Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра ЭВМ

А.М. Ковальчук

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ ПО ЯЗЫКУ ПРОГРАММИРОВАНИЯ C++

Учебное пособие по курсу
«Конструирование программ и языки программирования»
для студентов специальности 40 02 01
«Вычислительные машины, системы и сети»
дневной и дистанционной форм обучения

Минск – 2006

Оглавление

Лабораторная работа №1

Динамическое распределение памяти. Конструкторы и деструкторы.

Цель работы: Изучить принцип работы функции конструктор с параметрами и с параметрами по умолчанию, перегрузку конструкторов. Использовать функцию деструктор в разработанных программах.

Краткие теоретические сведения

Использование функций для установки начальных значений данных объекта неестественно и часто приводит к ошибкам. В связи с этим введена специальная функция, позволяющая инициализировать объект в процессе его объявления. 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. Пример:

```
#include <iostream.h>
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)
        {
            cout << “i=” << i << “str=” << str << endl;
        }
        void main(void)
        {
            int a=10, *b;
            b=&a;
            Over my_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();       // Активен конструктор Over()
            my_over1.print();     // Активен конструктор Over(char *s)
            my_over2.print();     // Активен конструктор Over(char *s, int x)
            my_over3.print();     // Активен конструктор Over(int *y)
        }
```

Результаты выполнения программы представляются в 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:

```
class Time
{
    public:
        Time(int = 0, int = 0, int = 0);  // конструктор 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::printStandard()
{
        cout << (( hour == 0 || hour == 12) ?  12 : hour % 12)
             << ":" << ( minute < 10 ?  "0" : "" ) << minute
             << ":" << ( second  < 10 ?  "0" : "" ) << second
             <<  ( hour < 12 ?  "AM" : "PM" );
}
void main(void)
{
        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. printStandard();
        cout << "\nЧасы заданы; минуты и секунды по умолчанию";
        t2. printStandard();
        cout << "\nЧасы и минуты заданы;  секунды по умолчанию";
        t3. printStandard();
        cout << "\nЧасы, минуты и секунды заданы:";
        t4. printStandard();
        cout << "\nВсе значения заданы неверно:";
        t5. printStandard();
}
```

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.

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, как и класс, но перед именем записывается знак тильда ~. Кроме того, деструктор не может иметь аргументы, возвращать значение и наследоваться.

Пример:

```
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;
}
void main(void)
{
        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
}
```

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.

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников [1,2,3,4,5].
3. Разработать алгоритм программы.
4. 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. 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. 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. Деструктор должен освобождать память, выделенную под массивы.

4. Создайте класс employee. Класс должен включать поле int для хранения номера сотрудника и поле float для хранения величины его оклада. Методы класса должны позволять пользователю вводить и отображать данные класса. Расширьте содержание класса employee, включив в него класс date и перечисление etype. Объект класса date будет использоваться для хранения даты приема сотрудника на работу. Перечисление будет использовано для хранения статуса сотрудника: лаборант, секретарь, менеджер и т.д. Вам необходимо разработать методы getemploy() и putemploy(), предназначены соответственно для ввода и отображения информации о сотруднике. Напишите программу, которая будет выдавать сведения о сотрудниках.

5. Создайте класс, одно из полей которого хранит «порядковый номер» объекта, т.е. для первого созданного объекта значение этого поля равно 1, для второго созданного объекта значение равно 2 и т.д. Для того чтобы создать такое поле, вам 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 класс следует включить метод, который будет выводить на экран свой порядковый номер, например: *Мой порядковый номер: 2* и т.п.

Лабораторная работа №2

Дружественные функции. Перегрузка операций

Цель работы: Понять назначение дружественных функций и классов. Изучить принципы использования перегрузки операций

Краткие теоретические сведения

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

- не является компонентом-функцией этого класса;
- имеет доступ ко всем компонентам этого класса (private, public и protected).

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

```
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)
{  return (double)i/x.i;
}
main()
{ kls obj(2,3);
   cout  << obj.max() << endl;
   cout << fun(3,obj) << endl;
   return 1;
}
```

Функции со спецификатором friend, не являясь компонентами класса, не имеют и, следовательно, не могут использовать this указатель (будет рассмотрен несколько позже). Следует также отметить ошибочность следующей заголовочной записи функции double kls :: fun(int i,int j), так как 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 компоненты-функции класса осуществляется с использованием операции доступа к компоненте (.) или(->). Вызов же friend-функции производится по ее имени (так как friend-функции не являются его компонентами). Следовательно, как это будет показано далее, в fried-функции 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 (так как она объявлена со спецификатором в классе Y).

Ниже приведен пример программы расчета суммы двух налогов на зарплату.

```
#include "iostream.h"
#include "string.h"
#include "iomanip.h"
class nalogi;                  // неполное объявление класса nalogi
class work
{  char s[20];                 // фамилия работника
   int zp;                     // зарплата
 public:
   float raschet(nalogi);   // компонента-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 соц. 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()
{ 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, так как в прототипе функции raschet класса work используется объект класса nalogi, объявляемого далее. В то же время полное объявление класса nalogi не может быть выполнено ранее (до объявления класса work), так как в нем содержится friend-функция, описание которой должно быть выполнено до объявления friend-функции. В противном случае компилятор выдаст ошибку.

Если функция raschet в классе work также будет использоваться со спецификатором friend, то приведенная выше программа будет выглядеть следующим образом:

```
#include "iostream.h"
#include "string.h"
#include "iomanip.h"

class nalogi;            // неполное объявление класса nalogi
class work
{  char s[20];           // фамилия работника
   int zp;               // зарплата
 public:
   friend float raschet(work,nalogi); // friend-функция класса work
   void inpt()
   { cout << "вводите фамилию и зарплату" << endl;
     cin >> s >>zp;
   }
   work(){}              // конструктор по умолчанию
   ~work(){}             // деструктор по умолчанию
};
class nalogi
{   float pd,            // подоходный налог
    st;                  // налог на соц. страхование
    friend float raschet(work,nalogi); // friend-функция класса nalogi
  public:
    nalogi(float f1,float f2) : pd(f1),st(f2){};
    ~nalogi(void){};
};
float raschet(work wr,nalogi nl)
{ cout << wr.s << setw(6) << wr.zp <<endl;// доступ к данным класса work
  cout << nl.pd << setw(8) << nl.st <<endl;// доступ к данным класса nalogi
  return wr.zp*nl.pd/100+wr.zp*nl.st/100;
}
int main()
{ nalogi nlg((float)12,(float)2.3);  // описание и инициализация объекта
  work wr[2];
  for(int i=0;i<2;i++) wr[i].inpt();  // инициализация массива объктов
```

```
  cout<<setiosflags(ios::fixed)<<setprecision(3)<<endl;
  cout << raschet(wr[0],nlg) << endl; // расчет налога для объекта wr[0]
  cout << raschet(wr[1],nlg) << endl; // расчет налога для объекта wr[1]
  return 0;
}
```

Все функции одного класса можно объявить со спецификатором friend по отношению к другому классу следующим образом.

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

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

```
#include <iostream.h>
class A
{   int i;              // компонента-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;              // компонента-данное класса B
 public:
   friend A;           // объявление класса A другом класса B
   B():j(2){}          // конструктор
   ~B(){}              // деструктор
   void f1_B(A &a){ cout<<a.i+j<<endl;}  // метод, оперирующий
                                         // данными обоих классов
};
void A :: f1_A(B &b)
{ cout<<i<<' '<<b.j<<endl;}
void 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* (то есть если класс A есть friend классу B, то это не означает, что B также является friend классу A), *ни транзитивным* (то есть 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.

Указатель this

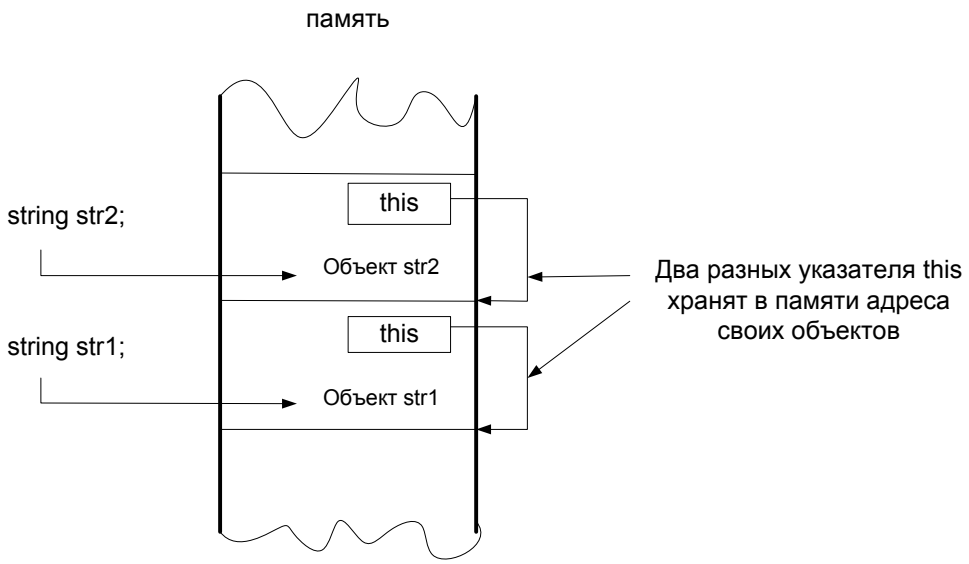
Каждый новый объект имеет скрытый от пользователя указатель на этот объект. По-другому это можно объяснить так. Когда объявляется объект, под него выделяется память. В памяти есть специальное поле, содержащее скрытый указатель, который адресует начало выделенной под объект памяти. Получить значение указателя в компонентах-функциях объекта можно с помощью ключевого слова this.

