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

Инженерно-экономический факультет

Кафедра экономической информатики

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-40 05 01 «Информационные системы и технологии» (по направлениям)

Минск БГУИР 2015

УДК 004.42(076.5)
ББК 32.973.26-018.2я73
О-29

А в т о р ы:
В. Н. Комличенко, Ю. А. Луцик, А. М. Ковальчук, Е. Н. Унучек

Р е ц е н з е н т ы:
кафедра дискретной математики и алгоритмики
Белорусского государственного университета
(протокол №10 от 21.02.2014);

доцент кафедры систем автоматизированного проектирования
Белорусского национального технического университета,
кандидат технических наук, доцент И. Л. Ковалева

Объектно-ориентированное программирование. Лабораторный прак-
О-29 тикум : пособие / В. Н. Комличенко [и др.]. – Минск : БГУИР, 2015. – 75 с. : ил.
ISBN 978-985-543-127-6.

Приведены описания лабораторных работ с рассмотрением кратких теоретических сведений по изучаемому материалу и примеров программ, их поясняющих. Так же присутствуют контрольные вопросы, дающие возможность определить подготовленность студента к выполнению задания. Приведены варианты заданий.

Может быть использовано студентами всех форм обучения, магистрантами и аспирантами, обучающимися по направлениям специальности «Информационные системы и технологии».

УДК 004.42(076.5)
ББК 32.973.26-018.2я73

ISBN 978-985-543-127-6

Лабораторная работа №1

Тема работы: организация ввода/вывода, динамическое выделение памяти.

Цель работы: изучить организацию ввода/вывода и работу с динамической памятью при программировании алгоритмов в С++.

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3−8].

Ввод/вывод. В С++ ввод и вывод данных производится потоками байт. Поток (последовательность байт) − это логическое устройство, которое выдает и принимает информацию от пользователя и связано с физическими устройствами ввода/вывода. При операциях ввода байты направляются от устройства в основную память. В операциях вывода – наоборот.

Имеется четыре потока (связанных с ними объекта), обеспечивающих ввод и вывод информации и определенных в заголовочном файле iostream.

В файле iostream перегружаются два оператора побитового сдвига

```
<<          // поместить в выходной поток
>>          // считать со входного потока
```

и объявляются три стандартных потока:

```
cout        // стандартный поток вывода (экран)
cin         // стандартный поток ввода (клавиатура)
cerr        //  стандартный поток диагностики (ошибки)
```

Объект cin. Для ввода информации с клавиатуры используется объект cin. Формат записи cin имеет следующий вид:

```
cin [>>имя_переменной];
```

При вводе необходимо, чтобы данные вводились в соответствии с форматом переменных.

Объект cout. Объект cout позволяет выводить информацию на стандартное устройство вывода – экран. Формат записи cout имеет следующий вид:

```
cout << data [ << data];
```

где data − это переменные, константы, выражения или комбинации всех трех типов.

Пример использования объектов cin и cout.

```
#include<iostream>
using namespace std;
int main()
{
    setlocale(LC_ALL,"Russian");
    int i;
    double x;
    cout << "Введите число с двойной точностью" << endl;;
    cin >> x;                              // ввод числа с плавающей точкой
    cout << "Введите положительное число" << endl;
    cin  >> i;                             // ввод целого числа
    cout << "i * x=" <<  i*x  << endl;     // вывод результата
    return 0;
}
```

Идентификатор endl называется манипулятором. Он очищает поток cerr и добавляет новую строку.

Для управления выводом информации в языке C++ используются манипуляторы. Для их использования необходим заголовочный файл iomanip. Манипуляторы hex и oct используются для вывода числовой информации в шестнадцатеричном или восьмеричном представлении. Применение их можно видеть на примере следующей простой программы:

```
#include<iostream>
using namespace std;
void main()
{
    setlocale(LC_ALL,"Russian");
    int a=0x11, b=4,     // целые числа: шестнадцатеричное и десятичное
        c=051, d=8,      //                  восьмеричное и десятичное
    i, j;
    i=a+b;
    j=c+d;
    cout << i << ' ' << hex << i << ' ' << oct << i << ' ' << dec << i << endl;
    cout << hex << j << ' ' << j << ' ' << dec << j << ' ' << oct << j << endl;
}
```

Манипуляторы изменяют значение некоторых переменных в объекте cout. Эти переменные называются флагами состояния. Когда объект посылает данные на экран, он проверяет эти флаги.

Пример использования манипуляторов форматирования информации.

```
#include<iostream>
#include <iomanip>
using namespace std;
void main()
{
     int a=0x11;
     double d=12.362;
     cout << setw(4) << a << endl;
     cout << setw(10) << setfill('*') << a << endl;
     cout << setw(10 ) << setfill(' ') << setprecision(3) << d << endl;
}
```

Манипуляторы setw(), setfill(' ') и setprecision() позволяют изменять флаги состояния объекта cout. Синтаксис их показывает, что это функции, позволяющие изменять флаги состояния объекта cout. Функции имеют следующий формат:

```
setw(количество_позиций_для_вывода_числа)
setfil(символ_для_заполнения_пустых_позиций)
setprecision (точность_при_выводе_дробного_числа)
```

Наряду с перечисленными выше манипуляторами в C++ используются также манипуляторы **setiosflags**() и **resetiosflags**() для установки определенных глобальных флагов, используемых при вводе и выводе информации. На эти флаги ссылаются как на *переменные состояния*. Функция setiosflags() устанавливает указанные в ней флаги, а resetiosflags() сбрасывает (очищает) их. Для того чтобы установить или сбросить некоторый флаг, могут быть использованы

функции **setf**() или **unsetf**(). Флаги формата объявлены в классе **ios**. Рассмотрим пример, демонстрирующий использование манипуляторов.

```
#include<iostream>
#include <iomanip>
using namespace std;
int main()
{
    setlocale(LC_ALL,"Russian");
    char s[]="Минск БГУИР ";
    cout << setw(30) << setiosflags(ios::right) << s << endl;
    cout << resetiosflags(ios::right);
    cout << setw(30) << setiosflags(ios::left) << s << endl;
    return 0;
}
```

Динамическое распределение памяти. Память под массивы можно выделять динамически, т. е. размещать в свободной памяти (free store). Свободная память – это предоставляемая системой область памяти для объектов, время жизни которых устанавливается программистом. В C++ для операций выделения и освобождения памяти используются встроенные операторы **new** и **delete**.

Оператор new имеет один операнд. Оператор имеет две формы записи:

```
[::] new [(список_аргументов)] имя_типа [(инициализирующее_значение)]
[::] new [(список_аргументов)] (имя_типа) [(инициализирующее_значение)]
```

В простейшем виде оператор new можно записать следующим образом:

```
new имя_типа;
new имя_типа(выражение);
new имя_типа[выражение];
```

Оператор new выделяет надлежащий объем свободной памяти для хранения указанного типа и возвращает базовый адрес объекта. Когда память недоступна, оператор new возвращает NULL либо возбуждает соответствующее исключение, например:

```
#include<iostream>
#include <iomanip>
using namespace std;
void main()
{
    setlocale(LC_ALL,"Russian");
    int *p, *q, size,i;
    p=new int(5);        // выделение памяти и инициализация
    cout << "Введите размер массива" << endl;
    cin >> size;
    q=new int[size];     // выделение памяти под массив
    cout << "Введите элементы массива" << endl;
    for(i=0; i < size; i++)
        cin >> q[i];
    cout << " Массив q" << endl;
    for(i=0; i < size; i++)
        cout << setw(4)<<  q[i];
    cout << endl;
```

```
    delete p;
      delete [] q;
}
```

Можно динамически выделить память под двухмерный массив, используя «указатель на указатель». В языке C++ допустимо объявлять переменные, имеющие тип «указатель на указатель». Объявляется «указатель на указатель» следующим образом:

```
int **mas;
```

Фактически «указатель на указатель» – это адрес ячейки памяти, хранящей адрес указателя. Например:

```
#include<iostream>
#include <iomanip>
using namespace std;
void main()
{
   setlocale(LC_ALL,"Russian");
   int **mas;                          // «указатель на указатель» на массив
   int n, m;                           // количество строк и столбцов массива
   cout << "Введите количество строк и столбцов" << endl;
   cin >> n >> m;                      // ввод количества строк и столбцов
   mas=new int*[n];                    // выделение памяти под массив указателей
   for(int j=0; j < n; j++)
       mas[j]=new int[m];
   cout << "Введите элементы в массив" << endl;
   for(int i=0; i < n; i++)
       for(int j=0; j < m; j++)
           cin >> mas[i][j];
   cout << "Двухмерный массив" << endl;
   for(int i=0; i < n; i++)
   {
        cout << endl;
        for(int j=0; j < m; j++)
        cout << setw(4) << mas[i][j];
    }
   cout << endl;
   for(int i=0; i < n; i++)
      delete [] mas;
}
```

Оператор delete уничтожает объект, созданный с помощью new.

Оператор delete имеет две формы записи:

```
[::] delete  переменная_указатель       // для указателя на один элемент
[::] delete [] переменная_указатель     // для указателя на массив
```

Первая форма используется, если соответствующее выражение new размещало не массив. Во второй форме присутствуют пустые квадратные скобки, показывающие, что изначально размещался массив объектов. Оператор delete не возвращает значения.

Контрольные вопросы

1. Как осуществляется ввод/вывод в C++?
2. Что такое манипулятор ввода/вывода?
3. Для чего необходимы операторы new и delete? В чем их отличие от функций malloc() и free()?
4. Как создать и удалить массив объектов?
5. Как можно выделить память под двухмерный массив?

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Написать программу, печатающую все вводимые символы в нижнем регистре. В строку выводится символ, шестнадцатеричный и восьмеричный код.
2. Написать программу, печатающую строчные и прописные буквы русского алфавита. В строку выводится символ, шестнадцатеричный и восьмеричный код.
3. Написать программу, которая получает данные либо по Фаренгейту в виде 59 F и преобразует их в данные по Цельсию 15 ˚C, либо наоборот. Нуль по Цельсию равен 32 по Фаренгейту. Один градус Цельсия равен 1,8 по Фаренгейту. Установить ширину поля в 10 символов, точность − 4 цифры, заполнить пробелы символом «\» с помощью функций и манипуляторов.
4. Написать программу решения квадратного уравнения. Установить ширину поля в 10 символов, точность − 4 цифры, заполнить пробелы символом «#».
5. Написать программу, печатающую все вводимые символы. В строку выводится символ, шестнадцатеричный и восьмеричный код.
6. Написать программу решения линейного уравнения. Установить ширину поля в 10 символов, точность − 4 цифры, заполнить пробелы символом «%».
7. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и символ заполнения «$».
8. Реализовать функции для работы с матрицами:

а) функция производит перемножение матриц;

б) функция производит сложение двух матриц.

Память под матрицы выделять динамически. Необходимо освобождать память, выделенную под матрицы.

9. Реализовать функции для работы с одномерными массивами:

а) получить пересечение элементов массивов;

б) получить объединение элементов массивов.

Память под массивы отводить динамически. Необходимо освобождать память, выделенную под массивы.

10. В квадратной матрице порядка n найти наибольший элемент по модулю. Получить квадратную матрицу порядка n − 1 путем выбрасывания из исходной матрицы какой-либо строки и столбца, на пересечении которых расположен элемент с найденным значением. Память под матрицу выделять динамически. Необходимо освобождать память, выделенную под матрицу.

11. В двухмерном массиве среди чисел, стоящих на четных местах, определить минимальный положительный элемент массива и его индексы. Память под массив выделять динамически. Необходимо освобождать память, выделенную под массив.

12. Рассортировать положительные элементы каждой строки матрицы по убыванию. Отрицательные элементы оставить на своих местах. Память под матрицу выделять динамически. Необходимо освобождать память, выделенную под матрицу.

13. Рассортировать отрицательные элементы каждого столбца матрицы по возрастанию. Положительные элементы оставить на своих местах. Память под матрицу выделять динамически. Необходимо освобождать память, выделенную под матрицу.

14. Рассортировать элементы побочной диагонали квадратной матрицы порядка n по возрастанию. Память под матрицу выделять динамически. Необходимо освобождать память, выделенную под матрицу.

15. Рассортировать элементы главной диагонали квадратной матрицы порядка n по возрастанию. Память под матрицу выделять динамически. Необходимо освобождать память, выделенную под матрицу.

Лабораторная работа №2

Тема работы: классы и объекты.

Цель работы: изучить структуру класса, атрибуты доступа к компонентам класса; рассмотреть принцип работы конструкторов (с параметрами, с параметрами по умолчанию), конструктора копирования и деструктора при работе с объектом, статические и константные данные и методы.

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3−8].

Основная идея введения классов заключается в том, чтобы предоставить программисту средства для создания новых типов данных, которые могут использоваться так же, как и встроенные типы. Класс – это абстракция, точнее, это тип, определяемый пользователем. Например, мы можем задать структуру с именем point, которая содержит координаты ***x*** и ***y*** точки на экране дисплея:

```
struct point
{
   int x,y;
};
```

Пусть нам нужны две функции, которые позволяют нарисовать точку set_pixel() и прочитать ее координаты get_pixel():

```
void set_pixel(int, int);
void get_pixel(int *, int *);
```

В нашем примере данные и функции, работающие с этими данными, отделены друг от друга. Связь между ними можно установить, если задать структуру в виде

```
struct point
{
   int x,y;
   void set_pixel(int, int);
   void get_pixel(int *, int *);
};
```

Данные x и y, объявленные в приведенной структуре, называются компонентами-данными, а функции – компонентами-функциями, или методами. Объект объявляется следующим образом:

```
имя_класса имя_объекта;
```

Теперь мы можем обратиться к данным и вызвать функции, только указав имя объекта, к которому они принадлежат. Для этих целей можно использовать те же операции: точка «.» и «->». Рассмотрим пример.

```
#include<iostream>
#include <iomanip>
using namespace std;
struct point                          // объявление структуры
{
    int x,y;
    void set_pixel(int, int);         // установить значения переменных x и y
    void get_pixel(int *, int *);     // получить значения x и y
};
```

```
int main()
{
    setlocale(LC_ALL,"Russian");
    int a,b;
    point my_point, *pointer;
    pointer = &my_point;
    my_point.set_pixel(50,100);
    pointer -> get_pixel(&a,&b);
    cout << "a= " << a << " b= " << b << endl;
    return 0;
}
void point::set_pixel(int a, int b)
{   x=a;
    y=b;
}
void point::get_pixel(int *a, int *b)
{
    *a=x;
    *b=y;
}
```

Поскольку различные структуры могут иметь функции с одинаковыми именами, при описании функции необходимо указывать, для какой структуры она описывается:

```
void point::set_pixel(int x, int y)
{
    тело функции
}
```

Синтаксис описания функции, принадлежащей структуре, имеет следующий вид (рис. 1).

