ввода/вывода

Если форматный ввод/вывод позволяет работать с различными типами данных, то неформатный – с символами и строками символов.

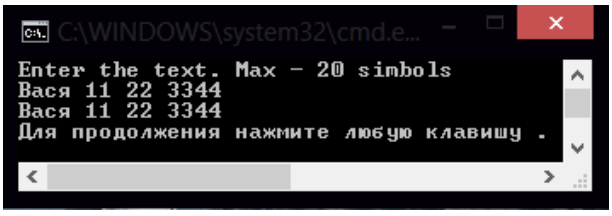
Для ввода строк применяется функция *gets (char *string),*
Для вывода – *puts (char *string),*

где *string* задает область памяти, в которую помещаются символы вводимой строки и откуда они выводятся на экран.

Данные функции подобны функциям *printf( )* и *scanf( )*. Отличие в том, что функция *gets( )* передает в программу все символы (и пробел в том числе) до символа *'\n'*.

Пример 3.9.

```
// VU_051.cpp : main project file.

#include "stdafx.h"
#include <iostream>
#include <stdio.h>

void main()
{
	char str[20];
	printf("Enter the text. Max - 20 simbols\n");
	gets (str);
	puts (str);
}
```

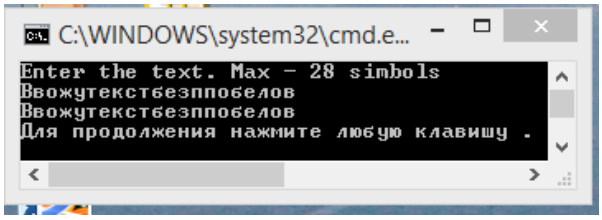


3.3.2.3. Ввод-вывод средствами консольного ввода/вывода языка Си++.

В Си++ в отличие от Си для ввода/вывода используются не функции, а операторы >> и << в сочетании с названиями стандартных потоков *cout, cin*. Причем ввод выполняется до первого пробельного символа (пробела, табуляции, перевода каретки)

Пример 3.10

```
// VU_052.cpp : main project file.

#include "stdafx.h"
#include <iostream>
using namespace std;

void main()
{
        char str[28];
        cout<<"Enter the text. Max - 28 simbols\n";
        cin>> str;
        cout<< str<< "\n";
}
```



3.3.2.4. Операции со строками

Пример программы, работающей со строками приведен ниже на рис.3.8.

```
// VU-053.cpp : main project file.
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
using namespace std;

void main()
{
        char s[40], ct[40];
        int i;
        printf("Enter text \n");
        gets(ct);
                i=strlen(ct);
                cout<<endl<<"kol_simvolov="<<i<<endl;
        strcpy(s,ct);
        puts(s);
                strcat(s,ct);
                puts(s);
}
```

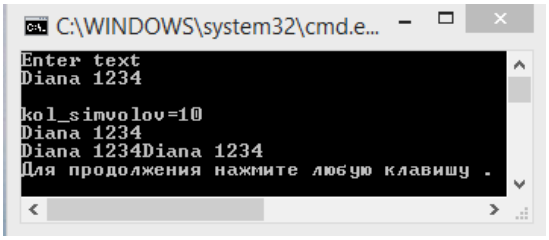


Рис.3.8 – Операции со строками

Помимо ввода/вывода со строками с помощью функций можно производить и другие операции. Функции, проводящие операции со строками определены в заголовочном файле *<string.h>*.

Рассмотрим некоторые функции.

strcpy(s,ct)– копирует строку *ct* в строку *s*,включая *'\0'*; возвращает *s*.
Здесь и далее *ct* и *s* принадлежат типу *shar**.
strcat(s,ct)– приписывает *ct* к *s*; возвращает *s*.
strcmp(s,ct)– сравнивает *s*и *ct*; возвращает
<0, если *s <ct;*
0, если *s = ct***;**
>0, если *s>st.* (Сравнение здесь лексикографическое.)
strlen(s)– возвращает длину *s*.

Вопросы и задания.

1. Объясните разницу между переменными и указателями.
2. Дайте определение указателя и порядок объявления указателей в программе.
3. Повторите все примеры и добейтесь выполнения приведенных программ, используя компилятор VC++2005-2012. Объясните работу операторов и особенности программ.
4. Составить программу сортировки массива, элементы которого вводятся с клавиатуры.
5. Составить программу сортировки массива, элементы которого вводятся с клавиатуры. Массив инициировать с помощью указателей.
6. Из массива, введенного с клавиатуры найти минимальное (максимальное) число.
7. Из массива, введенного с клавиатуры найти минимальное (максимальное) число., с использованием указателей.
8. Составить программу передачи массива в функцию в качестве параметра.
9. Составить программу сложения чисел с использованием указателей.

4. ОСНОВНЫЕ СВЕДЕНИЯ О СТРУКТУРАХ

В реальных задачах информация, которую требуется обработать может иметь достаточно сложную структуру. Помимо встроенных типов данных: целые, указатели и т.д. язык С++ позволяет программисту определить свои типы данных и правила работы с ними. Это структуры, объединения и др. Эти типы данные иногда *называют тип данных, которые создал программист.*

Структуры в С++ обладает практически теми же возможностями, что и классы. Но чаще их используют просто для логического объединения связанных между собой данных.

Структура – это одна или несколько переменных (возможно, различных типов), которых для удобства работы с ними сгруппировали под одним именем.

Структуры помогают в организации сложных данных, поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов с различными именами, а как единое целое.

Пример из работы аэропорта.

При работе диспетчера самолет характеризуется набором данных: дальность (D), азимут (As), высота (H), признак или название самолета, например, «РФ/Заграница» или «Ил62» (Type) и т.д.

Предположим, что наша программа должна оперировать с этими данными, причем количество самолетов может исчисляться десятками. Получится, что нам необходимо будет использовать несколько массивов данных, в одном из которых будет содержаться информация о высоте, в другом – о дальности и т.д. Такое представление вызывает определенные трудности. В самом деле, чтобы вывести на экран характеристики одного из самолетов, необходимо указывать несколько массивов с различными именами. Было бы проще использовать некую конструкцию, в которой все данные об одном конкретном самолете были бы объединены. Такая конструкция в языке Си имеется и называется *«структура».*

Представим сведения о воздушной цели в *виде структуры*, в которой заключены указанные данные о дальности, азимуте, высоте полета и признаке «свой/чужой».

4.1. ОБЪЯВЛЕНИЕ СТРУКТУРЫ

- начинается с ключевого слова *struct* и *имени структурного типа*, которое предложил программист (его называют *тегом*);
- далее в фигурных скобках разместятся описания элементов, которые будут входить в каждый объект типа *имени структурного типа*, т.е. разработанной структуры. Перечисленные в структуре переменные называются *элементами,* или *полями* структуры.

- после фигурных скобок можно (необязательный параметр) можно указать переменные данного структурного типа. (в примере 4.1-они указаны)

Пример 4.1 // переменные данного структурного типа указаны.

```
struct Cell
        int D;
        int As;
        floatH;
charType[20];}
jet, plane;                    //список структурных переменных
```

В этом примере объявлены две переменные *jet* и *plane* структурного типа *Cell*. Каждая из объявленных переменных имеет поля, указанные в структуре, в которые можно будет записывать данные о двух объектах. Причем поля имеют различный тип, а переменная *Type[20]* представляет собой символьный массив.

Объявление переменных заданной структуры приводит к выделению памяти определенного размера. В примере1 память будет выделена под две структуры с именами *jet* и *plane***.**

Если объявить структуру, не содержащую списка переменных, то память не резервируется, а просто описывает *шаблон структуры***.** Если шаблон задан, то описать структурную переменную можно короче:

Пример 4.2 // переменные данного структурного типа НЕ указаны.

```
structCell {
        intD;
        intAs;                  /*Задали шаблон структуры
                                  (т.к. переменных нет)*/
        floatH;
char Type[20];}
void main()
{Cellgruppa[20];
/* В программе объявили массив с именем gruppa из 20 элементов. Каждый элемент этого массива представляет собой структуру типа Cell.*/
}
```

*Инициализация***.** При определении структуры ее можно инициализировать в виде списка константных выражений, например, запись:

Celljet={250, 30, 2,5, ”Боинг”};

означает, что в структуре jet переменной D (дальность) будет присвоена 250, As (азимут) 30, H (высота) 2,5, a type(тип) – “Боинг”.

Доступ к элементам структуры. Чтобы обратиться к элементу структуры необходимо указать имя переменной данной структуры и через точку-имя элемента данной структуры. Например, чтобы вывести на экран значение дальности до цели, необходимо записать

Cout<<gruppa.D

Точка **(.)** означает оператор доступа к элементам структуры и соединяет имя структуры и имя элемента.

Пример 4.3. Пример использования переменной данных рассмотренных выше представлен на рисунке ниже.

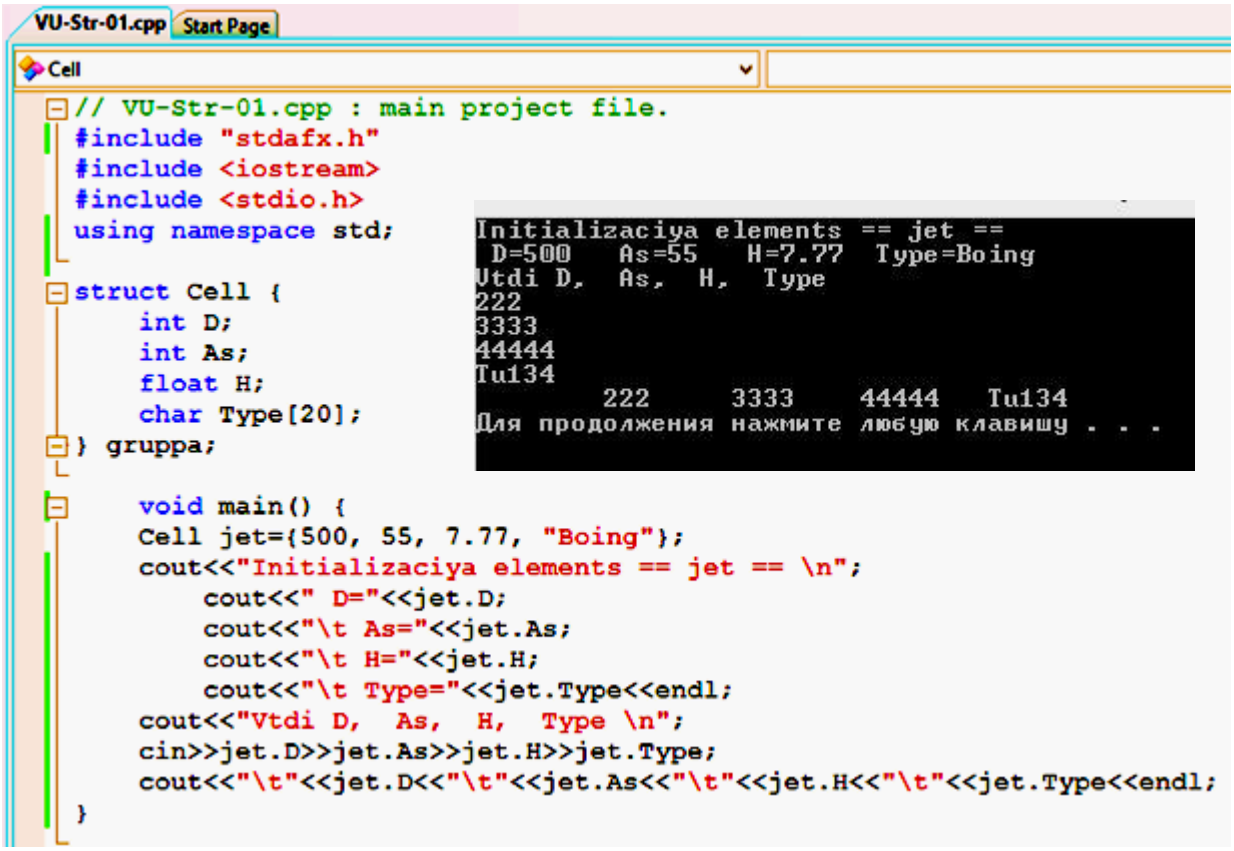


Рисунок 4.1. Использование структуры в программе.

Структуры могут быть вложены друг в друга.

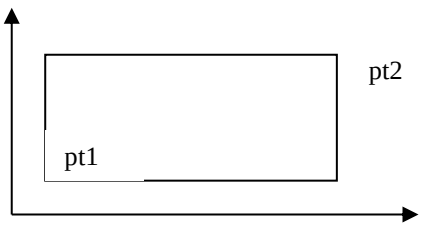
Т.е. элементом структуры может быть другая структура.

Для примера создадим две структуры: -

✓координаты точки (целые числа) поместим в первую структуру,

✓вторая структура, представляющая прямоугольник, будет содержать две первых

– для левой нижней точки и правой верхней.



Пример 4.4

```
struct point {                  // Структура для точки
        int x;
        int y;}
                struct rect {                   //Структура  для прямоугольника
                        int point pt1;
                        int point pt2;}
void main()
{  struct rectscreen;//переменная screen структурного типа rect
   screen.pt1.x                 //ссылка на координату x точки pt1.
```

Еще один пример программы с использованием структуры для вычисления модуля комплексного числа, представлен на рис.4.2.

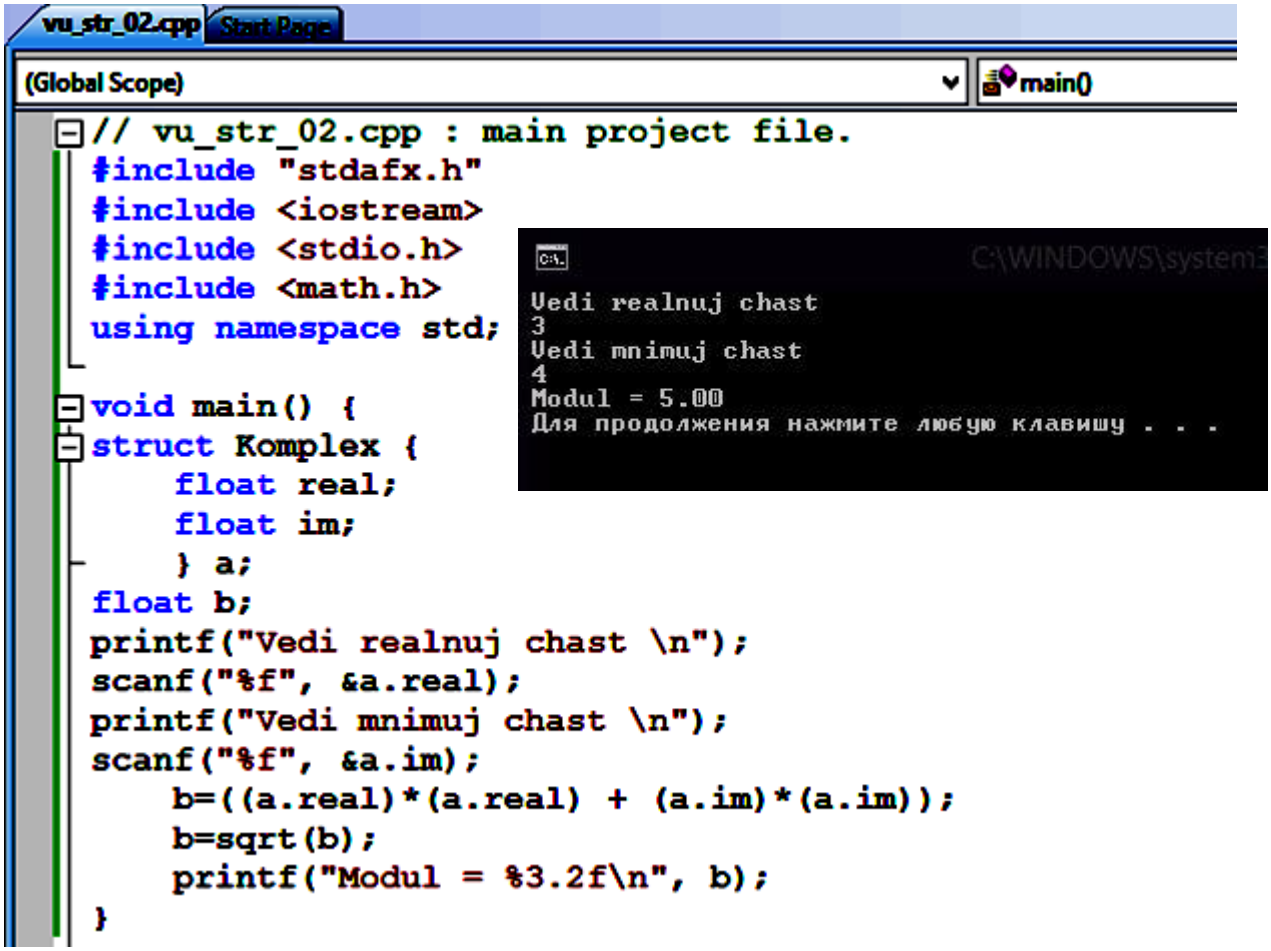


Рис.4.2. Вычисление модуля комплексного числа с использованием структур.