Для любой функции, принадлежащей классу my_class, указатель this неявно объявлен так: my_class *const this;

Если объявлен класс и объекты

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

Память распределится следующим образом



Основные свойства и правила использования указателя this.

- каждый новый объект имеет свой скрытый указатель this;
- this указывает на начало своего объекта в памяти компьютера;
- this не надо дополнительно объявлять;
- this передается как скрытый аргумент во все нестатические (т.е. не имеющие спецификатора static) компоненты-функции своего объекта;
- this – это локальна переменная, которая недоступна за пределами объекта (она доступна только во всех нестатических компонентах-функциях своего обекта);
- разрешается обращаться к указателю this непосредственно в виде this или *this.

Пример:

```
#include<stdio.h>
#include<string.h>
class String
{
        char str[100];
        int k;
        int *r;
      public:
        String() {k=123; r=&k;}
        void read(){ gets(str);}
        void print() { puts(str); }
        void read_print();
};
void String::read_print()
{
        char c;
        printf("k=%3d",this->k);
        printf("\n*r=%3d",*(this->k));
```

```
        puts(“Введите строку”);
/*Ключевое слово this содержит скрытый указатель на класс String. Поэтому конструкция this->read() выбирает через указатель функцию read() этого класса*/
        this->read();
        puts(“Введенная строка”);
        this->print();
/* 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;
        }
        puts(“Измененная строка”);
        (*this).print();
}
void main(void)
{
        String S;
        S.read_print();
}
```

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 определены 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 операция % должна быть приложена лишь к объектам целого типа. Из этого следует: операторы языка можно применить к тем объектам, для которых они были определены.

К сожалению, лишь определенное число типов непосредственно поддерживается любым языком программирования. Например, языки C и C++ не позволяют выполнять операции с комплексными числами, матрицами, строками, множествами и т.п. Однако все эти операции можно определить через классы в языке C++. Рассмотрим пример. Пусть заданы множества A и B:

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

И мы хотим выполнить операции типа пересечение & и объединение | 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 операторы & и | можно будет использовать как и раньше, а также снабдить их дополнительными функциями (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.

Операции, которые не могут быть перегружены:

., .*, ::, ?:, sizeof

Для перегрузки (доопределения) оператора разрабатываются функции являющимися 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 компонентой 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 его старое предназначение сохраняет силу. Функция operator является компонентом класса. При этом в случае унарной операции operator не будет иметь аргументов, а в случае бинарной операции 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 его можно будет использовать для выполнения операции объединения множеств.

```
#include<iostream.h>
class set
{                                       // класс «множество»
      char str[80];
    public:
      set(char *ss):str(ss) {}          // это конструктор
      char * operator&(set);            // объявление функции operator
};
char * set::operator&(set S)            // описание функции operator
{
      int t=0, l=0;
      while(str[t++]!='\0');
      char *s1=new char[t];
      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;
}
void main(void)
{
      Set S1="1f2bg5e6", S2="abcdef";           // задаются два множества
      cout << (S1 & S2) << endl;                // резудьтат fbe
      cout << (set("123") & set("426")) << endl;  // результат 2
}
```

Приведем еще пример программы перегрузки оператора "-" для использования его при вычитании из одной строки другой.

```
#include<iostream.h>
#include<string.h>
class String                            // Описание класса String
{
      char str[80];
```

```
    public:
        void init(char *s);                 // функция инициализации
        int operator – (String s_new);
};
void String::init(char *s)
{
        Strcpy(str,s);
}
int String::operator – (String s_new)
{
        for(int i=0; str[i]==s_new.str[i]; i++)
         if(!str[i]) return 0;
        return(str[i]-s_new.str[i]);
}
void main(void)
{
        char S1[51], S2[51];
        cout << “Введите первую строку” << endl;
        cin >> S1;
        cout << “Введите вторую строку” << endl;
        cin >> S2;
        String my_string1, my_string2;
         my_string1.init(S1);            // инициализация объекта my_string1
         my_string2.init(S2);            // инициализация объекта my_string2
        cout << “\n String1 – String2=”;
        cout << (my_string1 - my_string2) << endl;
}
```

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников [1,2,№,4,5].
3. Разработать алгоритм программы.
4. 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 операций ввода-вывода в поток.

2. Реализовать класс String для работы со строками символов. Перегрузить операторы !(пуст ли String), ()(int,int) – возвращение подстроки, [].

Предоставить конструктор копирования. Определить friend функции для операций ввода-вывода в поток.

3. Создать класс, в котором перегрузить операторы:
& для перемножения двух матриц;
+ для сложения двух матриц.
Память под матрицы отводить динамически. Предоставить конструктор копирования. Определить friend функции для операций 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 операций ввода-вывода в поток.

5. Создайте класс bMoney, в котором перегрузите арифметические операции для работы с денежным форматом. Перегрузите два оператора следующим образом:
long double * bMoney //умножить число на деньги
long double / bMoney //делить число на деньги
Эти операции требуют наличия дружественных функций, так как справа от оператора находится объект, а слева – обычное число. Убедитесь, 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.

Лабораторная работа №3

Наследование и виртуальные функции

Цель работы: Изучить принципы наследования. Использовать виртуальные функции в наследуемых классах.

Краткие теоретические сведения

Наследование

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.

Шаблон объявления производного класса можно представить следующим образом:

ключ_класса имя_производного_класса:необязательный_модификатор_доступа имя_базового_класса

```
{
        Тело производного класса
};
```

Пример:

```
class Location
{
        int x,y;
      public:
         . . .
};
class Point:public Location
{
        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:

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:

```
class A
{
        int a1;
     public:
         int a2;
         void f1(void);
};
class B:A
{
        int b1;
      public:
        void f1(void)
        {
            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
        }
};
void main(void)
{
        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
}
```

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

```
#include<iostream.h>
#include<string.h>
#define N 10
class book
{
       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
{
        char fm[10];                // фамилия автора
      public:
        avt(char *, int, char *); // конструктор класса avt
        ~avt();                     // деструктор класса avt
        void see();
};
enum razd{teh, hyd,uch};
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";
        strcpy(naz,s1);
        kl=i;
}
book::~book()
{
        cout << "\nДеструктор класса book";
}
avt::avt(char *s1, int i, char s2):book(s1,i)
{
        cout << "\n  Конструктор класса avt";
        strcpy(fm,s2);
}
 avt::~avt()
{
        cout << "\nДеструктор класса avt";
}
void avt::see()
{
        cout << "\nНазвание книги" << naz << endl << "\nКоличество страниц" <<
kl ;
}
rzd::rzd(char *s1, int i, razd tp):book(s1,i)
{
```

```
        cout << “\n Конструктор класса rzd”;
        rz=tp;
}
rzd::~rzd()
{
        cout << “Деструктор класса rzd”;
}
void rzd::see()
{
        switch(rz)
        {
                case teh: cout << “\n Раздел технической литературы”; break;
                case hyd: cout << “\nРаздел художественной литературы”; break;
                case uch: cout << “\n Раздел учебной литературы”; break;
        }
}
void main(void)
{
        avt av(“Книга 1”, 123, “автор 1”);
        rzd rz(“Книга 1”, 123, the);
        av.see();
        rz.see();
}
```

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);
}
```

Множественное наследование

Множественное наследование – это когда производный класс образуется из нескольких базовых классов.

В языке C++ допускается образовывать производный класс от нескольких базовых классов. Поэтому общий синтаксис описания конструктора производного класса представляется в виде:

имя_конструктора_производного_класса(список аргументов для производного класса): имя_базового_класса1(список аргументов для конструктора базового класса1), … , имя_базового_классаN(список аргументов для базового классаN)

```
{
        тело конструктора
}
```

Пример:

```
#include<iostream.h>
class Base_1
{
        int a;
      protected:
        int b;
       public:
        Base_1(int x, int y);
        ~Base_1();
        void show1(void)
        {
                cout << “a=” << a << “b=” << b;
        }
};
class Base_2
{
       protected:
        int c;
       public:
        Base_2(int r);
        ~Base_2();
        void show2(void)
        {
                cout << “c=” << c << endl;
        }
};
class Derive: public Base_1, public Base_2
{
        int p;
       public:
        Derive(int x, int y, int z);
        ~Derive();
```

```
        void show3(void)
        {
                cout << “a+b+c=” << p+b+c << endl;
        }
};
Base_1::Base_1(int x, int y)
{
        a=x;
        b=y;
        cout << “\n Конструктор Base_1”;
}
Base_1::~Base_1()
{
        cout << “\nДеструктор Base_1”;
}
 Base_2::Base_2(int x)
{
        c=x;
        cout << “\n Конструктор Base_2”;
}
Base_2::~Base_2()
{
        cout << “\nДеструктор Base_2”;
}
Derive::Derive(int x, int y, int z):Base_1(x,y), Base_2(z)
{
        p=x;
        cout << “\n Конструктор Derive”;
}
void main(void)
{
        int x,y,z;
        cout << “\nВведите значения x, y, z”;
        cin >> x >> y >> z;
        Derive my_d(x,y,z);
        my_d.show1();
        my_d.show2();
        my_d.show3();
}
```

