Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра «Вычислительные методы и программирование»

В.Л. Бусько, А.Г. Корбит, Т.М. Кривоносова

ОСНОВЫ ОБЪЕКТНО-ОРИЕТИРОВАННОГО ПРОГРАМИРОВАНИЯ. C++

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

для студентов всех специальностей и форм обучения БГУИР

Под общей редакцией В.Л. Бусько

Минск 2005

ДК 681.3.06 (075.8)
ББК 32.973.26-018.1 я 73
Б 92

Бусько В.Л.
Б 92 Основы объектно-ориентированного программирования. С++: Лабораторный практикум для студ. всех спец. и форм обуч. БГУИР /В.Л. Бусько, А.Г. Корбит, Т.М. Кривоносова; Под общ. ред. В.Л. Бусько. – Мн.: БГУИР, 2005. – 38 с.: ил.
ISBN 985-444-809-6

В работе приведены краткие теоретические сведения по основам объектно-ориентированного программирования на языке С++. В практикум вошло 8 лабораторных работ и даны по 15 индивидуальных заданий к каждой из них.

УДК 681.3.06 (075.8)
ББК 32.973.26-018.1 я 73

ISBN 985-444-809-6

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1.
Введение в пользовательский тип данных «Классы»

Цель работы: изучить основные способы работы с пользовательским типом данных «Классы», составляющим одну из основных концепций объектно-ориентированного программирования (ООП).

Краткие теоретические сведения

ООП – методология, основанная на представлении программ в виде совокупности объектов, которые между собой взаимодействуют, причем каждый объект является реализацией конкретного класса. То есть основным элементом ООП является класс.

Класс – это тип данных, вводимый пользователем. Классы, как правило, организованы иерархически. Основное назначение класса – описание состава, основных свойств и поведения будущих объектов этого типа данных (введение в проекте некоей абстракции). При этом каждый класс имеет общедоступную часть – интерфейс, а также недоступную, скрытую от пользователя личную часть или реализацию, которая и представляет внутреннее строение будущих объектов данного типа.

Таким образом, в ООП ***абстрагирование*** – процесс введения типа данных «класс», т.е. таких существенных характеристик некоторых будущих объектов, которые и будут отличать их от других видов объектов.

Общий формат декларации типа данных «класс» следующий:

```
вид_класса  ID_класса (идентификатор данного типа данных)  {
        элементы-данные;                    //  Далее – просто «данные»
        элементы-функции;                   //  Далее – просто «методы»
        } ;
```

Рассмотрим краткую характеристику элементов, составляющих описание класса.

Атрибут «вид_класса»

Виды типа данных «класс»: ***union*** ***struct*** ***class***

Атрибуты доступа

Элементы, входящие в шаблон класса, могут иметь следующую степень защищенности:

public – глобальный, общедоступный атрибут доступа; в этом случае элементы общедоступны из любой функции проекта;

private – локальный, частный; элементы закрыты от общего доступа, и с ними могут работать только методы класса;

protected – защищенные элементы класса, к ним имеют доступ методы данного класса и методы классов, производных от данного класса.

Как правило, закрытые элементы – это внутреннее строение объектов, которое нас не должно интересовать.

Все три вида класса имеют степень защиты элементов, которую компилятор установит по умолчанию.

Вид *union* – степень защиты *public*, управлять ею нельзя.

Вид *struct* – степень защиты *public*, управлять защитой можно.

Вид *class* – *private*, степенью защиты также можно управлять.

Для управления защитой элементов, входящих в состав класса, необходимо явно указать нужный атрибут, после которого добавить символ **:** (двоеточие).

Существует два способа создания объектов с таким составом данных.

1. Между символами «}» и «;» помещается список идентификаторов объектов, и уже на этапе компиляции будут созданы объекты с указанным типом данных. При таком способе задания атрибут *ID_класса* можно опускать.

2. Декларация объектов класса в любом месте программы по мере надобности:

ID_класса список ID объектов;

Данные, входящие в состав класса, – любой допустимый вид данных языка C++, за исключением файлов.

Методы, входящие в состав класса, можно разделить на три группы.

1. Методы, определяющие операции над данными.

2. Функции-конструкторы, основная задача которых – создание и инициализация объектов данного класса разнообразными способами.

3. Функции для уничтожения объектов после их использования – деструкторы.

Внешние функции, не входящие в состав класса, могут работать только с элементами класса, имеющими атрибут доступа *public*.

Декларация методов класса возможна в двух формах:

– в форме определения, при которой приводится полный текст метода;

– в форме описания (прототип метода). В данном случае вне шаблона должно быть полное определение метода, при этом необходимо указать, к какому классу принадлежит данный метод. Для этого используется операция привязки «**::**». Назначение этой операции – привязать функцию к конкретному классу.

Использование этой операции вводит новое определение:

Полное квалификационное имя

Запись ***X* :: *a*** означает: «Элемент ***a*** принадлежит классу ***X***».

Запись ***X* :: *f*1()** означает: «Функция ***f*1** принадлежит классу ***X***».

И в заключение рассмотрим некоторые приемы работы объектами и методами.

Пусть в процессе работы программы создан объект класса

***ID_класса ID_объекта*;**

и пусть в составе класса есть метод с идентификатором *ID_метода*.

Тогда:

1. *Прямой вызов метода*

ID_объекта . ID_метода;

– использована операция привязки «точка».

2. *Использование косвенной адресации*

Пусть объявлен указатель с типом *ID_класса*:

ID_класса *ID_указателя;

Теперь указатель типа *ID_класса* можно устанавливать на существующие объекты этого класса:

ID_указателя = & ID_объект;

– указатель идентифицирован адресом объекта.

Косвенный вызов метода:

ID_указателя -> ID_метода;

– использована операция привязки «стрелка».

3. *Создание объектов* (безымянных) *в динамической области памяти*

ID_класса *ID_указателя = new ID_класса;

Компилятор создает объект на этапе работы проекта.

Косвенный вызов метода:

ID_указателя -> ID_метода;

Задание к лабораторной работе № 1

Общая постановка. Написать программу, иллюстрирующую прямой и косвенный способы обращения к методам. Пользовательский класс должен содержать необходимые элементы-данные, метод установки их начальных значений:

void Set (double X, …);

метод печати:

void Print (void);

метод, решающий поставленную задачу:

void Run (void);

Коды методов – вне пространства определения класса.

Пример линейного алгоритма

Вычислить $$h = \frac{x^{y+1} + e^{y-1}}{1 + x * |y - tgz|} * (1 + |y - x|) + \frac{|y - x|^2}{2} - \frac{|y - x|^3}{3},$$

при x = 2,444, y = 0,00869, z = –130 должно быть получено – 0,49871.

Текст метода ***Run*** может иметь следующий вид:

```
. . .
class X{
```

```
. . .
pulic:
void Set(double X, …);
void Print(void);
void Run(void);
};
. . .
#include <math.h>
    void X::Run(void)  {
    double dop,a,b,c;
        puts(" Работа 1 – Линейный алгоритм ");
        dop = fabs(y–x);
        a = pow(x,y+1) + exp(y–1);
        b = 1 + x*fabs(y–tan(z));
        c = 0.5*pow(dop,2) – pow(dop,3)/3;
        rezult = a/b*(1+dop) + c;
        }
```

где *x*, *y*, *z*, *rezult* – вещественные элементы-данные класса *X*.

Индивидуальные задания

Составить метод ***Run*** для вычисления выражения.

1. $t = \dfrac{2\cos\left(x - \dfrac{\pi}{6}\right)}{0.5 + \sin^2 y}\left(1 + \dfrac{z^2}{3 - z^2/5}\right)$

При $x = 14{,}26$, $y = -1{,}22$, $z = 3{,}5\times10^{-2}$ $\boldsymbol{t = 0{,}564849.}$

2. $u = \dfrac{\sqrt[3]{8 + |x-y|^2} + 1}{x^2 + y^2 + 2} - e^{|x-y|}\left(\mathrm{tg}^2 z + 1\right)^x.$

При $x = -4{,}5$, $y = 0{,}75\times10^{-4}$, $z = 0{,}845\times10^{2}$ $\boldsymbol{u = -55{,}6848.}$

3. $v = \dfrac{1 + \sin^2(x+y)}{\left|x - \dfrac{2y}{1 + x^2y^2}\right|}x^{|y|} + \cos^2\left(\mathrm{arctg}\dfrac{1}{z}\right).$

При $x = 3{,}74\times10^{-2}$, $y = -0{,}825$, $z = 0{,}16\times10^{2}$, $\boldsymbol{v = 1{,}0553.}$

4. $w = \left|\cos x - \cos y\right|^{\left(1 + 2\sin^2 y\right)}\left(1 + z + \dfrac{z^2}{2} + \dfrac{z^3}{3} + \dfrac{z^4}{4}\right).$

При $x = 0{,}4\times10^{4}$, $y = -0{,}875$, $z = -0{,}475\times10^{-3}$ $\boldsymbol{w = 1{,}9873.}$

5. $\alpha = \ln\left(y^{-\sqrt{|x|}}\right)\left(x - \dfrac{y}{2}\right) + \sin^2 \mathrm{arctg}(z).$

При $x = -15{,}246$, $y = 4{,}642\times10^{-2}$, $z = 20{,}001\times10^{2}$ a $\boldsymbol{= -182{,}036.}$

6. $\beta = \sqrt{10\left(\sqrt[3]{x} + x^{y+2}\right)}\left(\arcsin^2 z - |x-y|\right)$.