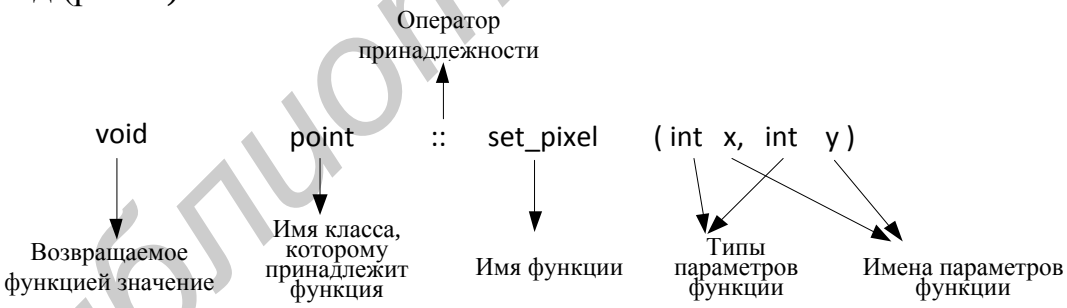

Рис. 1. Синтаксис описания функции, принадлежащей структуре

Оператор принадлежности «::» иначе называется оператором разрешения области видимости. Этот оператор самого высокого приоритета.

```
::i          // унарный оператор :: указывает на внешнюю область видимости
point::i     //  бинарный оператор :: указывает на область видимости класса
```

В языке C++ структура – это тоже класс. С другой стороны, мы можем записать вместо ключевого слова struct ключевое слово class, например:

```
class point
{
    int x,y;
```

```
    void set_pixel(int, int);
    void get_pixel(int *, int *);
};
```

В языке C++ класс, определяемый посредством ключевых слов struct, class, union, включает в себя функции и данные, создавая новый тип объектов. Компоненты класса имеют ограничения на доступ. Эти ограничения определяются ключевыми словами private, protected, public. Для ключевого слова class по умолчанию все компоненты будут private. Это означает, что они (их имена) будут недоступны для использования вне компонентов класса. Ограничения доступа для некоторого компонента можно изменить, записав перед ним атрибут модификации доступа – ключевое слово public или protected и двоеточие. Таким образом, упрощенную форму описания класса можно записать в виде

```
class имя_класса
{
      данные и функции с атрибутом private (по умолчанию)
  protected:
      данные и функции с атрибутом protected
  public:
      данные и функции с атрибутом public
} объекты этого класса через запятую.
```

Обычно ограничения на уровень доступа касаются элементов данных: данные имеют атрибут private или protected, а методы – public.

Смысл атрибутов доступа следующий:

− **private** – член класса с атрибутом private может использоваться только методами собственного класса и функциями-«друзьями» этого же класса; по умолчанию все члены класса, объявленного с ключевым словом class, имеют атрибут доступа private;

− **protected** – то же, что и private, но дополнительно член класса может использоваться методами и функциями-«друзьями» производного класса, для которого данный класс является базовым;

− **public** – член класса может использоваться любой функцией программы, т. е. защита на доступ снимается.

Явно ограничения на доступ могут переопределяться записью атрибута перед компонентами класса. Элементы класса типа структуры (struct) и объединения (union) по умолчанию принимаются как public. Для ключевого слова struct атрибут можно явно переопределить на private или protected. Для ключевого слова union явное переопределение атрибута доступа невозможно. Класс или структура может содержать любое количество секций с заданными атрибутами. Секция начинается с ключевого слова и двоеточия после него. Секция заканчивается в конце описания структуры (класса) или началом другой секции.

Пример использования атрибутов доступа к элементам класса.

```
#include<iostream>
#include <iomanip>
using namespace std;
 class String
```

```
{
     char str[25];                    // атрибут доступа private
  public:
     void set_string(char *);         // функция инициализации строки str
     void display_string();           // функция вывода строки str на экран
     char * return_string();          // функция возвращения строки
};
void String::set_string(char *s)
{
    strcpy(str,s);                    // копирование s в str
}
void String::display_string()
{
     cout << str << endl;
}
char * String::return_string()
{
     return str;
}
int main()
{
    setlocale(LC_ALL,"Russian");
    String str1;                      // объявление объекта
    str1.set_string("Минск");
    str1.display_string();
    cout << str1.return_string() << endl;
    return 0;
}
```

Использование функций для установки начальных значений данных объекта часто приводит к ошибкам. В связи с этим введена специальная функция, позволяющая инициализировать объект в процессе его декларирования (определения). Эта функция называется конструктором. Функция-конструктор имеет то же имя, что и соответствующий класс.

```
class String
{
    char str[25];          // атрибут доступа private
  public:
    String(char *s)        // конструктор
    {
       strcpy(str,s);
    }
};
```

Конструктор может иметь и не иметь аргументы и он никогда не возвращает значение (даже типа void). Класс может иметь несколько конструкторов, что позволяет использовать несколько различных способов инициализации соответствующих объектов. Иначе можно сказать – конструктор является функцией, а значит он может быть перегружен. Конструктор вызывается, когда связанный с ним тип используется в определении. Например:

```
#include<iostream>
using namespace std;
 class Over
{
      int i;
      char *str;
   public:
      Over()
      {
           str="Первый конструктор";
           i=0;
      }
      Over(char *s)           // Второй конструктор
      {
          str=s;
          i=50;
      }
      Over(char *s, int x)     // Третий конструктор
      {
          str=s;
          i=x;
      }
      Over(int *y)
      {
          str="Четвертый конструктор\n";
          i=*y;
      }
      void print();
};
void Over::print()
{
      cout << "i= " << i << " str= " << str << endl;
}
int main()
{     setlocale(LC_ALL,"Russian");
      int a=10, *b;
      b=&a;
      Over my_over;                // Активен конструктор Over()
      Over my_over1("Для конструктора с одним параметром");
      Over my_over2("Для конструктора с двумя параметрами", 100);
      Over my_over3(b);          // Для четвертого конструктора
      my_over.print();
      my_over1.print();
      my_over2.print();
      my_over3.print();
      return 0;
}
```

Результаты выполнения программы представляются в следующем виде:

i=0; str= Первый конструктор

i=50; str= Для конструктора с одним параметром

i=100; str= Для конструктора с двумя параметрами
i=10; str= Четвертый конструктор

Конструктор может содержать значения аргументов по умолчанию. Задание в конструкторе аргументов по умолчанию позволяет гарантировать, что объект будет находиться в непротиворечивом состоянии, даже если в вызове конструктора не указаны никакие значения. Созданный программистом конструктор, у которого все аргументы по умолчанию, называется конструктором с умолчанием, т. е. конструктором, который можно вызывать без указания каких-либо аргументов. Для каждого класса может существовать только один конструктор с умолчанием.

Пример использования конструктора по умолчанию.

```
#include<iostream>
using namespace std;
class Time
{
   public:
       Time(int = 0, int = 0, int = 0);  // конструктор с параметрами по умолчанию
       void  setTime(int, int, int);      // установка часов, минут, секунд
       void printStandart();              // печать времени в стандартном формате
   private:
        int hour;                         // 0–23
        int minute;                       // 0–59
        int second;                       // 0–59
}
  // Конструктор Time инициализирует члены класса нулевыми значениями
Time::Time(int hr, int min, int sec)
{
              setTime(hr, min, sec);
}
// Установка нового значения времени. Выполнение проверки корректности
// значений данных. Установка неправильных значений на 0.
void Time::setTime(int h, int m, int s)
{
    hour    = (h >= 0 && h < 24)  ?  h : 0;
    minute = (m >= 0 && m < 60)  ?  m : 0;
    second = (s >= 0 && s < 60) ?  s : 0;
}
 // Печать времени в стандартном формате
void Time::printStandart()
{
    cout << (( hour == 0 || hour == 12) ?  12 : hour % 12)
         << ":" << ( minute < 10 ?  "0" : "" ) << minute
         << ":" << ( second  < 10 ?  "0" : "" ) << second
         <<  ( hour < 12 ?  "AM" : "PM" );
}
int main()
{
    setlocale(LC_ALL,"Russian");
```

```
    Time t1,                    // все аргументы являются умалчиваемыми
     t2(2),                  // минуты и секунды являются умалчиваемыми
     t3(21, 34),             // секунды являются умалчиваемыми
     t4(12, 25, 42),         //  все значения указаны
     t5(27, 74, 99);         // все неправильные значения указаны
     cout << "\nВсе аргументы по умолчанию ";
     t1.printStandart();
     cout << "\nЧасы заданы; минуты и секунды по умолчанию ";
     t2.printStandart();
     cout << "\nЧасы и минуты заданы;  секунды по умолчанию ";
     t3.printStandart();
     cout << "\nЧасы, минуты и секунды заданы: ";
     t4.printStandart();
     cout << "\nВсе значения заданы неверно: ";
     t5.printStandart();
     cout << endl;
     return 0;
}
```

Конструктор имеет следующие отличительные особенности:

− всегда выполняется при создании нового объекта, т. е. когда под объект отводится память и когда он инициализируется;

− может определяться пользователем или создаваться по умолчанию;

− не может быть вызван явно из пределов программы (не может быть вызван как обычный метод). Он вызывается явно компилятором при создании объекта и неявно при выполнении оператора new для выделения памяти объекту;

− всегда имеет то же имя, что и класс, в котором он определен;

− никогда не должен возвращать значения;

− не наследуется.

Копирующий конструктор. Рассмотрим следующую программу, демонстрирующую использование копирующего конструктора.

```
#include<iostream>
#include<iomanip>
using namespace std;
class Massiv
{
    int *mas;                   // указатель на массив
    int n;              // количество элементов массива
 public:
    Massiv(int n1=0);       // конструктор с параметрами по умолчанию
    void vvod();            // функция ввода значений массива
    void display();         // функция вывода значений массива
    int fun(Massiv ob);     // функция вычисления суммы элементов массива
    ~Massiv();              // деструктор
 };
Massiv::Massiv(int n1)      // определение конструктора
{
     n=n1;
     mas=new int[n];
```

```
}
Massiv::~Massiv()            // определение деструктора
{    delete [] mas;    }

void Massiv::vvod()          // определение функции ввода значений массива
{
     for(int i=0; i < n; i++)
          cin >> mas[i];
}
void Massiv::display()       // определение функции вывода значений массива
{
     for(int i=0; i < n; i++)
          cout << setw(4) << mas[i];
     cout << endl;
}
int Massiv:: fun(Massiv ob)   // функция вычисления суммы элементов массива
{
     int sum=0;
     for(int i=0; i < n; i++)
          sum+=ob.mas[i];
     return sum;
}
int  main()                          // головная функция
{
     setlocale(LC_ALL,"Russian");
     int summa;
     int n;                    // размер массива
     cout << "Введите размер массива" << endl;
     cin >> n;
     Massiv ob(n);             // объявление объекта
     cout << "Введите элементы массива" << endl;
     ob.vvod();                 // вызов функции vvod
     cout << "Исходный массив" << endl;
     ob.display();              // вызов функции display
     summa=ob.fun(ob);          // вызов функции fun
     cout << "сумма элементов массива" << setw(4) << summa << endl;
      ob.display();             // при вызове функции display() возникает ошибка
}
```

В функцию fun() передается значение объекта типа Massiv. Даже если вызванная функция ничего не будет выполнять, произойдет ошибка, связанная с динамическим выделением и освобождением памяти. Параметр в функции fun() является локальным (автоматическим объектом) в теле этой функции. Любой автоматический объект конструируется тогда, когда встречается его объявление, и разрушается, когда блок, в котором он описан, прекращает существование. После завершения функция прекращает существовать, в результате вызывается деструктор объекта ob.

В функции main() выполняются следующие действия:

− описание Massiv ob(n); задается конструирование нового объекта ob. Конструктор объекта ob выделяет (динамически) память под массив mas с помощью оператора new;

− вызывается функция fun(ob);

− значение объекта ob копируется из функции main в стек функции fun();

− копия объекта ob содержит указатель на ту же динамическую память (указатель на динамическую память в объекте-оригинале и объекте-копии имеет одинаковые значения);

− функция fun() завершается;

− вызывается деструктор для копии объекта ob, который разрушает динамически выделенную память под массив mas;

− указатель в оригинале объекта адресует несуществующую удаленную память.

Если в приведенном примере изменить заголовок функции fun(Massiv ob) на заголовок fun(Massiv& ob), то ошибка будет устранена. При необходимости можно оставить и исходное объявление функции. В этом случае надо устранить ошибку в самом классе Massiv. Когда объект ob копируется из функции main() в функцию fun, то должен вызываться конструктор для копирования. Общий вид конструктора копирования имеет следующий вид:

```
имя_класса (const имя_класса  & );
```

Так как в нашем классе такого конструктора нет, то вызывается конструктор, заданный по умолчанию. Этот конструктор строит точную копию всех данных объекта ob, что и приводит к ошибке. Если в классе Massiv задать явно конструктор для копирования, например:

```
Massiv(const Massiv& ob)
{
    n=ob.n;
    mas=new int[n];
    for(int i=0; i < n; i++)
        mas[i]=ob.mas[i];
}
```

то ошибка будет устранена. Таким образом, если в конструкторе некоторого класса Massiv осуществляется динамическое выделение памяти, такой класс должен иметь соответствующий конструктор для копирования, а также деструктор (для освобождения памяти).

Деструктор. Противоположные действия по отношению к действиям конструктора выполняют функции-деструкторы (destructor), или разрушители, которые уничтожают объект. Деструктор может вызываться явно или неявно. Неявный вызов деструктора связан с прекращением существования объекта из-за завершения области его определения. Явное уничтожение объекта выполняет оператор delete. Деструктор имеет то же имя, что и класс, но перед именем записывается знак тильда «~». Кроме того, деструктор не может иметь аргументы, возвращать значение и наследоваться.

Пример использования деструктора.

```
#include<iostream>
```

```
using namespace std;
class String
{
     int i;
  public:
     String(int j);            // объявление конструктора
     ~String();                // объявление деструктора
     void show_i(void);
};                             // конец объявления класса
String::String(int j)         // определение конструктора
{
     i=j;
    cout << "Работает конструктор" << endl;
}
void String::show_i(void)    // определение функции
{
    cout << "i= " << i << endl;
}
String::~String()                // определение деструктора
{
    cout << "Работает деструктор" << endl;
}
int main()
{
    setlocale(LC_ALL,"Russian");
    String my_ob1(25);     // инициализация объекта my_ob1
    String my_ob2(36);     // инициализация объекта my_ob2
    my_ob1.show_i();       // вызов функции show_i() класса String для my_ob1
    my_ob2.show_i();       // вызов функции show_i() класса String для my_ob2
    return 0;
}
```

Результаты работы программы следующие:

Работает конструктор
Работает конструктор
i=25
i=36
Работает деструктор
Работает деструктор

Деструкторы выполняются в обратной последовательности по отношению к конструкторам. Первым разрушается объект, созданный последним.