Каждый из классов Base_1 и Base_2 содержит свой конструктор с параметрами. В связи с этим производный класс тоже должен содержать свой конструктор с параметрами. В функции main() объявлен объект my_d. В результате будут вызваны конструкторы базовых и производного классов. Конструктор производного класса

```
Derive::Derive(int x, int y, int z):Base_1(x,y), Base_2(z)
{ … }
```

Последовательно вызываются конструкторы базовых классов в соответствии с их появлением в списке, т.е. сначала вызывается конструктор Base_1, затем конструктор Base_2, после вызывается конструктор производного класса Derive. Выполнение деструкторов осуществляется в обратном порядке. На основании сказанного, результаты работы программы представляются в следующем виде:

```
Конструктор Base_1
Конструктор Base_2
Конструктор Derive
a=3 b=5 c=7
a+b+c=15
Деструктор Derive
Деструктор Base_2
Деструктор Base_1
```

Виртуальные функции

Один из основных принципов объектно-ориентированного программирования предполагает использование идеи «один интерфейс – множество методов реализации». Эта идея заключается также в том, что базовый класс обеспечивает все элементы, которые производные классы могут непосредственно использовать, плюс набор функций, которые производные классы должны реализовать путем их переопределения. Наряду с механизмом перегрузки функций это достигается использованием виртуальных (virtual) функций. *Виртуальная* функция – это функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или нескольких производных от этого классах. При вызове объекта базового или производных классов динамически (во время выполнения программы) определяется, какую из функций требуется вызвать, основываясь на типе объекта.

Рассмотрим пример использования виртуальной функции.

```
#include "iostream.h"
#include "iomanip.h"
#include "string.h"
class grup                          // БАЗОВЫЙ класс
{ protected:
     char *fak;              //  наименование факультета
     long gr;                //  номер группы
 public:
      grup(char *FAK,long GR) : gr(GR)
      {  if (!(fak=new char[20]))
         { cout<<"ошибка выделения памяти"<<endl;
           return;
```

```
          }
          strcpy(fak,FAK);
        }
        ~grup()
        { cout << "деструктор класса grup " << endl;
          delete fak;
        }
       virtual void see(void);        // объявление виртуальной функции
};
class stud : public grup              // ПРОИЗВОДНЫЙ класс
{     char *fam;                  //  фамилия
      int oc[4];                  //  массив оценок
 public:
      stud(char *FAK,long GR,char *FAM,int OC[]): grup(FAK,GR)
      { if (!(fam=new char[20]))
        { cout<<"ошибка выделения памяти"<<endl;
          return;
        }
        strcpy(fam,FAM);
        for(int i=0;i<4;oc[i]=OC[i++]);
      }
      ~stud()
      { cout << "деструктор класса stud " << endl;
        delete fam;
      }
      void see(void);
};
void grup::see(void)      // описание виртуальной функции
{ cout << fak << gr << endl;}
void stud::see(void)      //
{ grup ::see();               // вызов функции базового класса
  cout <<setw(10) << fam << "  ";
  for(int i=0; i<4; cout << oc[i++]<<' ');
  cout << endl;
}
int main()
{ int OC[]={4,5,5,3};
  grup gr1("факультет 1",123456), gr2("факультет 2",345678), *p;
  stud st("факультет 2",150502,"Иванов",OC);
  p=&gr1;           // указатель на объект базового класса
  p->see();         // вызов функции базового класса объекта gr1
  (&gr2)->see();  // вызов функции базового класса объекта gr2
  p=&st;            // указатель на объект производного класса
  p->see();        // вызов функции производного класса объекта st
```

```
  return 0;
}
```
Результат работы программы:
```
факультет 1   123456
факультет 2   345678
факультет 2",150502
      Иванов   4 5 5 3
```
Объявление virtual void see(void) говорит о том, что функция see может быть различной для базового и производных классов. Тип виртуальной функции не может быть переопределен в производных классах. Исключением является случай, когда возвращаемый тип виртуальной функции является указателем или ссылкой на порожденный класс, а виртуальная функция основного класса - указателем или ссылкой на базовый класс. 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. Вызов функций должен производиться с учетом списка параметров.

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 в процессе выполнения. Доступ к виртуальной функции осуществляется через этот указатель и соответствующую таблицу (то есть выполняется косвенный вызов функции).

Если функция вызывается с использованием ее полного имени grup::see(), то виртуальный механизм игнорируется. Игнорирование этого может привести к серьезной ошибке:
```
void stud::see(void)
{ see();
   . . . .
}
```
в этом случае инструкция see() приводит к бесконечному рекурсивному вызову функции see().

Заметим, что деструктор может быть виртуальным, а конструктор нет.

Замечание. В чем разница между виртуальными функциями (методами) и переопределением функции?

Что изменилось, если бы функция see() не была бы описана как виртуальная? В этом случае решение о том, какая именно из функций see() должна быть выполнена, будет принято при ее компиляции.

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 классов. Если в программе имеется сам объект, то уже на стадии компиляции известен его тип и, следовательно, механизм виртуальности не используется. Например:

```
func(cls obj)
{
     obj.vvod();    // вызов компоненты-функции obj::vvod
}
func1(cls &obj)
{
     obj.vvod();    // вызов компоненты-функции в соответствии
}                   // с типом объекта, на который  ссылается obj
```

Виртуальные функции позволяют принимать решение в процессе выполнения.

```
#include "iostream.h"
#include "iomanip.h"
#include "string.h"
#define N 5
class base                  // БАЗОВЫЙ класс
{ public:
     virtual char *name(){ return " noname ";}
     virtual double area(){ return 0;}
};
class rect : public base   // ПРОИЗВОДНЫЙ класс (прямоугольник)
{    int h,s;
  public:
     virtual char *name(){ return " прямоугольника ";}
      rect(int H,int S): h(H),s(S) {}
     double area(){ return h*s;}
};
class circl : public base   // ПРОИЗВОДНЫЙ класс (прямоугольник)
{    int r;
  public:
    virtual char *name(){ return " круга ";}
    circl(int R): r(R){}
    double area(){ return 3.14*r*r;}
};

int main()
{ base *p[N],a;
  double s_area=0;
  rect b(2,3);
  circl c(4);
  for(int i=0;i<N;i++) p[i]=&a;
  p[0]=&b;
  p[1]=&c;
```

```
  for(i=0;i<N;i++)
  cout << "площадь" << p[i]->name() << p[i]->area() << endl;
  return 0;
}
```

Массив указателей p хранит адреса объектов базового и производных классов и необходим для вызова виртуальных функций этих классов. Виртуальная функция базового класса необходима тогда, когда она явно вызывается для базового класса или не определена (не переопределена) для некоторого производного класса.

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.

```
class A
{  . . .
   public:  virtual void fun() {}
};
class B : public A
{  . . .
   public:  void fun() {}
};

class C : public B
{  . . .
   public:
   . . .   // в объявлении класса C отсутствует описание функции fun()
};
main()
{ A *p=&a;
  B b;
  C c;
  p->fun();    // вызов версии виртуальной функции fun для класса A
  p=&b;
  p->fun();    // вызов версии виртуальной функции fun для класса B
  p=&c;
  p->fun();    // вызов версии виртуальной функции fun для класса C
}
```

Если в производном классе виртуальная функция не переопределяется, то используется ее версия из базового класса.

```
class A
{  . . .
   public:  virtual void fun() {}
};
class B : public A
```

```
{  . . .
   public:  void fun() {}
};

class C : public B
{  . . .
   public:  void fun() {}
};
main()
{ A *p=&a;
  B b;
  C c;
  p->fun();    // вызов версии виртуальной функции fun для класса A
  p=&b;
  p->fun();    // вызов версии виртуальной функции fun для класса B
  p=&c;
  p->fun();    // вызов версии виртуальной функции fun для класса A
}
```

Основное достоинство данной иерархии состоит в том, что код не нуждается в глобальном изменении, даже при добавлении вычислении площадей новых фигур. Изменения вносятся локально и благодаря полиморфному характеру кода распространяются автоматически.

Рассмотрим механизм вызова виртуальной функции базового и производного классов из компонент-функций этих классов, вызываемых через указатель на базовый класс.

```
class A
{ public :
     virtual void f()
     { return; }
     void fn()
     { f(); }        // вызов функции f
};
class B : public A
{ public:
    void f()
     { return;  }
    void fn()
     { f(); }         // вызов функции f
};
main()
{ A *pa=&a;
  B *pb=&b;
  pa = &b;
```

```
  pa->fn();  //  вызов виртуальной ф-ции f класса b через  A::fn()
  pb->fn();  //  вызов виртуальной ф-ции f класса b через  B::fn()
}
```

В инструкции pa->fn() выполняется вызов функции fn() базового класса A, так как указатель **pa** – указатель на базовый класс и компилятор выполняет вызов функции базового класса. Далее из функции fn() выполняется вызов виртуальной функции f() класса B, так как указатель **pa** инициализирован адресом объекта B.