При $x = 16{,}55\times10^{-3}$, $y = -2{,}75$, $z = 0{,}15$ b = **– 40,630.**

7. $\gamma = 5\,\mathrm{arctg}(x) - \frac{1}{4}\arccos(x)\frac{x + 3|x-y| + x^2}{|x-y|z + x^2}$.

При $x = 0{,}1722$, $y = 6{,}33$, $z = 3{,}25\times10^{-4}$ g = **–205,305.**

8. $\varphi = \frac{e^{|x-y|}|x-y|^{x+y}}{\mathrm{arctg}(x) + \mathrm{arctg}(z)} + \sqrt[3]{x^6 + \ln^2 y}$.

При $x = -2{,}235\times10^{-2}$, $y = 2{,}23$, $z = 15{,}221$ j = **39,374.**

9. $\psi = \left| x^{\frac{y}{x}} - \sqrt[3]{\frac{y}{x}} \right| + (y-x)\frac{\cos y - \frac{z}{(y-x)}}{1 + (y-x)^2}$.

При $x = 1{,}825\times10^{2}$, $y = 18{,}225$, $z = -3{,}298\times10^{-2}$ y = **1,2131.**

10. $b = y^{\sqrt[3]{|x|}} + \cos^3(y)\frac{|x-y|\left(1 + \frac{\sin^2 z}{\sqrt{x+y}}\right)}{e^{|x-y|} + \frac{x}{2}}$.

При $x = 6{,}251$, $y = 0{,}827$, $z = 25{,}001$ ***b* = 0,7121.**

11. $c = 2^{\left(y^x\right)} + \left(3^x\right)^y - \frac{y\left(\mathrm{arctg}(z) - \frac{\pi}{6}\right)}{|x| + \frac{1}{y^2+1}}$.

При $x = 3{,}251$, $y = 0{,}325$, $z = 0{,}466\times10^{-4}$ ***c* = 4,25.**

12. $f = \frac{\sqrt[4]{y + \sqrt[3]{x-1}}}{|x-y|\left(\sin^2 z + \mathrm{tg}\, z\right)}$.

При $x = 17{,}421$, $y = 10{,}365\times10^{-3}$, $z = 0{,}828\times10^{5}$ ***f* = 0,33056.**

13. $g = \frac{y^{x+1}}{\sqrt[3]{|y-2|} + 3} + \frac{x + \frac{y}{2}}{2|x+y|}(x+1)^{-1/\sin z}$.

При $x = 12{,}3\times10^{-1}$, $y = 15{,}4$, $z = 0{,}252\times10^{3}$ ***g* = 82,8257.**

14. $h = \frac{x^{y+1} + e^{y-1}}{1 + x|y - tgz|}\left(1 + |y-x|\right) + \frac{|y-x|^2}{2} - \frac{|y-x|^3}{3}$.

При $x = 2{,}444$, $y = 0{,}869\times10^{-2}$, $z = -0{,}13\times10^{3}$ ***h* = – 0,49871.**

15. $b = y^{\sqrt[3]{|x|}} + \cos^3(y)\dfrac{|x-y|\left(1+\dfrac{\sin^2 z}{\sqrt{x+y}}\right)}{e^{|x-y|}+\dfrac{x}{2}}.$

При x = 6,251, y = 0,827, z = 25,001 $\boldsymbol{b}$ **= 0,7121.**

Контрольные вопросы

1. Что в ООП означает понятие «класс» и что входит в его состав?
2. Перечислите виды классов, и укажите их различия.
3. Укажите способы управления защитой элементов, входящих в состав класса.
4. Что такое операция привязки, ее основное назначение?
5. Дайте понятие полного квалификационного имени.

ЛАБОРАТОРНАЯ РАБОТА № 2.
Пользовательский тип данных «Классы».
Использование конструктора и деструктора.
Работа с динамической памятью

Цель работы: изучить основные способы работы по созданию конструкторов с захватом динамической памяти и деструкторов для ее освобождения.

Краткие теоретические сведения

Для создания и инициализации объекта используется функция, получившая название ***Конструктор***, которая определяет (контролирует), как создается и инициализируется объект.

После использования объектов их необходимо уничтожить. Для этого предназначена функция – ***Деструктор***, которая определяет, как и когда уничтожить объект.

Если этих методов явно в программе нет, компилятор при создании экземпляра класса вызовет стандартный конструктор, который и создаст объект после его декларации. После выхода из функции (из блока), в которой был создан объект, компилятор вызовет деструктор «по умолчанию», который его уничтожит. Если же объект создавался как глобальный, деструктор будет вызван по завершении работы программы.

Некоторые особенности конструктора и деструктора

1. Эти функции не имеют возвращаемого значения (даже *void*).
2. Они не наследуются производными классами, хотя и вызываются из них.

3. Нельзя работать с адресами этих функций.

4. К конструктору нельзя обратиться напрямую как к обычному методу.

5. Деструктор можно вызывать, используя его полное квалификационное имя.

Рассмотрим эти два метода более подробно.

Конструктор

Конструктор – функция-член класса, определяющая способ создания объекта. Имя конструктора должно совпадать с именем класса.

Существует несколько форм конструктора. Наиболее часто используется конструктор с параметрами. Эти параметры используются для одновременной инициализации создаваемых объектов. Например:

```
. . .
class X {
        int m;          // Будущие объекты – целочисленные переменные
        public:
// Конструктор с параметром, через который будущим объектам будут
// присваиваться конкретные значения
        X (int i) {
                m = i;
                }
// Метод переустановки значений существующих объектов
        void Set (int i) {
                m = i;
                }
// Метод просмотра текущего состояния существующих объектов
        void Print (void) {
                cout << “ m = ” << m  << endl;
                }
                . . .
};
                . . .
        X a1 (5);               // Явный вызов конструктора: объект создан
                                // и инициализирован значением 5
        a1. Print();            // Просмотр текущего значения
        a1. Set(8);             // Изменение текущего значение объекта на 8
        a1. Print();            // Просмотр нового текущего значения
                . . .
```

Конструктор – это специальная функция класса, поэтому допускается вторая форма его декларации: в виде прототипа и его определения вне пространства класса. Конструктор как функция может иметь умалчиваемые значения параметров. В программе может быть несколько конструкторов. Такие конструкторы часто называют конструкторы с перекрытием. Здесь работает механизм перегрузки функций.

Деструктор

Деструктор – специальная функция класса, которая отвечает за уничтожение объекта. Имя деструктора совпадает с именем класса, перед которым ставится символ «тильда» – ~.

Если декларирован

class X { . . . } ;

то деструктор для него ~X (){}

То есть если элементы-данные класса имеют известный размер в байтах, например: *int m*; (2 байта), *double x*; (4 байта) и т.д. Компилятор при создании объектов выделит соответствующие участки памяти, и в этом случае деструктор не нужен, т.к. он будет иметь пустое определение.

Если же объект создается в динамической области памяти, например, с помощью операции захвата памяти *new*, то деструктор необходимо задавать явно, т.к. он должен уничтожить объект после его использования с помощью операции *delete*. Рассмотрим кратко эти две операции.

Операции new и delete (C++)

Управление программным размещением объектов в памяти, т.е. выделение памяти в процессе работы программы под переменные и массивы в языке C++, осуществляется с помощью операций *new* и *delete*.

Операция *new* автоматически определяет размер выделяемой памяти для указанного типа, а также позволяет инициализировать вновь создаваемый объект.

Формат операций:

ID_указателя = ***new*** *тип* (*значение*); // Захват памяти

. . .

(работа с динамическим объектом через косвенную адресацию – операция «*»)

delete *ID_указателя*; // Освобождение памяти

// после работы с объектом

где *ID_указателя* – идентификатор переменной-указателя на заданный тип;

тип – тип значений, для которых выделяется память;

значение – константа (необязательный атрибут), определяющая начальное значение динамической переменной.