Пример использования конструкторов и деструктора. Разработать класс вектор (одномерный динамический массив). Методы класса: конструкторы, деструктор и несколько методов, выполняющих преобразование в массиве (например, нахождения максимального значения и сортировки).

```
#include <iostream>
#include <new>
#include<iomanip>
using namespace std;
// объявление класса
class vect
```

```
{
    int *v;              //  вектор (одномерный массив)
    int n;               //  размерность вектора
  public:
    vect();              //  конструктор без параметров
    vect(int, int*);     //  конструктор с двумя параметрами
    ~ vect();            //  деструктор
    void set();          //  инициализация вектора
    void print();        //  вывод вектора на экран
    int funk1();         //  нахождения максимума в массиве
};
// реализация методов класса
vect :: vect() : n(0),v(0){}    //  конструктор без параметров
vect :: vect(int nn, int *vv)   //  конструктор с двумя параметрами
{
    n=nn;                       // инициализация размерности
    v=new int[n];               // выделение памяти под вектор
    for(int i=0; i<n; i++)
       *(v+i)=*(vv+i);
}
vect :: ~ vect()                //  деструктор
{ delete [] v; }
void vect :: set()              //  инициализация вектора
{
    if(!v)
    {
       cout << "не выделена память под вектор" << endl;
       cout << "введите размерность вектора" << endl;
       cin >> n;
       v=new int[n];
   }
   cout << "Введите элементы в вектор" << endl;
   for(int i=0; i<n; i++)
      cin >> *(v+i);
}
void vect :: print()            //  функция вывода вектора на экран
{
    if(!v)
    {
       cout << "вектор пустой" << endl;
       return ;
    }
   for(int i=0; i<n; i++)
      cout << setw(4) << *(v+i);
   cout << endl;
}
int vect :: funk1()             // функция  нахождения максимума в массиве
{
    int max=*v;                 // инициализация максимального элемента
    for(int i=0; i < n; i++)
        if(max < *(v+i))
```

```
            max=*(v+i);
        return max;
}
int main()
{
      setlocale(LC_ALL,"Russian");
      int rez;                    // максимальное число объекта
      vect v1;                    // объявление объекта, вызывается конструктор
                                  // без параметров

      int ms[]={2,1,6,4,5};                    // объявление и инициализация массива
      vect v2((sizeof(ms)/sizeof(int)),ms);  // объявление объекта, вызывается
                                               // конструктор с параметрами
      v1.set();                                 // функция инициализирует объект v1
      cout << "Вектор v1" << endl;
      v1.print();                               // функция выводит на экран объект v1
      cout << "Вектор v2" << endl;
      v2.print();                               // функция выводит на экран объект v2
      rez=v2.funk1();                           // функция возвращает максимальное
                                                // число объекта
      cout << "Максимальное число = " << rez << endl;
      return 0;
}
```

Указатель this. Каждый новый объект имеет скрытый от пользователя свой указатель. Иначе это можно объяснить так. Когда объявляется объект, под него выделяется память. В памяти есть специальное поле, содержащее скрытый указатель, который адресует начало выделенной под объект памяти. Получить значение указателя в компонентах-функциях объекта можно с помощью ключевого слова this (рис. 2). Для любой функции, принадлежащей классу my_class, указатель this неявно объявлен так:

```
my_class *const this;
```

Если объявлен класс и объекты, то размещение объектов в памяти будет выглядеть следующим образом:

```
class string
{ . . .} str1, str2;
```

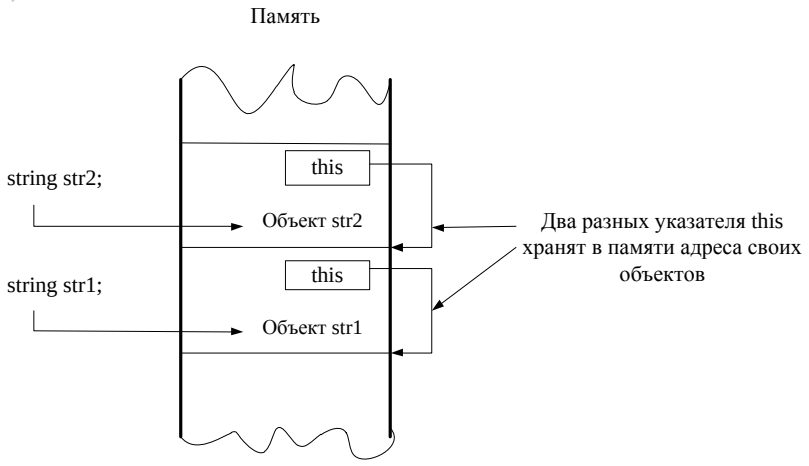

Рис. 2. Указатель this

Основные свойства и правила использования указателя this:

− каждый новый объект имеет свой скрытый указатель this;

− указывает на начало своего объекта в памяти компьютера;

− не надо дополнительно объявлять;

− передается как скрытый аргумент во все нестатические (т. е. не имеющие спецификатора static) компоненты-функции своего объекта;

− является локальной переменной, которая недоступна за пределами объекта (она доступна только во всех нестатических компонентах-функциях своего объекта);

− разрешается обращаться к указателю this непосредственно в виде this или *this.

Пример использования указателя this.

```
#include<iostream>
using namespace std;
#include<string.h>
class String                          // объявление класса
{
     char str[100];
     int k;
     int *r;
 public:
    String() {k=123; r=&k;}           // конструктор
    void read(){ gets(str);}          // функция чтения строки
    void print() { puts(str); }       // функция печати строки
    void read_print();
};
void String::read_print()
{
    char c;
    cout <<  this->k << endl;
    cout << "Введите строку" << endl;
    // ключевое слово this содержит скрытый указатель на класс String,
    // поэтому конструкция this->read() выбирает через указатель
    // функцию read() этого класса
    this->read();
    cout << "Введенная строка" << endl;
    this->print();
     // ниже, в цикле for, одинаково удаленные от середины строки
     // символы меняются местами
    for(int i=0, j=strlen(str)-1; i<j; i++,j--)
    {
        c=str[i];
        str[i]=str[j];
        str[j]=c;
    }
    cout << "Измененная строка" << endl;
    (*this).print();
}
```

```
int main()
{
    setlocale(LC_ALL,"Russian");
    String S;                    // объявление объекта
    S.read_print();              // вызов функции класса
}
```

static-члены (данные) класса. Компоненты-данные могут быть объявлены с модификатором класса памяти static. Класс, содержащий статические (static) компоненты-данные, объявляется как глобальный (локальные классы не могут иметь статических членов). static-компонент совместно используется всеми объектами этого класса и хранится в одном месте. Статический компонент глобального класса должен быть явно определен в контексте файла. Основные правила использования статических компонентов:

− статические компоненты будут одними для всех объектов данного класса, т. е. ими используется одна область памяти;

− статические компоненты не являются частью объектов класса;

− объявление статических компонентов-данных в классе не является их описанием; они должны быть явно описаны в контексте файла;

− локальный класс не может иметь статических компонентов;

− к статическому компоненту st класса cls можно обращаться cls::st независимо от объектов этого класса, а также при помощи операций «.» и «->» при использовании объектов этого класса;

− статический компонент существует даже при отсутствии объектов этого класса;

− статические компоненты можно инициализировать, как и другие глобальные объекты, только в файле, в котором они объявлены.

Компоненты-функции static и const. В C++ компоненты-функции могут использоваться с модификатором static и const. Обычный компонент-функция имеет явный список параметров и неявный список параметров. Неявные параметры можно представить как список параметров, доступных через указатель this. Статический (static) компонент-функция не может обращаться к любому из компонентов посредством указателя this. Компонент-функция const не может изменять неявные параметры.

Основные свойства и правила использования static- и const-функций:

− статические компоненты-функции не имеют указателя this, поэтому обращаться к нестатическим компонентам класса можно только с использованием «.» или «->»;

− не могут быть объявлены две одинаковые функции с одинаковыми именами и типами аргументов, чтобы при этом одна была статической, а другая нет;

− статические компоненты-функции не могут быть виртуальными.

Пример использования статических и константных (static, const) данных и методов.

```
#include <iostream>
using namespace std;
```

```
class cls
{
    int kl;                  // количество изделий
    double zp;               // зарплата на производство одного изделия
    double nl1,nl2;          // два налога на зарплату
    double sr;               // количество сырья на производство одного изделия
    static double cs;        // цена сырья на одно изделие
  public:
    cls(){}                  // конструктор по умолчанию
    ~cls(){}                 // деструктор
    void inpt(int);
    static void vvod_cn(double);
    double seb() const;
};
double cls::cs;              // явное определение static-члена в контексте файла
void cls::inpt(int k)
{
     kl=k;
     cin >> nl1 >> nl2 >> zp;
}
void cls::vvod_cn(double c)
{
     cs=c;           // можно обращаться в функции только к static-компонентам
}
double cls::seb() const
{
     return kl*(zp+zp*nl1+zp*nl2+sr*cs); // в функции нельзя изменить ни один
}                                        // неявный параметр (kl zp nl1 nl2 sr)
int main()
{
    setlocale(LC_ALL,"Russian");
    cls c1,c2;
    cout << "Введите зарплату и два налога для объекта c1" << endl;
    c1.inpt(100);                        //  инициализация первого объекта
    cout << "Введите зарплату и два налога для объекта c2" << endl;
    c2.inpt(200);                        //  инициализация второго объекта
    cls::vvod_cn(500.);
    cout << "\nc1 = " << c1.seb() << "\nc2 = " << c2.seb() << endl;
    return 0;
}
```

Контрольные вопросы

1. В чем разница между struct, class и union?
2. Что такое указатель this? Приведите пример использования этого указателя.
3. Какова основная форма конструктора копирования и когда он вызывается?
4. Когда вызывается деструктор?
5. Приведите пример использования константных и статических данных и методов.

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Создать класс, в котором реализовать функции для работы с матрицами:

а) функция производит перемножение матриц;

б) функция производит сложение двух матриц.

Память под матрицы отводить динамически. Использовать конструктор с параметрами, конструктор копирования. Деструктор должен освобождать память, выделенную под матрицы.

2. Создать класс, в котором реализовать функции для работы с одномерными массивами:

а) получить пересечение элементов массивов;

б) получить объединение элементов массивов.

Память под массивы отводить динамически. Использовать конструктор с параметрами, конструктор копирования. Деструктор должен освобождать память, выделенную под массивы.

3. Создать класс, в котором реализовать функции для работы с двухмерными массивами:

а) получить пересечение элементов массивов;

б) получить объединение элементов массивов.

Память под массивы отводить динамически. Использовать конструктор с параметрами, конструктор копирования. Деструктор должен освобождать память, выделенную под массивы.

4. Создать класс employee. Класс должен включать поле int для хранения номера сотрудника и поле float для хранения величины его оклада. Расширить содержание класса employee, включив в него класс date и перечисление etype. Объект класса date использовать для хранения даты приема сотрудника на работу. Перечисление использовать для хранения статуса сотрудника: лаборант, секретарь, менеджер и т. д. Разработать методы getemploy() и putemploy(), предназначенные соответственно для ввода и отображения информации о сотруднике. Написать программу, которая будет выдавать сведения о сотрудниках.

5. Создать класс, одно из полей которого хранит «порядковый номер» объекта, т. е. для первого созданного объекта значение этого поля равно 1, для второго − 2 и т. д. Для того чтобы создать такое поле, необходимо иметь еще одно поле, в которое будет записываться количество созданных объектов. Каждый раз при создании нового объекта, конструктор может получить значение этого поля и в соответствии с ним назначить объекту индивидуальный порядковый номер. В

классе разработать метод, который будет выводить на экран свой порядковый номер, например «Мой порядковый номер: 2».

6. Создать класс «Время» с полями: часы (0−23), минуты (0−59), секунды (0−59). В классе реализовать функции конструктора, деструктора, установки времени, получения часа, минуты и секунды, а также две функции-члены печати: печать по шаблону «16 часов 18 минут 3 секунды» и «4:18:3 p.m.». Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

7. Создать класс «Квадрат». Поле класса – длина стороны квадрата. Функции-члены вычисляют площадь, периметр, устанавливают поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Разработать функцию вывода на экран результата.

8. Создать класс «Окружность». Данные класса: радиус окружности. Функции-члены класса: вычисление площади, вычисление длины окружности, установление радиуса окружности и возвращение значения радиуса окружности, вывод данных на экран. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

9. Создать класс «Прямоугольник». Данные класса: высота и ширина прямоугольника. Функции-члены класса: вычисление площади, периметра, установление данных класса и возвращение их значений, вывод данных на экран. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

10. Создать класс типа «Односвязный список». Функции-члены добавляют элемент к списку, удаляют элемент из списка, печатают элементы с начала списка. Найти элемент в списке.

11. Создать класс типа «Стек». Функции-члены вставляют элемент в стек, удаляют элемент из стека, печатают элементы списка.

12. Создать класс, в котором реализуются функции для работы с одномерными массивами:

а) функция должна найти максимальное число в одном массиве;

б) функция должна найти минимальное число во втором массиве;

в) функция должна поменять местами минимальное и максимальное значения в массивах.

Память под массивы отводить динамически. Использовать конструктор с параметрами. Деструктор должен освобождать память, выделенную под массивы.

13. Создать класс, в котором реализовать функции для работы с двухмерными массивами:

а) функция должна обнулить строку и столбец с минимальным элементом;

б) функция должна в каждой строке массива первый найденный нечетный элемент поменять местами с первым элементом этой строки.

Память под массивы отводить динамически. Использовать конструктор с параметрами. Деструктор должен освобождать память, выделенную под массивы.

14. Создать класс, в котором надо реализовать функции для работы с двухмерными массивами:

а) функция находит минимальный элемент ниже главной диагонали;

б) функция находит максимальный элемент выше главной диагонали.

Память под массивы отводить динамически. Использовать конструктор с параметрами. Использовать константные и статические данные и методы. Деструктор должен освобождать память, выделенную под массивы.

15. Создать класс, в котором требуется реализовать функции для работы с матрицами:

а) функция находит минимальное число в матрице, максимальное число и меняет их местами;

б) функция выполняет обмен элементов между строками матрицы: первой и второй, третьей и четвертой и т. д.

Память под матрицы отводить динамически. Использовать конструктор с параметрами по умолчанию. Использовать константные и статические данные и методы. Деструктор должен освобождать память, выделенную под матрицы.

Лабораторная работа №3

Тема работы: дружественные функции и классы, перегрузка операторов.

Цель работы: понять назначение дружественных функций и классов, изучить принципы перегрузки бинарных и унарных операций.

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3−8].

Дружественные функции. Иногда возникает необходимость организации доступа к локальным данным нескольких классов из одной функции. Для реализации этого в С++ введен спецификатор friend. Если некоторая функция определена как friend-функция для некоторого класса, то эта функция называется дружественной и она:

− не является компонентом-функцией этого класса;

− имеет доступ ко всем компонентам этого класса (private, public и protected).

Пример использования дружественной функции.

```
#include <iostream>
using namespace std;
class kls
{
    int i,j;
 public:
    kls(int I,int J) : i(I),j(J) {}          // конструктор
    int max() {return   i>j? i : j;}         // функция-компонент класса kls
    friend double fun(int, kls&);            // friend-объявление внешней функции fun
};
double fun(int i, kls &x)                    // описание дружественной функции
{
    return (double)i/x.i;
}
int main()
{
    kls obj(2,3);                            // объявление объекта
    cout  << obj.max() << endl;
    cout << fun(3,obj) << endl;              // вызов дружественной функции
    return 0;
}
```

Функции со спецификатором friend, не являясь компонентами класса, не имеют и, следовательно, не могут использовать this указатель. Следует также отметить ошибочность следующей заголовочной записи функции

```
double kls :: fun(int i,int j),
```

т. к. fun не является компонентом-функцией класса kls.

В общем случае friend-функция является глобальной независимо от секции, в которой она объявлена (public, protected, private), при условии, что она не объявлена ни в одном другом классе без спецификатора friend. Функция friend, объявленная в классе, может рассматриваться как часть интерфейса класса с внешней средой.

Вызов компонента-функции класса осуществляется с использованием

операции доступа к компоненту «.» или «->». Вызов же friend-функции производится по ее имени. В friend-функцию не передается this-указатель и доступ к компонентам класса выполняется либо явно «.», либо косвенно «->».

Компонент-функция одного класса может быть объявлена со спецификатором friend для другого класса:

```
class X{  …….
            void fun (….);
          };
class Y{  …….
            friend void X:: fun (….);
          };
```

В приведенном фрагменте функция fun() имеет доступ к локальным компонентам класса Y. Запись вида friend void X:: fun (….) говорит о том, что функция fun принадлежит классу X, а спецификатор friend разрешает доступ к локальным компонентам класса Y (т. к. она объявлена со спецификатором в классе Y).