Перечислим основные свойства и 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 объявить только в классах (class) и структурах (struct);
- виртуальными функциями могут быть только нестатические функции (без спецификатора static), так как характеристика virtual унаследуется. Функция порожденного класса автоматически становится virtual;
- виртуальные функции можно объявить со спецификатором friend для другого класса;
- виртуальными функциями могут быть только не глобальные функции (то есть 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. Если спецификатор virtual опущен, то новые версии функции далее не будут рассматриваться;
- для вызова виртуальной функции требуется больше времени, чем для не виртуальной. При этом также требуется дополнительная память для хранения виртуальной таблицы;
- при использовании полного имени при вызове некоторой виртуальной функции (например, grup ::see();), виртуальный механизм не применяется.

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников [1,2,3,4,5].
3. Разработать алгоритм программы.
4. 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. Реализовать класс «Статья» с полями «название статьи» и «имя автора» и методами, позволяющими посмотреть эти поля. Реализовать класс «Библиотечная карточка» хранящая автора, название, авторский знак, инвентарный номер, код по тематическому каталогу. Класс должен иметь методы, возвращающие эти поля и иметь метод для алфавитного поиска, возвращающий объединенное значение полей «автор» и «название».

Реализовать производные от него классы:

1) «Карточка для самостоятельного издания» с методами, возвращающими издательство, год издания, тираж, количество страниц и производные от него классы:

а) «карточка для сборника статей» с методом, возвращающим карточки, описывающие статьи (см. ниже);

б) «карточка для книги» (книга состоит из одной статьи), дополнительный метод должен возвращать статью.

2) «Карточка для статьи» с методами, возвращающими кроме имени автора, статью и ссылку на карточку для издания, в котором статья опубликована. Реализовать класс «Каталог», с производными «Тематический каталог», «Алфавитный каталог» с методами, реализующими поиск по шифру или автору (названию).

3. При наличии классов Circle(круг), Square (квадрат) и Triangle(треугольник), производных от Shape(фигура), определите 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.

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

Реализовать производные классы:

1) «Преподаватель университета» с полями: должность, ученая степень, специальность, список научных трудов (массив строк);

2) «Член комиссии» с полями: название комиссии, год назначения в комиссию, номер свидетельства, автобиография (массив строк);

3) «Преподаватели – члены комиссии» (производный от 2 и 3). Дополнительное поле – список работ выполненных в комиссии.

Классы должны содержать методы доступа и изменения всех полей.

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

Реализовать производные классы:

1) «Предприниматель» - содержит номер лицензии, адрес регистрации, УНН, данные о налоговых платежах (массив пар вида <дата, сумма>).
2) «Турист» - содержит данные паспорта(строка), данные о пересечении границы в виде массива пар <дата, страна>.
3) «Челнок» (производный от 2 и 3) – добавляется массив строк – список адресов, по которым покупается товар

Классы должны содержать методы доступа и изменения всех полей.

Лабораторная работа №4

Реализация шаблонов классов

Цель работы: Научиться использовать шаблоны функций для создания группы однотипных функций. Научиться использовать шаблоны классов для создания группы связанных типов классов.

Краткие теоретические сведения

Шаблоны

Параметризированные классы

Параметризированный класс – некоторый шаблон, на основе которого можно строить другие классы. Этот класс можно рассматривать как некоторое описание множества классов, отличающихся только типами их данных. В C++ используется ключевое слово template для обеспечения параметрического полиморфизма. Параметрический полиморфизм позволяет использовать один и тот же код относительно различных типов (параметров тела кода). Это наиболее полезно при определении контейнерных классов. Шаблоны определения класса и шаблоны определения функции позволяют многократно использовать код, корректно по отношению к различным типам, позволяя компилятору автоматизировать процесс реализации типа.

Шаблон класса определяет правила построения каждого отдельного класса из некоторого множества разрешенных классов.

Спецификация шаблона класса имеет вид:

template <список параметров>
class объявление класса

Список параметров класса-шаблона представляет собой идентификатор типа, подставляемого в объявление данного класса при его генерации. Рассмотрим пример шаблона класса работы с динамическим массивом и выполнением контроля за значениями индекса при обращении к его элементам.

```
#include "iostream.h"
#include "string.h"
template <class T>

class vector
{       T *ms;
        int size;
  public:
        vector() : size(0),ms(NULL) {}
        ~vector(){delete [] ms;}

        void decrem(const T &t)  // увеличение размера массива на 1 элемент
        {  T *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
        }

        void inkrem(void)          // уменьшение размера массива на 1 элемент
        {  T *tmp = ms;
            if(size>1) ms=new T[--size];
            if(tmp)
            { memcpy(ms,tmp,sizeof(T)*size); // перезапись без посл.элемента
              delete [] tmp;          // удаление временного массива
            }
        }
        T &operator[](int ind)  // определение обычного метода
        {  // if(ind<0 || (ind>=size)) throw IndexOutOfRange; // возбуждение
                                  // исключительной ситуации IndexOutOfRange
          return ms[ind];
        }
};

void main()
{  vector <int> VectInt;
   vector <double> VectDouble;
   VectInt.decrem(3);
   VectInt.decrem(26);
   VectInt.decrem(12);        // получен int-вектор (массив) из 3 атрибутов
   VectDouble.decrem(1.2);
   VectDouble.decrem(.26);//получен double-вектор (массив) из 2 атрибутов
```

```
  int a=VectInt[1];          // a = ms[1]
  cout << a << endl;
  int b=VectInt[4];          // будет возбуждена исключительная ситуация
  cout << b << endl;
  double d=VectDouble[0];
  cout << d << endl;
  VectInt[0]=1;
  VectDouble[1]=2.41;
}
```

Класс vector наряду с конструктором и деструктором имеет 2 функции: decrem – добавление в конец вектора нового элемента, inkrem – уменьшение числа элементов на единицу и операция [] обращения к i-му элементу вектора.

Параметр шаблона vector – любой тип, у которого определены операция присваивания и операция new. Например, при задании объекта типа vector <int> происходит генерация конкретного класса из шаблона и конструирование соответствующего объекта VectInt, при этом тип T получает значение типа int. Генерация конкретного класса означает, что генерируются все его компоненты-функции, что может привести к существенному увеличению кода программы.

Выполнение функций

```
VectInt.decrem(3);
VectInt.decrem(26);
VectInt.decrem(12);
```

приведет к созданию вектора (массива) из трех атрибутов (3, 26 и 12).

Сгенерировать конкретный класс из шаблона можно, явно записав:

template vector<int>;

При этом не будет создано никаких объектов типа vector<int>, но будет сгенерирован класс со всеми его компонентами.

В некоторых случаях желательно описания некоторых компонент-функций шаблона класса выполнить вне тела шаблона, например:

```
#include "iostream.h"

template <class T1,class T2>
T1 sm1(T1 aa,T2 bb)                 //  описание шаблона глобальной
{ return (T1)(aa+bb);               //  функции суммирования значений
}                                   //  двух аргументов

template <class T1,class T2>
class cls
{   T1  a;
    T2  b;
 public:
    cls(T1 A,T2 B) : a(A),b(B) {}
    ~cls(){}
     T1 sm1()                       // описание шаблона функции
```

```
      { return (T1)(a+b);          // суммирования компонент объекта  obj_
      }
 T1 sm2(T1,T2);               // объявление шаблона функции
};
template <class T1,class T2>
T1 cls<T1,T2>::sm2(T1 aa,T2 bb)        // описание шаблона функции
{ return (T1)(aa+bb);                  // суммирования внешних данных
}
void main()
{ cls <int,int> obj1(3,4);
  cls <double,double> obj2(.3,.4);
  cout<<"функция суммирования компонент объекта 1       = "
      <<obj1.sm1()<<endl;
  cout<<"функция суммирования внешних данных (int,int)  = "
       <<obj1.sm2(4,6)<<endl;
  cout<<"вызов глобальной функции суммирования (int,int)    = "
      <<sm1(4,.6)<<endl;
  cout<<"функция суммирования компонент объекта 2          = "
      <<obj2.sm1()<<endl;
  cout<<"функция суммирования внешних данных (double,double)= "
      <<obj2.sm2(4.2,.1)<<endl;
}
```

Передача в шаблон класса дополнительных параметров

При создании экземпляра класса из шаблона в него могут быть переданы не только типы, но и переменные и константные выражения:

```
#include "iostream.h"
template <class T1,int i=0,class T2>
class cls
{    T1  a;
     T2  b;
 public:
    cls(T1 A,T2 B) : a(A),b(B){}
    ~cls(){}
    T1 sm() //описание шаблона ф-ции суммирования компонент объекта
    { //  i+=3;    // error member function 'int __thiscall cls<int,2>::sm(void)'
      return (T1)(a+b+i);
    }
};
void main()
{ cls <int,1,int> obj1(3,2);       //  в шаблоне const i инициализируется  1
  cls <int,0,int> obj2(3,2,1);     //  error 'cls<int,0>::cls<int,0>':no overloaded
```

```
                                        //  function takes 3 parameter s
  cls <int,int,int> obj13(3,2,1); //  error 'cls' : invalid template argument for 'i',
                                        //  constant expression expected
  cout<<obj1.sm()<<endl;
}
```

Результатом работы программы будет выведенное на экран число 6.

В этой программе инструкция **template <class T1,int i=0,class T2>** говорит о том, что шаблон класса cls имеет три параметра, два из которых - имена типов (T1 и T2), а третий (int i=0) - целочисленная константа. Значение константы i может быть изменено при описании объекта cls <int,**1**,int> obj1(3,2).