4.2. СТРУКТУРЫ И ФУНКЦИИ

Ранее были рассмотрены две операции над структурами – это доступ к элементам структуры и операция присвоения.

Кроме них разрешенными операциями являются: копирование и взятие адреса (&).

vu_st_03.cpp* Start Page

(Global Scope)

```
pt1     10      20
pmidl   20 *    20
add pt1         20      40
Для продолжения нажмите любую клавишу . . .
```

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
//#include <math.h>
using namespace std;
struct point {          //Задали шаблон структуры
    int x;              //Точка
    int y;
    };
struct rect {           //Задали шаблон структуры
    struct point pt1;   //Прямоугольник
    struct point pt2;
    };
struct point makepoint(int x, int y) {
    struct point temp;  //функция получает два значения
    temp.x=x;           // и возвращает структуру
    temp.y=y;           //Создали временную структуру
    return temp;
    }
struct point addpoint(struct point p1,
                      struct point p2) {
    p1.x +=p2.x;        //Создали временную структуру
    p1.y +=p2.y;        //сложения точек
    return p1;          //p1.y=p1.y + p2.y
    }
void main() {
struct point pt1;       //создали структурную переменную
pt1=makepoint(10, 20);  //инициализировали переменную
                        //координаты точки
cout<<"pt1 \t"<<pt1.x<<" \t"<<pt1.y<<endl;

struct rect screen; //структурная переменная - прямоугольник
struct point;       //Стр.перем-я -точка посреди прямоугольника
    screen.pt1=pt1;
    screen.pt2=makepoint(30, 40);
    pmidl=makepoint( (screen.pt1.x +screen.pt2.x)/2,
                     (screen.pt1.y +screen.pt2.y)/2 );
    cout<< "pmidl \t"<<pmidl.x<<" *\t"<<pt1.y<<endl;
pt1=addpoint(pt1, pt1); // складываем точки и выводим координаты
cout<<"add pt1 \t"<<pt1.x<<" \t"<<pt1.y<<endl;
}
```

Рис. 4.3 Работа с процедурами

Например,

А) если *struct name a, b*; то допустимо *a=b*; и *b=a*.

Б) если *Struct name*p;*
то допустимо *p=&t;*
и тогда верно **p=t*.

Структуры могут передаваться функциям в качестве аргументов. В функцию можно передавать:

а) компоненты структуры по отдельности;

б) всю структуру целиком;

в) указатель на структуру.

а) б) Функции могут возвращать отдельную переменную структуры или всю структуру целиком. Пример, приведенный на рисунке 4.3, демонстрирует процесс передачи функциям аргументов и возврат ими как обычных переменных, так и структур.

Рассмотрим еще один пример - пример 4.4, представляющий собой листинг программы, в котором функция используется для инициализации переменных структуры.

Пример 4.4

```
#include<iostream.h>
struct target          //В начале на внешнем уровне описывается шаблон
                       //структуры типа target.
        int b;
        {int d;
        float h; };
struct target init( )                  //  Затем определяется функция init( ),
{struct target temp;                    //  которой не передаются аргументы,
cout<< "Insert d:"; cin>>temp.d; /*но, которая возвращает структуру
                                        типа target, предварительно
                                        заполнив ее поля */
cout<<"Insert b:"; cin>>temp.b;
cout<<"Insert h:"; cin>>temp.h;
return temp;
}
void main( )                  // В главной функции объявляется структура jet
        {struct target jet;     // типа target, в которую копируется структура,
        jet = init( );          // возвращаемая функцией init( ). Поля скопиро-
        cout<<jet.d<<endl ;     // ванной структуры выводятся на экран.
        cout<<jet.b<<endl;
        cout<<jet.h<<endl;
}
```

Вернемся к примеру 2. Опишем функцию, которой передается структура.

double dg (struct target ff)
{return(Sgrt(ff.d ff.d + ff.h* ff.h));}*

В данном примере вычисляется горизонтальная дальность до цели. Функция *dg* имеет один параметр – структуру *ff* типа *target;* возвращает значение типа *double* т.к. функция выдает результат в формате *double*.

в) Если функции передается большая структура, то чем копировать ее целиком, эффективней передать указатель на нее. Указатели на структуры ничем не отличаются от указателей на обычные переменные.

Объявление

*Struct point *pp;* сообщает, что *pp*– это указатель на структуру типа *struct point.*

Если *pp* указывает на структуру *point* , то

**pp* – это сама структура, а *(*pp).x и (*pp).y* ее элементы.

Скобки здесь необходимы, т.к. приоритет оператора точка **.** выше чем приоритет звездочка *****.

Выражение **pp.x* будет интерпретировано как **(pp.x)*, что неверно, так как *pp.x* указателем не является.

Пример на рис. 4.4. демонстрирует сказанное.

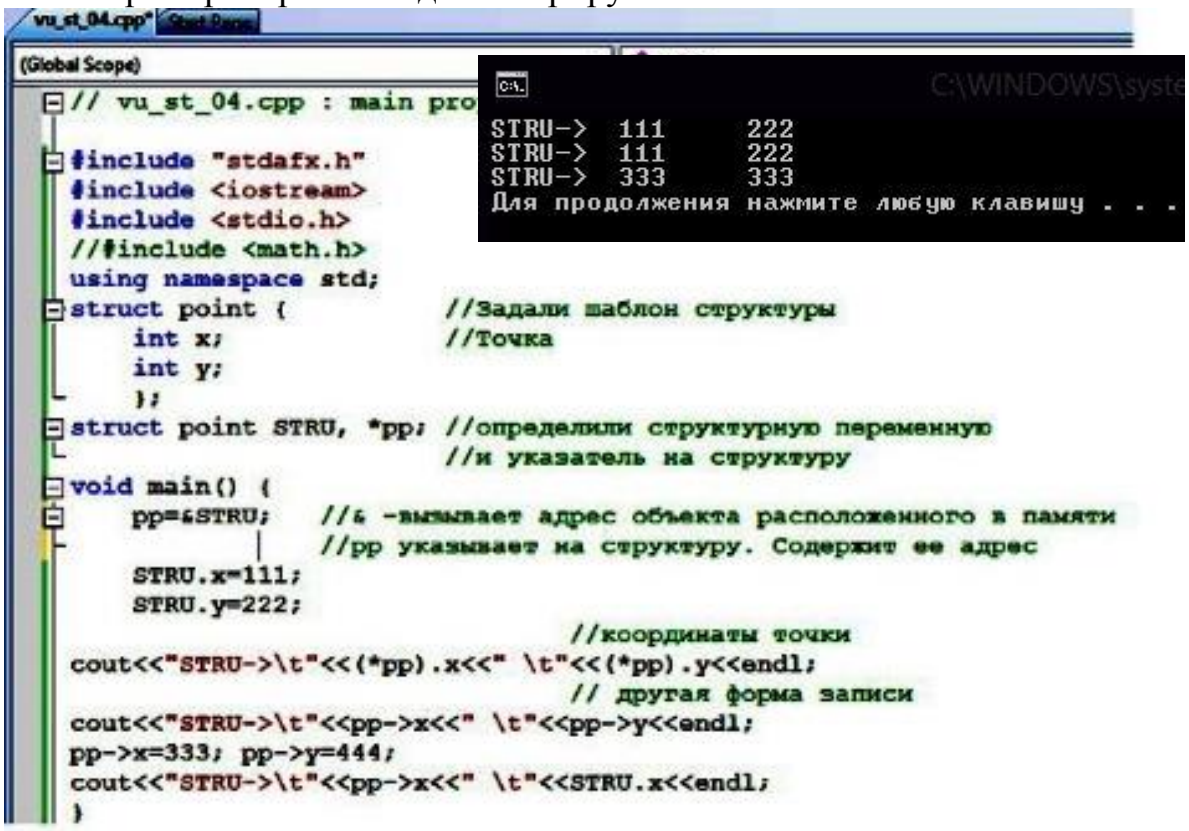


Рис.4.4 – Использование указателей на структуру

Для доступа к элементам структуры через указатель существует другая форма записи. Если *pp*- указатель на структуру *point*, то к элементу структуры *STRU* обращаются так:

pp ->STRU.x

Пример, приведенный выше, демонстрирует такую форму обращения.

4.3 ОБЪЕДИНЕНИЯ

При программировании может возникнуть ситуация, когда в каждый момент времени требуется использовать лишь одну переменную из нескольких переменных разного типа. В этом случае для всех переменных отводится одна область памяти, где в различные моменты активизируется та или иная переменная. Для организации такой работы служит тип данных – объединение, который обозначается ключевым словом ***union.***

Объединение –это переменная, которая в различные моменты времени представляется разными именами и разными типами из заранее указанного множества.

Ее описание по форме похоже на описание структуры.

Пример объединения A:

```
union medley {   inta;
                 float b;
                 char c [10];}   mu;
```

Объединение инициализируется только значением первого поля ***mu.a = 100***. Как видно, доступ к полям объединения такой же, как и к полям структуры.

Различия объединения и структуры рассмотрим на примере. Зададим структуру, подобную объединению из Примера А.

Пример структуры B:

```
struct str  { int  a; float d; char c[10]; }  SS;
```

Сравним расположение структуры и объединения в памяти (рис.4.5):

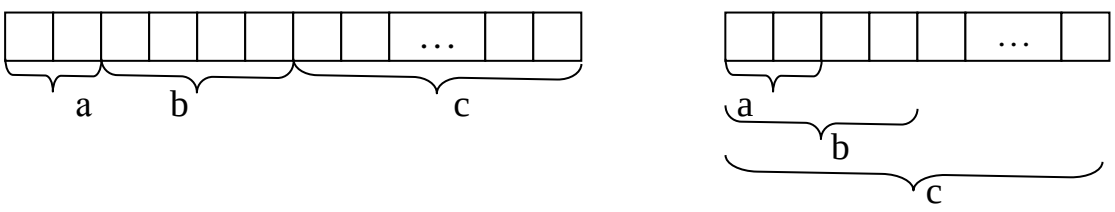


Рис.4.5 – Разница между структурами и объединениями

Поля в структуре располагаются последовательно друг за другом, без наложений. В объединении поля начинаются с одного и того же адреса и поэтому – с наложением.

Рассмотрим программу, в которой использовано объединение. Она приведена на рис.4.6.

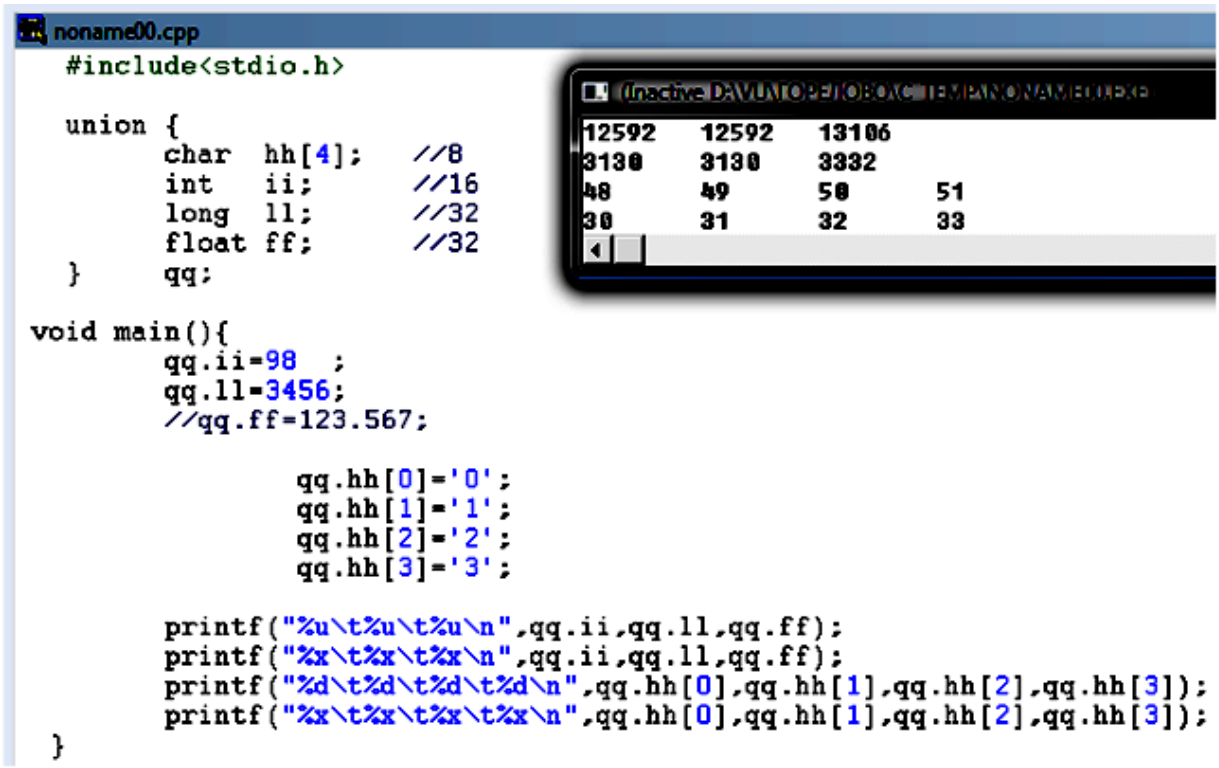


Рис.4.6. – Листинг программы с объединениями

Задано объединение *qq*. Оно включает:

- массив *char[4]*, каждый элемент которого содержит 8 бит (1 байт),
- целую переменную *int ii* - 16 бит,
- целую переменную *long ll* – 32 бита,
- вещественную переменную *float ff* – 32 бита.

Поскольку последним записывается массив символов *char[4]*, вся остальная информация будет стерта.

Каждый из символов будет сохранен в своем байте, причем ему будет соответствовать свой ASSCI код: 0 соответствует -48_{10} (11000_2), 1 – 49, 2 – 50, 3- 51. Что и видно в окне результата в третьей строчке. Ниже, в 4-й строке приведены те же числа, но в шестнадцатеричной системе счисления.

В памяти ПК эти числа хранятся в двоичной системе счисления. Рисунок ниже (рис.4.7) демонстрирует сказанное.

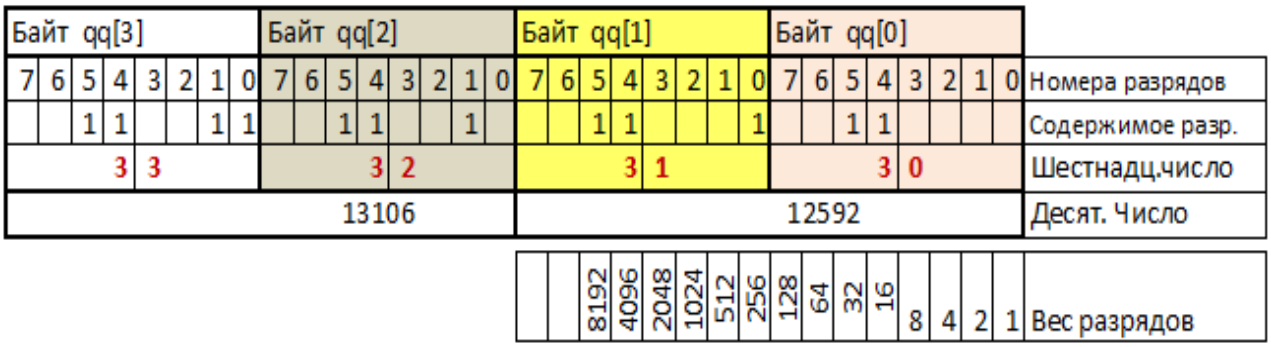


Рис.4.7. – Хранение переменных в памяти ПК

Заремим массив *hh[4]*.

Для случая, когда длинное целое равно предельному значению *int ii* все нормально

```
qq.ii=32767 ;
qq.ll=32767;
```

```
ii      32767 ll        32767
Шестнад ii ll ff        7fff
```

В том случае, когда значение превышает допустимое *int ii* при его считывании получим результат, которого, как правило, не ждем.