Ниже приведен пример программы расчета суммы двух налогов на зарплату. Используется дружественная функция.

```
#include <iostream>
#include "string.h"
#include <iomanip>
using namespace std;
class nalogi;                  // неполное объявление класса nalogi
class work
{
    char s[20];                // фамилия работника
    int zp;                    // зарплата
 public:
    float raschet(nalogi);     // компонент-функция класса work
    void inpt()
    {
        cout << "вводите фамилию и зарплату" << endl;
        cin >> s >> zp;
    }
    work(){}
    ~work(){};
};
class nalogi
{
   float pd,                                  // подоходный налог
   st;                                        // налог на социальное страхование
   friend float work::raschet(nalogi); // friend-функция класса nalogi
 public:
   nalogi(float f1,float f2) : pd(f1),st(f2){};
  ~nalogi(void){};
};
float work::raschet(nalogi nl)
{
   cout << s << setw(6) << zp <<endl;          // доступ к данным класса work
```

```
    cout << nl.pd << setw(8) << nl.st <<endl;   // доступ к данным класса nalogi
    return zp*nl.pd/100+zp*nl.st/100;
}
int main()
{
    setlocale(LC_ALL,"Russian");
    nalogi nlg((float)12,(float)2.3);     // описание и инициализация объекта
    work wr[2];                           // описание массива объектов
    for(int i=0;i<2;i++)
       wr[i].inpt();                      // инициализация массива объктов
    cout << setiosflags(ios::fixed) << setprecision(3) << endl;
    cout << wr[0].raschet(nlg) << endl;     // расчет налога для объекта wr[0]
    cout << wr[1].raschet(nlg) << endl;     // расчет налога для объекта wr[1]
    return 0;
}
```

Следует отметить необходимость выполнения неполного (предварительного) объявления класса nalogi, т. к. в прототипе функции raschet класса work используется объект класса nalogi, объявляемого далее. Все функции одного класса можно объявить со спецификатором friend по отношению к другому классу следующим образом:

```
class X{ ……..
            friend class Y;
            ……..
         };
class Y{  …….
         };
```

В этом случае все компоненты-функции класса Y имеют спецификатор friend для класса X (имеют доступ к компонентам класса X). Класс Y является дружественным классу X.

В приведенном ниже примере оба класса являются дружественными.

```
#include <iostream>
using namespace std;
class A
{
    int i;                // компонент-данное класса A
  public:
    friend class B;   // объявление  класса B другом класса A
    A():i(1){}            // конструктор
    ~A(){}                // деструктор
    void f1_A(B &); // метод, оперирующий данными обоих классов
};
class B
{
    int j;                // компонент-данное класса B
  public:
    friend A;             // объявление класса A другом класса B
    B():j(2){}            // конструктор
    ~B(){}                // деструктор
void f1_B(A &a) // метод, оперирующий данными обоих классов
```

```
        {
        cout << a.i + j << endl;
    }
};
void A :: f1_A(B &b)
{
    cout << i << ' ' << b.j<< endl;
}
int main()
{
    A aa;
    B bb;
    aa.f1_A(bb);
    bb.f1_B(aa);
}
```

Результат выполнения программы:

```
1  2
3
```

В объявлении класса A содержится инструкция friend class B, являющаяся и предварительным объявлением класса B, и объявлением класса B дружественным классу A. Отметим также необходимость описания функции f1_A после явного объявления класса B (в противном случае не может быть создан объект b еще не объявленного типа).

Отметим основные свойства и правила использования спецификатора friend:

− friend-функции не являются компонентами класса, но имеют доступ ко всем его компонентам независимо от их атрибута доступа;

− friend-функции не имеют указателя this;

− friend-функции не наследуются в производных классах;

− отношение friend не является *ни симметричным* (т. е. если класс A friend классу B, то это не означает, что B также является friend классу A), *ни транзитивным* (т. е. если A friend B и B friend C, то не следует, что A friend C);

− друзьями класса можно определить перегруженные функции. Каждая перегруженная функция, используемая как friend для некоторого класса, должна быть явно объявлена в классе со спецификатором friend.

Перегрузка операторов. Программы на языке C++ используют некоторые ранее определенные простейшие классы (типы), такие, как int, char, float и т. д. Мы можем описать объекты указанных классов, например:

```
int a,b;
char c,d,e;
float f;
```

Здесь переменные a, b, c, d, e, f можно рассматривать как простейшие объекты. В языке определено множество операций над простейшими объектами, выражаемых через операторы, такие, как «+», «-», «*», «/», «%» и т. д. Каждый оператор можно применить к операндам определенного типа.

```
float a, b=3.123, c=6.871;
a=c+b;          // нет ошибки
```

```
a=c%b;        // ошибка
```

Второе является ошибочным, поскольку оператор «%» должен быть приложен лишь к объектам целого типа. Из этого следует, что операторы языка можно применить к тем объектам, для которых они были определены.

К сожалению, лишь определенное число типов непосредственно поддерживается любым языком программирования. Однако многие операторы можно определить через классы в C++. Рассмотрим пример.

Пусть заданы множества A и B:

A={a1,a2,a3}; B={a3,a4,a5};

Необходимо выполнить операции пересечения множеств «&» и объединения множеств «|»:

A&B={a1,a2,a3} & {a3,a4,a5}={a3};

A|B={a1,a2,a3} | {a3,a4,a5}={a1,a2,a3a4,a5};

Языки C/C++ не поддерживают непосредственно эти операции, однако в языке C++ можно объявить класс, назвав его set (множество). Далее можно определить операции над этим классом, выразив их с помощью знаков операторов, которые уже есть в языке C++, например «&» и «|». В результате операции «&» и «|» можно будет использовать, как и раньше, а также снабдить их дополнительными функциями (объединения и пересечения множеств). Как определить, какую функцию должен выполнять оператор: старую или новую? Надо посмотреть на тип операндов в соответствующем выражении. Если операнды – это объекты целого типа, то нужно выполнить операцию «побитового И» или «побитового ИЛИ». Если же операнды – это объекты типа set, то надо выполнить объединение или пересечение соответствующих множеств.

Функция operator. Функция operator может быть использована для расширения области приложения следующих операторов: «+», «-», «*», «%», «&», «|» и т. д. Операторы, которые не могут быть перегружены: «.», «.*», «::», «?:», sizeof и typeid.

Для перегрузки (доопределения) оператора разрабатываются функции, являющиеся либо компонентами, либо friend-функциями того класса, для которого они используются. Для того чтобы перегрузить оператор, требуется определить действие этого оператора внутри класса. Общая форма записи функции-оператора, являющейся компонентом класса, имеет вид

```
тип_возв_значения имя_класса::operator#(список аргументов)
{
    действия, выполняемые применительно к классу
}
```

Вместо символа «#» ставится значок перегружаемого оператора. Оператор всегда определяется по отношению к компонентам некоторого класса. В результате старое предназначение оператора сохраняет силу. Функция operator является компонентом класса. При этом в случае унарного оператора функция operator не будет иметь аргументов, а в случае бинарного оператора будет иметь один аргумент. В качестве отсутствующего аргумента используется указатель this на тот объект, в котором определен оператор. Объявление и вызов функции

operator осуществляется так же, как и любой другой функции. Единственное ее отличие заключается в том, что разрешается использовать сокращенную форму ее вызова. Так, выражение operator#(a,b), можно записать в сокращенной форме a#b.

Рассмотрим пример. Программа доопределяет значение оператора «&». В результате ее можно будет использовать для выполнения операции пересечения множеств.

```
#include<iostream>
using namespace std;
class set                          // класс «множество»
{
    char str[80];                  // строка
  public:
    set(char *ss)                  // это конструктор
    {
      strcpy(str,ss);
    }
    char * operator&(set);         // объявление функции operator
};
char * set::operator&(set S)       // описание функции operator
{
    int t=0, l=0;
    while(str[t++]!='\0');         // вычисление длины строки
    char *s1=new char[t];          // выделение памяти под строку
    for(int j=0;  str[j]!='\0'; j++)
        for(int k=0; S.str[k]!='\0'; k++)
            if(str[j]==S.str[k])
            {
                s1[l]=str[j];
                l++;
                break;
            }
            s1[l]='\0';
            return s1;
}
int main()
{
    set S1="1f2bg5e6", S2="abcdef";          // задаются два множества
    cout << (S1 & S2) << endl;               // результат fbe
    cout << (set("123") & set("426")) << endl;  // результат 2
}
```

Ниже приведен пример программы перегрузки унарного оператора.

```
#include <iostream>
using namespace std;
class dek_koord
{
     int x,y;                          //  декартовы координаты точки
  public:
    dek_koord(){};                     // конструктор без параметров
```

```
    dek_koord(int X,int Y)            // конструктор с параметрами
    {
        x=X;
        y=Y;
    }
    void operator++();                // перегрузка префиксного оператора ++
    void operator++(int);             // перегрузка постфиксного оператора ++
   dek_koord operator=(dek_koord);// перегрузка оператора =
   void see();
};
void dek_koord::operator++()          // определение перегрузки оператора ++A
{
    x++;
}
void dek_koord::operator++(int)       // определение перегрузки оператора A++
{
    y++;
}
dek_koord dek_koord::operator =(dek_koord a)
{
     x=a.x;                           // определение перегрузки оператора =
     y=a.y;
     return *this;
}
void dek_koord::see()
{
     cout << "координата x = " << x << endl;
     cout << "координата y = " << y << endl;
}
int main()
{
    setlocale(LC_ALL,"Russian");
    dek_koord A(1,2), B, C; // объявление объектов
    A.see();
    A++;                    //  увеличение значения  компонента x объекта A
    A.see();                //  просмотр содержимого объекта A
   ++A;                     //  увеличение значения  компонента y объекта A
   A.see();                 //  просмотр содержимого объекта A
   C=B=A;                   // множественное присваивание
   B.see();
   C.see();
   return 0;
}
```


Контрольные вопросы

1. Почему может потребоваться перегрузка оператора присваивания?
2. Можно ли изменить приоритет перегруженного оператора?
3. Когда следует переопределять операторы с помощью дружественных функций, а когда с помощью функций элементов класса?

4. Назовите особенности дружественных функций.

5. Опишите особенности перегрузки постфиксных и префиксных операторов «++» и «--».

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Реализовать класс String для работы со строками символов. Перегрузить операторы «=», «+=», «==», «<», «>», «!=», «<=», «>=». Предоставить конструктор копирования. Определить friend-функции для операторов ввода/вывода в поток.

2. Реализовать класс String для работы со строками символов. Перегрузить для объектов класса String операторы «!» (пуст ли объект класса), «()(int,int)» (возвращение подстроки объекта), «[]» (возвращение некоторого символа строки объекта). Предоставить конструктор копирования. Определить friend-функции для операторов ввода/вывода в поток.

3. Создать класс, в котором перегрузить операторы:

а) «&» для перемножения двух матриц;

б) «+» для сложения двух матриц.

Память под матрицы отводить динамически. Предоставить конструктор копирования. Определить friend-функции для операторов ввода/вывода в поток.

4. Реализовать класс String для работы со строками символов. Перегрузить операторы «=», «+=» так, чтобы производилось сложение строки и объекта. Предоставить конструктор копирования. Определить friend-функции для операторов ввода/вывода в поток.

5. Создать класс bMoney, в котором необходимо перегрузить арифметические операторы для работы с денежным форматом. Перегрузить два оператора следующим образом:

```
long double * bMoney      // умножить число на деньги
long double / bMoney      // делить число на деньги
```

Эти операторы требуют наличия дружественных функций, т. к. справа от оператора находится объект, а слева – обычное число. Необходимо убедиться, что main() позволяет пользователю ввести две денежные строки и число с плавающей запятой, а затем корректно выполнить все семь арифметических действий с соответствующими парами значений.

6. Написать функцию инкремента единственного параметра. Написать функцию, возвращающую ссылку на передаваемый параметр. Изменить его при вызове функции. Что произойдет, если все ссылки сделать const?

7. Написать функцию декремента единственного параметра. Написать функцию, возвращающую ссылку на элемент глобального массива. Изменить его при вызове функции. Что произойдет, если все ссылки сделать const?

8. Реализовать класс String для работы со строками символов. Перегрузить операторы «=», «+=» так, чтобы производилось сложение строки и объекта. Предоставить конструктор копирования. Определить friend-функции для операторов ввода/вывода в поток.

9. Реализовать класс String для работы со строками символов. Перегрузить операторы «+» (сложение строк), «>» (сравнение строк). Предоставить конструктор копирования.

10. Реализовать класс String для работы со строками символов. Перегрузить оператор «-» (минус) так, чтобы определить, насколько одна строка длиннее другой. Предоставить конструктор копирования.

11. Реализовать класс String для работы со строками символов. Перегрузить оператор «>» так, чтобы возвратить разность кодов первой пары несовпадающих символов в строках. Предоставить конструктор копирования.

12. Создать класс, в котором перегрузить оператор «&» для пересечения двух множеств. Память под матрицы отводить динамически. Предоставить конструктор копирования.

13. Создать класс, в котором перегрузить оператор «+» для объединения двух множеств. Память под матрицы отводить динамически. Предоставить конструктор копирования.

14. Реализовать класс String для работы со строками символов. Перегрузить унарные операторы «++» (префиксную и постфиксную). Предоставить конструктор копирования.

15. Создать два класса: вектор и матрица. Определить конструкторы (по умолчанию, с параметрами, копирования), деструкторы. Определить функцию умножения матрицы на вектор как дружественную.

Лабораторная работа №4

Тема работы: наследование, простое наследование.

Цель работы: изучить принципы наследования.

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3−8].

Наследование – это механизм получения нового класса из существующего. Существующий класс может быть дополнен или изменен для создания производного класса. При создании нового класса вместо написания полностью новых данных и функций программист может указать, что новый класс должен наследовать данные и функции ранее определенного базового класса. Этот новый класс называется производным классом. Каждый производный класс сам является кандидатом на роль базового класса для будущих производных классов. При простом наследовании класс порождается одним базовым классом. При множественном наследовании производный класс наследуется несколькими базовыми классами. Производный класс обычно добавляет свои данные и функции, так что производный класс в общем случае больше своего базового.

Механизм наследования помогает сделать разработку более экономной и читаемой. Совокупность классов можно описать в виде такой иерархической структуры, что, если класс B наследует структуру и поведение класса A, то класс A называется базовым, а класс B – производным (рис. 3).

Рис. 3. Диаграмма наследования классов

Различают простое наследование и множественное. В первом случае производный класс имеет один базовый класс (рис. 4).

Рис. 4. Простое наследование классов

Шаблон объявления производного класса можно представить следующим образом:

```
ключ_класса имя_производного_класса :
необязательный_модификатор_доступа имя_базового_класса
{
    Тело производного класса
};
```

Пример объявления производного класса.

```
class Location                  // базовый класс
{
       int x,y;
   public:
         . . .
};
class Point:public Location  // производный класс
{
    Тело класса Point
};
```

Класс Location является базовым и наследуется с атрибутом public. Класс Point – производный класс. Двоеточие (:) отделяет производный класс от базового. Атрибут класса (модификатор прав доступа) может задаваться ключевыми словами public и private. Атрибут может опускаться – в этом случае принимается атрибут по умолчанию (для ключевого слова class – private, для struct – public). Объединение (union) не может быть базовым или производным классом.

Модификатор прав доступа используется для изменения прав доступа к наследуемым элементам класса в соответствии с правилами, представленными в табл. 1.

Таблица 1

Ограничения на доступ в базовом классе	Модификатор наследования прав	Ограничения на доступ в производном классе
private	private	Нет доступа
protected	private	private
public	private	private
private	public	Нет доступа
protected	public	protected
public	public	public

Отметим, что в производных классах права на доступ к элементам базовых классов не могут быть расширены, а только более ограничены.