Шаблоны функций

В C++, так же как и для класса, для функции (глобальной, то есть не являющейся компонентой-функцией) может быть описан шаблон. Это позволит снять достаточно жесткие ограничения, накладываемые механизмом формальных и фактических параметров при вызове функции. Рассмотрим это на примере функции, вычисляющей сумму нескольких аргументов.

```
#include "iostream.h"
#include "string.h"

template <class T1,class T2>
T1 sm(T1 a,T2 b)                    // описание шаблона
{
   return (T1)(a+b);                // функции с 2 параметрами
}

template <class T1,class T2,class T3>
T1 sm(T1 a,T2 b,T3 c)       // описание шаблона функции
{ return (T1)(a+b+c);       // функции с 3 параметрами
}

void main()
{cout<<"вызов ф-ции суммирования sm(int,int)       =   "<<sm(4,6)<<endl;
 cout<<"вызов ф-ции суммирования sm(int,int,int)   = "<<sm(4,6,1)<<endl;
 cout<<"вызов ф-ции суммирования sm(int,double)  = "<<sm(5,3)<<endl;
 cout<<"вызов ф-ции суммирования sm(double,int,short)= " <<
          sm(.4,6,(short)1)<<endl;
// cout<<sm("я изучаю","язык C++")<<endl;  error cannot add two pointers
}
```

В программе описана перегруженная функция sm(), первый экземпляр которой имеет 2, а второй 3 параметра. При этом тип формальных параметров функции определяется при вызове функции типом ее фактических параметров. Используемые типы T1, T2, T3 заданы как параметры для функции с помощью

выражения template <class T1,class T2,class T3>. Это выражение предполагает использование типов T1, T2 и T3 в виде ее дополнительных параметров. Результат работы программы будет иметь вид:

вызов функции суммирования sm(int,int) = 10
вызов функции суммирования sm(int,int,int) = 11
вызов функции суммирования sm(int,double) = 8
вызов функции суммирования sm(double,int,short)= 7.4

В случае попытки передачи в функцию sm() двух строк, то есть типов, для которых не определена данная операция, компилятор выдаст ошибку. Чтобы избежать этого, можно ограничить использование шаблона функции sm(), описав явным образом функцию sm() для некоторых конкретных типов данных. В нашем случае:

```
char *sm(char *a,char *b)          //  явное описание функции объединения
{ char *tmp=a;                     //  двух строк
  a=new char[strlen(a)+strlen(b)+1];
  strcpy(a,tmp);
  strcat(a,b);
  return a;
}
```

Добавление в main() инструкции, например,

```
cout<<sm("я изучаю"," язык C++")<<endl;
```

приведет к выводу кроме указанных выше сообщения:

я изучаю язык C++

Следует отметить, что шаблон функции не является ее экземпляром. Только при обращении к функции с аргументами конкретного типа происходит генерация конкретной функции.

Совместное использование шаблонов и наследования

Шаблонные классы, как и обычные, могут использоваться повторно. Шаблоны и наследование представляют собой механизмы повторного использования кода и могут включать полиморфизм. Шаблоны и наследования связаны между собой следующим образом:

- шаблон класса может быть порожден от обычного класса;
- шаблонный класс может быть производным от шаблонного класса;
- обычный класс может быть производным от шаблона класса.

Ниже приведен пример простой программы, демонстрирующей наследование шаблонного класса oper от шаблонного класса vect.

```
#include "iostream.h"
template <class T>
class vect                    // класс-вектор
{protected:
     T *ms;                   // массив-вектор
```

```
        int size;                       // размерность массива-вектора
 public:
        vect(int n) : size(n)           // конструктор
        { ms=new T[size];}
        ~vect(){delete [] ms;}          // деструктор
        T &operator[](const int ind)    // доопределение операции []
        { if((ind>0) && (ind<size)) return ms[ind];
          else return ms[0];
        }
};
template <class T>
class oper : public vect<T>     // класс операций над вектором
{ public:
        oper(int n): vect<T>(n) {}      // конструктор
        ~oper(){}                       // деструктор
        void print()                    // функция вывода содержимого вектора
        { for(int i=0;i<size;i++)
          cout<<ms[i]<<' ';
          cout<<endl;
        }
};
void main()
{ oper <int> v_i(4);                    //   int-вектор
  oper <double> v_d(4);                 //  double-вектор
  v_i[0]=5;     v_i[1]=3;     v_i[2]=2;    v_i[3]=4;     // инициализация int
  v_d[0]=1.3;  v_d[1]=5.1;  v_d[2]=.5;  v_d[3]=3.5;  // инициализация double
  cout<<"int вектор = ";
  v_i.print();
  cout<<"double вектор = ";
  v_d.print();
}
```

Как следует из примера, реализация производного класса от класса шаблона в основном ничем не отличается от обычного наследования.

Некоторые примеры использования шаблона класса

При использовании в программе указателя на объект, память для которого выделена с помощью оператора new, в случае если объект становится не нужен, то для его разрушения необходимо явно вызвать оператор delete. 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 тем, что наряду с указателем на

объект хранится счетчик числа других указателей на этот же объект. Объект может быть уничтожен только в том случае, если счетчик ссылок станет равным нулю.

```
#include "iostream.h"
#include "string.h"

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
    //   Point &operator=(T *ptr);          // перегрузка
    T *operator->() const;
    T &operator*() const;
};
```

Приведенный ниже конструктор Point инициализирует объект указателем. Если указатель равен NULL, то указатель на структуру Status, содержащую указатель на объект и счетчик других указателей, устанавливается в NULL. В противном случае создается структура Status

```
template <class T>
Point<T>::Point(T *ptr)      // описание конструктора
{ if(!ptr) StatPtr=NULL;
  else
  { StatPtr=new Status<T>;
    StatPtr->RealPtr=ptr;
    StatPtr->Count=1;
  }
}
```

Копирующий конструктор StatPtr не выполняет задачу копирования исходного объекта, а так как новый указатель ссылается на тот же объект, то мы лишь увеличиваем число ссылок на объект.

```
template <class T>          // описание конструктора копирования
Point<T>::Point(const Point &p):StatPtr(p.StatPtr)
{ if(StatPtr) StatPtr->Count++;    // увеличено число ссылок
}
```

Деструктор уменьшает число ссылок на объект на 1, и при достижении значения 0 объект уничтожается

```
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 <=0,
    { delete StatPtr->RealPtr;      // то уничтожается объект
      delete StatPtr;
    }
  }
}
template <class T>
T *Point<T>::operator->() const
{ if(StatPtr) return StatPtr->RealPtr;
  else return NULL;
}
template <class T>
T &Point<T>::operator*() const          // доступ к StatPtr осуществляется
{ if(StatPtr) return *StatPtr->RealPtr;   // посредством this-указателя
  else throw bad_pointer;                 // исключительная ситуация
}
```

При выполнении присваивания вначале необходимо указателю слева от знака = отсоединиться от ”своего” объекта и присоединиться к объекту, на который указывает указатель справа от знака =.

```
template <class T>
Point<T> &Point<T>::operator=(const Point &p)
{                           // отсоединение объекта справа от = от указателя
 if(StatPtr)
 { StatPtr->Count--;
   if(StatPtr->Count<=0)            // так же как и в деструкторе
   { delete StatPtr->RealPtr;       // освобождение выделенной под объект
 delete StatPtr;                    // динамической памяти
   }
 }
                          // присоединение к новому указателю
 StatPtr=p.StatPtr;
 if(StatPtr) StatPtr->Count++;
 return *this;
}
```

struct Str

```
{  int a;
   char c;
};
void main()
{ Point<Str> pt1(new Str); // генерация класса Point, конструирование
                           // объекта pt1, инициализируемого указателем
                           // на стр-ру Str, далее с объектом можно обра-
                           // щаться 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=12;              // operator*() получает this указатель на pt1
  (*pt1).c='b';
  int X=pt1->a;             // operator->() получает this-указатель на pt1
  char C=pt1->c;
}
```

Порядок выполнения работы

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

Варианты заданий

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

2. Напишите шаблон класса для работы с очередью. Определите несколько очередей разных типов и поработайте с их данными.

3. Создайте функцию swaps(), обменивающую значения двух аргументов, посылаемых ей. Сделайте из функции шаблон, чтобы она могла использоваться с любыми числовыми типами данных. Напишите main() для тестирования функции.

4. Напишите простой шаблон предикатной функции isEqualTo, которая сравнивает два своих параметра при помощи операции проверки равенства (==) и возвращает true, если они равны, и false, если не равны. Используйте этот шаблон функции в программе, которая вызывает isEqualTo с различными встроенными типами аргументов. Затем напишите отдельную версию

программы, которая вызывает isEqualTo с определенным пользователем типов и неперегруженной операцией равенства. Что случится, когда вы попытаетесь выполнить эту программу? Теперь перегрузите операцию равенства. Что получится, если теперь вы попытаетесь выполнить эту программу?

5. Напишите шаблон класса односвязного списка, который принимает элементы любого типа. В классе реализуйте функции для работы с односвязным списком.

Лабораторная работа №5

Классы–контейнеры и классы-итераторы

Цель работы: Научиться использовать контейнеры и итераторы

Краткие теоретические сведения

Общее понятие о контейнере

Контейнер – это некоторое множество таких классов, которые отличаются только типами используемых в них данных. Таким образом, контейнер и шаблон(template) – понятия одного порядка. Обычно контейнеры содержат некоторые типовые структуры, такие, как массив или стек, и типовые операции над данными, которые могут быть записаны в эти структуры либо прочитаны из этих структур.