```
qq.ii=32767 ;
qq.ll=32769;
```

```
ii      -32767 ll       32769 ff
Шестнад ii ll ff        8001    8001
```

Для корректного ответа следует перевести число *int ll* в шестнадцатеричную систему счисления, потом в двоичную, и оценив полученное перейти к *int ii.*

И наконец зададим *qq.ff.* Результат для этого случая приведен ниже.

```
qq.ll=32769;
qq.ff=123.567;
```

```
ii      8782 ll         1123492430 ff   123.567001
Шестнад ii ll ff        224e    224e    42f7
```

Все кроме *qq.ff* нуждается в корректировке и объяснении.

4.4. БИТОВЫЕ ПОЛЯ

Битовое поле может быть только элементом структуры или объединения. Битовые поля не имеют адресов и не могут объединяться в массивы.

Назначение битовых полей:

- обеспечить удобный доступ к отдельным битам данных,
- формировать объекты с длиной внутреннего представления не кратного байту,
- плотно упаковывать память и экономить ее.

Формат структуры с битовыми полями:

strukt { тип_1 имя_ поля_1 : ширина_поля_1;
тип_2 имя_ поля_2 : ширина_поля_2;
……..
} имя_структуры

тип_i – тип поля, который может быть только *int*, возможно со спецификаторами *unsigned* или *signed*

*ширина_поля_**i*** - целое неотрицательное десятичное число, не превышающего длины слова конкретной машины.

Для обращения к полям используются те же конструкции, что и для обращения к обычным элементам структуры.

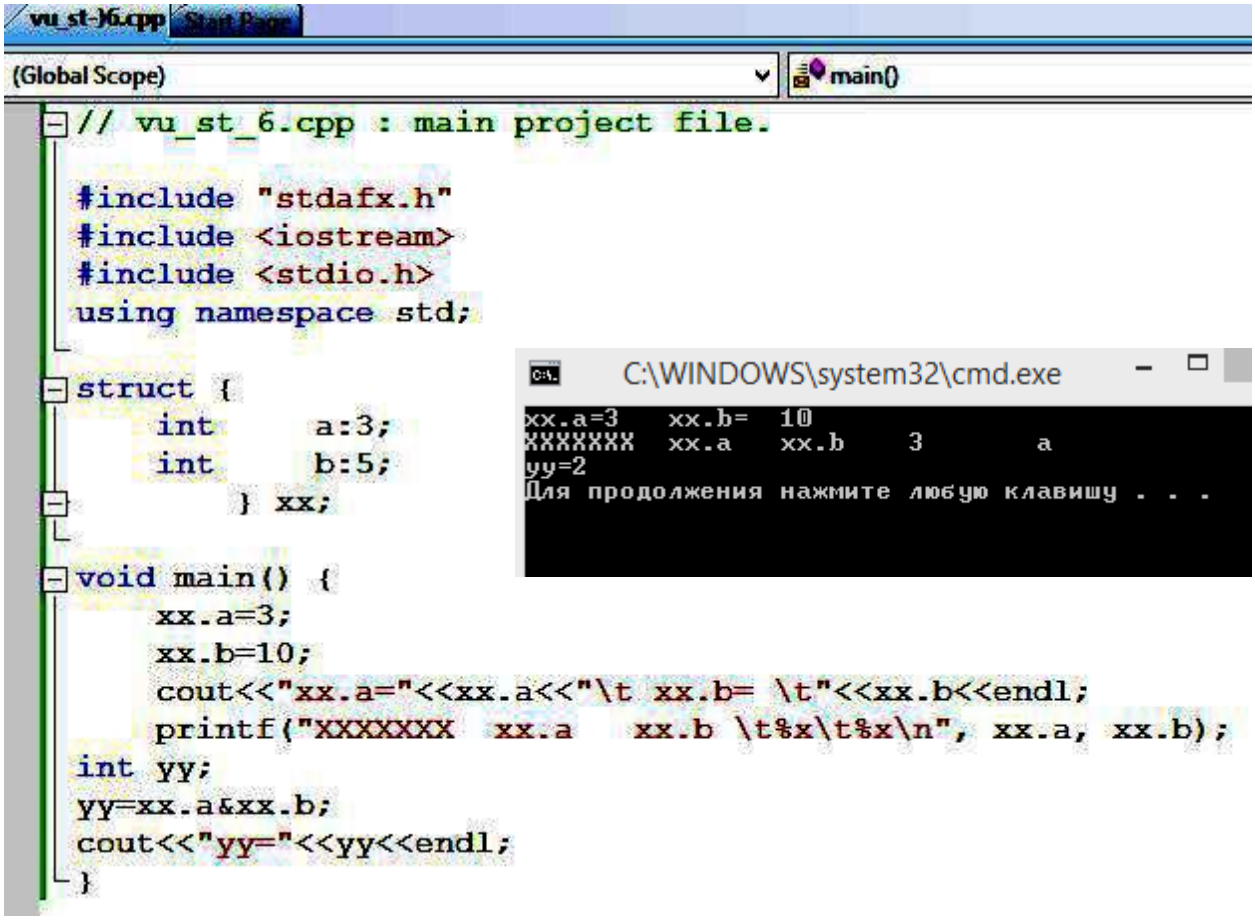


Рис.4.8 – Листинг программы, использующей битовые поля

Работу с битовыми полями демонстрирует пример, приведенный выше. В байте отдельно сформирован доступ к 3 младшим и пяти старшим разрядам. Используя поразрядные операции можно добираться к отдельным битам и т.д.

Выводы

1. Структуры и объединения представляют собой сложные типы данных, которые внутри себя объединяют переменные простых типов, причем внутри структуры — это совмещение статическое, а в объединении – динамическое.
2. Перечисление является удобным средством для определения именованных констант.

Вопросы и задания.

1. Повторите все примеры и добейтесь выполнения приведенных программ, используя компилятор VC++2005-2012. Объясните работу операторов и особенности программ.
2. Приведите свой пример программы, использующей структуры.

5 КЛАССЫ И ОБЪЕКТЫ СВОЙСТВА ЯЗЫКОВ ООП

Практика использования языка Си показала, что он хорошо подходит для разработки программ средней сложности. С увеличением длины программы она становилась трудновоспринимаемой и трудноуправляемой. Необходимы были новые подходы к программированию. И они были разработаны. Достаточно полно все

В 1979 году был создан язык Си++, автором которого явился Бьярн Страуструп. Си++ развивался на платформе языка Си и поэтому полностью включил его в себя. Достаточно полно все нюансы по программированию на Си++ изложены в литературе [1, 2, 4, 5] и др.

Язык Си реализовывал концепцию процедурного программирования, согласно которой каждой задаче приводился в соответствие свой блок данных и своя прикладная программа (функция). Вся программа представляла собой набор таких процедур. Язык Си++ создан для поддержки концепции объектно-ориентированного программирования (ООП).

5.1 ОСНОВНЫЕ ПОНЯТИЯ

Основное понятие языка ООП это класс. Класс можно представить как некую замкнутую систему. Такой подход изображен на рис.5.1.

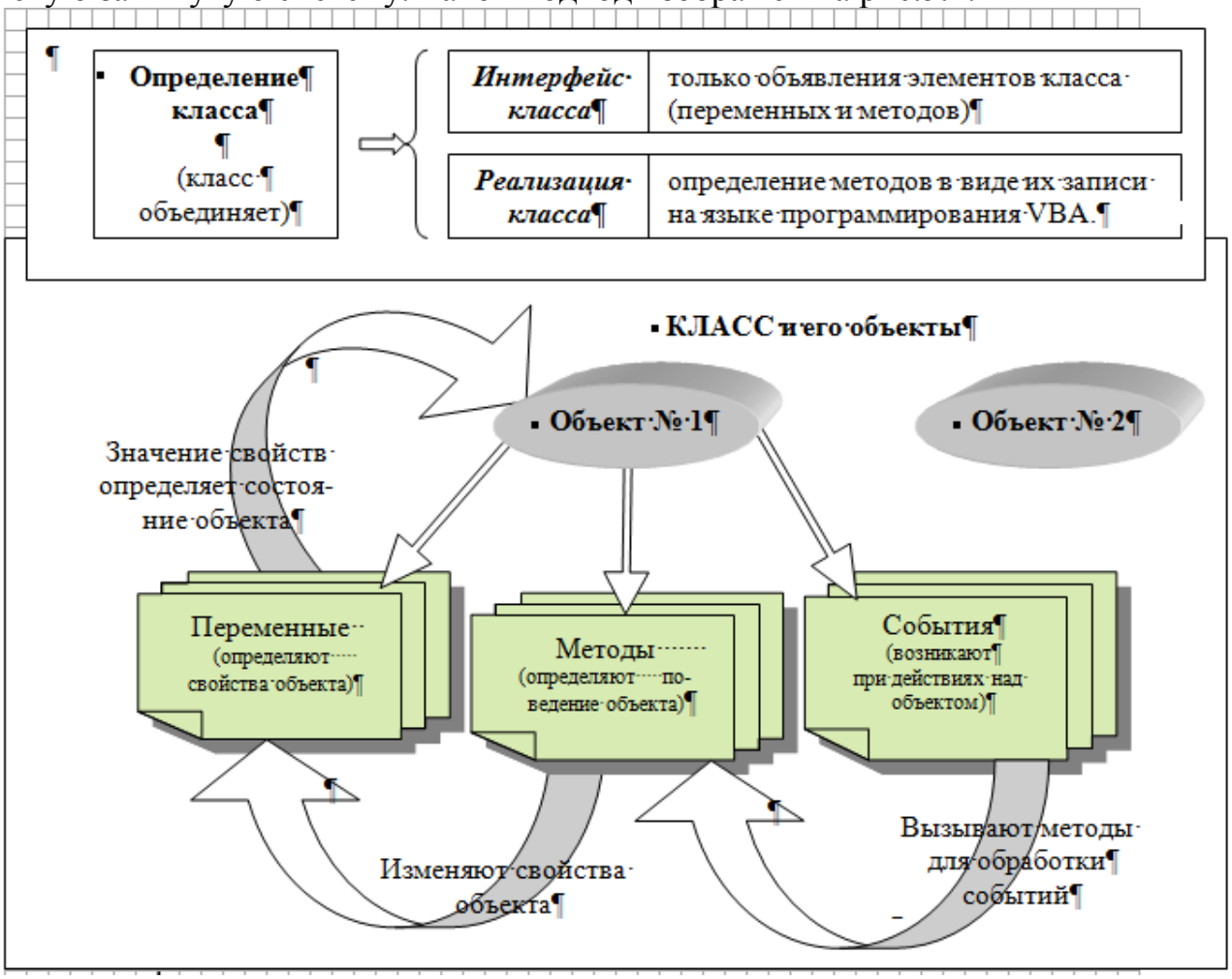


Рис.5.1- Класс, как замкнутая система

Класс определяет имена объектов, информацию о них, действия, которые они выполняют, их свойства и по этим признакам отделяет один класс от другого.

Объект - это экземпляр (элемент) класса. Например, определив класс - «собаки», мы совершенно точно знаем, что это животные, которые лают и выглядят совершенно определенным образом. И хотя каждый экземпляр класса отличен от другого, тем не менее, по общим свойствам и поступкам мы никогда не спутаем его с другим классом – коровой или тигром.

Класс отличается от структуры языка Си тем, что в него помимо собственно данных различных типов входят также функции, обрабатывающие эти данные.

Под *объектом* понимается переменная особого типа. Этот особый тип в Си++ создан на базе типа данных *структура* (*struct*), который мы уже умеем использовать в программах, и называется *класс* (*class*). Так вот, объявленная переменная типа *class* и есть *объект*.

5.2 СВОЙСТВА ЯЗЫКОВ ООП

Любой язык ООП должен удовлетворять таким свойствам как

- инкапсуляция,
- наследование,
- полиморфизм.

Наследование - предполагает то, что при создании нового класса объектов «А» его можно объявить наследником базового класса «В». При этом он будет наследовать все методы и свойства родительского класса, но главное, что они могут быть дополнены и расширены. Этими изменениями созданный класс «А» будет отличаться от базового.

Инкапсуляция - еще одно важное свойство. Дословно - это помещение в капсулу, а что более понятно – скрытие данных и реализаций класса от внешних по отношению к нему программ, использующих класс.

Если привязаться к определению класса, данному на рис.1, то получается, что интерфейс класса должен быть виден и доступен всем тем, кто его использует. А вот его реализация, то есть как написаны программы на языке С, может быть известно только программисту, который создавал этот класс.

Допустим есть класс (объект класса) - автомат, который умеет складывать два числа. Интерфейс представлен тремя полями и кнопкой «сложить» (см. рис.5.2). В первые два поля вводим слагаемые, а в третьем после щелчка по кнопке получаем результат. Эти поля и кнопка (*это интерфейс*) видны и понятны, иначе все теряет смысл.

Процесс же складывания скрыт и может быть изменен лишь программистом, создавшим класс. Он может написать программу для сложения первого числа со вторым, а может и наоборот, может, в конце концов, сам спрятаться за лицевой панелью автомата и, складывая в уме, вводить результат сложения вручную – этот результат будет один и тот же, *но реализация разная.*

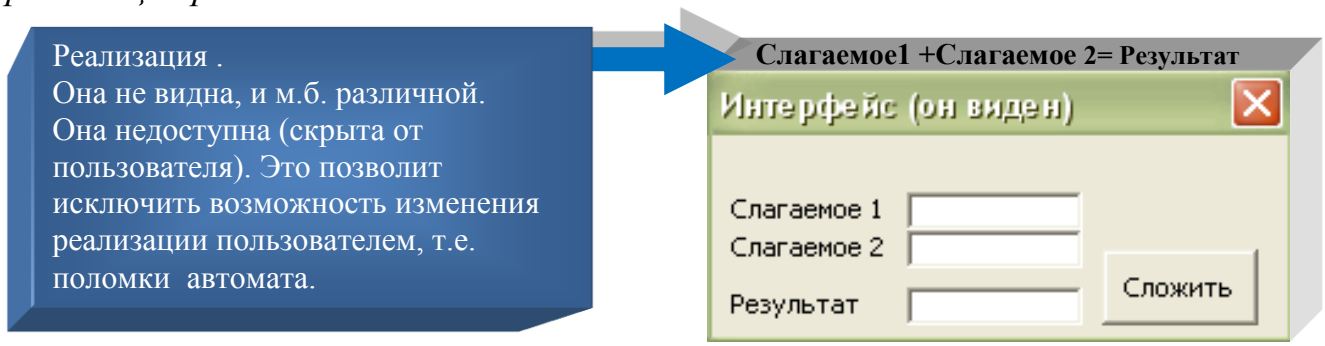


Рис.5.2 –Пояснение инкапсуляции

Для того чтобы определить степень инкапсуляции элементов класса, в С используют спецификаторы:

Public– публичный, доступный, открытый, общий. Элемент виден и доступен из любой программы;

Private–приватный, закрытый. Элемент доступен только методам данного класса и не доступен вне его описания.

Инкапсуляция защищает данные от внешнего вмешательства или неправильного использования. В то же время объект может оснащаться средствами программной связи с другими объектами. Итак, инкапсуляция – это первый «кит», на котором базируется ООП.

Полиморфизм

Полиморфизм по греческий обозначает много форм. Объекты, имеющие общего предка, могут принимать разные формы, оставаясь при этом схожими.

Например, в СИ можно использовать одни и те же имена или знаки для указания компьютеру на различные, но похожие действия. При этом вариант действий будет определен исходя из используемых типов данных. Например, складываем целые числа и числа с плавающей точкой, используя при этом знак + *(сложить)*.

Выводы. Данные и функции, входящие в класс, называются *представителями* или *членами* этого класса. Иногда в литературе *члены-функции* класса называют *методами* этого класса.

Члены класса разделяются на *закрытые (private)* и *открытые (public).* К закрытым членам обращение возможно только внутри класса и из других частей программы они недоступны. Обычно таким образом защищаются

данные от внешнего вмешательства. К открытым членам класса может производиться обращение извне.

По умолчанию все члены класса являются закрытыми. Но такая конструкция не представляет практического интереса, т. к. члены класса не могут быть использованы в программе. Для открытия доступа к ним обязательно необходимы public-члены, в качестве которых используют функции, оперирующие закрытыми членами класса. Таким образом, public-функции являются посредниками между private-членами класса и другими частями программы.

5.3 ОБЪЯВЛЕНИЕ КЛАССОВ И СОЗДАНИЕ ОБЪЕКТОВ

Класс объявляется с помощью ключевого слова ***class***, за которым указывается имя класса. Затем в фигурных скобках перечисляются члены класса.

Синтаксис объявления класса похож на синтаксис объявления структуры. Рассмотрим *пример объявления класса.*

Пример 5.1