Пример использования прав доступа к элементам базового класса.

```
#include<iostream>
using namespace std;
#include<string.h>
class A                         // базовый класс
{
     int a1;
  public:
```

```
        int a2;
        void f1();
};
class B:A                    // производный класс
{
     int b1;
   public:
     void f1()
    {
          a1=1;          // ошибка, a1 – private-переменная класса A и доступна
                         // только для методов и дружественных функций
                         // собственного класса
          b1=0;          // доступ к переменной типа private из метода класса
          a2=1;          // a2 унаследована из класса A с атрибутом доступа private
                         // и поэтому доступна в методе класса
    }
};
int main()
{
    A a_ob1;         // объявление объекта a_ob1 класса A
    B b_ob1;         // объявление объекта  b_ob1 класса B
    b_ob1.a2+=1;  // ошибка, т. к. a2 private
    a_ob1.a2+=1;  // допустимая операция
    return 0;
}
```

Рассмотрим еще пример, демонстрирующий наследование прав доступа к элементам базовых классов.

```
#include<iostream>
using namespace std;
#include<string.h>
#define N 10
class book                   // базовый класс
{
  protected:
       char naz[20];          // название книги
       int kl;                // количество страниц
  public:
     book(char *, int);       // конструктор класса book
     ~book();                 // деструктор класса book
};
class avt: public book        // производный класс
{
     char fm[10];             // фамилия автора
  public:
     avt(char *, int, char *); // конструктор класса avt
     ~avt();                   // деструктор класса avt
     void see();
};
enum razd{teh, hyd,uch};   // перечисление
class rzd: public book
```

```
{
      razd rz;
  public:
      rzd(char *, int, razd);    // конструктор класса rzd
      ~rzd();                    // деструктор класса rzd
      void see();
};
book::book(char *s1, int i)
{
    cout << "\n Конструктор класса book" << endl;
    strcpy(naz,s1);
    kl=i;
}
book::~book()
{
     cout << "\nДеструктор класса book" << endl;
}
avt::avt(char *s1, int i, char* s2):book(s1,i)
{
      cout << "\n  Конструктор класса avt" << endl;
      strcpy(fm,s2);
}
 avt::~avt()
{
     cout << "\nДеструктор класса avt" << endl;
}
void avt::see()
{
     cout << "\nНазвание книги  " << naz << endl << "\nКоличество страниц " <<
        kl <<endl;
}
rzd::rzd(char *s1, int i, razd tp):book(s1,i)
{
     cout << "\n Конструктор класса rzd" << endl;
     rz=tp;
}
rzd::~rzd()
{
     cout << "Деструктор класса rzd" << endl;
}
void rzd::see()
{
    switch(rz)
   {
       case teh:  cout << "\n Раздел технической литературы" << endl; break;
       case hyd: cout << "\nРаздел художественной литературы" << endl;
                 break;
       case uch: cout << "\n Раздел учебной литературы" << endl; break;
   }
}
```

```
int main()
{
    setlocale(LC_ALL,"Russian");
    avt av("Книга 1", 123, "автор 1");
    rzd rz("Книга 1", 123, teh);
    av.see();
    rz.see();
    return 0;
}
```

Если базовый класс имеет конструктор с одним или более аргументами, то любой производный класс должен иметь конструктор. В нашем примере конструктор класса book задан в виде

```
book::book(char *s1, int i)
{
    cout << "\nКонструктор класса book";
    strcpy(naz,s1);
    kl=i;
}
```

Теперь объявление объекта в функции main (либо в другой функции) может осуществляться

```
book my_ob("Дейтел", 1113);
```

В соответствии со сделанными выше замечаниями производный класс avt тоже должен иметь конструктор. В нашем примере он задан следующим образом:

```
avt::avt(char *s1, int i, char s2):book(s1,i)
{
     cout << "\n  Конструктор класса avt";
     strcpy(fm,s2);
}
```

Конструкторы и деструкторы производных классов. Если у базового и производного классов имеются конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы − в обратном порядке. Общий синтаксис конструктора производного класса следующий:

```
конструктор_производного_класса(арг) : base(арг)
{
        тело конструктора производного класса
}
```

Контрольные вопросы

1. Опишите модификаторы доступа и наследования. Как изменяются атрибуты доступа элементов класса при наследовании?
2. Как работают конструкторы при наследовании?
3. Как работают деструкторы при наследовании?
4. Какой класс называется производным?
5. Как объявляются производные классы?

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. При наличии классов Circle (круг), Square (квадрат) и Triangle (треугольник), производных от Shape (фигура), определить функцию intersect() (пересечение), которая принимает две фигуры Shape* в качестве аргументов и вызывает подходящие функции для определения того, пересекаются ли эти фигуры. Для решения этой задачи вам придется добавить подходящие (виртуальные) функции к классам. Не пишите код, проверяющий пересечение, просто убедитесь, что вызываются правильные функции. Такой подход часто называют двойной диспетчеризацией или мульти-методом.

2. Реализовать класс «Человек», включающий в себя фамилию, имя, отчество, год рождения и методы, позволяющие изменять/получать значения этих полей.

Реализовать следующие производные классы:

- «Преподаватель университета» с полями: должность, ученая степень, специальность, список научных трудов (массив строк);
- «Член комиссии» с полями: название комиссии, год назначения в комиссию, номер свидетельства, автобиография (массив строк);
- «Преподаватели – члены комиссии» (производный от первого и второго класса). Дополнительное поле – список работ, выполненных в комиссии.

Классы должны содержать методы доступа и изменения всех полей.

3. Реализовать класс «Человек», включающий в себя фамилию, имя, отчество, год рождения и методы, позволяющие изменять/получать значения этих полей.

Реализовать следующие производные классы:

- «Предприниматель» с полями: номер лицензии, адрес регистрации, УНН, данные о налоговых платежах (массив пар вида «дата, сумма»);
- «Турист» – содержит данные паспорта (строка), данные о пересечении границы в виде массива пар «дата, страна»;
- «Челнок» (производный от второго и третьего класса) – добавляется массив строк списка адресов, по которым покупается товар.

Классы должны содержать методы доступа и изменения всех полей.

4. Разработать базовый класс Fakultet, включающий в себя название факультета. Реализовать производный класс Student, включающий следующие компоненты данных: ФИО студента, год рождения, результаты сдачи последней сессии. Классы должны содержать методы доступа и изменения всех полей. Написать программу, которая выдает информацию об успеваемости студентов.

5. Разработать иерархию классов для нахождения корней квадратного уравнения $ax^2 + bx + c = 0$. A – базовый для класса B, B – базовый для класса C. Компоненты классов A, B, C – коэффициенты a, b, c. Корни уравнения и методы решения находятся в классе C.

В следующих заданиях требуется создать базовый класс и определить общие методы show( ), get( ), set( ) и другие специфические методы для данного класса. Создать производные классы, в которых добавить специфические свойства и методы. Часть методов переопределить. Создать массив объектов базового класса и заполнить объектами производных классов. Объекты производных классов идентифицировать конструктором по имени или идентификационному номеру. Использовать объекты производных классов для моделирования реальных ситуаций и объектов.

6. Создать базовый класс «Транспортное средство» и производные классы «Автомобиль», «Велосипед», «Повозка». Подсчитать время и стоимость перевозки пассажиров и грузов каждым транспортным средством.

7. Создать базовый класс «Грузоперевозчик» и производные классы «Самолет», «Поезд», «Автомобиль». Определить время и стоимость перевозки для указанных городов и расстояний.

8. Создать базовый класс «Пассажироперевозчик» и производные классы «Самолет», «Поезд», «Автомобиль». Определить время и стоимость передвижения пассажиров.

9. Создать базовый класс «Учащийся» и производные классы «Школьник» и «Студент». Создать массив объектов базового класса и заполнить этот массив объектами. Вывести информацию о студентах и школьниках.

10. Определить базовый класс для работы с матрицами, предусмотрев ввод, вывод матриц и выполнение следующих операций:

а) сложение матриц;

б) умножение матрицы на скаляр;

в) перестановка строк матрицы по заданному вектору транспозиции.

В производном классе реализовать указанные операции для квадратных матриц, добавив выполнение следующих операций:

а) транспонирование матрицы;

б) умножение матриц.

11. Расширить возможности стандартного класса Time так, чтобы была возможность выводить время дня: утро, вечер и т. д.

12. Расширить возможности стандартного класса Date так, чтобы была возможность выводить время года: зима, лето и т. д.

13. Написать программу для обслуживания клиентов магазина. Реализовать класс «Товар», «Отдел», «Корзина», «Банк». Каждый товар имеет характеристики (группа, тип, индивидуальные особенности, страна происхождения и т. д.) и штрих-код. При обслуживании клиента необходимо подготовить электронный чек, в котором должно быть указано название товара, его цена, количество, общая сумма покупки, дата и время покупки.

14. Написать программу, управляющую работой библиотеки. Создать классы: «Книга», «Отдел», «Библиотека». В классах реализовать следующие функции: добавление, удаление книг из отделов, выдача книг на абонемент. Классы должны содержать методы доступа и изменения всех полей.

15. Написать иерархию наследования для классов Quadrilateral (четырехугольник), Trapezoid (трапеция), Parallelogram (параллелограмм), Rectangle (прямоугольник) и Square (квадрат). Использовать Quadrilateral как базовый класс иерархии. Сделать иерархию настолько глубокой (т. е. настолько многоуровневой), насколько это возможно. Закрытыми данными класса Quadrilateral должны быть пары координат (x,y) четырех угловых точек Quadrilateral. Написать программу драйвер, который создает объекты каждого из этих классов.

Лабораторная работа №5

Тема работы: принцип полиморфизма, виртуальные функции, абстрактные классы.

Цель работы: изучить реализацию принципа полиморфизма через использование виртуальных функций при наследовании.

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3−8].

Виртуальные функции. В языке C++ полиморфизм реализуется посредством виртуальных функций. Виртуальная функция – это функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или нескольких производных этого класса. Объявление

```
virtual void print(void);
```

говорит о том, что функция print может быть различной для базового и разных производных классов. В производных классах функция может иметь список параметров, отличный от параметров виртуальной функции базового класса. В этом случае эта функция будет не виртуальной, а перегруженной. Механизм вызова виртуальных функций можно пояснить следующим образом. При создании нового объекта для него выделяется память. Для виртуальных функций (и только для них) создается указатель на таблицу функций, из которой выбирается требуемая функция в процессе выполнения. Если в некотором классе задана хотя бы одна виртуальная функция, то все объекты этого класса содержат указатель на связанную с их классом виртуальную таблицу. Эта таблица содержит адреса (указатели на первые инструкции) действительных функций, которые будут вызваны. Доступ к виртуальной функции осуществляется через этот указатель и соответствующую таблицу (т. е. осуществляется косвенный вызов функции). Если функция вызвана с использованием ее полного имени, то виртуальный механизм игнорируется. Свойство виртуальности проявляется только тогда, когда обращение к функции идет через указатель или ссылку на объект. Указатель или ссылка могут указывать как на объект базового, так и на объект производного класса.

Рассмотрим пример использования виртуальной функции.

```
#include<iostream>
#include<iomanip>
#include<string.h>
using namespace std;
class base                          //  базовый класс
{
  public:
     virtual char * name(void)
     {
        return "noname";
     }
     virtual double area(void)
     {   return 0;     }
};
class rect: public base         // производный класс «Прямоугольник»
{
```

```
    int h,s;                    // размеры прямоугольника
  public:
    rect(int H, int S)          // конструктор
    {
        h=H;
        s=S;
    }
    char * name(void);          // вывод на экран названия фигуры
    double area(void);          // вывод на экран площади фигуры
};
char * rect::name(void)         // вывод на экран названия фигуры
{
    return "прямоугольник";
}
double rect:: area(void)        // вывод на экран площади фигуры
{
    return h*s;
}
class circl: public base        // производный класс «Окружность»
{
   int r;                       // радиус окружности
public:
    circl(int R)                // конструктор
    {   r=R;    }
    char * name(void);          // вывод на экран названия фигуры
    double area(void);          // вывод на экран площади фигуры
};
char * circl::name(void)        // вывод на экран названия фигуры
{
    return  "круг";
}
double circl::area(void)        // вывод на экран площади фигуры
{
    return 3.14*r*r;
}
int main()
{
   setlocale(LC_ALL,"Russian");
   base *p[2];                  //  массив указателей на базовый класс
   rect obj1(3,4);
   circl obj2(5);
   p[0]=&obj1;
   p[1]=&obj2;
   for(int i=0; i < 2; i++)
      cout  << "площадь " << p[i]->name() << setw(10) << p[i]->area() << endl;
   return 0;
}
```

Массив указателей p хранит адреса объектов производных классов и необходим для вызова виртуальных функций этих классов. Если функции name() и area() в базовом классе объявлены как virtual и мы вызываем эти функции че-

рез указатель базового класса, указывающий на объекты производных классов, то программа будет динамически (т. е. во время выполнения программы) выбирать соответствующие функции name() и area() производного класса. Это называется динамическим связыванием (dynamic binding). Когда виртуальная функция вызывается путем обращения к заданному объекту по имени и при этом используется операция доступа к элементу «точка», тогда эта ссылка обрабатывается во время компиляции и это называется статическим связыванием.

Если функция была объявлена как виртуальная в некотором классе (базовом классе), то она остается виртуальной независимо от количества уровней в иерархии классов, через которые она прошла.

Приведем основные правила использования виртуальных функций:

– виртуальный механизм поддерживает полиморфизм на этапе выполнения программы. Это значит, что требуемая версия программы выбирается на этапе выполнения программы, а не компиляции;

– класс, содержащий хотя бы одну виртуальную функцию, называется полиморфным;

– виртуальные функции можно объявлять только в классах (class) и структурах (struct);

– виртуальными функциями могут быть только нестатические функции (без спецификатора static), т. к. характеристика virtual наследуется. Функция производного класса автоматически становится virtual;

– виртуальные функции можно объявлять со спецификатором friend для другого класса;

– виртуальными функциями могут быть только не глобальные функции (т. е. компоненты класса);

– если виртуальная функция объявлена в производном классе со спецификатором virtual, то можно рассматривать новые версии этой функции в классах, наследуемых из этого производного класса;

– для вызова виртуальной функции требуется больше времени, чем для невиртуальной. При этом также требуется дополнительная память для хранения виртуальной таблицы;

– при использовании полного имени при вызове виртуальной функции виртуальный механизм не поддерживается.