Таким образом, операция *new* устанавливает указатель на участок свободной динамической оперативной памяти размером *sizeof*(*тип*). Если выделить память не удается, то возвращается нулевой указатель, что дает возможность проконтролировать процесс захвата участка динамической памяти.

Объект существует до тех пор, пока память не будет освобождена при помощи операции *delete* (или до окончания работы программы).

Следующий участок программы демонстрирует работу с целочисленной динамической переменной.

```
. . .
int *p;
if ( ! (p = new int (5))) {
    printf (" Ошибка \n"); getch();
    exit (1);
```

```
  }
*p = 66;
printf  ("\n Динамическая переменная = %d  ", *p); getch();
delete p;
        . . .
```

Создание динамических массивов и работа с ними будут рассмотрены далее.

Теперь можно видоизменить рассмотренный ранее пример, включив в него работу с динамической памятью, применяя косвенную адресацию, а также добавив метод увеличения текущего состояния объекта в два раза:

```
class X {
int *m;   // Будущие объекты – целочисленные динамические переменные
public:
// Конструктор с параметром, через который будущим объектам будут
// присваиваться конкретные значения
X (int i) {
        m = new int(i);
        }
// Деструктор
~X(){
        delete m;
        }
// Метод переустановки значений существующих объектов
void Set (int i) {
        *m = i;
        }
// Метод просмотра текущего состояния существующих объектов
void Print (void) {
        cout << “ m = ” << *m << endl;
        }
void run(void){
        *m *= 2;
        }
        . . .
};
        . . .
X a1 (5);           // Явный вызов конструктора: объект создан
                    // и инициализирован значением 5
a1. Print();        // Просмотр текущего значения
a1. Set(8);         // Изменение текущего значения объекта на 8
a1. Print();        // Просмотр нового текущего значения
a1.run();           // Решение поставленной задачи
a1. Print();        // Просмотр нового значения
        . . .
```

Задание к лабораторной работе № 2

Общая постановка. Разработать программу, иллюстрирующую прямой и косвенный способы обращения к методам. Пользовательский класс ***X*** должен содержать необходимые элементы-данные, которые создаются в динамической области памяти. Конструктор с параметрами (исходные значения по умолчанию) для создания объектов (*new*) и установки их начального состояния: ***X* (…);**

Деструктор: **~ *X* ();**
Метод печати текущего состояния: ***void Print* (…);**
Метод переустановки текущего состояния: ***void Set* (…);**
Функция, решающая поставленную задачу: ***void Run* (…);**
Код методов – вне пространства определения класса.

Индивидуальные задания

Составить метод ***Run*** для вычисления выражения, приведенного в лабораторной работе № 1.

Контрольные вопросы

1. Что такое «конструктор», какие его основные особенности?
2. Что такое «деструктор», какие его основные особенности?
3. Когда наличие деструктора обязательно?
4. Назначение операций *new* и *delete*.
5. Сколько существуют динамические объекты?

ЛАБОРАТОРНАЯ РАБОТА № 3.
Класс «Одномерный динамический массив»

Цель работы: изучить создание одномерных динамических массивов при помощи конструкторов с захватом динамической памяти и деструкторов для их уничтожения.

Краткие теоретические сведения

Как вы уже знаете, при стандартной декларации массивов компилятор воспользуется векторной организацией памяти и выделит фиксированный участок памяти, достаточный для хранения всех его элементов. Использование динамической памяти и списковой ее организации позволяет создавать массивы переменной длины. Помимо уже известных операций по захвату и освобождению динамической памяти *new* и *delete* для этих целей используют стандартные библиотечные функции, декларацию которых содержит заголовочный файл ***alloc.h*.**

Приведем сведения о наборе функций манипулирования памятью:

void ***malloc*** (unsigned *n*); – выделение памяти для размещения блока размером *n* байт; возвращает указатель на распределенную область или *NULL* при неудаче;

void ***calloc*** (unsigned *n*, unsigned *size*); – выделение памяти для размещения *n* объектов размером *size* байт и заполнение полученной области нулями; возвращает указатель на захваченную область памяти или *NULL* при неудаче;

unsigned ***coreleft*** (void); – получение размера свободной памяти в байтах:

void ***free***(void *b); – освобождение блока памяти, адресуемого указателем *b*;

void ***realloc*** (void *b, unsigne *n*); – изменение размера размещенного по адресу *b* блока на новое значение *n* и копирование (при необходимости) содержимого блока; возвращает указатель на перераспределенную область памяти или *NULL* при неудаче.

Пример выделения памяти для массива действительных чисел размером *n*:

```
        . . .
double *x;
int n;                                  // Количество элементов массива
        . . .
x = (double*)calloc(n, sizeof(double));  // Захват памяти для n элементов
        . . .
free(x);                                // Освобождение памяти
        . . .
```

Общий формат операций *new* и *delete* для работы с динамической памятью следующий:

```
type *name;                 // Декларировали указатель на начало массива
        . . .
name = new  type[size];     // Захватили память
        . . .
delete [ ] name;            // Освободили память
```

где *type* – тип элементов,

size – максимальное количество элементов массива *name.*

Задание к лабораторной работе № 3

Общая постановка. Пользовательский класс *Array* должен содержать конструктор с параметром для создания динамических целочисленных массивов (операция *new* или стандартная библиотечная функция *calloc*) и установки начальных значений их элементов: ***Array*(…)** (реальный размер массива передается через параметр);

Деструктор: **~ *X* ();**

Метод печати текущего состояния массива: ***void Print*(…);**

Метод переустановки текущего состояния массива: ***void Set*(…);**

Функция, решающая поставленную задачу: ***void Run*(…);**

Код методов – вне пространства определения класса. Программа иллюстрирует косвенный способ обращения к элементам массива.

Индивидуальные задания

Составить метод ***Run***, который позволит выполнить следующие действия с одномерными массивами (если задачу решить нельзя, то сообщить об этом).

1. Найти произведение элементов массива, расположенных между максимальным и минимальным элементами.
2. Найти произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.
3. Найти сумму элементов массива, расположенных после минимального элемента.
4. Найти сумму модулей элементов массива, расположенных после минимального по модулю элемента.
5. Найти сумму элементов массива, расположенных до минимального элемента.
6. Найти сумму элементов массива, расположенных после первого положительного элемента.
7. Преобразовать массив так, чтобы сначала располагались элементы, целая часть которых лежит в интервале [a, b], а потом – все остальные.
8. Преобразовать массив так, чтобы сначала располагались отрицательные элементы, а потом – положительные (0 считать положительным).
9. Найти сумму элементов массива, расположенных после максимального.
10. Заменить все отрицательные элементы массива их квадратами.
11. Найти сумму элементов массива, расположенных между первым и вторым отрицательными элементами.
12. Преобразовать массив так, чтобы в первой половине располагались элементы, стоявшие в нечетных позициях, а во второй половине – в четных.
13. Преобразовать массив таким образом, чтобы элементы, равные нулю, располагались после всех остальных.
14. Сжать массив, удалив из него элементы, величина которых находится в интервале [a, b]. Освободившиеся в конце массива элементы заполнить нулями.
15. Найти сумму элементов массива, расположенных после минимального.

Контрольные вопросы

1. Формат и назначение библиотечной функции *malloc*().
2. Формат и назначение библиотечной функции *calloc*().
3. Формат и назначение библиотечной функции *free*().
4. Как создать и уничтожить динамический одномерный массив при помощи операций?
5. Как программно контролируется захват памяти под массив?

ЛАБОРАТОРНАЯ РАБОТА № 4.
Класс «Двухмерный динамический массив»

Цель работы: изучить методику создания и уничтожения двухмерных динамических массивов при помощи конструкторов с захватом динамической памяти и деструкторов для ее освобождения. Научиться работать с классом через функции-друзья этого класса.