```
#include<iostream.h>
class myclass                  // класс под именем myclass
{
int a;                         // закрытая переменная
public:                        // ключевое слово (для открытых членов)
      void set_a(int n);       // функции – открытые члены класса -
      int get_a();             // для доступа к закрытой переменной
};
```

✓ В этом примере объявлен класс *myclass*, который содержит одну закрытую переменную и две функции, которые являются открытыми членами класса.

✓ Перед закрытой переменной ключевое слово *private* может не ставиться. Оно предполагается по умолчанию.

✓ *Перед открытыми членами записывается ключевое слово public* ***с*** *двоеточием.* В конце описания класса, после закрывающейся фигурной скобки, ставится точка с запятой (**;**).

✓ Функции *set_a()* и *get_a()* только объявлены внутри класса, но еще не определены (не описаны). Эти функции, являясь открытыми членами класса, должны обеспечивать доступ к закрытым членам.

✓ *Доступ к закрытой переменной предполагает две операции:*

1). присваивание переменной определенного значения (инициализация);

2) получение значения переменной.

✓ Таким образом, для «обслуживания» одной закрытой переменной требуются две функции - открытые члены класса. При определении такой функций необходимо указать не только их функциональное назначение, но и принадлежность к классу, в котором они объявлены.

```
void myclass::set_a(int n)              //Пример 5 - А)
        {              // определения функций класса вне описания класса
          a=n;
        }
int myclass::get_a()
        {
        return a;
        }
```

Для обозначения принадлежности функции к классу достаточно лишь перед её именем указать имя класса и *два двоеточия*, которые *называются оператором расширения области видимости*.

Функция set_a() предназначена для присваивания целочисленных значений закрытой переменной a. Поэтому она имеет один параметр, через который и передаются эти значения; функция, в свою очередь, ничего не возвращает. *Вторая функция- get_a()* – служит для получения значения закрытой переменной. Поэтому она имеет возвращаемое значение, но при её вызове аргументы ей не передаются.

Описание функций – членов можно производить и внутри класса. Тогда они называются *встраиваемыми*. Такой способ удобен, когда функции не объёмные и за счёт этого не затеняют структуру класса. Приведем *пример 5.2 - Б)* описания того же класса, но уже со встраиваемыми функциями.

```
class myclass                                   // Пример 2 – Б)
{                                       // Определение функций внутри класса
int a;
public:
        void set_a(int n) {a=n;}
        int get_a() {return a;}
};
```

Как можно видеть, запись получилась более компактной.

Объявление класса myclass не задает ни одного объекта типа *myclass.* Оно определяет только *тип объекта (шаблон).* Чтобы создать объект, необходимо указать имя класса - как спецификатор типа данных, после которого записать имя объекта.

Пример создания двух объектов типа *myclass* .

```
myclass obj1,  obj2;
```

Создание объекта связано с выделением под него памяти. При объявлении же класса память не выделяется.

После того, как объекты созданы, можно обращаться к открытым членам класса. Как и при работе со структурами, обращение к открытым членам класса производится через точку **(.)**. В следующем фрагменте показано *обращение к функциям- членам класса myclass*, который используется в данном параграфе.

```
obj1.set_a(10);      // обращение к функции set_a( ) первого объекта
obj2.set_a(22);      // обращение к функции set_a( ) второго объекта
cout<<obj1.get_a( )<<'\t';    /*обращение к функции get_a( ) первого
                                объекта */
cout<<obj2.get_a( )<<'\n';    /*обращение к функции get_a( ) второго
                                объекта */
```

Функции *set_a( )* первого объекта в качестве аргумента передаётся численное значение *10*, которое присваивается переменной ***a*** этого объекта. Аналогично переменной ***a*** второго объекта присваивается значение *22.*

Каждый объект имеет свою переменную ***a***, свою функцию *set_a( )* и функцию *get_a( ).* Поэтому объекты еще называют экземплярами классов. В третьей строке фрагмента на экран выводится значение функции *get_a( )* первого объекта, а в четвертой – второго объекта. После выполнения указанных операций на экране будет: *10 22.*

5.4. МАССИВЫ ОБЪЕКТОВ

Из рассмотренного выше видно, что объекты – это переменные, и они имеют те же возможности и признаки, что и переменные любых других типов. Поэтому объекты могут объединяться в массивы. Синтаксис объявления массива объектов совершенно аналогичен тому, который используется для объявления массива переменных любого другого типа. То же касается и доступа к элементам массива. На рис.5.3 представлен пример работы с массивами объектов.

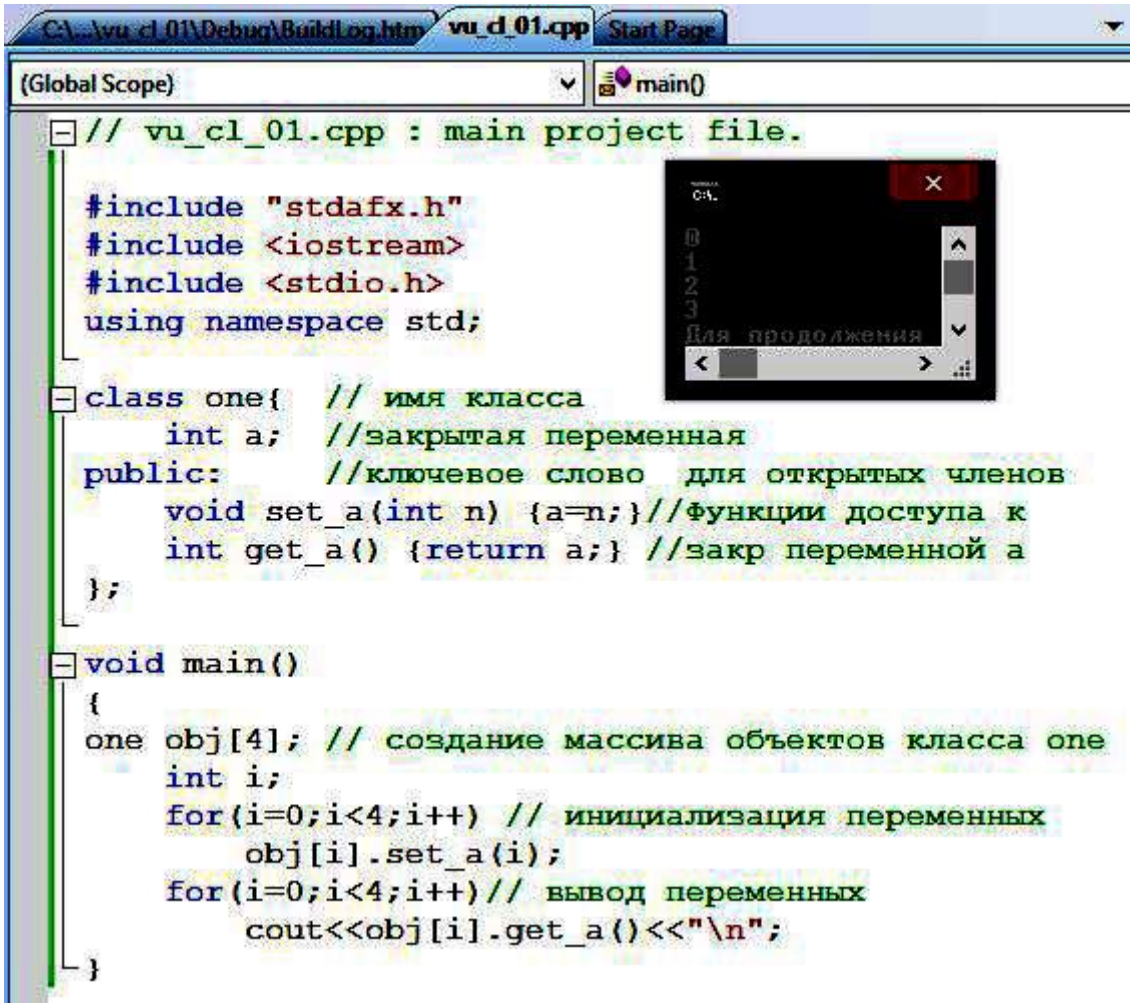


Рис.5.3. - Работа с массивом объектов

В данном примере объявлен класс *one*, в котором имеется закрытая переменная и две функции доступа к ней. В главной функции *main( )*создан массив из 4-х объектов, в котором циклической операцией производится инициализация закрытой переменной каждого элемента массива с помощью функции *set_a( ).* Во второй циклической операции производится вывод на экран значения переменной каждого объекта путем использования функции *get_a( ).*

5.5. УКАЗАТЕЛИ НА ОБЪЕКТЫ

На объекты можно создавать указатели. В этом случае доступ к члену объекта производится с помощью оператора стрелка (->), как и в случае работы со структурой.

Объявляется указатель на объект так же, как и указатель на переменную любого другого типа. Для этого следует задать имя класса этого объекта, а затем имя переменной со звездочкой перед ним.

Для получения адреса объекта перед ним необходим оператор **&,** точно так же, как это делается для получения адреса переменной другого типа. Если для указателя, например, производится операция инкремент, то он после этого будет указывать на объект такого же типа.

Пример использования указателя на объект содержит рис.5.4 .

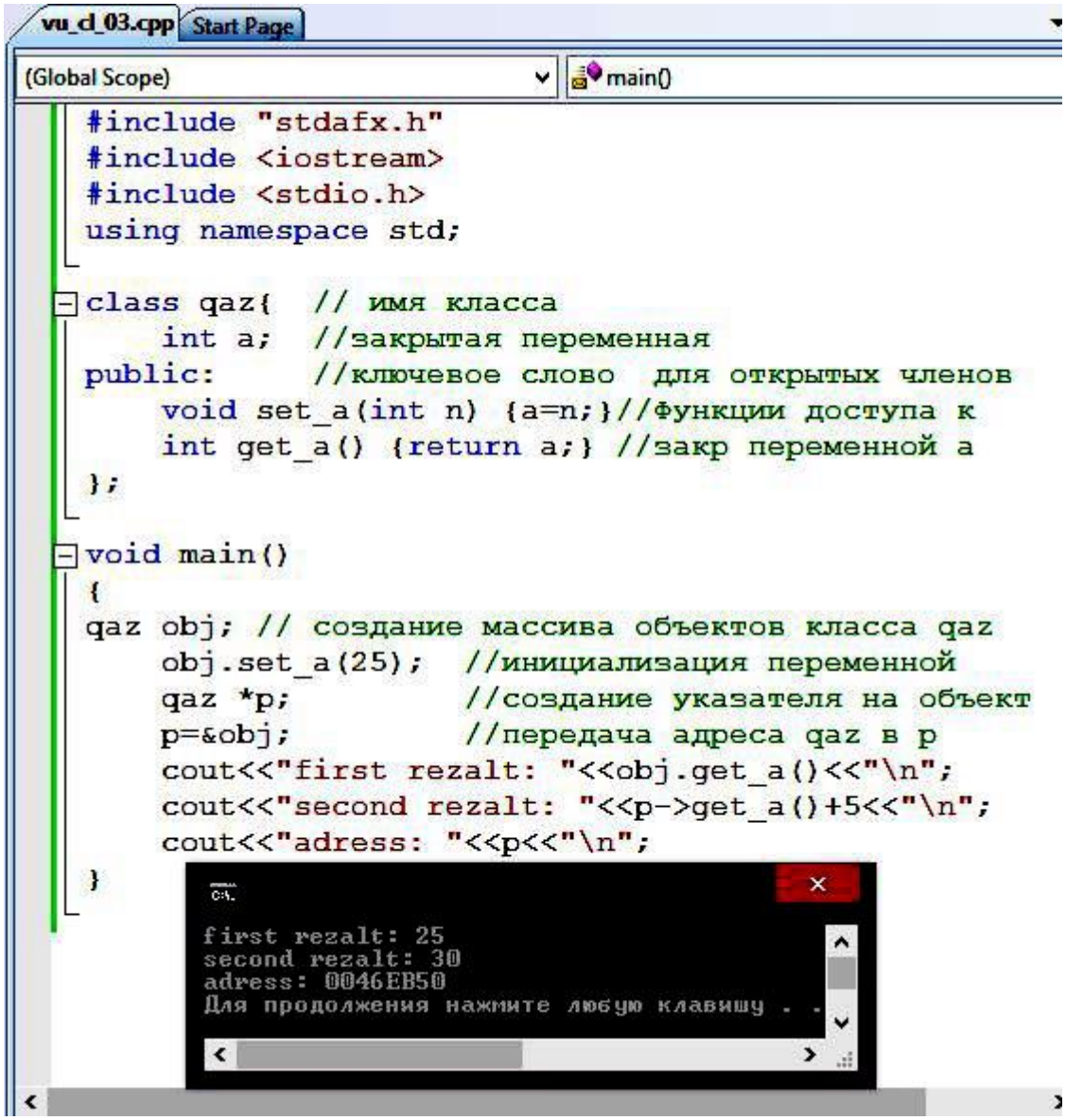


Рис.5.4. Листинг программы с использованием указателей на объект.

В примере класс *qaz* имеет закрытую переменную и две функции доступа к ней.

После создание объекта, в нём инициализируется закрытая переменная.

Затем объявляется указатель на объект класса two. Здесь важно понимать, что объявление указателя не создает объект, а только создает указатель на него.

В следующей строке адрес объекта передается переменной p (указателю).

В заключении программы показано обращение к членам объекта через имя объекта и через указатель.

Вопросы и задания.

1. Как объявить новый класс объектов.
2. Что такое члены и функции класса. Ка их объявить.
3. Поясните понятие открытых и закрытых членов класса.
4. Поясните что такое указатель на объект и как его использовать.
5. Повторите все примеры и добейтесь выполнения приведенных программ, используя компилятор VC++2005-2012. Объясните работу операторов и особенности программ.

6. КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

Среди членов класса могут быть две особенные функции. Их особенность заключается в том, что обращение к ним производится по умолчанию.

Первая функция называется *конструктор*. Она вызывается автоматически при создании объекта. Её назначение состоит в том, чтобы присвоить начальные значения (инициализировать) переменным объекта.

Вторая функция называется *деструктор*. Она служит для ликвидации последствий использования объекта (например, для освобождения памяти). Поэтому она автоматически вызывается при удалении объекта.

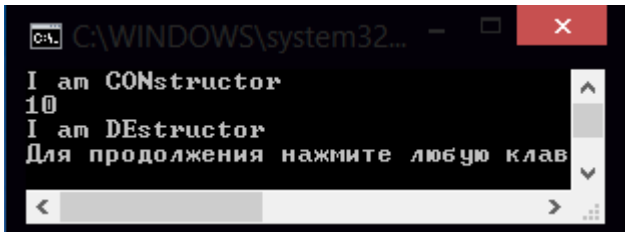
Конструктор имеет то же имя, что и класс, частью которого он является. Имя деструктора также совпадает с именем класса, но перед ним ставится символ *~(тильда)*. Конструктор и деструктор не возвращают никаких значений.

Рассмотрим пример использования конструкторов и деструкторов в программах.

На рис.6.1 приведен простейший пример. В данном примере класс *abc* содержит одну переменную и три функции, две из которых являются конструктором и деструктором. Следует отметить, что перед конструктором и деструктором слово *void не ставится* (оно предполагается по умолчанию). Определение функций производится вне класса.

В главной функции создается объект *obj*. В момент создания объекта вызывается конструктор. Затем вызывается функция *show( )* объекта и программа завершается. При окончании программы объект перестает существовать, что автоматически приводит к вызову деструктора.

После выполнения программы на экран будет выведено:



В этом примере конструктор не содержал параметров. Но обычно параметры присутствуют, т.к. основное назначение конструкторов – производить инициализацию переменных объекта.

В следующем примере используется конструктор с параметрами.

Класс *black* содержит:

- две закрытые переменные,
- две функции, одна из которых конструктор с параметрами.

vu_cl_05.cpp | Start Page

(Global Scope) | main()

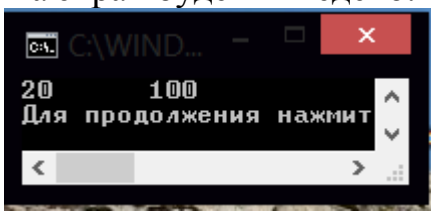
```
// vu_cl_05.cpp : main project file.