Приведем еще пример использования виртуальной функции.

```
#include<iostream>
#include<iomanip>
using namespace std;
#include<string.h>
class grup                          // базовый класс
{
  protected:
    char *spec;                     // название специальности
    long gr;                        // номер группы
  public:
```

```
    grup(char *SPEC, long GR)       // конструктор
    {
       spec=new char[40];
       strcpy(spec,SPEC);
       gr=GR;
    }
   ~grup()                          // деструктор
   {
       cout << "деструктор класса grup" << endl;
       delete [] spec;
   }
   virtual void print(void);        // объявление виртуальной функции
};
class asp: public grup              // производный класс
{
    char *fam;                      // фамилия
    int oc[4];                      // массив оценок
  public:
    // конструктор производного класса
    asp(char *SPEC, long GR, char *FAM, int OC[]):grup(SPEC,GR)
    {
        fam=new char[30];
        strcpy(fam,FAM);
        for(int i=0; i < 4; i++)
           oc[i]=OC[i];
    }
    ~asp()
    {
         cout << "Деструктор класса asp" << endl;
         delete [] fam;
    }
      void print(void);
};
void grup::print(void)              // определение виртуальной функции
{
      cout  <<  setw(10) << "Специальность" << setw(10) << "Группа" << endl;
      cout << setw(10) << spec << setw(10) << gr << endl;
}
void asp::print(void)
{
    grup::print();                  // вызов функции базового класса
    cout << setw(10) << "ФИО" << setw(10) << "Оценки" << endl;
    cout << setw(10) << fam ;
    for(int i=0; i < 4; i++)
  cout << setw(4) << oc[i];
    cout << endl;
}
int main()
{
    setlocale(LC_ALL,"Russian");
```

```
    int OC[]={4,5,5,4};          // массив оценок
    char SP[40];
    long GR;
    char FAM[40];
    cout << "Введите название специальности" << endl;
    gets(SP);
    cout << "Введите номер группы" << endl;
    cin >> GR;
    cout << "Введите фамилию" << endl;
    fflush(stdin);
    gets(FAM);
    grup ob1(SP,GR), *p;
    asp ob2(SP,GR,FAM,OC);
    cout << "Результат" << endl;
    //p=&ob1;              // указатель на объект базового класса
    //p->print();          // вызов функции базового класса
    p=&ob2;
    p->print();            // вызов функции производного класса
}
```

Абстрактные классы. Базовый класс обычно содержит ряд виртуальных функций, которые часто фиктивны и имеют пустое тело. Эти функции существуют как некоторая абстракция, конкретное значение им придается в производных классах. Такие функции называются чисто виртуальными (pure virtual function), т. е. такими, тело которых не определено. Общая форма записи чисто виртуальной функции имеет вид

virtual прототип функции = 0;

Если класс является производным класса с чисто виртуальной функцией и эта функция в нем не описана, тогда функция остается чисто виртуальной и в этом производном классе. Следовательно, такой производный класс является абстрактным. Хотя иерархия классов не требует обязательного включения в нее каких-либо абстрактных классов, однако программы, использующие объектно-ориентированное программирование, все же имеют иерархию, порожденную абстрактным базовым классом. Абстрактные классы могут составлять несколько уровней иерархии. В качестве примера можно привести иерархию форм (рис. 5).

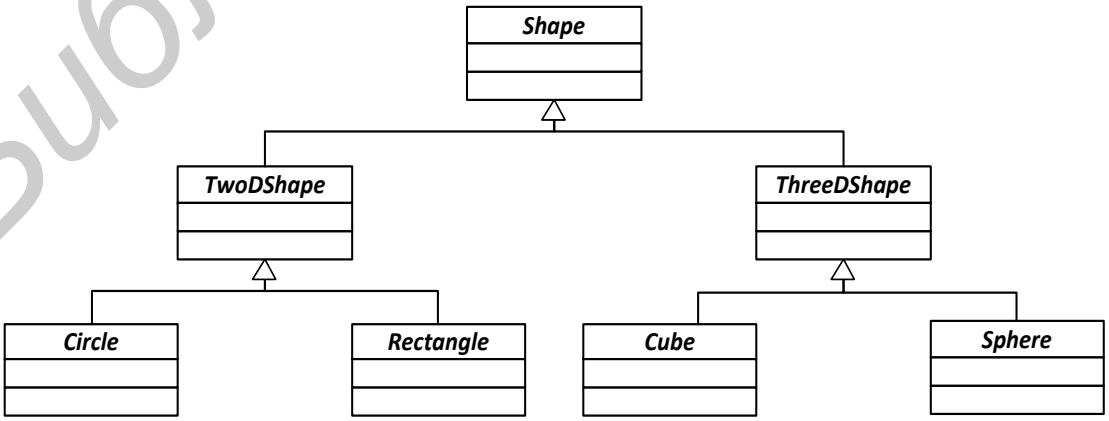

Рис. 5. Иерархия наследования классов

Иерархия может порождаться абстрактным базовым классом *Shape*. На уровень ниже можно получить два абстрактных класса *TwoDShape* и *ThreeDShape*. При переходе еще на один уровень ниже можно определить конкретные классы для двухмерных (Circle, Rectangle) и трехмерных (Cube, Sphere) форм. Согласно языку UML имя абстрактного класса пишется курсивом.

Рассмотрим пример программы приведенной иерархии классов, в которой будет выводиться название фигуры, площадь двухмерных и объем трехмерных фигур.

```
#include<iostream>
#include<iomanip>
using namespace std;
#include<string.h>
const double pi=3.14159;
class Shape                          // абстрактный базовый класс
{
  public:                            // чисто виртуальные функции
    virtual void print()=0;          // печать названия фигуры
    virtual void area() = 0;         // вычисление площади фигуры
    virtual void volume()=0;         // вычисление объема фигуры
};
class TwoDShape: public Sha          // абстрактный производный класс
{
  protected:
    float r;
  public:
    TwoDShape(float r1)              // конструктор с параметрами
    {
        r=r1;
    }
    virtual void area()=0;           // вычисление площади фигуры
    void volume(){}                  // определение функции вычисления
                                     // объема фигуры
};
class ThreeDShape: public Shape // абстрактный производный класс
{
  protected:
    float h;
  public:
    ThreeDShape(float h1)
    {
        h=h1;
    }
    virtual void volume()=0;         // вычисление объема фигуры
    void area(){}                    // определение функции вычисления
                                     // площади фигуры
};
class Circle: public TwoDShape   // класс «Окружность»
{
  public:
```

```
        Circle(float r):TwoDShape(r) {}
        void print()
        {
            cout << "Окружность" << endl;
        }
        void area()
        {
            cout << "Площадь окружности" << setw(10) << pi*r*r << endl;
        }
};
class Rectangle: public TwoDShape     // класс «Квадрат»
{
   public:
      Rectangle(float r):TwoDShape(r)  {}
      void print()
      {      cout << "Квадрат" << endl;    }
      void area()
      {
            cout << "Площадь квадрата" << setw(7) << r*r << endl;
      }
};
class Sphere: public ThreeDShape          // класс «Сфера»
{
   public:
      Sphere(float h): ThreeDShape(h)   {}
      void print()
      {
          cout << "Сфера" << endl;
      }
      void volume()
     {
          cout << "Объем сферы" << setw(10) << (4.0*pi*h*h*h)/3.0 << endl;
      }
};
class Cube: public ThreeDShape            // класс «Куб»
{
   public:
      Cube(float h): ThreeDShape(h)  {}
      void print()
      {
          cout << "Куб" << endl;
      }
      void volume()
      {
          cout << "Объем куба" << setw(5) << h*h*h << endl;
      }
};
int main()
{
   setlocale(LC_ALL,"Russian");
```

```
    Shape *ptr[4];          //  массив указателей на абстрактный базовый класс
    Circle okr(5);          // объект класса «Окружность»
    Rectangle pr(5);        // объект класса «Прямоугольник»
    Sphere sf(5);           // объект класса «Сфера»
    Cube kb(5);             // объект класса «Куб»
    ptr[0]=&okr;            // инициализация массива указателей ptr
    ptr[1]=&pr;
    ptr[2]=&sf;
    ptr[3]=&kb;
    for(int i=0; i < 4; i++)
    {
        ptr[i]->print();        // вывод названия фигуры
        ptr[i]->area();         // вывод площади фигуры
        ptr[i]->volume();       // вывод объема фигуры
    }
    return 0;
}
```

Контрольные вопросы

1. Какая функция называется виртуальной?
2. Чем виртуальные функции отличаются от перегружаемых?
3. Какой класс называется абстрактным?
4. В чем состоит различие раннего и позднего связывания?
5. Опишите назначение виртуального деструктора.

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Реализовать абстрактный класс *Shape*, содержащий интерфейс иерархии. Создать производные абстрактные классы *TwoDShape* и *ThreeDShape*, от которых унаследовать всевозможные конкретные формы. Реализовать виртуальные функции print (для вывода типа и размера объектов каждого класса), area, draw и volume.

2. Разработать программу для обработки информации о строительных материалах: поступление, учет и отгрузка. Для этого требуется разработать классы. Базовый класс должен быть абстрактным. Первый производный класс определяет количество стройматериалов каждого типа, например 50 перекрытий. Другой класс хранит данные о каждом виде стройматериалов, например площадь и стоимость за квадратный метр. Далее следует класс, хранящий описание каждого вида стройматериалов, которое включает название и сорт материала.

3. Разработать иерархию наследования, включающую следующие классы: город (наименование), магазин (наименование, тип), товар (наименование, сорт, количество, цена), банк (номер счета, сумма денег на счете) и покупатель (фамилия, сумма денег, сумма покупки). При этом класс «Город» – абстрактный базовый для классов «Магазин» и «Банк», класс «Товар» – для класса «Магазин», а классы «Магазин» и «Банк» – базовые для класса «Покупатель». Класс «Покупатель» содержит также методы выполнения различных операций с товаром.

4. Написать программу учета наличия транспортных средств (автобусы) в автопарке. Разработать следующие классы: абстрактный базовый класс «Автобус», содержащий поля с информацией о марке автобуса и госномере; производные классы «Мягкий автобус», «Жесткий автобус». Производные классы содержат следующие поля: количество мест, водители. По запросу выдавать информацию о свободных местах в автобусах, об автобусах в рейсе.

5. Создать базовый абстрактный класс «Книга», включающий название книги, фамилию автора. Реализовать производный класс «Отдел», включающий в себя название отдела. Написать программу, позволяющую добавлять и удалять книги из отдела.

6. Разработать иерархию наследования, включающую следующие классы: абстрактный базовый класс «Товар» (название товара, производитель, цена), производные классы: «Отдел» (название отдела), «Магазин» (наименование, тип), «Корзина» (сумма покупки). Класс «Банк» (номер счета, сумма денег на счете) не наследуется. Все классы должны содержать функции получения и изменения всех полей. Написать программу, позволяющую производить покупки.

7. Создать абстрактный базовый класс *Person*, описывающий обычного человека. Создайте производный класс *Student*, описывающий типичного студента. От класса *Student* наследуйте класс *GradStudent*, описывающий типичного аспиранта. Все классы должны содержать функции получения и изменения всех полей. Написать программу, позволяющую получать сведения о студентах и аспирантах.

8. Создать абстрактный базовый класс «Часы», а также производные классы «Механические часы» и «Электронные часы». Все классы должны содержать функции получения и изменения всех полей. Написать программу, позволяющую получать сведения о часах.

9. Создать абстрактный базовый класс «Студент», а также производные классы «Выпускник» и «Старшекурсник». Все классы должны содержать функции получения и изменения всех полей. Написать программу, позволяющую получать сведения о студентах.

10. Создать абстрактный базовый класс «Служащий», содержащий фамилию и имя служащего. Производный класс «Работники с почасовой оплатой» содержит следующие поля: оплата за час и часы, отработанные за неделю. Производный класс «Работники со сдельной оплатой» содержит следующие поля: оплата единицы продукции и число единиц продукции за неделю. Все классы

должны содержать функции получения и изменения всех полей. Написать программу, позволяющую получать сведения о служащих.

11. Написать программу учета книг. Создать абстрактный базовый класс «Книга», содержащий следующие поля: название книги, фамилия автора, издательство. Получить производные классы:

- «Отдел технической книги», который содержит поля с названиями отдела и отраслей техники;
- «Отдел художественной книги», который содержит поля с названиями отдела и направлений в литературе;
- «Абонент», который наследуется от классов «Отдел технической книги» и «Отдел художественной книги». Класс «Абонент» включает следующие поля: ФИО студента; факультет; группа; название книги, имеющейся у студента, и время ее возврата.

Все классы должны содержать функции получения и изменения всех полей. Программа должна выдавать сведения о студентах, просрочивших время возврата книг.

Лабораторная работа № 6

Тема работы: множественное наследование, виртуальное наследование.

Цель работы: изучить принципы и получить практические навыки при использовании множественного наследования; рассмотреть случаи, когда необходимо использовать виртуальное наследование.

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3–8].

В языке C++ имеется возможность образовывать производный класс от нескольких базовых классов. Общая форма такого наследования имеет вид

class имя_произв_класса : имя_базового_кл 1,…,имя_базового_кл N
{ содержимое класса
};

Иерархическая структура, в которой производный класс наследует от нескольких базовых классов, называется множественным наследованием. В этом случае производный класс, имея собственные компоненты, имеет доступ к protected- и public-компонентам базовых классов.

Конструкторы базовых классов при создании объекта производного класса вызываются в том порядке, в котором они указаны в списке при объявлении производного класса. Ниже приведен пример программы, использующей множественное наследование.

Задание (множественное наследование). Разработать иерархию классов, реализующих множественное наследование. Первый базовый класс Firma содержит название фирмы, производный от него класс Otdel содержит информацию о табельном номере работника и номере отдела. Второй базовый класс Bank – информацию о наименовании банка и величине счета в банке. Класс Work, производный от этих двух классов, содержит фамилию работника. Все классы содержат функции просмотра полей.

```
#include <iostream>
#include <string.h>
#include <locale.h>
using namespace std;
class Firma
{
  protected:
      char naz[20];          //  название фирмы
  public:
     Firma(char *);
     ~Firma() { cout << "деструктор класса A" << endl; }
     void Firma_prnt();
};
class Otdel : public Firma
{
   protected:
      long tn;          // табельный номер
      int nom;          // номер подразделения
   public:
```

```
      Otdel(char*,long ,int);
      ~Otdel() { cout << "деструктор класса B1" << endl; }
      void Otdel_prnt();
};
class Bank
{
  protected:
      char naz[30];        // название банка
      double  schet;       // сумма денег
  public:
      Bank(char *, double);
      ~ Bank (){ cout << "деструктор класса B2" << endl; }
      void Bank_prnt();
};
class Work : public Otdel, public Bank
{
      char *fam;
  public:
      Work(char *,char *,long ,int ,char *,double) ;
      ~Work() { cout << "деструктор класса C" << endl; }
      void Work_prnt();
};
Firma::Firma(char *NAZ) {strcpy(naz,NAZ);}
Otdel::Otdel(char *NAZ, long TN,int NOM): Firma(NAZ),tn(TN),nom(NOM) {}
Bank::Bank(char *NAZ, double SCHET): schet(SCHET) {strcpy(naz,NAZ);}
Work::Work(char *FAM,char *NAZ1,long TN,int NOM,char *NAZ2,double ZP) :
        Otdel(NAZ1,TN,NOM), Bank(NAZ2,ZP)
{
   fam = new char[strlen(FAM)+1];
   strcpy(fam,FAM);
}
void Firma::Firma_prnt() {cout << naz << endl;}
void Otdel::Otdel_prnt()
{
   Firma::Firma_prnt();
   cout << " таб. N " << tn <<"  подразделение = " << nom <<endl;
}
void Bank::Bank_prnt()
{
     cout << " банк   : " << naz << endl;
     cout << " счет   = " << schet << endl;
}
void Work::Work_prnt()
{
   Otdel::Otdel_prnt();
   Bank::Bank_prnt();
   cout << " фамилия " << fam<<endl;
}
int main()
{
```

```
    Work rb("Иванов","предприятие",1234,2,"банк",555.6);
    Work *pt=&rb;  // указатель на созданный объект. Далее можно использо-
                   //  вать для вызова методов либо сам объект car, либо
                   // указатель на него pt
    setlocale(LC_ALL,"Russian" );
    rb.Otdel_prnt();      // вызов метода Otdel_prnt, используя объект
    pt->Otdel_prnt();    // вызов метода Otdel_prnt, используя указатель на объект

    rb.Bank_prnt();
    pt->Bank_prnt();

    rb.Work_prnt();
    pt->Work_prnt();
    return 0;
}
```

При применении множественного наследования возможно возникновение нескольких конфликтных ситуаций. ***Первая*** − конфликт имен методов или атрибутов нескольких базовых классов:

```
class A
{   public: void fun(){}
};
class B
{   public: void fun(){}
};
class C : public A, public B
{  };
int main()
{
    C obj;
    obj.fun();            //  error C::f' is ambiguous
    return 0;
}
```

При таком вызове функции fun() компилятор не может определить, к какой из двух функций классов A и B выполняется обращение. Неоднозначность можно устранить, явно указав, какому из базовых классов принадлежит вызываемая функция:

```
obj.A:: fun();          или        obj.B::fun();
```

Вторая проблема возникает при многократном включении некоторого базового класса:

```
class A
{ public: void fun(){}
};
class B : public A
{    // компоненты класса B
};
class C : public A
{    // компоненты класса C
};
class D : public B, public C
```

```
{ };

int main()
{
  D obj;
  obj.fun();            // ambiguous access of 'fun'
  return 0;
}
```

Проблема состоит в том, что при создании объекта класса D создаются два (одинаковых) объекта базового класса A (для объектов класса B и C соответственно). Решение этой проблемы состоит в использовании виртуального наследования:

```
class B : virtual public A
class C : virtual public A.
```

В этом случае при создании объекта класса D происходит однократное создание виртуального объекта класса A. Рассмотрим пример программы.