Контейнер - это хранилище объектов (как встроенных, так и определённых пользователем типов). Как правило, контейнеры реализуются в виде шаблонов классов. Простейшие виды контейнеров (статические и динамические массивы) встроены непосредственно в язык C++. Кроме того, стандартная библиотека включает в себя реализации таких контейнеров, как вектор (vector), список (list), очередь (deque), ассоциативный массив (map), множество (set) и некоторых других.

Алгоритм - это функция для манипулирования объектами, содержащимися в контейнере. Типичные примеры алгоритмов - сортировка и поиск. В STL реализовано порядка 60 алгоритмов, которые можно применять к различным контейнерам, в том числе к массивам, встроенным в язык C++.

Итератор - это абстракция указателя, то есть объект, который может ссылаться на другие объекты, содержащиеся в контейнере. Основные функции итератора - обеспечение доступа к объекту, на который он ссылается (разыменование), и переход от одного элемента контейнера к другому (итерация, отсюда и название итератора). Для встроенных контейнеров в качестве итераторов используются обычные указатели. В случае с более

сложными контейнерами итераторы реализуются в виде классов с набором перегруженных операторов.

Остановимся более подробно на рассмотрении введенных понятий.

Контейнеры. Каждый контейнер предоставляет строго определённый интерфейс, через который с ним будут взаимодействовать алгоритмы. Этот интерфейс обеспечивают соответствующие контейнеру итераторы. Важно подчеркнуть, что никакие дополнительные функции-члены для взаимодействия алгоритмов и контейнеров не используются. Это сделано потому, что стандартные алгоритмы должны работать, в том числе со встроенными контейнерами языка C++, у которых есть итераторы (указатели), но нет ничего, кроме них. Таким образом, при создании собственного контейнера реализация итератора - необходимый минимум.

Каждый контейнер реализует определённый тип итераторов. При этом выбирается наиболее функциональный тип итератора, который может быть эффективно реализован для данного контейнера. "Эффективно" означает, что скорость выполнения операций над итератором не должна зависеть от количества элементов в контейнере. Например, для вектора реализуется итератор с произвольным доступом, а для списка - двунаправленный. Поскольку скорость выполнения операции [] для списка линейно зависит от его длины, итератор с произвольным доступом для списка не реализуется.

Общее понятие об итераторе

Для структурированных итераций, например, при обработке массивов:

for(i=0;i<size;i++) sm+=a[i];

порядок обращения к элементу управляется индексом i, который изменяется явно. Можно зафиксировать получение следующего элемента в компоненте-функции.

```
class vect
{  int *p;        // массив чисел
   int size;      // размерность массива
   int ind;       // текущий индекс
 public:
   vect();              // размерность массива const
   vect(int SIZE);   // размерность массива size
  ~vect();
  int ub(){return size-1;}
  int next()
  { if(ind==pv->size)
       return pv->p[(ind=0)++];
    else
       return pv->p[ind++];
  }
};
```

Это соответствует тому, что обращение к объекту ограничивается использованием одного индекса ind. Другая возможность состоит в том, чтобы

создать множество индексов и передавать функции обращения к элементу один из них. Это ведет к существенному увеличению числа переменных. Более удобным представляется создание отдельного, связанного с vect класса (класса итераций), в функции которого входит обращение к элементам класса vect.

```
#include <iostream.h>
class vect
{  friend class vect_iterator;          // предварительное friend-объявление
   int *p;                              // базовый указатель (массив)
   int size;                            // размерность массива
 public:
   vect(int SIZE):size(SIZE)
   { p=new int[size];
     for(int i=0; i<size; *(p+i++)=i);
   }
   int ub(){return size-1;}             // возвращается размер массива
   void add()                           // изменение содержимого массива
   { for(int i=0; i<size; *(p+i++)+=1);}
   ~vect(){delete [] p;}
};
class vect_iterator
{  vect *pv;                            // указатель на класс vect
   int ind;                             // текущий индекс в массиве
 public:
   vect_iterator(vect &v): ind(0),pv(&v){}
   int &next();//возвращается текущее значение из массива (с индекса ind)
};
int &vect_iterator::next()
{ if(ind==pv->size)
    return pv->p[(ind=0)++];
  else
return pv->p[ind++];
}
void main()
{ vect v(5);
  vect_iterator v_i1(v),v_i2(v);            // создано 2 объекта-итератора
                                            // для прохода по объекту vect
  cout<<v_i1.next()<<' '<<v_i2.next()<<endl;
  v.add();                                  // модификация объекта v
  cout<<v_i1.next()<<' '<<v_i2.next()<<endl;
  for(int i=0;i<v.ub();i++)
  cout<<v_i1.next()<<endl;
}
```

Результат работы программы:

```
0   0
2   2
3
4
5
1
```

Полное отсоединение обращения от составного объекта позволяет объявлять столько объектов итераторов, сколько необходимо. При этом каждый из объектов итераторов может просматривать объект vect независимо от других.

Итератор представляет собой операцию, обеспечивающую последовательный доступ ко всем частям объекта. Итераторы имеют свойства, похожие на свойства указателей, и могут быть использованы для указания на элементы контейнеров первого класса. Итераторы реализуются для каждого типа контейнера. Также имеется целый ряд операций (*, ++ и другие) с итераторами, стандартными для контейнеров.

Если итератор **a** указывает на некоторый элемент, то **++a** указывает на следующий элемент, а ***a** ссылается на элемент, на который указывает **a**.

Объект типа **iterator** может использоваться для ссылки на элемент контейнера, который может быть модифицирован, а **const_iterator** для ссылки на немодифицируемый элемент контейнера.

Порядок выполнения работы

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

Варианты заданий.

1. Реализовать шаблон класса Stack. Реализовать пассивный итератор для класса Stack, в который будет передоваться определяемая пользователем функция. Реализовать активный итератор для класса Stack, предоставляющий функции продвижения по стеку и получения текущего элемента стека. Предоставить 3-4 функции для выполнения математических операций (найти среднее значение элементов стека и т.п.), операции поиска по стеку, замены значения элемента стека на указанное, в которые как параметры передаются 2 активных итератора и необходимые добавочные данные.

2. Создайте список целых чисел. Создайте два итератора: один для продвижения в прямом направлении, другой для продвижения в обратном направлении. Используйте их для переворачивания содержимого списка.

3. Создайте шаблон класса Stack и заполните его четными числами. Создайте шаблон класса Stack1 и заполните егонечетными числами. С помощью итератора объедините эти контейнеры в вектор.

4. Придумайте и реализуйте дважды связанный список, которым можно пользоваться посредством итератора. Итератор иметь действия для движения вперед и назад, действия для вставления и удаления элементов списка, и способ доступа к текущему элементу.

5. Определите класс Triangle (треугольник) и класс Circle (круг). Определите функцию, которая рисует линию, соединяющую две фигуры, отыскивая две ближайшие «точки соприкосновения» и соединяя их.

Лабораторная работа №6

Генерация и обработка исключений

Цель работы: Понять, как обрабатываются исключения. Использовать try, throw и catch для отслеживания, индикации и обработки исключений.

Краткие теоретические сведения

Исключения в C++

Исключения – возникновение непредвиденных ошибочных условий, например, деление на ноль, невозможность выделения памяти при создани нового объекта и т.д. Обычно эти условия завершают выполнение программы с системной ошибкой. C++ позволяет восстанавливать программу из этих условий и продолжать ее выполнение. В то же время исключение – это более общее, чем ошибка, понятие и может возникать и тогда, когда в программе нет ошибок.

Механизм обработки исключительных ситуаций на сегодняшний день является неотъемлемой частью языка C++. Этот механизм предоставляет программисту средство реагирования на нештатные события и позволяет преодолеть ряд принципиальных недостатков следующих традиционных методов обработки ошибок:

- возврат функцией кода ошибки;
- возврат значений ошибки через аргументы функций;
- использование глобальных переменных ошибки;
- использование оператора безусловного перехода goto или функций setjmp/longjmp;
- использование макроса assert.

Возврат функцией кода ошибки является самым обычным и широко применяемым методом. Однако этот метод имеет существенные недостатки. Во-первых, нужно помнить численные значения кодов ошибок. Эту проблему можно обойти, используя перечисляемые типы. Но в некоторых случаях функция может возвращать широкий диапазон допустимых (неошибочных) значений, и тогда сложно найти диапазон для возвращаемых кодов ошибки. Это и является вторым недостатком. И, в-третьих, при использовании такого механизма сигнализации об ошибках вся ответственность по их обработке ложится на программиста, и могут возникнуть ситуации, когда серьезные ошибки будут оставлены без внимания.

Возврат кода ошибки через аргумент функции или использование глобальной переменной ошибки снимают, прежде всего, вторую проблему, однако по-прежнему остаются первая и третья. Кроме того, использование глобальных переменных не является особо позитивным фактором.

Использование оператора безусловного перехода в любых ситуациях является нежелательным, кроме того, оператор goto действует только в пределах функции. Пара функций setjmp/longjmp является довольно мощным средством, однако и этот метод имеет серьезнейший недостаток: он не обеспечивает вызов деструкторов локальных объектов при выходе из области видимости, что, естественно, влечет за собой утечки памяти.

И, наконец, макрос assert является скорее средством отладки, чем средством обработки нештатных событий, возникающих в процессе использования программы.