Краткие теоретические сведения

Создание двухмерного динамического массива

Учитывая тот факт, что имя двухмерного массива – это указатель на указатель *int **m*; двухмерный динамический массив создается и уничтожается за два шага. Например, требуется создать целочисленный массив *m*[3][4]:

```
m = new  int* [3];              // Захват памяти для указателей – рис. а
for ( int i=0; i<3; i++)        // Захват памяти для элементов – рис. б
      m[i] = new  int [4];
      . . .
for ( i=0; i<3; i++)        // Обработка элементов, например обнуление
  for ( j=0; j<4; j++)
      m[i][j] = 0;          // или  *(*(m+i)+j) = 0;
      . . .
for ( i=0; i<3; i++)        // Освобождение памяти, занятой под элементы
      delete [ ] m[i];
delete [ ] m;               // Освобождение памяти, занятой под указатели
```

В динамической памяти была создана такая конструкция:

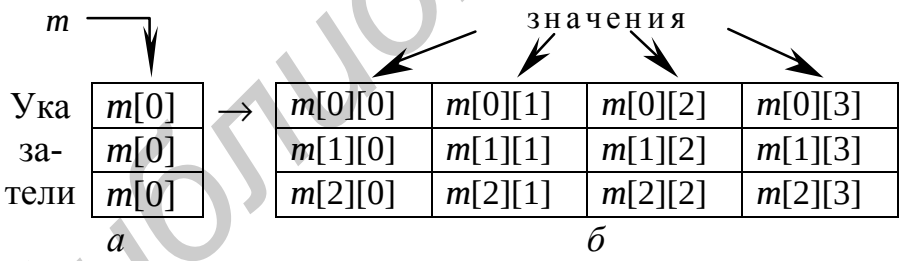

Рис. Схема размещение двухмерного массива в памяти:
а – размещение указателей, *б* – размещение элементов

Участок программы, в котором для создания двухмерного динамического массива используются библиотечные функции:

```
            . . .
int **a;
puts("\n Input n,m: ");
scanf("%d %d", &n, &m);
      a=(int**)calloc(n, sizeof(int*));          // Захват памяти
       for(i=0; i<n; i++)
            a[i]=(int *)calloc(m, sizeof(int));
```

```
        . . .
for(i=0; i<n; i++)   free(a[i]);                    // Освобождение памяти
free(a);
        . . .
```

Краткая характеристика функции-друга класса

Механизм «функции-друга» класса обеспечивает возможность доступа к любым элементам класса тем функциям, которые не входят в состав этого класса. То есть функция-друг класса – это обычная функция пользователя, которая благодаря данному механизму имеет полные права доступа ко всем без исключения элементам класса, независимо от их степени защищенности. Чтобы обычная функция стала другом класса, в заголовке ее необходимо указать ***friend***.

Некоторые особенности:

1. Функцию-друга класса необходимо декларировать в описании этого класса. Декларация возможна в двух формах:

– в форме описания (приводится прототип), а ее полное определение – вне шаблона класса, при этом не нужно указывать операцию привязки к данному классу, т.к. она не является членом данного класса;

– в форме полного определения, в шаблоне класса приводится ее полный текст.

2. При обращении к функции-другу не надо использовать операцию привязки (**.**) или (**->**).

3. Так как функция-друг – обычная функция, у нее отсутствует первый скрытый параметр *this*.

Для обращения элементам класса из функции-друга используют или ссылку, или указатель, что и иллюстрирует следующий пример:

```
        . . .
class X {
        int a, b;
        public:
        void Print(void);                   // Метод печати
        friend void f1Set (X &);
        friend void f2Set (X *);
        };
void X :: Print (void) {
                cout << " a = " << a << " b = " << b << endl;
                        }
// f1Set() и f2Set() – обычные функции, поэтому операция привязки не нужна
        void f1Set (X &r) {
                r.a = 1; r.b = 2;
                        }
        void f2Set (X *p) {
                p -> a = 3; r -> b = 4;
                        }
        void main(void)  {
```

```
X  x1;                          // Установили значение
f1Set (x1);    x1.Print();
f2Set (&x1);  x1.Print();
}
```

Отметим, что функции-друзья класса кроме ссылки или указателя на объекты класса могут иметь обычные параметры, например для инициализации данных объектов класса. Например, функция-друг класса *X* f2Set() может иметь следующий вид:

```
void f2Set (X *p, int i, int j) {
        p -> a = i; r -> b =j;
              }
```

а обращение к ней: f2Set (&x1, 5, 6);

Задание к лабораторной работе № 4

Общая постановка. Пользовательский класс *Array* должен содержать конструктор с параметрами для создания динамических целочисленных массивов (операция *new* или стандартная библиотечная функция *calloc*) и установки начальных значений их элементов: ***Array*(…)** (реальные размеры массива – число строк и столбцов передается в конструктор через параметры);

Деструктор: ***~ X* ();**

Метод печати текущего состояния массива: ***void Print*(…);**

Метод переустановки текущего состояния массива: ***void Set*(…);**

Функция-друг, решающая поставленную задачу: ***friend void Run*(…);**

Код методов и функции-друга – вне пространства определения класса. Программа иллюстрирует косвенный способ обращения к элементам массива

Индивидуальные задания

Составить функцию ***Run***, которая позволит выполнить следующие действия с двухмерными массивами (если задачу решить нельзя, то сообщить об этом).

1. Определить сумму элементов в тех строках, которые не содержат отрицательных элементов.
2. Определить количество строк, содержащих хотя бы один нулевой элемент.
3. Определить номер первой из строк, не содержащих ни одного положительного элемента.
4. Определить номер первого из столбцов, не содержащих ни одного отрицательного элемента.
5. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.
6. Определить номер первой из строк, содержащих хотя бы один положительный элемент.
7. Определить количество строк, среднее арифметическое элементов которых меньше заданной величины.

8. Найти сумму модулей элементов, расположенных выше главной диагонали.

9. Определить количество строк, не содержащих ни одного нулевого элемента.

10. Определить максимальное из чисел, встречающихся в заданной матрице более одного раза.

11. Определить количество столбцов, не содержащих ни одного нулевого элемента.

12. Найти произведение элементов в тех строках, которые не содержат отрицательных элементов.

13. Определить сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

14. Определить сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.

15. Найти произведение элементов в тех строках, которые не содержат элементов, кратных 3.

Контрольные вопросы

1. Чем является имя двухмерного массива?
2. Как косвенно обратиться к элементу двухмерного массива?
3. Объясните понятие «функция-друг» класса?
4. Какие существуют способы обращения к элементам класса из функции-друга?

ЛАБОРАТОРНАЯ РАБОТА № 5.
Класс «Динамическая строка» и перегрузка операций

Цель работы: изучить методику по созданию одномерных динамических символьных массивов при помощи конструкторов с захватом динамической памяти и деструкторов для их уничтожения, а также способа работы со строковыми объектами. Познакомиться с механизмом перегрузки операций.

Краткие теоретические сведения

Напомним, что работа со строками в языке Си реализована путем использования одномерных массивов типа ***char***, т.е. строка символов – это одномерный массив типа ***char***, заканчивающийся нулевым байтом. Нулевой байт – это байт, каждый бит которого равен нулю, при этом для нулевого байта определена символьная константа ´\0´ (признак окончания строки или нуль-терминатор). Поэтому, если строка должна состоять из k символов, то в описании массива необходимо указать размер k+1, а при ручном формировании строки в ее окончание нужно явно добавить признак ее окончания.

Операции над строками выполняются только через стандартные функции. Декларации функций для работы со строками размещены в файле ***string.h***. Перечислим некоторые из них, наиболее часто используемые.

1. Функция ***strcpy***(*S*1, *S*2) – копирует содержимое строки *S*2 в строку *S*1.

2. Функция ***strcat***(*S*1, *S*2) – присоединяет строку *S*2 к строке *S*1 и помещает ее в массив, где находилась строка *S*1, при этом строка *S*2 не изменяется. Нулевой байт, который завершал строку *S*1, заменяется первым символом строки *S*2.

3. Функция ***strcmp***(*S*1, *S*2) сравнивает строки *S*1 и *S*2 и возвращает значение 0, если строки равны, т.е. содержит одно и то же число одинаковых символов; значение <0, если *S*1<*S*2;значение >0, если *S*1>*S*2.

4. Функция ***strlen***(*S*) возвращает длину строки, т.е. количество символов, начиная с нулевого и до нуль-терминатора, нулевой байт не учитывается.

5. Функции преобразования строки S в число:
целое: int ***atoi***(*S*); длинное целое: long ***atol***(*S*); действительное: double ***atof***(*S*); при ошибке возвращает значение 0.

6. Функции преобразования числа *V* в строку *S*:
целое: ***itoa***(int *V*, char *S*, int *kod*); длинное целое: ***ltoa***(long *V*, char *S*, int *kod*); 2<=*kod*<=36, для отрицательных чисел *kod*=10.

Помимо традиционной декларации строки, например char *s*1[51]; можно создавать динамические строки.

Пример создания с использованием библиотечной функции:

```
        . . .
char *s;
        s = (char*)calloc(k,sizeof(char));  // Захват памяти для строки длиной k
        . . .
```

Можно воспользоваться и операциями захвата и освобождения динамической памяти.