Задание (виртуальное наследование). Разработать иерархию классов, реализующих виртуальное наследование. Базовый класс Car содержит название марки автомобиля, первый производный класс Color – цвет и число дверей, второй производный класс Power – мощность двигателя и величину расхода топлива. Класс Shop, производный от этих двух классов, содержит название автомагазина. Все классы содержат методы отображения своих компонентов.

```
#include <iostream>
#include <string.h>
#include <locale.h>
#include <iomanip>
using namespace std;
class Car
{
    char *naz;        //  марка автомобиля
  public:
    Car(char *NAZ)
    {
      naz=new char[strlen(NAZ)+1];
      strcpy(naz,NAZ);
    }
     ~Car()
    {
     delete [] naz;
     cout << "деструктор класса A" << endl;
    }
     void car_prnt(){ cout <<"марка а/м "<<naz<< endl; }
};
class Color : virtual public Car
{
  protected:
      char *cv;        // цвет автомобиля
      int  kol;        // количество дверей
```

```
  public:


      Color(char *NAZ,char *CV,int KOL): Car(NAZ),kol(KOL)
      {
        cv=new char[strlen(CV)+1];
        strcpy(cv,CV);
      }
      ~Color()
      {
        delete [] cv; ;
        cout << "деструктор класса B1" << endl;
      }
      void color_prnt()
      {
        Car::car_prnt();
        cout << "цвет а/м "<<cv<<" количество дверей "<<kol<<endl;
      }
};
class Power : virtual public Car
{
   protected:
      int pow;          // мощность двигателя автомобиля
      double  rs;       // расход топлива
   public:
      Power(char *NAZ,int POW,double RS): Car(NAZ),pow(POW),rs(RS) {};
     ~Power(){ cout << "деструктор класса B2" << endl; }
     void power_prnt()
     {
       Car::car_prnt();
       cout <<"мощность двигателя "<<pow<<" расход топлива "<<rs<<endl;
     }
};
class Shop : public Color,public Power
{
    char *mag;        // название магазина
  public: Shop(char *NAZ,char *CV,int KOL,int POW,double RS,char *MAG):
     Color(NAZ,CV,KOL),Power(NAZ,POW,RS),Car(NAZ)
     {
      mag =new char[strlen(MAG)+1];
      strcpy(mag,MAG);
     }
     ~Shop()
    {   delete [] mag;
        cout << "деструктор класса C" << endl;
    }
    void shop_prnt()
   {
      Car::car_prnt();
      Color::color_prnt();
```

```
    Power::power_prnt();
    cout << " название магазина " <<mag<<endl;
  }
};
int main()
{
  Shop car("BMW","красный",4,140,8.5,"Автомагазин"); // объект класса Shop
  Shop *pt=&car; // указатель на созданный объект. Далее можно использо-
                 //  вать для вызова методов либо сам объект car, либо
                 // указатель на него pt
  setlocale(LC_ALL,"Russian" );
  car.car_prnt();
  pt->car_prnt();
  car.color_prnt();
  pt->color_prnt();
  car.power_prnt();
  pt->power_prnt();
  car.shop_prnt();
  pt->shop_prnt();
  return 0;
}
```

Контрольные вопросы

1. Для чего используется множественное наследование? Чем оно отличается от простого наследования?

2. Каков механизм вызова конструкторов при множественном и виртуальном наследовании?

3. Какие проблемы возможны при множественном наследовании?

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Описать базовый класс «Точка» со следующими полями: координаты и цвет. Методы класса – конструктор, получение значения полей, изменение значения полей, отображение координат точки на экране и перемещение точки.

Описать базовый класс «Отрезок» со следующими полями: координаты и цвет. Методы класса – конструктор, получение значения полей, изменение значения полей, отображение координат отрезка на экране и перемещение отрезка.

Используя базовые классы «Точка» и «Отрезок», определить класс «Квадрат». Предусмотреть следующие действия с объектами создаваемого класса: со-

здание объектов, отображение координат объектов на экране, перемещение объектов и изменение параметров объектов (цвет, размер, координаты, и т. д).

2. Для базовых классов «Двигатель», «Кузов», «Цвет» разработать производный класс «Автомобиль». Содержимое классов продумать самостоятельно.

3. Имеется два базовых класса: «Работник» (данные класса – фамилия, массив зарплат за квартал), «Налог» (данные класса – процент подоходного налога на зарплату). Разработать от этих классов производный класс «Форма» для вывода итоговых форм, например фамилия, величина зарплаты, сумма налога.

4. Разработать иерархию классов («Город», «Магазин», «Банк», «Покупатель»). Класс «Город» является базовым для классов «Магазин», «Банк», а они являются базовыми для класса «Покупатель». Содержимое классов следующее:

− данные класса «Город» – название города, методы – конструкторы, деструктор, show();

− данные класса «Магазин» – наименование нескольких товаров (массив), цена и количество каждого из товаров, методы – конструкторы, деструктор, show();

− данные класса «Банк» – номер счета и сумма на счете, методы – конструкторы, деструктор, show(), popolnenie_sheta() (пополнение счета в банке);

− данные класса «Покупатель» – сумма сделанной покупки, методы – конструкторы, деструктор, show(), pokupka() (совершение покупки товара).

Каждый из методов show() отображает содержимое своего класса и вызывает show() базового для него класса.

5. При иерархии классов, описанной в предыдущем задании, продумать и реализовать возможность создания нескольких объектов класса Pokup таким образом, чтобы результат покупки каждым предыдущим объектом изменял информацию о количестве товара для последующих объектов.

6. Построить иерархию классов согласно схеме наследования, приведенной на рис. 6. Каждый класс должен содержать инициализирующий конструктор и функцию show() для вывода значений. Функция main() должна иллюстрировать иерархию наследования.

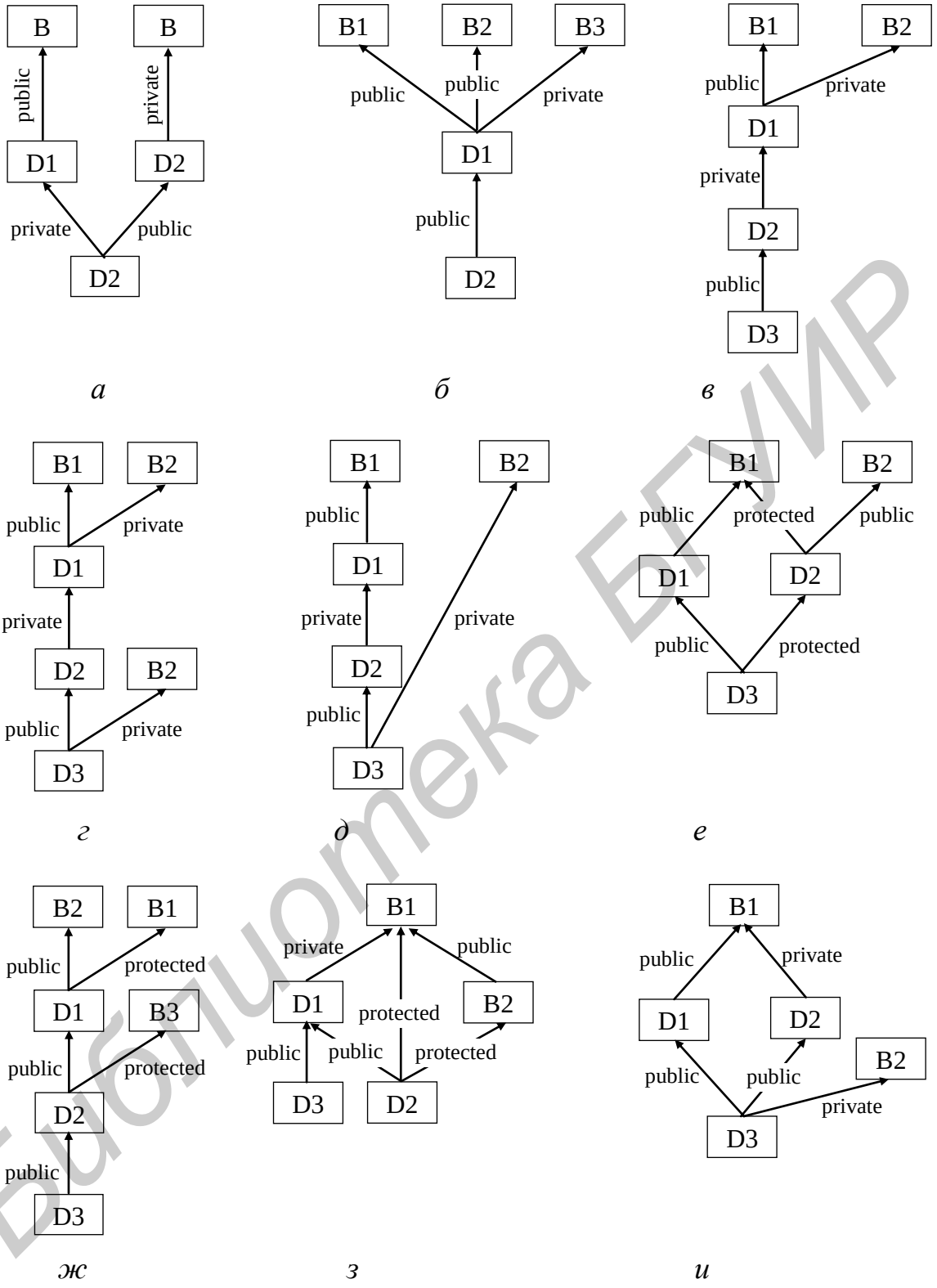

Рис. 6. Схемы множественного наследования классов

Лабораторная работа №7

Тема работы: реализация шаблонов классов.

Цель работы: научиться использовать шаблоны функций для создания группы однотипных функций, а также шаблоны классов для создания группы связанных типов классов.

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3−8].

Шаблон семейства классов определяет способ построения отдельных классов подобно тому, как класс определяет правила построения и формат отдельных объектов. Этот класс можно рассматривать как некоторое описание множества классов, отличающихся только типами их данных. В C++ используется ключевое слово template для обеспечения параметрического полиморфизма. Шаблоны определения класса и шаблоны определения функции позволяют многократно использовать код, корректно по отношению к различным типам.

Формат шаблона класса имеет вид

```
template <список параметров>
class объявление класса
```

Список параметров шаблона класса представляет собой идентификатор типа, подставляемого в объявление данного класса при его генерации. Идентификатору типа предшествует ключевое слово **class** или **typename**.

Определение шаблона может быть только глобальным.

Далее будет приведен пример определения шаблона класса вектора (одномерного массива). Какой бы тип не имели элементы массива (целый, вещественный, с двойной точностью и т. д.), в этом классе должны быть определены одни и те же базовые операции, например доступ к элементу по индексу и т. д. Если тип элементов вектора задавать как параметр шаблона класса, то система будет формировать вектор нужного типа (и соответствующий класс) при каждом определении конкретного объекта.

В программе шаблон семейства классов с общим именем vector используется для формирования двух классов с массивами целого и символьного типов. В соответствии с требованием синтаксиса имя параметризованного класса, определенное в шаблоне (в примере − vector), используется в программе только с последующим конкретным фактическим параметром (аргументом), заключенным в угловые скобки. Параметром может быть имя стандартного или определенного пользователем типа. В данном примере использованы стандартные типы int и char. Использовать имя vector без указания фактического параметра шаблона нельзя, т. к. никакое умалчиваемое значение при этом не предусматривается.

В списке параметров шаблона могут присутствовать формальные параметры, не определяющие тип, точнее – это параметры, для которых тип фиксирован.

Рассмотрим пример использования шаблонного класса, реализующего одномерный вектор. Для этого реализуем некоторые методы для работы с вектором, а также перегрузим операцию [].

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

```
#include <locale.h>
#include <iomanip>
#include <typeinfo.h>
using namespace std;
template <class T>
class vector
{      T *ms;                        // указатель на вектор
       int size,ind;                 // размер и индекс
  public:
       vector() : size(0),ind(0),ms(NULL) {}
       vector(int);
       ~vector(){delete [] ms;}
       void set(const T&);        // увеличение размера массива на один элемент
       T* get_vect();             // возвращается указатель на массив
       int get_size();            // возвращается размер массива
       T &operator[](int);        // определение обычного метода
};
template <class T>
vector<T>::vector(int SIZE) : size(SIZE), ind(0)
{
  ms=new T[size];
  const type_info & t=typeid(T);    // получение ссылки t на
  const char* s=t.name();           // объект класса type_info
  for(int i=0;i<size;i++)           // в зависимости от типа T
  if(!strcmp(s,"char")) *(ms+i)=' ';  // заносим пустой символ
  else *(ms+i)=0;                   // или заносим цифру нуль
}
template <class T>
void vector<T>::set(const T &t)  // увеличение размера массива на один элемент
{
   T *tmp = NULL;
   if(++ind>=size)
   {
      tmp=ms;
      ms=new T[size+1];          // ms-указатель на новый массив
   }
   if(tmp) memcpy(ms,tmp,sizeof(T)*size); // перезапись tmp -> ms
   ms[size++]=t;                  // добавление нового элемента
   if(tmp) delete [] tmp;         // удаление временного массива
}
template <class T>
T* vector<T>::get_vect()
{   return ms;
}
template <class T>
int vector<T>::get_size()
{      return size;
}
template <class T>
T & vector<T>::operator[](int n)  // определение обычного метода
```

```
{
  if(n<0 || (n>=size)) n=0;          // контроль за выходом из вектора
  return ms[n];
}
int main()
{
   vector <int> VectInt;
   vector <char> VectChar;
   VectInt.set(3);
   VectInt.set(26);
   VectInt.set(12);                   // получен int-вектор из трех атрибутов
   VectChar.set('a');
   VectChar.set('c');                 // получен char-вектор из двух атрибутов
   cout << VectInt[0]<< endl;
   cout << VectChar[0] << endl;
   VectInt[0]=1;
   VectChar[1]='b';
   int *m_i=VectInt.get_vect();
   for(int i=0; i<VectInt.get_size(); i++ )
     cout<<setw(3)<< *(m_i+i); cout<< endl;  // вывод целочисленного вектора
   char *m_c=VectChar.get_vect();
   for(int i=0; i<VectChar.get_size(); i++ )
     cout<<setw(3)<< *(m_c+i); cout<< endl;  // вывод символьного вектора
   return 0;
}
```

Контрольные вопросы

1. Для чего используются шаблоны классов? Что у них общего с шаблонами функций?
2. Как описываются шаблоны классов?
3. Как создать объект на основе класса, порожденного шаблоном?
4. Каких типов могут быть фактические параметры шаблонов классов?
5. Можно ли описывать в списке параметров шаблона параметры, не определяющие тип?