#include "stdafx.h"
#include <iostream>
#include <stdio.h>
using namespace std;

class abc{                    // объявление класса abc
    int a;                    // закрытая переменная
public:
    abc();                    // конструктор
    ~abc();                   // деструктор
void show();
};

abc::abc(){                   // описание конструктора
cout<<"I am CONstructor \n";
a=10;
}

abc::~abc() {                 // описаниедеструктора
cout<<"I am DEstructor \n";
}

void abc::show(){             // описание функции- члена класса
    cout<<a<<"\n";
}

void main(){
    abc obj;                  // создание объекта
    obj.show();
}
```

Рис.6.1 Программа с использованием конструкторов

Значения, которые передаются конструктору, будут присвоены закрытым переменным.

При создании объекта, в этом случае, после его имени в скобках через запятую указываются аргументы – численные значения, которые будут переданы параметрам ***x*** и ***y***.

На экран будет выведено:



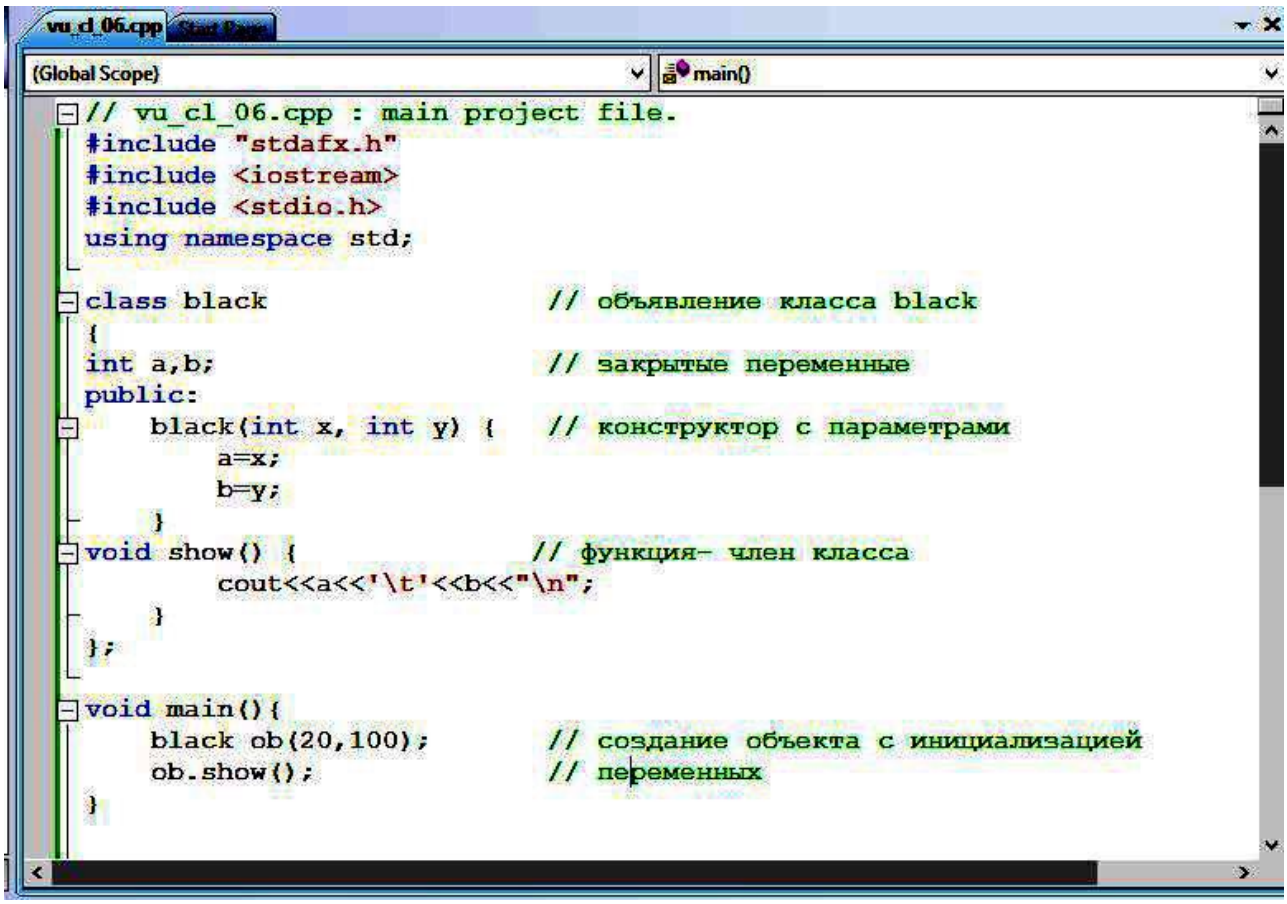


Рис.6.2 – Листинг программы, использующей конструктор с параметрами

Вопросы и задания.

1. Объясните разницу между конструктором и деструктором.
2. Повторите все примеры и добейтесь выполнения приведенных программ, используя компилятор VC++2005-2012. Объясните работу операторов и особенности программ.
3. Приведете свой вариант программы, использующей конструкторы и деструкторы.

7. ОБЪЕКТЫ И ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

7.1. ОБЪЕКТЫ И ДЕЙСТВИЯ С НИМИ

Объекты можно передавать функциям в качестве аргументов точно также, как передаются данные других типов. Для этого параметр функции объявляется типом *class*, а в качестве аргумента при вызове функции используется объект этого класса. Рассмотрим пример, приведенный на рис.7.1.

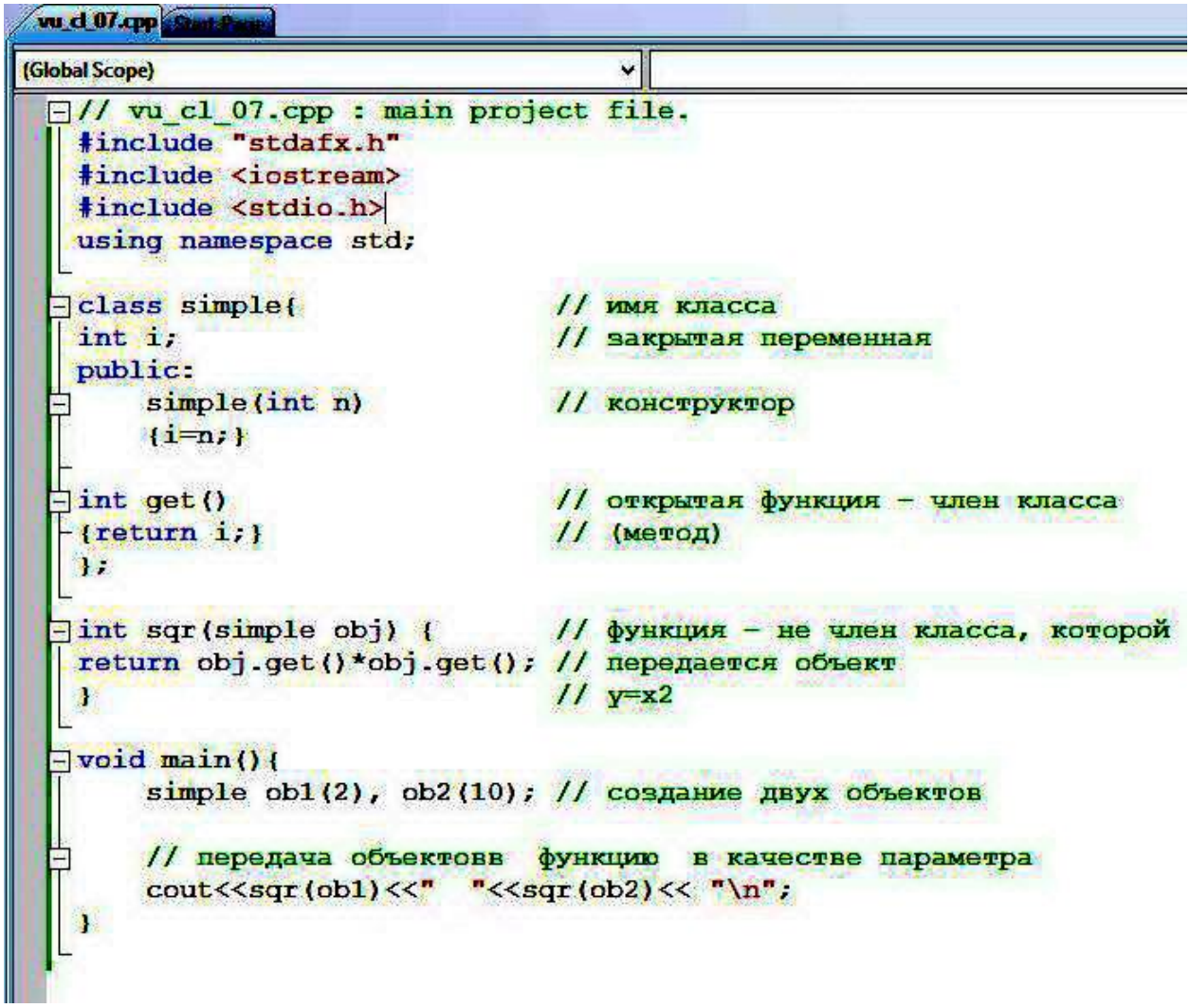
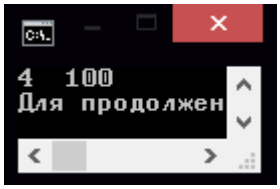


Рис.7.1. – Использование объектов в качестве параметров

В этом примере объявляется *класс simple*, который содержит одну переменную, конструктор и функцию, возвращающую значение переменной.

Функция *sqr( )* в качестве параметра имеет объект типа ***simple*,** получает из него значение переменной *i* и возвращает её квадрат. Здесь необходимо отметить, что *функция sqr( )не является членом класса.*

В главной функции создаются два объекта с инициализацией. На экран выводится значение квадрата переменной каждого объекта:



Возвращение объектов функциям производится также, как и значений других типов данных. Для этого необходимо объявить функцию так, чтобы её возвращаемое значение имело имя типа класса, а в операторе *return* указать имя возвращаемого объекта.

Возвращенный объект должен быть скопирован в месте вызова функции в объект такого же класса, другими словами, *операция присваивания может производиться только между объектами одного класса.*

7.2. ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

При программировании возникают ситуации, когда требуется функция, которая имела бы доступ к закрытым членам класса, но сама не являлась бы членом этого класса. Для этого введены дружественные функции.

Дружественная функция определяется в программе как обычная функция. Чтобы показать, к какому классу она дружественна, её объявляют в этом классе с ключевым словом *friend.*

Особенность дружественной функции состоит в следующем:

дружественная функция не является членом класса. К закрытым членам класса она *не может* обращаться непосредственно. Она это может сделать только через объекты этого класса. Поэтому в качестве *параметра* у дружественной функции должен указываться *объект* со спецификатором дружественного класса.

Рассмотрим использование дружественной функций на примере, представленном на рис.7.2.

В классе *data* объявлены две переменные: *a* и *b,* а также конструктор с параметрами.

Кроме того, объявлена дружественная функция *diff(data ob).* Необходимо обратить внимание на порядок её объявления: сначала записывается ключевое слово *friend,* затем указывается тип возвращаемого значения функции, после чего – имя функции и в скобках – класс и имя объекта.

В главной функции создается объект; при этом запускается конструктор и инициализируются переменные *a* и *b.* Созданный объект передается в функцию *diff( ),* которая вычисляет разность *a* и *b*. Разность возвращается переменной *d* и затем выводится на экран.

vu_d_09.cpp | Start Page

(Global Scope)