Пример класса с конструктором для создания динамических строковых объектов и деструктором для их уничтожения после использования в программе.

```
        …
class String1 {
                char *str;
                public:
String1 (char *s = “\0”) {                      // Конструктор
                str = new char [strlen(s)+1];
                strcpy(str,s);
                        }
~ String1 () {                                  // Деструктор
                delete str;
                }
void Print(char *s1);
        . . .
};
```

```
void main(void) {
        String1 s1("Minsk");
        String1 s2("Moskow");
        s1 . Print ("S1");
        s2 . Print ("S2");
        . . .
        }
```

Получим: S1 : Minsk
S2 : Moskow

Перегрузка операций в C++ (краткая характеристика)

В языке C++ наряду с обычными конструкциями операций над операндами со стандартными типами (арифметические, операции отношений, логические операции и т.д.) существует механизм перегрузки этих операций, позволяющий манипулировать объектами классов пользователя, используя обычный синтаксис языка, т.е. привычную (удобную) запись операций.

Если компилятор обнаружил, что в программе есть перегрузка операций для объектов конкретного типа, введенных пользователем, то, найдя эти операции, компилятор анализирует их. И, если

– операнды имеют встроенный тип данных, то будет заложена стандартная операция;

– операнды имеют тип данных пользователя, т.е. являются объектами класса, для которого операция была перегружена, то будет заложена перегруженная операция, запрограммированная пользователем.

Операции для конкретного класса перегружаются или методом этого класса, или функцией-другом для этого класса следующего:

```
mun   operator@ (список параметров с указанием их типа) {
            код, определяющий смысл указанной операции,
        }
```

@ – символ перегружаемой операции.

Так как операция @ перегружается для объекта конкретного класса, а сам класс – это тип этого объекта, то *mun* – это идентификатор этого класса.

Отличия перегрузки операций при помощи метода класса или функции-друга следующие:

а) *@a* – унарная операция может быть перегружена методом без параметров и функцией-другом с одним параметром, т.к. у метода имеется первый скрытый от нас параметр (указатель *this*), который автоматически устанавливается на единственный операнд унарной операции, а функция-друг такого указателя не имеет, потому при перегрузке ее параметров, ее операнд и передают;

б) *a*1 *@ a*2 – бинарная операция перегружается методом класса с одним параметром и функцией-другом с двумя параметрами, т.к. при ее перегрузке методом на первый операнд устанавливается скрытый указатель (*this*), а второй пере-

даем через параметр; при перегрузке функцией-другом первый операнд передается функции через первый параметр, второй – через второй параметр.

Рассмотрим пример перегрузки (с использованием функции-друга) операции для класса, порождающего комплексные числа:

Класс, порождающий комплексные числа:

$$z1 = re1 + j \cdot im1 = (re1, im1) ;$$

т.е. комплексное число – это пара вещественных чисел.

$$z2 = re2 + j \cdot im2 = (re2, im2) ;$$

Введем

$$z3 = (z1 + z2) = re3 + j \cdot im3 = (re3, im3) ;$$

где

$$re3 = re1 + re2; \quad im3 = im1 + im2;$$

это и составит смысл перегрузки операции «+».

```
. . .
class Complex {
    double re, im;
    public:
Complex(double r, double i) {          // Обычный конструктор с параметрами
        re = r; im = i;
    }
    . . .
// Декларация функции-друга типа operator
    friend Complex operator+(Complex, Complex);
    };
// Определение функции
    Complex operator+(Complex x, Complex y){
        Complex z;
        z.re = x.re + y.re;
        z.im = x.im + y.im;
        return z;
        }                                  // Перегрузили
        . . .
```

Участок основного кода, иллюстрирующий перегрузку операции «+»:

```
        . . .
double a = 1.5, b = 2.6, c;
        c = a + b;
// Компилятор нашел «+», анализирует данные – это обычные типы, выполняется
// стандартная операция
        cout << " C = " << endl;          // На экране: C = 4.1
        Complex z1(1, 2), z2(3, 4), z3(0, 0);
        z3 = z1 + z2;                     // Выполняется перегруженный «+»
```

```
cout << " Z3 : ";
z3.Print();   getch();                // На экране: Z3 : Re = 4  Im = 6
        . . .
```

Задание к лабораторной работе № 5

Общая постановка. Пользовательский класс *String* должен содержать необходимые элементы-данные, создаваемые в динамической области памяти.

Конструктор для создания строк: ***String* (…);**
Деструктор: **~*String*();**
Метод ввода исходной строки ***Set***();
Метод печати: ***void Print*(…);**

Код методов – вне пространства определения класса. Программа иллюстрирует прямой и косвенный способы обращения к методам.

Индивидуальные задания

Ввести строку символов *S*1, признак окончания ввода строки – нажатие клавиши *Enter*. Программа должна содержать перегруженную операцию **«=»**, использование которой скопирует *S*1 в *S*2 при следующих условиях:

1. Без двух первых и двух последних символов.
2. Без всех чисел, которые делятся на 2.
3. Без всех цифр.
4. Без всех *a..z*.
5. Без всех *A..Z*.
6. Без скобок всех видов.
7. Подстроку до первого пробела.
8. Подстроку в фигурных «{ }» скобках.
9. Подстроку до первой круглой скобки.
10. Подстроку после последнего пробела.
11. Подстроку со второго пробела.
12. Без каждого 3-го символа.
13. Подстроку до последнего пробела.
14. Подстроку от последней цифры.
15. Подстроку в квадратных «[ ]» скобках.

На печать вывести исходную и преобразованную строки.

Контрольные вопросы

1. Как создать динамическую строку?
2. В чем заключаются особенности работы со строками в языке Си?
3. Что такое «перегрузка операций»?
4. При помощи чего можно перегрузить операцию в языке Си?
5. Какие отличия в перегрузке унарных и бинарных операциях?

ЛАБОРАТОРНАЯ РАБОТА № 6.
Наследование и механизм виртуальных функций

Цель работы: изучить одну из базовых концепций ООП – наследование классов в C++, заключающуюся в построении цепочек классов, связанных иерархически. Познакомиться с механизмом виртуальных функций.

Краткие теоретические сведения

Наследование – такое соотношение между классами, когда один класс использует часть другого класса, добавляя этому классу нечто свое и таким образом расширяя возможности этого класса. При этом первый класс, который описывает наиболее общие свойства ряда объектов, называется базовый, второй класс – производный.

Определение производного класса имеет следующий синтаксис:

```
class ID_производного_класса : ID_базового_класса {
        код_производного_класса
        };
```

Одна из особенностей порожденного класса – видимость унаследованных компонент базового класса со степенью защиты *protected* и *public* (атрибуты базового класса). Компоненты базового класса с ключевым словом *private* производному классу недоступны.

Производный класс может служить базовым классом для создания следующего, производного класса на более низком уровне иерархии классов. Наследование называется простым, если производный класс имеет одного родителя. В C++ наследование может быть множественным. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

Видимые компоненты базового класса со степенью защиты *protected* и *public* в производном классе становятся *private*, а значит, не могут быть наследованы производными классами на более низких уровнях иерархии классов. Это связано с тем, что в заголовке производного класса перед *ID* базового класса по умолчанию компилятор указывает атрибут доступа *private*. Если же указать явно общедоступный атрибут

```
class ID_производного_класса : public ID_базового_класса
```

то унаследованные компоненты базового класса сохранят степень защиты в производном классе и могут быть унаследованы производным классам на следующем уровне иерархии классов.

Перед активизацией конструктора производного класса производится вызов конструктора базового класса. Если базовый класс является производным классом на более высоком уровне иерархии, то процесс вызова конструктора производится по своей цепи иерархии классов, пока не доберется до базового класса самого верхнего уровня иерархии. Таким образом, объекты базового класса всегда существуют в составе объектов производного класса. В противоположность этому де-

структор производного класса вызывается перед вызовом деструктора базового класса. Это объясняется тем, что уничтожение объекта базового класса влечет за собой уничтожение и объекта производного класса.

Если конструктор базового класса имеет параметры для инициализации своих объектов, то конструктор производного класса обязан обеспечить передачу ему исходных данных. Для этой цели используется форма конструктора со списком инициализаторов, в который включаются идентификаторы конструкторов базового класса (явный вызов конструктора базового класса). При этом последние должны быть объявлены с атрибутом доступа как минимум *protected*.