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Написать программу, в которой определяется шаблон для функции max(x,y), возвращающей большее из значений x и y. Написать перегруженную функцию max(char*,char*), возвращающую большую из передаваемых ей символьных строк. В каждой из функций предусмотреть вывод сообщения о том, что вы-

звана шаблонная или специализированная функция и вывод найденного большего.

2. Реализовать шаблон класса Tree, реализующий бинарное дерево. Для представления узлов дерева использовать шаблон класса TreeNode. Определить функции добавления узла, обхода всего дерева в нисходящем и в восходящем порядке, поиска по дереву. На основе созданного класса написать программу вычисления арифметических выражений.

3. Написать шаблон класса вектор, который принимает элементы любого типа. В классе реализовать методы, например, поиска, сортировки, удаления и т. д. При реализации методов использовать динамическую идентификацию типа элементов объекта.

4. Описать шаблонный класс для работы с односвязными списками. Для объектов класса определить операции проверки списка на пустоту, добавления элемента в начало списка, в конец списка, подсчет числа вхождений элемента в список, удаление элемента из списка. Продемонстрировать работу с шаблонным классом для списка с целыми элементами и с элементами-строками.

5. Для каждого варианта (табл. 2) разработать шаблон соответствующего класса, где поля могут иметь различные типы данных (некоторые поля могут быть статическими). Предусмотреть наличие в классе указанных методов и перегруженных операций.

Таблица 2

Номер варианта	Название класса	Поля	Методы	Перегружаемые операции
1	2	3	4	5
1	Стек	Указатель на вершину, текущий размер	Добавить, извлечь элемент, проверить на пустоту, распечатать	+= (поместить в стек), префиксный – – (извлечь из стека)
2	Элемент односвязного списка	Указатель на начало, информационное поле элемента, ссылка на следующий элемент	Добавить элемент в начало, удалить из начала, найти элемент	<< (распечатать весь список), == (равны ли размеры двух списков?)
3	Элемент двусвязного списка	Указатели начала и конца, информационное поле элемента, ссылки на следующий и предыдущий элементы	Добавить элемент в конец, удалить из конца, распечатать список	<< (распечатать весь список), == (равны ли размеры двух списков?)
4	Квадратное уравнение	Коэффициенты, дискриминант, корни	Вывести на экран уравнение, решить уравнение, вывести корни	Бинарный –, *= (каждый коэффициент умножить на число)

Окончание табл. 2

1	2	3	4	5
5	Матрица	Две размерности, указатель на элементы	Вывести на экран, найти максимум и среднее значение элементов	+ (сложение двух матриц), < (каждый элемент первой матрицы меньше соответствующего элемента второй)
6	Дата	Число, месяц, год	Изменить значения полей, определить, сколько дней осталось до нового года, вывести в формате «чч/мм/гггг»	!= (проверка двух дат на неравенство), бинарный – (промежуток между двумя датами в днях, месяцах, годах)
7	Время	Часы, минуты, секунды	Изменить значения полей, вывести в формате «чч:мм:сс», определить, является ли заданное время «до полудня» или «после полудня»	>> (ввод новых значений полей), += (добавить заданное количество секунд)
8	Множество	Элементы множества, текущее количество элементов	Проверка на включение элемента в множество, вывод на экран, удаление элемента из множества	+= (добавление элемента в множество), * (пересечение множеств)
9	Бинарное дерево	Указатель на корень, информационное поле узла, ссылки на левого и правого потомков	Распечатать дерево, определить среднее арифметическое элементов, найти и удалить заданный элемент	<< (обход дерева сверху вниз), += (добавить элемент в дерево)

Лабораторная работа №8

Тема работы: практические приемы использования шаблонов классов.

Цель работы: применить знания, полученные в предыдущих работах, к реализации практических задач (реализация smart-указателей, механизма транзакций).

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3–8].

smart-указатели. При работе с динамическими объектами, используя указатели на них, в случае если объект становится не нужен, то его необходимо уничтожить. В то же время на один объект могут ссылаться множество указателей и, следовательно, нельзя однозначно сказать, нужен ли еще этот объект или он уже может быть уничтожен. Рассмотрим пример реализации класса, для которого происходит уничтожение объектов, в случае если уничтожается последняя ссылка на него. Это достигается, например, тем, что создается дополнительно структура, в которой наряду с указателем на динамический объект хранится также счетчик числа указателей, «прикрепленных» к этому объекту. Динамический объект может быть уничтожен только в том случае, если счетчик ссылок на этот объект станет равным нулю.

```
#include <iostream>
#include <locale.h>
using namespace std;
struct Str                      // структура-объект
{
   int a;
   char c;
};
template <class T>
struct Status                   // состояние указателя
{
   T *RealPtr;                  // указатель
   int Count;                   // счетчик числа ссылок на указатель
};
template <class T>
class Point                     // класс-указатель
{
     Status<T> *StatPtr;
  public:
     Point(T *ptr=0);               // конструктор
     Point(const Point &);          // копирующий конструктор
     ~Point();
     Point &operator=(const Point &);     // перегрузка  операции =
     T *operator->() const;
};
template <class T>
Point<T>::Point(T *ptr)
{
   if(!ptr) StatPtr=NULL;           // указатель на объект пустой
   else
   {  StatPtr=new Status<T>;
```

```
      StatPtr->RealPtr=ptr;          // инициализирует объект указателем
      StatPtr->Count=1;              // счетчик «прикрепленных» объектов
    }
}
template <class T>                  // описание конструктора копирования
Point<T>::Point(const Point &p):StatPtr(p.StatPtr)
{
   if(StatPtr) StatPtr->Count++; // только увеличение числа ссылок
}
template <class T>
Point<T>::~Point()                  // описание деструктора
{
  if(StatPtr)
  {  StatPtr->Count--;              // уменьшается число ссылок на объект
     if(StatPtr->Count<=0)          // если число ссылок на объект меньше
     {                              // либо равно нулю, то уничтожается
        delete StatPtr->RealPtr;  // объект
        delete StatPtr;
     }
  }
}
template <class T>
T *Point<T>::operator->() const
{
  if(StatPtr) return StatPtr->RealPtr;
  else return NULL;
}
template <class T>
Point<T> &Point<T>::operator=(const Point &p)
{                                   // отсоединение объекта, расположенного слева
 if(StatPtr)                        // от знака «=» от указателя
 { StatPtr->Count--;                // уменьшаем счетчик «прикрепленных» объектов
    if(StatPtr->Count<=0)           // если объектов нет, то так же, как и в
    {                               // деструкторе выполняется освобождение
       delete StatPtr->RealPtr;  // выделенной под объект динамической памяти
       delete StatPtr;
    }
 }
 StatPtr=p.StatPtr;                // присоединение к новому указателю
 if(StatPtr) StatPtr->Count++; // увеличиваем счетчик «прикрепленных»
 return *this;                      // объектов
}
int main()
{
 setlocale(LC_ALL,"Russian" );
 Point<Str> pt1(new Str); // генерация класса Point, конструирование
                          // объекта pt1, инициализируемого указателем
                          // на структуру Str, далее с объектом можно
                          // обращаться, как с указателем
 Point<Str> pt2=pt1,pt3;   // для pt2 вызывается конструктор копирования,
```

```
                                    // затем создается указатель pt3
  pt3=pt1;                          // pt3 переназначается на объект указателя pt1
  pt1->a=2;                         // operator->() получает this-указатель на pt1
  pt1->c='b';
  return 0;
}
```

Концепция smart-указателей позволяет просто решать задачу поддержки транзакций.

Транзакция в информатике − группа последовательных операций, которая представляет собой логическую единицу работы с данными. Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда она не должна произвести никакого эффекта. При этом должно соблюдаться следующее:

− если клиент начал и не завершил транзакцию, то другие клиенты не видят его изменений;

− две транзакции не могут одновременно менять одни и те же данные.

Для поддержки механизма транзакции объект должен содержать два указателя (на текущий объект и на объект, представляющий его предыдущее состояние) и функции (начать, закрепить и отменить действие). Если требуется закрепить или отменить выполненные изменения, то необходимо хранить состояние объекта на заданный момент – начало транзакции и в момент принятия решения или уничтожать предыдущее состояние (закрепление), или возвращаться к нему (отмена). Механизм транзакций можно рассмотреть на примере следующей программы.

```
#include <iostream>
#include <locale.h>
using namespace std;
template <class T1>
class Cls          // класс, над объектами которого выполняются транзакции
{
    T1 x;
  public:
    Cls() : x(0){}
    void set(int X) {x=X;}      // инициализация объекта
    T1 get() {return x;}        // возвращается значение объекта
};
template <class T2>
class Tran          // класс поддержки транзакций
{
    T2* that;       // текущее значение объекта класса Cls
    T2* prev;       // предыдущее значение объекта класса Cls
  public:
    Tran():prev(NULL), that(new T2){}     // конструктор
    Tran(const Tran & obj):                   // конструктор копирования
          that(new T2(*(obj.that))), prev(NULL) {}
    ~Tran(){ delete that;  delete prev; }    // деструктор
```

```
    Tran& operator=(const Tran & obj);   // перегрузка операции присваивания
    void Show(int);        // отображение значений (предыдущего
                           // и текущего) объекта класса Cls
    int BeginTrans();      // начало транзакции
    void Commit();         // закрепление транзакции
    void DeleteTrans(); // отмена транзакции
    T2* operator->();      // перегрузка операции ->
};
template <class T2>
Tran<T2>& Tran<T2>::operator=(const Tran<T2> & obj)
{
  if (this!= &obj)        // проверка на случай  obj=obj
  {
     delete that;         // удаление текущего значения объекта
     that = new T2(*(obj.that)); // создание и копирование
  }
  return *this;
}
template <class T2>
T2* Tran<T2>::operator->()     // перегрузка операции ->
{ return that; }
template <class T2>
void Tran<T2>::Show(int fl)     // отображение состояние объекта
{
   cout<<"состояния объекта ";
   if(!fl) cout<<"до начала транзакции "<<endl;
   else cout<<"после выполнения транзакции "<<endl;
   if(prev) cout<<"prev = "<<prev->get()<<endl;    // предыдущее
   else cout<<"prev = NULL"<<endl;
   cout<<"that = "<<that->get()<<endl;                // текущее
}
template <class T2>
int Tran<T2>::BeginTrans()  // начало
{
  delete prev;             // удаление предыдущего значения
  prev = that;             // текущее становится предыдущим
  that = new T2(*prev); // новое значение текущего значения
  if(!that) return 0;      // ошибка (необходимо отменить действие)
  that->set(7);            // изменение состояния объекта
  return 1;                // успешное окончание транзакции
}
template <class T2>
void Tran<T2>::Commit ()  // закрепление
{
  delete prev;                // удаление предыдущего значения
  prev = NULL;                // предыдущего состояния  нет
}
template <class T2>
void Tran<T2>::DeleteTrans()  // отмена
{
```

```
    if (prev != NULL)
    {
        delete that;                // удаление текущего значения
        that = prev;                // предыдущее становится текущим
        prev = NULL;                // предыдущего состояния  нет
    }
}
int main()
{
    Tran<Cls<int> > tr;
    setlocale(LC_ALL,"Russian" );
    tr->set(5);            // начальная инициализация объекта
    tr.Show(0);
    if(tr.BeginTrans()) // начало транзакции (изменение объекта)
    {
        tr.Show(1);
        tr.Commit();      // закрепление транзакции
    }
    tr.DeleteTrans();   // отмена транзакции при ошибке
    tr.Show(0);
    if(tr.BeginTrans()) // начало транзакции (изменение объекта)
    {
        tr.Show(1);
        tr.Commit();      // закрепление транзакции
    }
    tr.Show(0);
    return 0;
}
```

Контрольные вопросы

1. С какой целью используются smart-указатели?
2. Что представляет собой smart-указатель?
3. В чем состоит механизм транзакций?
4. Какие основные операции используются при реализации механизма транзакций?

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Использовать smart-указатель в заданиях, выполненных в лабораторной работе №7. Предусмотреть удаление объекта указателя (обнуление его содержимого) только в том случае, если все объекты, на которые он указывает, уничтожены.

2. Использовать smart-указатель для моделирования делопроизводства в учреждении. Имеется несколько объектов класса (структуры) «Документ», некоторое количество сотрудников фирмы могут с ним работать. Документ считается сформированным, когда с ним уже никто не работает.

3. Реализовать механизм транзакций для задания 5 лабораторной работы №7. Предусмотреть возможность «отката» к предыдущему состоянию объекта, если текущее состояние является неудовлетворительным. Смоделировать данную ситуацию.

4. Используя механизм транзакций, реализовать простейший текстовый редактор. Редактор должен поддерживать возможность отмены (Undo) и восстановления (Redo) информации. Для хранения историй модификации объекта использовать списочный тип. Его организацию выбрать самостоятельно.

5. Реализовать программу, выполняющую банковские операции со счетом. В программе реализовать механизм транзакций. Если операция выполнена неверно (например, зачислена сумма меньше ожидаемой, произведена попытка списать со счета больше, чем на нем имеется), то должна быть выполнена отмена. Программа должна обеспечивать вывод информации о проведении банковских операций по запросу клиента.

Литература

1. Шилдт, Г. Искусство программирования на C++ / Г. Шилдт ; пер. с англ. – СПб. : БХВ-Петербург, 2005. – 928 с.

2. Дейтел, Х. Как программировать на C++ / Х. Дейтел, П. Дейтел ; пер. с англ. – М. : Бином-Пресс, 2009. – 1037 с.

3. Страуструп, Б. Программирование. Принципы и практика использования C++ / Б. Страуструп ; пер. с англ. – М. : Вильямс, 2011. – 1246 с.

4. Страуструп, Б. Язык программирования C++. Специальное издание / Б. Страуструп ; пер. с англ. – М. : Вильямс, 2012. – 1136 с.

5. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч ; пер. с англ. – М. : Вильямс, 2010. – 720 с.

6. Джамса, К. Учимся программировать на языке C++ / К. Джамса ; пер. с англ. – М. : Мир, 1997. – 320 с.

7. Павловская, Т. C/C++. Программирование на языке высокого уровня / Т. Павловская. – СПб. : Питер, 2013. – 464 с.

8. Луцик, Ю. А. Объектно-ориентированное программирование на языке C++ : учеб. пособие / Ю. А. Луцик, В. Н. Комличенко. – Минск : БГУИР, 2008. – 266 с.

Содержание

Св. план 2014, поз. 73

Учебное издание

Комличенко Виталий Николаевич
Луцик Юрий Александрович
Ковальчук Анна Михайловна
Унучек Евгений Николаевич

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редактор *Е. С. Чайковская*
Корректор *Е. И. Герман*
Компьютерная правка, оригинал-макет *Е. Г. Бабичева*

Подписано в печать 18.08.2015. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 4,53. Уч.-изд. л. 4,0. Тираж 150 экз. Заказ 225.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6