Таким образом, необходим некий другой способ обработки ошибок, который учитывал бы объектно-ориентированную философию. Таким способом и является механизм обработки исключительных ситуаций.

Основы обработки исключительных ситуаций

Обработка исключительных ситуаций лишена недостатков вышеназванных методов реагирования на ошибки. Этот механизм позволяет использовать для представления информации об ошибке объект любого типа. Поэтому можно, например, создать иерархию классов, которая будет предназначена для обработки аварийных событий. Это упростит, структурирует и сделает более понятной программу.

Рассмотрим пример обработки исключительных ситуаций. Функция div(), возвращающет частное от деления чисел, принимаемых в качестве аргументов. Если делитель равен нулю, то генерируется исключительная ситуация.

```
#include<iostream.h>
double div(double dividend, double divisor)
{  if(divisor==0)  throw 1;
   return dividend/divisor;
}
void main()
```

```
{ double result;
  try {
    result=div(77.,0.);
    cout<<"Answer is "<<result<<endl;
  }
  catch(int){
    cout<<"Division by zero"<<endl;
}
  }
```

Результат выполнения программы:
Division by zero

В данном примере необходимо выделить три ключевых элемента. Во-первых, вызов функции div() заключен внутрь блока, который начинается с ключевого слова **try**. Этот блок указывает, что внутри него могут происходить исключительные ситуации. По этой причине код, заключенный внутри блока try, иногда называют охранным.

Далее за блоком try следует блок **catch**, называемый обычно обработчиком исключительной ситуации. Если возникает исключительная ситуация, выполнение программы переходит к этому catch-блоку. Хотя в этом примере имеется один-единственный обработчик, их в программах может присутствовать множество и они способны обрабатывать множество различных типов исключительных ситуаций.

Еще одним элементом процесса обработки исключительных ситуаций является оператор **throw** (в данном случае он находится внутри функции div()). Оператор throw сигнализирует об исключительном событии и генерирует объект исключительной ситуации, который перехватывается обработчиком catch. Этот процесс называется вызовом исключительной ситуации. В рассматриваемом примере исключительная ситуация имеет форму обычного целого числа, однако программы могут генерировать практически любой тип исключительной ситуации.

Если в инструкции if(divisor==0) throw 1 значение 1 заменить на 1. , то при выполнении будет выдана ошибка об отсутствии соответствующего обработчика catch (так как возбуждается исключительная ситуация типа double).

Одним из главных достоинств использования механизма обработки исключительных ситуаций является обеспечение ***развертывания стека***. Развертывание стека – это процесс вызова деструкторов локальных объектов, когда исключительные ситуации выводят их из области видимости.

Сказанное рассмотрим на примере функции add() класса add_class, выполняющей сложение компонентов-данных объектов add_class и возвращающей суммарный объект. В случае если сумма превышает максимальное значение для типа unsigned short, генерируется исключительная ситуация.

```
#include<limits.h>
#include<iostream.h>
class add_class
{ private:
      unsigned short num;
  public:
      add_class(unsigned short a)
       {  num=a;
          cout<<"Constructor "<<num<<endl;
       }
      ~add_class()  { cout<<"Destructor of add_class "<<num<<endl; }
      void show_num()  { cout<<" "<<num<<" ";   }
      void input_num(unsigned short a)  { num=a; }
      unsigned short output_num(){return num;}
};
add_class add(add_class a,add_class b)
{ add_class sum(0);
  unsigned long s=(unsigned long)a.output_num()+
                        (unsigned long)b.output_num();
  if(s>USHRT_MAX) throw (int)1;
  sum.input_num((unsigned short) s);
  return sum;
}
void main()
{ add_class a(USHRT_MAX),b(1),s(0);
  try{
     s=add(a,b);
     cout<<"Result";
     s.show_num();
     cout<<endl;
  }
 catch(int){
     cout<<"Overflow error"<<endl;
  }
}
```

Результат выполнения программы:

```
Constructor 65535
Constructor 1
Constructor 0
Constructor 0
Destructor of add_class 0
Destructor of add_class 65535
```

```
Destructor of add_class 1
Overflow error
Destructor of add_class 0
Destructor of add_class 1
Destructor of add_class 65535
```

Сначала вызываются конструкторы объектов a, b и s, далее происходит передача параметров по значению в функцию (в этом случае происходит вызов конструктора копий, созданного по умолчанию, именно поэтому вызовов деструктора больше, чем конструктора), затем, используя конструктор, создается объект sum. После этого генерируется исключение и срабатывает механизм развертывания стека, то есть вызываются деструкторы локальных объектов sum, a и b. И, наконец, вызываются деструкторы s, b и a.

Рассмотрим более подробно элементы try, catch и throw механизма обработки исключений.

Блок try. Синтаксис блока:

try{
охранный код
}
список обработчиков

Необходимо помнить, что после ключевого слова try всегда должен следовать составной оператор, т.е. после try всегда следует {…}. Блоки try не имеют однострочной формы, как, например, операторы if, while, for.

Еще один важный момент заключается в том, что после блока try должен следовать, по крайней мере, хотя бы один обработчик. Недопустимо нахождение между блоками try и catch какого-либо кода. Например:

```
int i;
try{
     throw исключение;
}
i=0;                    //   'try' block starting on line '  номер  ' has no catch handlers
catch(тип аргумент){
   блок обработки исключения
}
```

В блоке try можно размещать любой код, вызовы локальных функций, функции-компоненты объектов, и любой код любой степени вложенности может генерировать исключительные ситуации. Блоки try сами могут быть вложенными.

Обработчики исключительных ситуаций catch. Обработчики исключительных ситуаций являются важнейшей частью всего механизма обработки исключений, так как именно они определяют поведение программы после генерации и перехвата исключительной ситуации. Синтаксис блока catch имеет следующий вид:

catch(тип 1 <аргумент>)

{
тело обработчика
}
catch(тип 2 <аргумент>))
{
тело обработчика
}
.
.
.
catch(тип N <аргумент>))
{
тело обработчика
}

Таким образом, так же как и в случае блока try, после ключевого слова catch должен следовать составной оператор, заключенный в фигурные скобки. В аргументах обработчика можно указать только тип исключительной ситуации, не обязательно объявлять имя объекта, если этого не требуется.

У каждого блока try может быть множество обработчиков, каждый из которых должен иметь свой уникальный тип исключительной ситуации. Неправильной будет следующая запись:

```
typedef int type_int;
try{
        . . .
}
catch(type_int error1){
        . . .
}
catch(int error2){
        . . .
}
```

Так, в этом случае type_int и int - это одно и то же.

Однако следующий пример верен.

```
class cls
{ public:
    int i;
};

try{
        . . .
}
catch(cls i1){
        ...
```

```
}
catch(int i2){
}
```

В этом случае cls – это отдельный тип исключительной ситуации. Существует также абсолютный обработчик, который совместим с любым типом исключительной ситуации. Для написания такого обработчика надо вместо аргументов написать многоточие (эллипсис).

```
catch (…){
   блок обработки исключения
}
```

Использование абсолютного обработчика исключительных ситуаций рассмотрим на примере программы, в которой происходит генерация исключительной ситуации типа char *, но обработчик такого типа отсутствует. В этом случае управление передается абсолютному обработчику.

```
#include <iostream.h>
#include <conio.h>
void int_exception(int i)
{ if(i>100) throw 1;
}
void string_exception()
{ throw "Error";
}
void main()
{ try{
     int_exception(99);
     string_exception();
  }
  catch(int){
      cout<<"Обработчик для типа Int";
      getch();
  }
 catch(...){
      cout<<"Абсолютный обработчик ";
      getch();
 }
```

Результат выполнения программы:
Абсолютный обработчик

Так как абсолютный обработчик перехватывает исключительные ситуации всех типов, то он должен стоять в списке обработчиков последним. Нарушение этого правила вызовет ошибку при компиляции программы.

Для того чтобы эффективно использовать механизм обработки исключительных ситуаций, необходимо грамотно построить списки обработчиков, а для этого, в свою очередь, нужно четко знать следующие правила, по которым осуществляется поиск соответствующего обработчика:

- исключительная ситуация обрабатывается первым найденным обработчиком, т. е. если есть несколько обработчиков, способных обработать данный тип исключительной ситуации, то она будет обработана первым стоящим в списке обработчиком;
- абсолютный обработчик может обработать любую исключительную ситуацию;
- исключительная ситуация может быть обработана обработчиком соответствующего типа либо обработчиком ссылки на этот тип;
- исключительная ситуация может быть обработана обработчиком базового для нее класса. Например, если класс B является производным от класса A, то обработчик класса A может обработать исключительную ситуацию класса B;
- исключительная ситуация может быть обработана обработчиком, принимающим указатель, если тип исключительной ситуации может быть приведен к типу обработчика, путем использования стандартных правил преобразования типов указателей.

Если при возникновении исключительной ситуации подходящего обработчика нет среди обработчиков данного уровня вложенности блоков try, то обработчик ищется на следующем охватывающем уровне. Если обработчик не найден вплоть до самого верхнего уровня, то программа аварийно завершается.

Следствием из правил 3 и 4 является еще одно утверждение: исключительная ситуация может быть направлена обработчику, который может принимать ссылку на объект базового для данной исключительной ситуации класса. Это значит, что если, например, класс B – производный от класса A, то обработчик ссылки на объект класса A может обрабатывать исключительную ситуацию класса B (или ссылку на объект класса B).