Продемонстрируем данный механизм на конкретном примере:

```
                    . . .
class B1 {                          // Базовый класс
        protected:
        int x;
        public:
        B1(int i) {                 // Конструктор с параметрами
                x = i; }
        };
class D: public B1  {               // Наследование с сохранением прав доступа
        protected:
        int a, b;
        public:
D(int i, int  j) : B1(i+3), a(i)  {   // Конструктор со списком инициализации
                b = j;
                }
void Print(void) {
        cout << " a = " << a << "  b = " << b << " x = " << x << endl;
        }
};
void main(void)  {
        D  d1(3, 4);
        d1.Print();
                }
```

На экране монитора: **a = 3 b = 4 x = 6**

Как правило, коды методов выносят из пространства класса. Тогда для предыдущего примера:

```
        . . .
class D {
        protected:
        int a, b;
        public:
        D(int, int);
        void Print(void);
        };
```

```
D :: D(int i, int j) : B1(i+3), a(i)  {
        b = j;
        }
void D :: Print(void)  {
        cout << " a = " << a << " b = " << b << " x = " << x << endl;
        }
        . . .
```

Использование косвенной адресации с установкой указателей на базовый класс

Механизм косвенной адресации рассмотрим на конкретном примере.

```
        . . .
class B  {
        public:
        int x;
        B() {                          // Конструктор по умолчанию
                x = 4; }
        };
class D : public B {                   // Производный класс
        public:
        int y;
        D()  {                         // Конструктор по умолчанию
                y = 5; }
};
void main(void)  {
        D  d;                          // Конструктор класса D создает объект d
        B *p;                          // Указатель установлен на базовый касс
        p = &d;                        // Указатель p инициализируется адресом d
// Косвенное обращение к объектам базового и производного классов
// Считываем их текущее состояние в переменные
        int i = p -> x;                // Базовый класс виден напрямую
        int j = ( ( D* ) p )p -> y;    // Прямое преобразование указателя на D
// Через переменные печатаем их текущее состояние
        cout << " x_i = " << i << "  y_j = " << j << endl;
        getch();
        }
```

На экране монитора: **x_i = 4 y_j = 5**

Механизм виртуальных функций

Механизм виртуальных функций – одна из основных концепций объектно-ориентированного программирования. Данный механизм предполагает использование идеи «один интерфейс – множество методов реализации». Эта идея заключается в том, что базовый класс обеспечивает для производных классов все элементы, которые эти классы могут использовать непосредственно, а также содержит набор функций, которые производные классы должны реализовать путем их

переопределения (создание собственного кода функции в производном классе, которые позволяют решить поставленную перед производным классом задачу).

Виртуальная функция – это функция, объявленная с ключевым словом ***virtual*** в базовом классе и переопределенная в одном или нескольких производных от этого классах. Обязательное требование: заголовок функции должен быть точно такой же, как в базовом классе (и имя, и список параметров должны быть одинаковые). Тогда при создании объекта или базового, или одного из производных классов компилятор определяет, какую из функций требуется вызвать, основываясь на типе (*ID* класса) объекта.

Таким образом, компилятор на этапе компиляции, обнаружив виртуальные функции с одинаковыми заголовками, но с разными кодами, не конкретизирует, какой из виртуальных методов будет вызываться, а отводит для их адресов место в специальной таблице адресов виртуальных методов. И далее на этапе выполнения зарезервированное в таблице место инициализируется адресом соответствующей виртуальной функции конструктором того класса, объект которого создается в данный момент выполнения программы.

Такой процесс в C++ получил название «*позднее связывание*».

Как правило, на практике в базовом классе приводят только прототип виртуального метода, который определяет общий интерфейс, указывающий, с какими данными необходимо работать. А в производном классе приводят полные коды, которые определяют способы обработки указанных данных.

Если требование полного сохранения заголовка виртуального метода в производном классе нарушено, то компилятор, обнаружив это, механизм виртуальных функций проигнорирует и произведет на этапе компиляции его перегрузку.

Задание к лабораторной работе № 6

Общая постановка. Программа должна содержать:

– базовый класс ***X***, включающий два элемента *x*1, *x*2 типа *int*;

– конструктор с параметрами для создания объектов в динамической области памяти,

– деструктор;

– виртуальные методы просмотра текущего состояния и переустановки объектов базового класса в новое состояние.

Производный класс ***Y***, включающий один элемент *y* типа *int*;

– конструктор с параметрами и списком инициализаторов, передающий данные конструктору базового класса;

– переопределенные методы просмотра текущего состояния объектов и их переустановки в новое состояние.

Индивидуальные задания

Создать в производном классе метод ***Run,*** определяющий:

1. Сумму компонент классов.
2. Произведение компонент классов.
3. Сумму квадратов компонент классов.

4. Значение $x1 + x2 – y$.
5. Значение $(x1 + x2)/y$.
6. Значение $(x1 + x2)*y$.
7. Значение $x1*y$ + x2.
8. Значение $x1 + x2*y$.
9. Произведение квадратов компонент класса.
10. Значение $x1*x2 + y$.
11. Значение $x1*x2/y$.
12. Значение $x1*x2 – y$.
13. Значение $(x1 – x2)*y$.
14. Значение $(x1 – x2)/y$.
15. Значение $x1/y – x2$.

Программа должна продемонстрировать работу конструкторов базового и производного классов, начальное значение задается конструкторами, а переустановка их в новое состояние производится методами классов через косвенную адресацию.

Контрольные вопросы

1. Что такое «наследование» и иерархия классов?
2. Какие элементы базового класса видны из производного? Как управлять степенью их защиты?
3. Какое наследование называют множественным?
4. Поясните механизм виртуальных функций.

ЛАБОРАТОРНАЯ РАБОТА № 7.
Шаблоны классов

Цель работы: изучить приемы создания и использования шаблонов классов.

Краткие теоретические сведения

Механизм **шаблонов** C++ – это средство построения обобщенных определений функций и классов, которые не зависят от используемых типов данных. Этот механизм позволяет сократить трудоемкость создания программ, т.к. повышает лаконичность текста, т.е. использование шаблонов избавляет от необходимости дублировать коды классов и функций для разных типов данных.

Компилятор по заданному типу аргументов на основе описания шаблона автоматически создает соответствующие экземпляры классов и функций, которые называются ***представители*** конкретных классов и функций.

Шаблоны класса в отличие от шаблона функции позволяют параметризовать, т.е. использовать в качестве параметров не только типы элементов данных, но и константы разных типов.

Синтаксис определения шаблона класса следующий:

template ***<список параметров шаблона >*** **обычное описание структуры класса;**

При этом список параметров шаблона не может быть пустым. Элементы в списке разделяются запятыми. Внутри пространства класса параметры шаблона должны быть хотя бы один раз упомянуты.

В список параметров могут входить два вида параметров:

1) типизированные параметры, начинающиеся со слов

class Идентификатор ,

компилятор при создании экземпляра класса заменит его на конкретный тип данных;

2) нетипированные параметры шаблона:

Стандартный тип числовых данных ***Идентификатор***

Таким образом, типированные параметры – это фиктивные имена типов данных, входящих в класс. Нетипированные параметры – это поименованные типы числовых констант. При этом им при декларировании можно присваивать умалчиваемые значения.

Каждый параметр является локальным в рамках пространства класса.

В описании класса хотя бы один раз необходимо упомянуть ID типированных и нетипированных параметров, конкретные значения для которых будут переданы в момент создания объекта этого класса через список аргументов, который указывается через запятые в треугольных скобках сразу после ID класса:

ID_ класса <список аргументов> ID_объектов;

ID_объектов – идентификаторы объектов, которые создает данный класс (записанные через запятые, например *a,b,c*). При этом в списке аргументов каждому типированному параметру шаблона должен соответствовать известный конкретный тип данных, а каждому нетипированному параметру – константное выражение соответствующего типа. Таким образом, между списком параметров шаблона и списком аргументов должно быть абсолютное соответствие по количеству, порядку их следования и типам. Если нетипированные параметры имеют умалчиваемые значения, их располагают в списке последними.