```
#include <iostream>
#include <stdio.h>
using namespace std;

class data{
int a,b;
public:
    data(int x,int y)           // конструктор с параметрами x и y
    { a=x; b=y; }
    friend int diff(data ob);   /* дружественнаяфункция.
                                   В классе объявляем ее прототип */
};

int diff(data ob)               // описание дружественной функции
{ return ob.a-ob.b; }

void main(){
int d;
data obj(8,5);                  //создание объекта и вызов конструктора
  d=diff(obj);                  // вызов дружественной функции
cout<<"Difference = "<<d<<"\n";
}
```

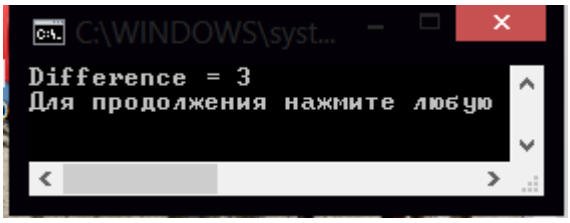


Рис.7.2 – Листинг программы с использованием дружественной функции

Сделаем выводы

1. Класс – это структура особого типа, включающая данные и функции работы с ними. Переменная такого типа есть объект.

2. Для инициализации объекта в состав класса входит особая функция – конструктор, которая вызывается автоматически при объявлении объекта.

3. При передаче объектов функциям и их возвращении соблюдаются правила работы как с обыкновенными переменными.

8. НАСЛЕДОВАНИЕ В С++

За счет механизма наследования, реализованного в любом языке объектно-ориентированного программирования, между объектами устанавливаются некоторые связи, которые можно отследить, построив модель или иерархию объектов.

Наследование– это процесс, посредством которого один объект может приобретать свойства другого. Так как объекты принадлежат к классам, то между классами в этом случае устанавливаются определенные взаимоотношения. Класс, который делегирует свойства называется *базовым (base)*, а который наследует свойства – *производным (derived - дирайвед)*. У одного базового класса может быть несколько производных, и наоборот, один производный класс может наследовать характеристики нескольких базовых классов.

Наследование может быть представлено в виде графов или схем наследования:

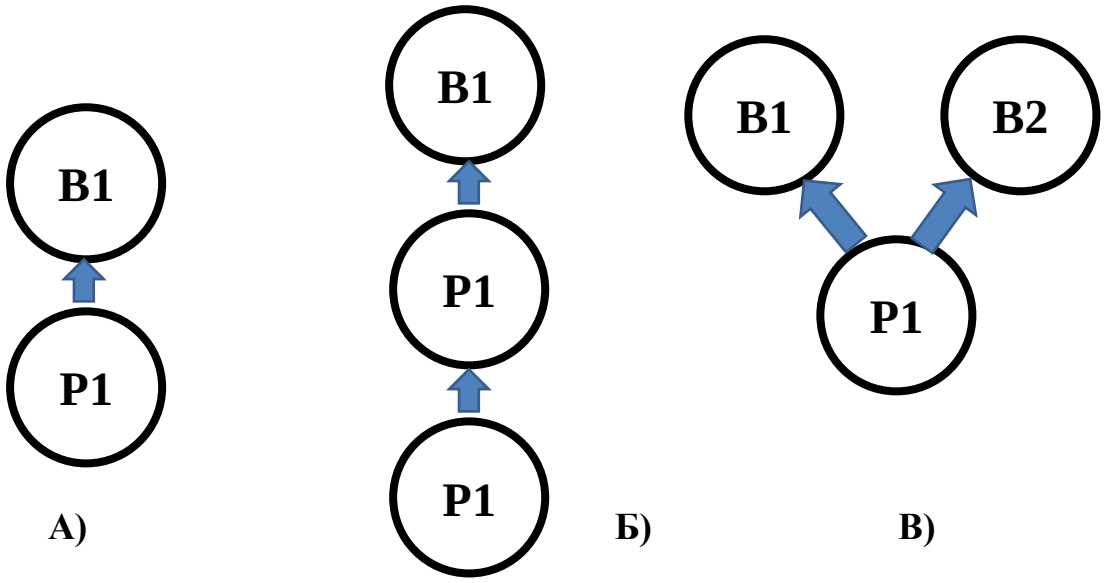


Рис.8.1 –Изображение наследования в виде графов

В примере *(А)* показана простейшая схема подчиненности с одним базовым *(B1)* и одним производным *(P1)* классом.

На схеме *(Б)* показано, что *B1* есть базовый класс для *P1*, который в свою очередь является базовым для *P2*. В этом последнем случае *B1* называется для *P2 косвенно-базовым*.

В примере *(В)* показан один производный *(P1)* класс от двух базовых *(B1* и *B2*).

Взаимосвязи классов в программе образуют *иерархию классов*.

Правила указания наследования.

1. Базовый класс описывается обычным образом. Например:

```
Class base
        {
                //содержание класса base
        };
```

2. Для указания наследования после имени производного класса ставится *двоеточие* и *указывается имя базового класса* со *спецификатором доступа (private, public).*

а) Для примера (А) описание будет выглядеть следующим образом:

```
class derived: public base
        {
                // содержание класса derived
        };
```

Спецификатор выбирается в зависимости от того, как предполагается использовать члены класса *base.*

в) Для примера (В) описание производного класса будет выглядеть, например, так:

```
class derived: public B1, public B2
        {
                //содержание класса derived
        };
```

б) В примере *(Б)* для производного класса *P2* базовым будет класс *P1*, поэтому описание *P2* соответствует схеме (а).

Преимущества, получаемые от механизма наследования:

1) повторное использование данных и функций, ранее созданных в базовом классе, без их дублирования в производном классе;
2) получение более компактной и обозримой программы;
3) повышение степени защищенности закрытых членов базового класса;
4) адекватность программной модели предметам реального мира.

8.1. УПРАВЛЕНИЕ ДОСТУПОМ К БАЗОВОМУ КЛАССУ

Базовый класс является первым в иерархии, он «не знает» сколько у него будет производных классов (или не будет вообще). Поэтому

1) он проектируется самодостаточным в плане доступа к закрытым членам и

2) члены производных классов в нем не упоминаются.

Как известно, *внутри* класса его члены открыты друг для друга: любая функция-член может обращаться к любой переменной-члену класса. Однако извне можно обратиться только к открытым членам класса.

Важно отметить, что при наследовании *производный класс* полностью включает в себя *базовый класс*. Однако не все члены классов открыты друг для друга в этом объединении. Здесь необходимо следовать следующим правилам.

1.«*Круг общения*» *членов производного класса* расширяется только за счет *public-членов базового класса*. Это означает:

- что *члены производного класса* могут обращаться *напрямую* только к открытым членам класса *base;*
- *private-члены базового класса* можно использовать в описании функций производного класса только через *открытые члены baseclass.*

2. Если наследование производится со спецификатором доступа public, то *открытые члены base* становятся *public-членами класса derived;*

3. Если наследование производится по типу *private*, то *public-члены базового класса* становятся *private-членами класса derived.*

4.Private-члены базового класса безусловно наследуются производным классом *derived*, но обращаться к ним члены класса *derived* могут только исключительно посредством *public-членов класса base.*

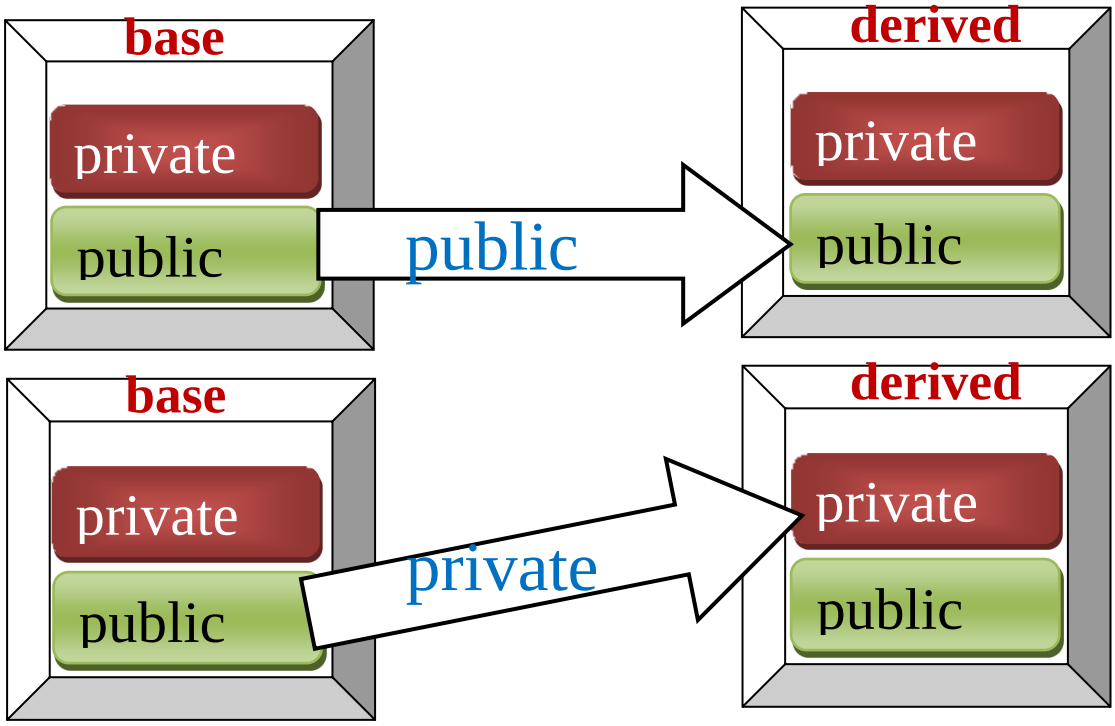


Рис.8.2 – Пояснение к процессу наследования классов

8.1.1. Базовый класс наследуется как открытый

Листинг программы для этого случая представлен на рис.8.3

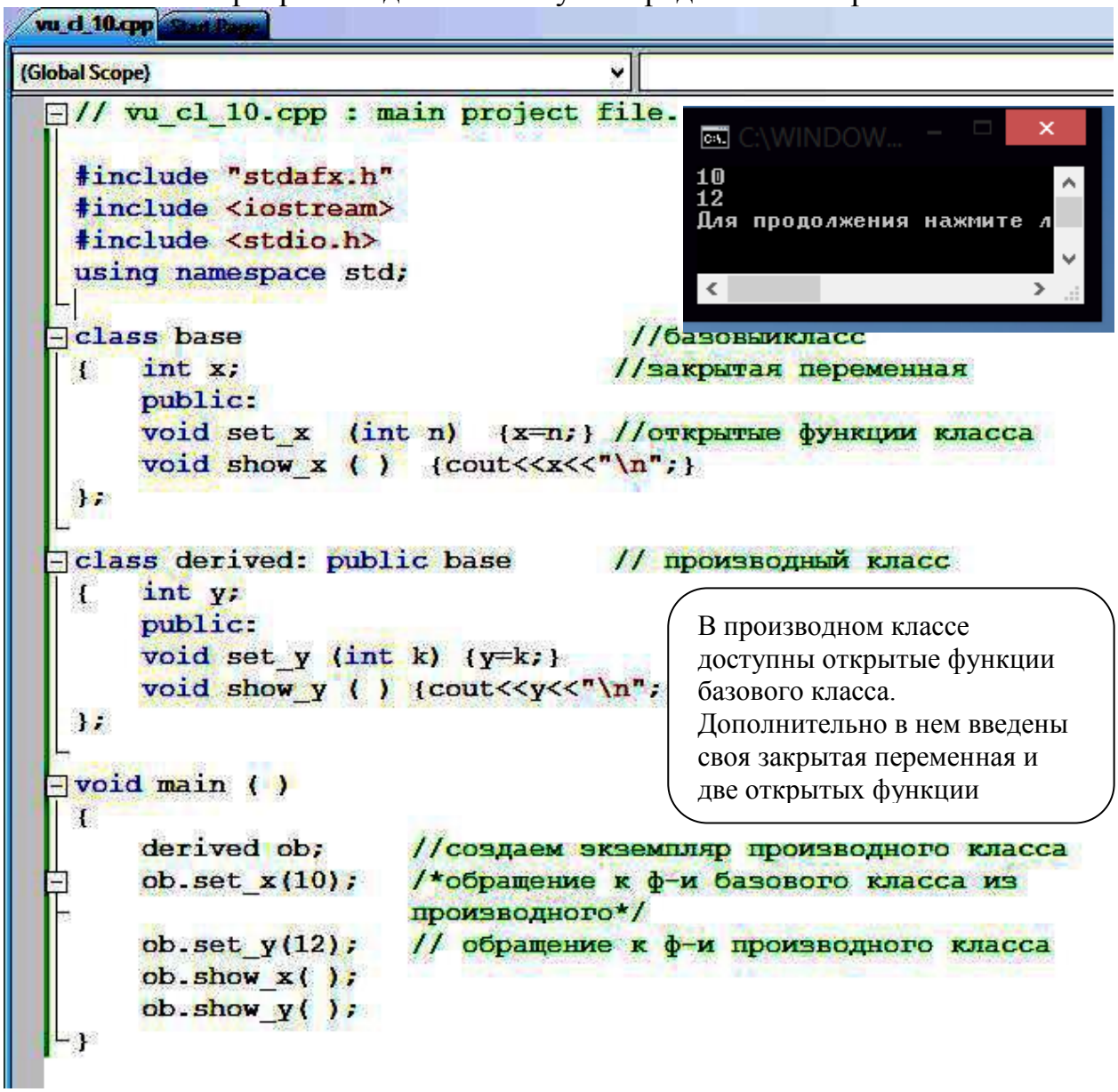


Рис.8.3 –Наследование базового класса как открытого

В данном примере base наследуется как открытый. Поэтому в объекте *ob*, который является экземпляром производного класса, его *public функции* доступны как обычные открытые члены.

В данном примере “добраться” до закрытого члена базового класса невозможно; и инструкция типа

cout<<ob.x; // Неверно!

будет ошибкой.

8.1.2. Базовый класс наследуется как закрытый

В следующем примере наследование производится с типом *private.*

В этом случае открытые члены базового класса в производном становятся закрытыми. Рассмотрим, как в этом случае получить доступ и к открытым и к закрытым членам базового класса через производный (см. рис. 8.4).

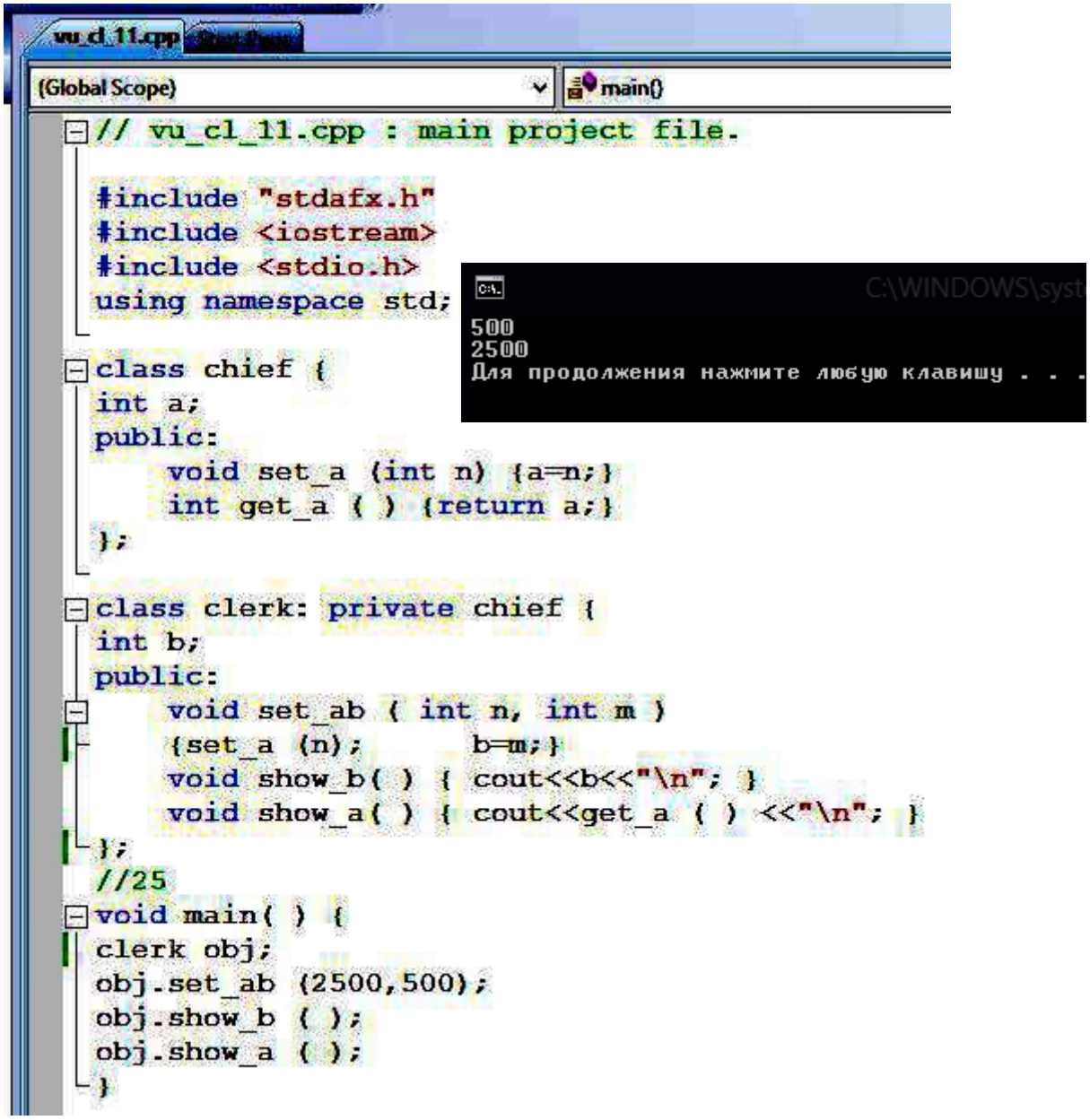


Рис. 8.4- Листинг программы с наследование класса типом *private.*

Т.к. при наследовании с типом *private* получить доступ к “наследству” можно только через открытые члены “наследника”, то в классе *clerk* проектируется функция для ввода значения закрытой переменной *“a”* базового класса и вторая функция – для вывода этого значения на экран.

Однако доступ к *“a”* возможен только через открытые члены базового класса, которые и выступают в этих функциях в качестве посредников.

8.1.3. Базовый класс наследуется как защищенный

Из вышеизложенного все больше проступает неудобство работы с закрытыми членами базового класса в производном классе. *Чтобы допустить членов производного класса к общению с private-членами базового класса введен спецификатор доступа protected (защищенный).*

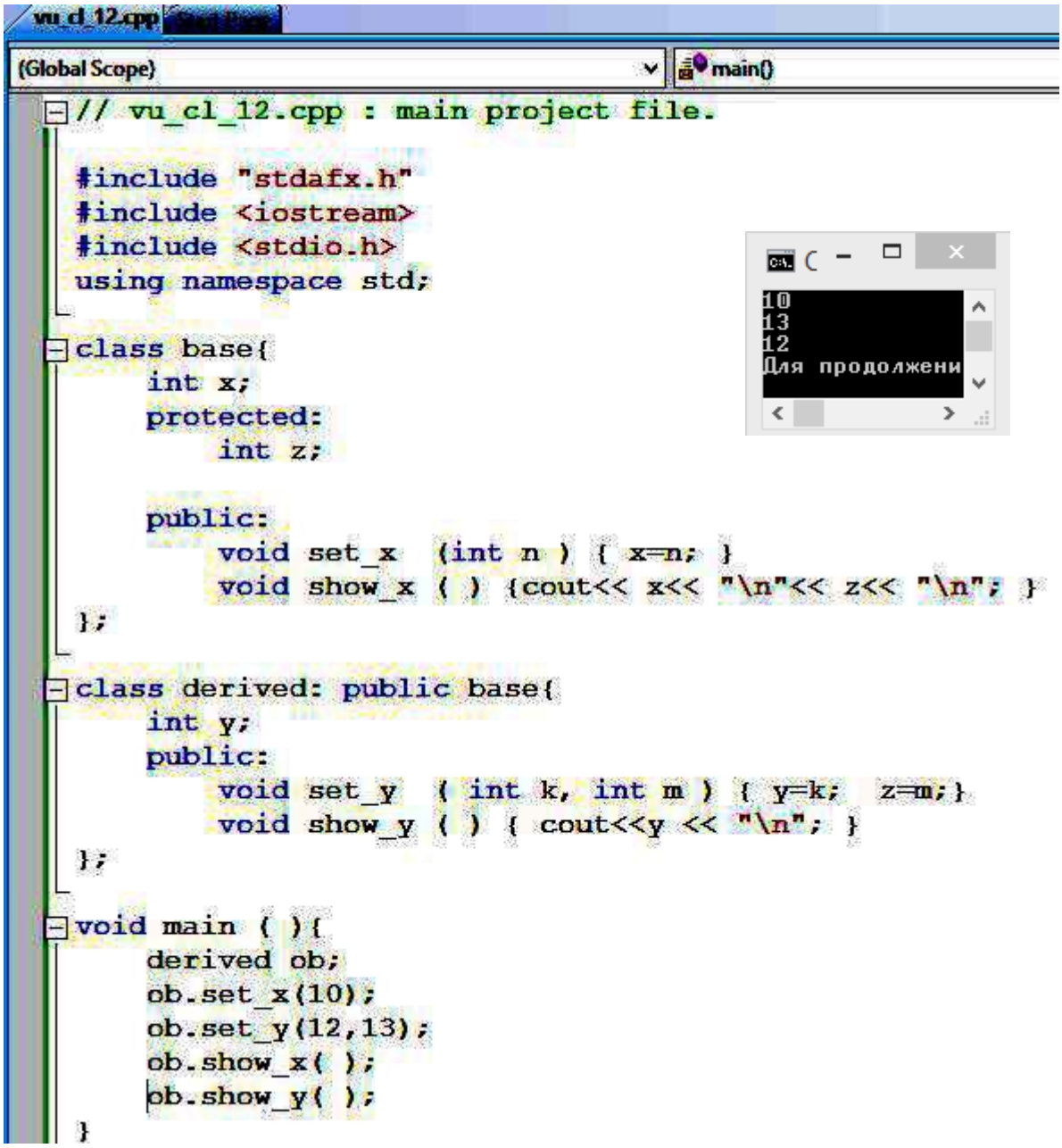


Рис.8.5. Наследование базового класса как защищенного

Теперь члены класса разделены на три группы: *private, protected и public*. Спецификатор *protected* может располагаться в любом месте описания класса, но обычно он располагается перед *public* . И это оправдано, потому что

protected-члены – это часть закрытых членов базового класса, к которым напрямую могут обращаться члены производных классов. Рассмотренное подтверждает программа, листинг которой представлен на рис.8.5

Таким образом, существует еще одно правило наследования:

Если базовый класс наследуется как защищенный, то открытые и защищенные члены базового класса наследуются как *protected*.

9. ПЕРЕГРУЗКА ОПЕРАТОРОВ И ФУНКЦИЙ

Познакомимся еще с одним «китом», на которой базируется ООП – понятием полиморфизма. Слово «полиморфизм» происходит от греческого πολυμορφία – многообразие. Оно характеризует способность чего-либо реализовываться в различных форматах. Применительно к языку Си++ под *полиморфизмом* понимается использование одних и тех же имен (или знаков) для указания ЭВМ на выполнение различных, но схожих по типу действий. При этом ЭВМ выбирает тот или иной вариант *в зависимости от используемых типов данных.*

Полиморфизм в скрытом виде присутствует и в языке Си. Например, при операции сложения ЭВМ реализует различные последовательности действий, если мы складываем целые числа или числа с плавающей запятой. Но и в том и в другом случае пишется знак «+». В Си++ полиморфные конструкции проектирует сам программист.

Если одно имя функции используется для различных последовательностей действий, то это называется *перегрузкой функции.* Если один оператор используется для обозначения различных действий, то такой тип полиморфизма называется *перегрузкой операторов*.

В более общем смысле полиморфизм позволяет реализовывать идею: один интерфейс – множество методов. Это означает, что можно создать один интерфейс для группы близких по смыслу действий. Его преимущество состоит в том, что он помогает снижать сложность программ, разрешая использовать один интерфейс для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор.

9.1. ПЕРЕГРУЗКА ФУНКЦИЙ

Если две или более функций имеют одинаковое имя, то говорят, что они перегружены. В этом случае функции должны отличаться типом и (или) числом своих аргументов. Перегрузить функцию не сложно – просто следует объявить и определить все требуемые варианты. На компилятор возлагается

задача выбора соответствующей конкретной версии вызываемой функции (а значит и метода обработки данных).

Рассмотрим пример перегрузки функции, представленный на рис.9.1.

В данном примере функция *date()* перегружается для получения даты либо в виде строки, либо в виде трех целых чисел. Данный пример показывает, что перегрузка функции может производиться и вне объектно-ориентированного подхода.

Компилятор различает перегруженные функции и в случае, когда параметры одного типа, но их разное количество.

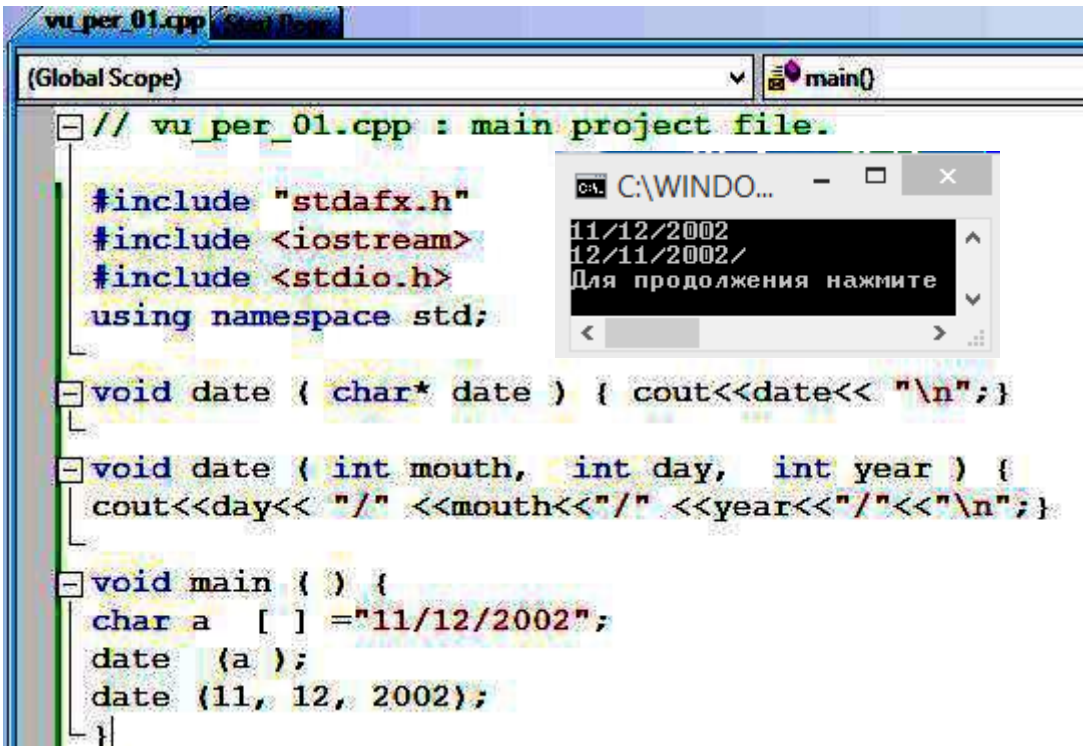


Рис. 9.1 – Пример перегрузки функций

9.2. ПЕРЕГРУЗКА КОНСТРУКТОРОВ

Перегрузка конструкторов в программах на Си++ - обычное дело. Она осуществляется для обеспечения гибкости и выполняет ряд других функций. Рассмотрим пример, представленный на рис.9.2.

В данном примере в классе *number* описаны два конструктора: один производит инициализацию, второй – нет.

Затем, в главной функции создается два объекта различными способами. У первого объекта x=5, у второго x=0.

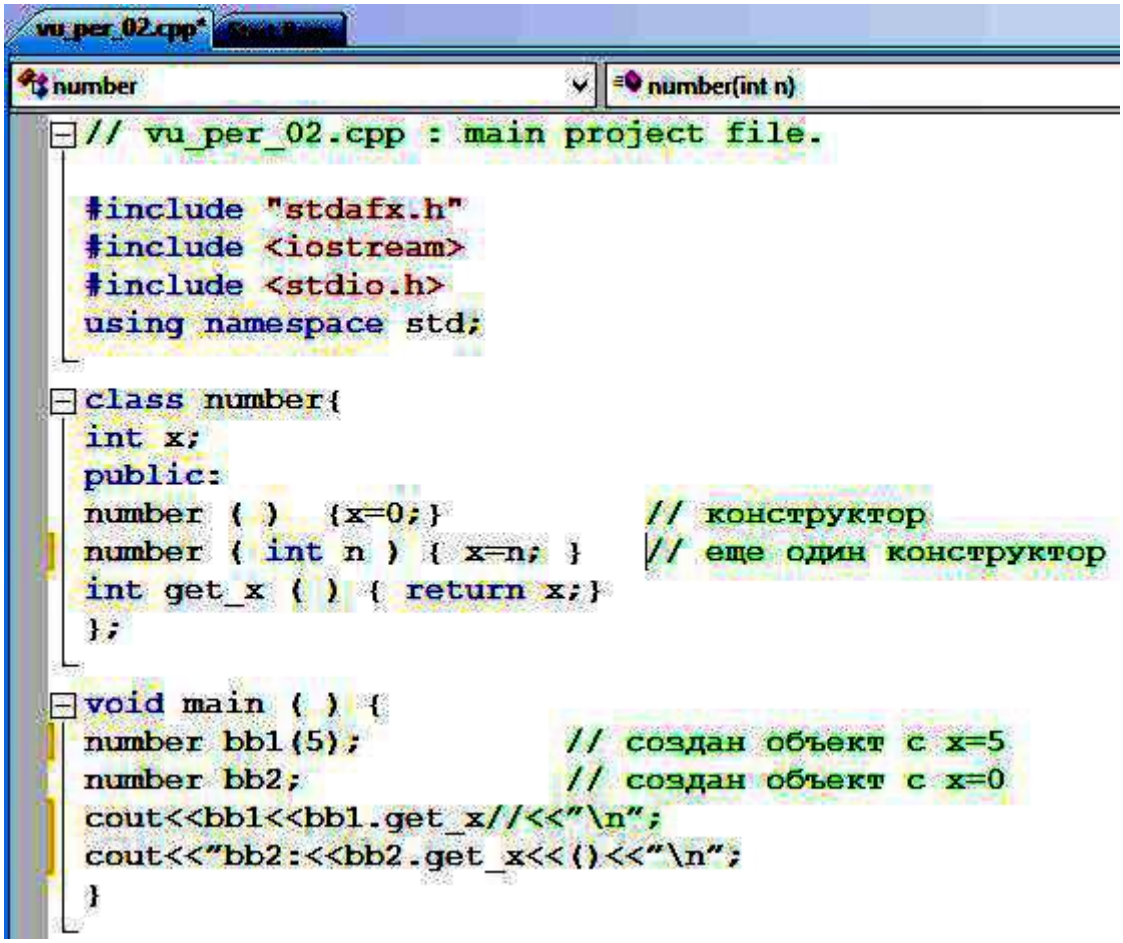


Рис. 9.2. Перегрузка конструкторов

9.3. ИСПОЛЬЗОВАНИЕ ПЕРЕГРУЖЕННЫХ КОНСТРУКТОРОВ ДЛЯ ИНИЦИАЛИЗАЦИИ МАССИВОВ ОБЪЕКТОВ

Массив объектов задается так же, как и массив любого другого типа данных. Например, для класса *number*, массив из трех объектов задается так:

numberobj[3];

Количество элементов массива указывается в прямоугольных скобках. Приведем пример инициализации массива объектов.

В примере на рис.9.3. используются два конструктора: один – с инициализацией, другой – без.

В функции *main* () создаются два массива по десять объектов.

Первый создается так, что у всех объектов x=0.

Для второго массива применяется инициализация с использованием конструктора с параметром.

В заключении на экран выводятся значения переменной x всех объектов.

В этом примере используются два конструктора: один – с инициализацией, другой – без.

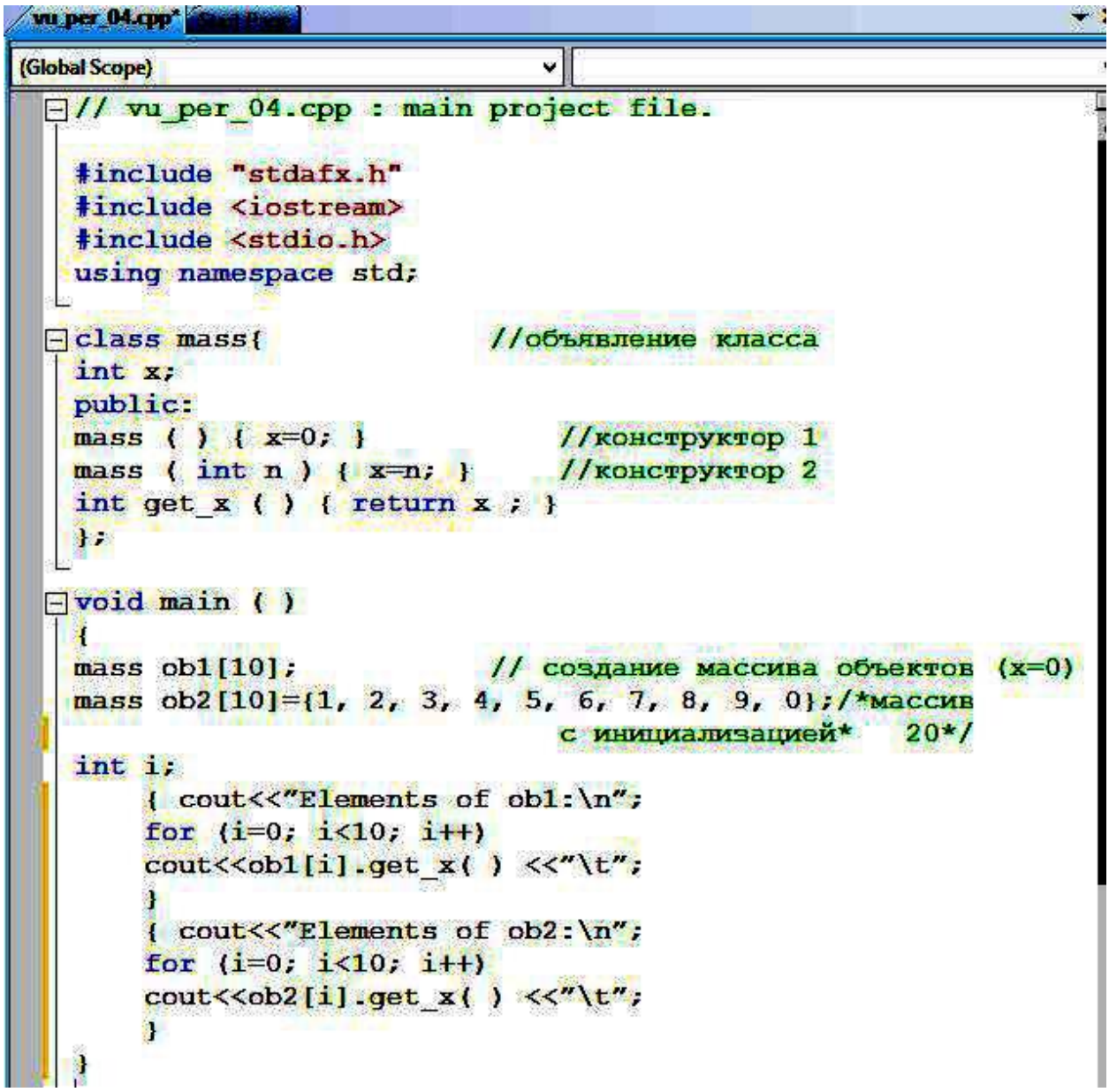


Рис. 9.3 – Перегрузка конструкторов для массивов

В функции *main* () создаются два массива по десять объектов. Первый создается так, что у всех объектов x=0. Для второго массива применяется инициализация с использованием конструктора с параметром.

В заключении на экран выводятся значения переменной x всех объектов.

Итак, отметим:

1. При наследовании из производного класса предоставляется возможность обращаться к членам базового класса, кроме его *private-членов*.

2. Для доступа к закрытым членам базового класса из производного класса, эти члены помечаются ключевым словом *protected.*

3. Полиморфизм позволяет использовать одно имя (оператор) для выполнения различных, но схожих по типу действий (процедур, функций).

Вопросы и задания.

1. Поясните, что такое дружественная функция.
2. Расскажите о свойствах ООП на Си++
    a. Наследование
    b. Инкапсуляция
    c. Полиморфизм
3. Что означает и как это происходит в случае