Пример с шаблоном класса, конструктор которого порождает объекты с двумя параметризованными полями:

```
. . .
template <class T1>
        class X {
                protected:
                T1 a, b;
                public:
                X(T1 i, T1 j) {                 // Конструктор
                        a = i; b = j;
                        cout << “\n Object created, size = ” << sizeof(T1) << endl;
```

```
        }
    void Print(void) {
        cout << “  a = ” << a << “  b = ” << b << endl;
        }
    };
void main(void) {
    X < int >  x1(2, 3);            // Целочисленные объекты
    x1.Print();                     // На экране: Object created, size = 2
    getch();                        //            a = 2  b = 3
    X < double >  x2(0.5, 1.2);     // Вещественные объекты
    x2.Print();                     // На экране: Object created, size = 4
    getch();                        //            a = 0.5  b = 1.2
    }
```

Методы шаблона класса можно определять ***вне его пространства***. В этом случае формат их определения будет следующим:

template <список параметров шаблона> тип результата ID класса <перечень через «,» ID из списка параметров шаблона> :: ID метода список параметров метода)

```
    {
        код метода
    }
```

В вышеприведенном примере вынесем за пределы пространства класса код конструктора:

```
    . . .
template <class T1>
    class X {
        protected:
        T1 a, b;
        public:
        X(T1, T1);
        void Print(void) {
            cout << “  a = ” << a << “  b = ” << b << endl;
            }
        };
// Определение конструктора вне состава класса
    template <class T1> X <T1> :: X(T1 i, T1 j) {
        a = i; b = j;
        cout << “ Object created, size = ” << sizeof(T1) << endl;
        }
void main(void) {
        // Код предыдущего примера
    }
```

Задание к лабораторной работе № 7

Общая постановка. Даны: число N и последовательность $a_1, a_2, \dots a_N$.

Создать шаблон класса, порождающего динамические одномерные массивы с элементами различных типов (вещественные, целочисленные, символьные и т.д.). Тип данных и результат являются параметрами по отношению к классу. Программа должна содержать: конструктор, деструктор, метод просмотра значений созданного массива, а также метод для решения задач формирования нового массива по соответствующему индивидуальному заданию.

Индивидуальные задания

1. $a_1,\ (a_1 + a_2), \dots ,\ (a_1 + a_2 + \dots + a_N)$.
2. $(a_1 * a_1),\ (a_1 * a_2), \dots ,\ (a_1 * a_N)$.
3. $|a_1|,\ |a_1 + a_2|, \dots ,\ |a_1 + a_2 + \dots + a_N|$.
4. $a_1,\ -a_1 * a_2,\ + a_1 * a_2 * a_3, \dots ,\ (-1)^{N-1} * a_1 * a_2 * \dots a_N$.
5. $-a_1,\ + a_2,\ - a_3, \dots ,\ (-1)^N * a_N$.
6. $(a_1 + 1),\ (a_2 + 2),\ (a_3 + 3), \dots ,\ (a_N + N)$.
7. $a_1 * 1,\ a_2 * 2,\ a_3 * 3, \dots ,\ a_N * N$.
8. $a_1 * a_2,\ a_2 * a_3, \dots ,\ a_{N-1} * a_N$.
9. $a_1/1,\ a_2/2,\ a_3/3, \dots ,\ a_N/N$.
10. $(a_1 + a_2),\ (a_2 + a_3), \dots ,\ (a_{N-1} + a_N)$.
11. $(a_1 + a_2 + a_3),\ (a_2 + a_3 + a_4),\ (a_3 + a_4 + a_5), \dots,\ (a_{N-2} + a_{N-1} + a_N)$.
12. $(N + a_1),\ (N-1 + a_2), \dots ,\ (1 + a_N)$.
13. $N * a_1,\ (N-1) * a_2, \dots ,\ (1 * a_N)$.
14. $a_1/\mathrm{N},\ a_2/\mathrm{N}, \dots ,\ a_N/1$.
15. $-a_1,\ +a_1 * a_2,\ - a_1 * a_2 * a_3, \dots , (-1)^N * a_1 * a_2 * \dots a_N$.

Контрольные вопросы

1. В чем в С++ заключается механизм шаблонов классов и функций?
2. Каков общий формат шаблона класса?
3. Перечислите виды параметров шаблона класса.
4. Как создать конкретный экземпляр класса, используя шаблон класса?
5. Как определить метод вне пространства шаблона класса?

ЛАБОРАТОРНАЯ РАБОТА № 8.
Обработка исключительных ситуаций

Цель работы: приобрести навыки по организации контроля за возникновением непредвиденных или аварийных ситуаций во время работы программы.

Краткие теоретические сведения

Исключения – возникновение непредвиденных ошибочных условий в момент работы программы. В то же время исключение – это более общее чем ошиб-

ка понятие, так как может возникать и тогда, когда в программе нет ошибок (например, сбой выделения памяти при создании объекта класса). В общем, исключение обозначает особую ситуацию, когда требуется безвозвратно переключить выполнение программы на блок обработки такой ситуации. Выявление особой ситуации производится только программным путём при помощи проверки нормального хода выполнения программы.

Средства обработки ошибочных ситуаций позволяют передать обработку исключений из кода, в котором возникло исключение, некоторому другому программному блоку, который выполнит в данном случае некоторые определенные действия. Таким образом, основная идея данного механизма состоит в том, что функция проекта, которая обнаружила непредвиденную ошибочную ситуацию, которую она не знает, как решить, генерирует сообщение об этом (бросок исключения). А система вызывает по этому сообщению программный модуль, который перехватит исключение и отреагирует на возникшее нештатное событие. Такой программный модуль называют «обработчик» или перехватчик исключительных ситуаций. И в случае возникновения исключения в его обработчик передаётся произвольное количество информации с контролем ее типа. Эта информация и является характеристикой возникшей нештатной ситуации.

Обработка исключений в С++ – это обработка с завершением. Это означает, что исключается невозможность возобновления выполнения программы в точке возникновения исключения.

Для обеспечения работы такого механизма были введены следующие ключевые слова:

try – проба испытания;
catch – перехватить (обработать);
throw – бросать.

Кратко рассмотрим их назначение.

Ключевое слово ***try*** открывает блок кода, в котором может произойти ошибка; это обычный составной оператор:

```
try {
        код
};
```

Код содержит набор операций и операторов, который и будет контролироваться на возникновение ошибки. В него могут входить вызовы функции пользователя, которые компилятор также возьмет на контроль.

Среди данного набора операторов и операций обязательно указывают операцию броска исключения ***throw***, которая имеет следующий формат:

throw выражение;

где *выражение* определяет тип информации, которая и описывает исключение (например конкретные типы данных).

Обработчик исключения ***catch*** перехватывает информацию:

```
catch ( тип и параметр) {
        код
        }
```

Через параметр обработчику передаются данные определенного типа, описывающие обрабатываемое исключение. Код определяет те действия, которые надо выполнить при возникновении данной конкретной ситуации. В С++ используют несколько форм обработчиков. Рассмотренный обработчик получил название параметризованный специализированный перехватчик.

Перехватчик должен следовать сразу же после блока контроля, т.е. между обработчиком и блоком контроля не должно быть ни одного оператора. При этом в одном блоке контроля можно вызывать исключения разных типов для различных ситуаций, поэтому обработчиков может быть несколько. В этом случае их необходимо расположить сразу же за контролирующим блоком последовательно друг за другом.

Кроме того, запрещены переходы как извне в обработчик, так и между обработчиками.

Можно воспользоваться универсальным или абсолютным обработчиком:

```
catch ( . . . ) {
        код
         }
```

где (...) – означают способность данного перехватчика обрабатывать информацию любого типа. Такой обработчик располагают последним в пакете специализированных обработчиков. Тогда, если исключение не будет перехвачено специализированными обработчиками, то будет выполнен последний – универсальный.

Если не возникнет исключение, набор обработчиков будет обойден, т.е. проигнорирован.

Если же исключение было «брошено» при возникновении критической ситуации, то будет вызван конкретный перехватчик при совпадении его параметра с выражением в операторе броска, т.е. управление будет передано найденному обработчику. После выполнения кода вызванного обработчика управление передается оператору, который расположен за последним перехватчиком, или проект корректно завершает работу.

Существенное отличие вызова конкретного обработчика от вызова обычной функции заключается в следующем: при возникновении исключения и передаче управления определенному обработчику система осуществляет вызов всех деструкторов для всех объектов классов, которые были созданы с момента начала контроля и до возникновения исключительной ситуации, с целью их уничтожения.

Блоки *try* как составные блоки могут быть вложены друг в друга. В случае возникновения исключения в некотором текущем блоке, поиск обработчика последовательно продолжается в блоках, предшествующих уровней вложенности с продолжением вызова деструкторов.

Пример обработки исключительных ситуаций. Функция *Divide*() возвращает частное от деления чисел, принимаемых в качестве аргументов (*a*, *b*), если делитель равен нулю (*b* = 0), то возникает исключительная ситуация.

```
#include <iostream.h>
#include <conio.h>
    double Divide(double, double);
```

```
void main(void)  {
        double a, b, result;
        cout << " Input a, b : " << endl;
        cin >> a >> b;
        try {
                result = Divide(a, b);
                cout << " Normal Work " << endl;
                cout << " a =  " << a << ",  b =  " << b << endl;
                cout << " Ansver : " << result << endl;
                getch();
        }
        catch(double z) {
                cout << " Division by zero... " << endl;
                cout << " b = " << z << endl;
                getch();
        }
}
double Divide(double a1, double b1) {
        if (b1==0) throw b1;
        return a1/b1;
        }
```

Результаты выполнения программы, введя значения 1 и 2, получим:

```
Input a, b :
1 2
Normal Work
a = 1,  b = 2
Ansver :  0.5
```

А при вводе значений 1 и 0, получим:

```
Input a, b :
1 0
Division by zero...
b = 0
```

Оператор *throw* сигнализирует об исключительном событии (попытку деления на ноль) и генерирует объект исключительной ситуации. В данном случае он находится внутри функции *Divide*(). Объект перехватывается обработчиком *catch*. Этот процесс, как уже известно, и называется вызовом исключительной ситуации. В рассмотренном примере исключительная ситуация имеет форму вещественной переменной – делителя.

Задание к лабораторной работе № 8

Общая постановка. Даны два выражения *Z*1 и *Z*2. Написать функции для вычисления этих выражений с организацией обнаружения нештатной ситуации

(например деление на ноль) и ее обработки. Передача аргументов в функции – по ссылкам.

В случае успеха значения *Z*1 и *Z*2 должны быть приблизительно одинаковыми.

Индивидуальные задания

1. $z_1 = \dfrac{\sqrt{2b + 2\sqrt{b^2 - 4}}}{\sqrt{b^2 - 4} + b + 2}, \quad z_2 = \dfrac{1}{\sqrt{b+2}}.$

2. $z_1 = \dfrac{x^2 + 2x - 3 + (x+1)\sqrt{x^2 - 9}}{x^2 - 2x - 3 + (x-1)\sqrt{x^2 - 9}}, \quad z_2 = \sqrt{\dfrac{x+3}{x-3}}.$

3. $z_1 = \dfrac{\sqrt{(3m+2)^2 - 24m}}{3\sqrt{m} - 2/\sqrt{m}}, \quad z_2 = -\sqrt{m}.$

4. $z_1 = \left(\dfrac{a+2}{\sqrt{2a}} - \dfrac{a}{\sqrt{2a} + 2} + \dfrac{2}{a - \sqrt{2a}} \right) \cdot \dfrac{\sqrt{a} - \sqrt{2}}{a+2}, \quad z_2 = \dfrac{1}{\sqrt{a} + \sqrt{2}}.$

5. $z_1 = \left(\dfrac{1 + a + a^2}{2a + a^2} + 2 - \dfrac{1 - a + a^2}{2a - a^2} \right)^{-1} (5 - 2a^2), \quad z_2 = \dfrac{4 - a^2}{2}.$

6. $z_1 = \dfrac{(m-1)\sqrt{m} - (n-1)\sqrt{n}}{\sqrt{m^3 n} + nm + m^2 - m}, \quad z_2 = \dfrac{\sqrt{m} - \sqrt{n}}{m}.$

7. $z_1 = \dfrac{\sqrt{2m + 2\sqrt{m^2 - 4}}}{m + \sqrt{m^2 - 4} + 2}, \quad z_2 = \dfrac{1}{\sqrt{m+2}}.$

8. $z_1 = \dfrac{(x+1)\sqrt{x^2 - 9} + x(x+2) - 3}{(x-1)\sqrt{x^2 - 9} + x^2 - 2x - 3}, \quad z_2 = \sqrt{\dfrac{x+3}{x-3}}.$

9. $z_1 = \left(2 + \dfrac{1 + x + x^2}{2x + x^2} - \dfrac{1 - x + x^2}{2x - x^2} \right)^{-1} (5 - 2x^2), \quad z_2 = \dfrac{4 - x^2}{2}$

10. $z_1 = \left(\dfrac{2}{x - \sqrt{2x}} + \dfrac{x+2}{\sqrt{2x}} - \dfrac{x}{\sqrt{2x} + 2} \right) \cdot \dfrac{\sqrt{x} - \sqrt{2}}{x+2}, \quad z_2 = \dfrac{1}{\sqrt{x} + \sqrt{2}}.$

11. $z_1 = \dfrac{\sqrt{(3x+2)^2 - 24x}}{3\sqrt{x} - 2/\sqrt{x}}, \quad z_2 = -\sqrt{x}.$

12. $z_1 = \dfrac{(a-1)\sqrt{a} - (b-1)\sqrt{b}}{\sqrt{a^3 b} + ba + a^2 - a}, \quad z_2 = \dfrac{\sqrt{a} - \sqrt{b}}{a}.$

13. $z_1 = \dfrac{\sqrt{2\sqrt{x^2-4}+2x}}{x+\sqrt{x^2-4}+2}$, $z_2 = \dfrac{1}{\sqrt{x+2}}$.

14. $z_1 = \left(\dfrac{m+2}{\sqrt{2m}} + \dfrac{2}{m-\sqrt{2m}} - \dfrac{m}{\sqrt{2m}+2}\right) \cdot \dfrac{\sqrt{m}-\sqrt{2}}{m+2}$, $z_2 = \dfrac{1}{\sqrt{m}+\sqrt{2}}$.

15. $z_1 = \dfrac{(x-1)\sqrt{x}-(y-1)\sqrt{y}}{\sqrt{x^3y}+xy+x^2-x}$, $z_2 = \dfrac{\sqrt{x}-\sqrt{y}}{x}$.

Контрольные вопросы

1. Что называют исключением?
2. Что такое «блок с контролем»?
3. Дайте характеристику обработчику исключений. Какие бывают виды обработчиков?
4. Какие правила налагаются на соотношения между блоком контроля и обработчиками?
5. Чем отличается вызов обработчика от вызова обычной функции?

ЛИТЕРАТУРА

1. Касаткин А.И., Вальвачев А.Н. Профессиональное программирование на языке Си. От Turbo C к Borland C++: Справочное пособие. – Мн.: Выш. шк., 1992.

2. Касаткин А.И. Профессиональное программирование на языке Си. Управление ресурсами: Справочное пособие. – Мн.: Выш. школа, 1992.

3. Касаткин А.И. Профессиональное программирование на языке Си. Системное программирование. – Мн.: Выш. шк., 1992.

4. Цимбал А.А., Майоров А.Г., Козодаев М.А. Turbo C++: язык и его применение. – М.: «Джен Ай Лтд.», 1993.

5. Страуструп Б. Язык программирования C++. – М.: Радио и связь, 1991.

6. Вайнер Р., Пинсон Л. C++ изнутри. – Киев.: Софт, 1993.

7. Бабэ Б. Просто и ясно о Borland C++. – М.: Бином, 1995.

8. Скляров В.А. Язык C++ и объектно-ориентированное программирование. – Мн.: Выш. шк., 1997.

9. Джефф Элджер. «C++». Бибилиотека программиста. – СПб.: ПИТЕР, 2000.

10. Крячков А.В. и др. Программирование на C и C++ (практикум). – М.: Горячая линия–Телеком, 2000.

11. Карпов Б. Баранова Т. C++ – специальный справочник. – СПб.: ПИТЕР, 2001.

12. Павловская Т.А., Щупак Ю.А. C++ Объектно-ориентированное программирование. – СПб.: ПИТЕР, 2004.

13. Герберт Шилдт. Программирование на Borland C++. – Мн.: ПОПУРРИ, 1998.

14. Бусько В.Л., Корбит А.Г. и др. Основы программирования на алгоритмическом языке С.: Лабораторный практикум. – Мн.: Ротапринт БГУИР, 2003.

15. Бусько В.Л., Корбит А.Г., Кривоносова Т.М. Основы алгоритмизации программирования: Конспект лекций для студентов всех специальностей и форм обучения БГУИР. – Мн.: БГУИР, 2004.

Св. план 2005, поз. 81

Учебное издание

Бусько Виталий Леонидович,
Корбит Анатолий Григорьевич,
Кривоносова Татьяна Михайловна

ОСНОВЫ ОБЪЕКТНО-ОРИЕТИРОВАННОГО ПРОГРАМИРОВАНИЯ. C++

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

для студентов всех специальностей и форм обучения БГУИР

Редактор Н.В. Гриневич

Подписано в печать 03.05.2005. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Печать ризографическая. Усл. печ. л. 2,44.
Уч.-изд. л. 2,5 Тираж 400 экз. Заказ 49.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия на осуществление издательской деятельности № 02330/0056964 от 01.04.2004.
Лицензия на осуществление полиграфической деятельности № 02330/0133108 от 01.04.2004.
220013, Минск, П. Бровки, 6.