    a. Базовой класс наследуется как открытый
    b. Базовой класс наследуется как закрытый
    c. Базовой класс наследуется как защищенный
4. Повторите все примеры и добейтесь выполнения приведенных программ, используя компилятор VC++2005-2012. Объясните работу операторов и особенности программ.
5. Создайте программу, использующую классы

ЛИТЕРАТУРА

1. Бьерн Страуструп. Язык программирования C++. –М.: Бином, 2016 – 1136с.

2. Подбельский В.В. Стандартный Си++: учебное пособие /В.В. Подбельский. –М: Финансы и статистика, 2014. -688 с.

3. Б.Керриган, Д.Ритчи, Язык программирования Си, -М.: Вильямс, 2016,-288с.

4. Т.А.Павловская, Ю.А. Щупак, C/C++. Структурное и объектно-ориентированное программирование. Практикум. - СПб.:Питер, 2011 -352с.

5. Шилдт Г. C++ Шаг за шагом. – Издательство «ЭКОМ Паблишерз», 2013-640с.

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

КАФЕДРА ЭКОНОМИКИ И СТРАТЕГИЧЕСКОГО МЕНЕДЖМЕНТА

Кафедра образована 9 февраля 2015 в результате реорганизации, проводимой в Университете ИТМО, на базе кафедр «Прикладной экономики и маркетинга», «Экономической теории и бизнеса» гуманитарного факультета и «Экономической теории и экономической политики» факультета экономики и экологического менеджмента.

Кафедры экономической теории и бизнеса и прикладной экономики и маркетинга организованы в 90-х годах XX века, в ответ на резко возросшие запросы общества на специалистов в области экономики и ведения бизнеса.

Характерной чертой кафедры является разветвленная прикладная *научная деятельность*, возглавляемая и координируемая доктором экономических наук, профессором О.В. Васюхиным, признанным специалистом в области организации производственных структур, на счету которого 63 опытно-конструкторских разработки, одна из которых удостоена бронзовой медали ВДНХ в 1982 г. С 1990 под его непосредственным управлением выполнялись крупные хоздоговорные разработки по организации опытного производства для ряда Ленинградских НИИ, заводов и опытных производств, включая ГОИ им. С. Н. Вавилова и др.

Кафедра экономической теории и экономической политики (ЭТиЭП) создана в 2007 году решением ректората университета. С момента организации кафедрой заведовала доктор экономических наук, профессор Наталья Александровна Шапиро, автор более 170 научных и учебно-методических работ, действительный член международной общественной организации «Академия философии хозяйства» (на базе МГУ им. М.В. Ломоносова), сотрудник международной лаборатории «Устойчивое развитие и ресурсная эффективность в продуктовой цепочке», заместитель главного редактора научного журнала Университета ИТМО «Экономика и экологический менеджмент», включенного в перечень журналов ВАК. С 09.02.2015 года кафедра ЭТиЭП вошла в состав кафедры экономики и стратегического менеджмента.

Сегодня в составе кафедры работают 7 профессоров, 21 доцент, 11 старших преподавателей, 11 ассистентов, 3 тьютора. *Основные учебные курсы:* «Экономическая теория», «Региональная экономика», «Национальная

экономика», «Государственное регулирование национальной экономики», «Социальное и экономическое прогнозирование», «Экономика общественного сектора», «Информатика», «Корпоративные информационные системы в экономике», «Экономика защиты информации», «Предметно-ориентированные экономические информационные системы», «Экономическая оценка инвестиций», «Бизнес-планирование» и др.

Кафедра постоянно обновляет *учебно-методическую базу* путем разработки и внедрения в учебный процесс новых методических материалов (пособий, электронных учебников, тестовых программ для дистанционного обучения). Авторские курсы отображают мнение как специалистов промышленности, так и ученых родственных кафедр. Связь с внешними организациями позволяет обновлять содержательную часть читаемых профессиональных дисциплин в пределах 10-15% ежегодно.

Обучение проводится на основе *материально-технической базы факультета. Компьютерные классы меж*кафедральной лаборатории и собственные ресурсы кафедры насчитывают несколько десятков новейших компьютеров. В учебном процессе активно используется INTERNET на основе постоянно действующего сервера, подключенного по оптоволоконным линиям связи. Сетевой класс также подключен к университетской сети RUNNET.

Кафедра проводит *научные исследования и разработки* по направлениям:

- анализ, обоснование и разработка антикризисных и стабилизационных программ в национальной экономике;
- регулирование производственной деятельности на разных уровнях управления;
- планирование кадровых стратегий и научных исследований, проводимых в соответствующих организациях;
- организационное проектирование предприятий;
- анализ и моделирование деятельности инновационно-ориентированных организаций, обоснование инновационных проектов.

В результате обширной научной деятельности кафедра установила и поддерживает эффективное сотрудничество с родственными кафедрами российских вузов, в том числе СПбГУ, СПбУЭиФ, СПбГУКиТ, СПбГМТУ, МУСЭИ, РЭУ им Г.В. Плеханова и другими. За последние 5 лет сотрудники кафедры участвовали в 50 международных и отечественных конференциях, 4 раза выезжали в научные командировки по приглашениям зарубежных партнеров. Опубликовано более 10 монографий, более 90 научных публикаций, в том числе учебных пособий и методических работ.

Основные цели развития кафедры: совершенствование методов научно-исследовательской работы студентов, аспирантов, научно-педагогических работников; организация и осуществление широкого круга прикладных научных исследований; развитие устойчивых связей с зарубежными партнерами.

КАФЕДРА БЕЗОПАСНЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра Безопасные информационные технологии осуществляет подготовку специалистов по специальности 075300 — Организация и технология защиты информации. Квалификация: специалист по защите информации. Кафедра является базовой кафедрой Государственной технической комиссии при Президенте Российской Федерации и функционирует на базе органа по сертификации и аттестации по требованиям безопасности информации Научно-технического центра "Критические информационные технологии".

На кафедре БИТ осуществляется выполнение научно-исследовательских работ в области безопасности и защиты информации телекоммуникационных вычислительных систем:

- Аттестация объектов информатизации.
- Кафедра БИТ является о разработчиком Концепции информационной безопасности региона, разработчиком Концепции защиты информации в банковских информационных технологиях.
- Кафедра БИТ имеет большой опыт эксплуатации и внедрения средств защиты сетевого взаимодействия пользователей корпоративных сетей при подключении к общедоступным каналам связи (в том числе Internet). Для обеспечения безопасного взаимодействия клиентов и защиты ресурсов собственной корпоративной сети предлагаются высокоэффективные решения на базе сертифицированных Гостехкомиссией России средств защиты информации. К таковым относятся межсетевые экраны, функционирующие на базе различных UNIX-платформ, защищенные программные маршрутизаторы, позволяющие организовывать виртуальные частные сети и ряд других средств, обеспечивающих безопасное сетевое взаимодействие. Указанные средства имеют гибкие механизмы настройки под определенную конфигурацию сети и требования пользователей. Специалисты кафедры БИТ проводили сертификационные испытания множества сетевых средств защиты, что позволяет им наиболее полно использовать достоинства каждой из систем в целях достижения максимальной степени защищенности и удобства для Заказчика.
- Специалисты кафедры БИТ ведут научные работы по созданию новой технологической схемы защиты информации от вирусных воздействий, по созданию иммунной системы защиты, методические

исследования в области проведения сертификационных испытаний средств защиты и автоматизированных систем обработки информации с имеющимися системами защиты. Специалисты кафедры БИТ являются авторами некоторых основополагающих Руководящих Документов Гостехкомиссии России.

Результаты научных работ и исследований, полученные специалистами кафедры БИТ, неоднократно обсуждались в научных статьях, опубликованы в ряде российских научно-технических и специализированных журналах, а также докладывались на многих конференциях, посвященных вопросам обеспечения информационной безопасности как в России, так и за рубежом.

Кафедрой заключены договора о сотрудничестве и ведутся работы с ведущими западными фирмами в области информационных технологий (Oracle, CyberGuard, Ericsson, Internet Security Systems, Microsoft).

Возглавлял кафедру БИТ Осовецкий Л.Г., доктор технических наук, профессор, академик Международной Академии Информатизации, лауреат Государственной Премии СССР.

В 2010 году на должность заведующего кафедрой избран профессор, доктор технических наук Зикратов Игорь Алексеевич.

Илья Сергеевич Лебедев
Вадим Юрьевич Петров

Информатика. Программирование

Учебно-методическое пособие. Часть 2.

В авторской редакции
Дизайн В.Ю.Петров
Верстка В.Ю.Петров
Редакционно-издательский отдел Санкт-Петербургского государственного университета информационных технологий, механики и оптики
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати _____
Заказ № _____
Тираж 100
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО

197101, Санкт-Петербург, Кронверкский пр., 49