Рассмотрим особенности выбора соответствующего обработчика на следующем примере. Пусть имеется класс C, являющийся производным от классов A и B; показано, какими обработчиками может быть перехвачена исключительная ситуация типа C и типа указателя на C.

```
#include<iostream.h>
class A{};
class B{};
class C : public A, public B {};
void f(int i)
{  if(i) throw C();      // возбуждение исключительной ситуации
```

```
                                // типа объект C
    else throw new C; //  возбуждение исключительной ситуации
                                // типа указатель на объект класса C
}
void main()
{ int i;
  try{
     cin>>i;
     f(i);
  }
  catch(A) {
     cout<<"A handler";
  }
  catch(B&) {
     cout<<"B& handler";
  }
  catch(C) {
     cout<<"C handler";
  }
  catch(C*) {
     cout<<"C* handler";
  }
  catch(A*) {
     cout<<"A* handler";
  }
  catch(void*) {
     cout<<"void* handler";
  }
}
```

В данном примере исключительная ситуация класса C может быть направлена любому из обработчиков A, B& или C, поэтому выбирается обработчик, стоящий первым в списке. Аналогично для исключительной ситуации, имеющей тип указателя на объект класса C, выбирается первый

подходящий обработчик A* или C*. Эта ситуация также может быть обработана обработчиками void*. Так как к типу void* может быть приведен любой указатель, то обработчик этого типа будет перехватывать любые исключительные ситуации типа указателя.

Генерация исключительных ситуаций throw. Исключительные ситуации передаются обработчикам с помощью ключевого слова throw. Как ранее отмечалось, обеспечивается вызов деструкторов локальных объектов при выходе из области видимости, то есть развертывание стека. Однако развертывание стека не обеспечивает уничтожение объектов, созданных динамически. Таким образом, перед генерацией исключительной ситуации необходимо явно освободить динамически выделенные блоки памяти.

Следует отметить также, что если исключительная ситуация генерируется по значению или по ссылке, то создается скрытая временная переменная, в которой хранится копия генерируемого объекта. Когда после throw указывается локальный объект, то к моменту вызова соответствующего обработчика этот объект будет уже вне области видимости и, естественно, прекратит существование. Обработчик же получит в качестве аргумента именно эту скрытую копию. Из этого следует, что если генерируется исключительная ситуация сложного класса, то возникает необходимость снабжения этого класса конструктором копий, который бы обеспечил корректное создание копии объекта.

Если же исключительная ситуация генерируется с использованием указателя, то копия объекта не создается. В этом случае могут возникнуть проблемы. Например, если генерируется указатель на локальный объект, к моменту вызова обработчика объект уже перестанет существовать и использование указателя в обработчике приведет к ошибкам.

Перенаправление исключительных ситуаций

Иногда возникает положение, при котором необходимо обработать исключительную ситуацию сначала на более низком уровне вложенности блока try, а затем передать ее на более высокий уровень для продолжения обработки. Для того чтобы сделать это, нужно использовать throw без аргументов. В этом случае исключительная ситуация будет перенаправлена к следующему подходящему обработчику (подходящий обработчик не ищется ниже в текущем списке - сразу осуществляется поиск на более высоком уровне). Приводимый ниже пример демонстрирует организацию такой передачи. Программа содержит вложенный блок try и соответствующий блок catch. Сначала происходит первичная обработка, затем исключительная ситуация перенаправляется на более высокий уровень для дальнейшей обработки.

```
#include<iostream.h>
void func(int i)
{  try{
      if(i) throw "Error";
   }
```

```
    catch(char *s) {
       cout<<s<<"-  выполняется первый обработчик"<<endl;
       throw;
    }
}
void main()
{  try{
      func(1);
    }
    catch(char *s) {
      cout<<s<<"- выполняется второй обработчик"<<endl;
    }
}
```

Результат выполнения программы:

Error - выполняется первый обработчик
Error - выполняется второй обработчик

Если ключевое слово trow используется вне блока catch, то автоматически будет вызвана функция terminate(), которая по умолчанию завершает программу.

Порядок выполнения работы

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

Варианты заданий.

1. Реализовать базовый класс исключений и многоуровневую иерархию производных классов с использованием наследования, таких как выход индекса за пределы массива, неправильный аргумент у функции, переполение сверху, переполнение снизу, ошибка диапазона, ошибка выделения памяти и др. На основе созданного в л.р. №2 класса String и любых других классов продемонстрировать генерацию исключений. Промоделировать ситуацию, в которой будет важна последовательность вызова обработчиков в иерархии исключений.

2. Напишите функцию, осуществляющую поиск значения в узлах двоичного дерева из char*. Если узел, содержащий слово «здравствуй», найден, функция find(“здравствуй”) возвратит указатель на этот узел. Воспользуйтесь исключением для индикации «не найдено».

3. Определите класс Int, который ведет себя точно также как и встроенный тип int, за исключением того, что он генерирует исключения, не допуская переполнения сверху и снизу.

4. Напишите программу, которая контролирует индексы, выходящие за пределы массива и генерирует исключения. В сообщение об ошибке должна входить информация о значении индекса, приведшего к сбою.

5. Перегрузите операцию + для объединения двух строк. Добавьте класс исключений, генерируйте исключения в конструкторе с одним аргументом в случае, если строка инициализации слишком длинная. Генерируйте еще одно исключение в перегруженном операторе +, если результат конкатенации оказывается слишком длинным. Сообщайте пользователю о том, какая именно ошибка произошла.

Лабораторная работа №7

Организация работы с файлами

Цель работы: Научиться создавать, читать, записывать и обновлять файлы. Овладель обработкой файлов последовательного и произвольного доступа.

Краткие теоретические сведения

В языке C++ для организации работы с файлами используются классы потоков **ifstream** (ввод), **ofstream** (вывод) и **fstream** (ввод и вывод) (рис. 1). Перечисленные классы являются производными от istream, ostream и iostream, соответственно. Операции ввода-вывода выполняются так же, как и для других потоков, то есть компоненты-функции, операции и манипуляторы могут быть применены и к потокам файлов. Различие состоит в том, как создаются объекты и как они привязываются к требуемым файлам.

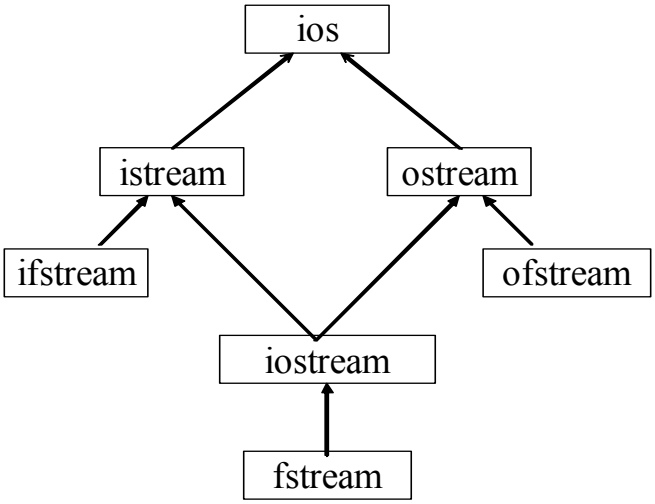


Рис. 1. Часть иерархии классов потоков ввода-вывода

В C++ файл открывается путем стыковки его с соответствующим потоком. Рассмотрим организацию связывания потока с некоторым файлом. Для этого используется конструкторы классов ifstream и ofsream:

ofstream(const char* Name, int nMode= ios::out, int nPot= filebuf::openprot);

ifstream(const char* Name, int nMode= ios::in, int nPot= filebuf::openprot);

Первый аргумент определяет имя файла (единственный обязательный параметр).

Второй аргумент задает режим для открытия файла и представляет битовое ИЛИ величин:

ios::app при записи данные добавляются в конец файла, даже если текущая позиция была перед этим изменена функцией **ostream::seekp**;

ios::ate указатель перемещается в конец файла. Данные записываются в текущую позицию (произвольное место) файла;

ios::in поток создается для ввода: если файл уже существует, то он сохраняется;

ios::out поток создается для вывода (по умолчанию для всех **ofstream** объектов);

ios::trunc если файл уже существует, его содержимое уничтожается (длина равна нулю). Этот режим действует по умолчанию, если **ios::out** установлен, а **ios::ate**, **ios::app** или **ios:in** не установлены;

ios::nocreate если файл не существует, функциональные сбои;

ios::noreplace если файл уже существует, функциональные сбои;

ios::binary ввод-вывод будет выполняться в двоичном виде (по умолчанию текстовой режим).

Третий аргумент – данное класса filebuf, используется для установки атрибутов доступа к открываемому файлу.

Возможные значения *nProt*:

filebuf::sh_compat совместно используют режим;

filebuf::sh_none режим Exclusive: никакое совместное использование;

filebuf::sh_read совместно использующее чтение;

filebuf::sh_write совместно использующее запись.

Для комбинации атрибутов **filebuf::sh_read** and **filebuf::sh_write** используется операция логическое ИЛИ (||).

В отличие от рассмотренного подхода можно создать поток, используя конструктор без аргументов. Позже вызвать функцию open, имеющую те же аргументы, что и конструктор, при этом второй аргумент не может быть задан по умолчанию

void open(const char* name, int mode, int prot=fileout::openprot);

Только после того как поток создан и соединен с некоторым файлом (используя либо конструктор с параметрами, либо функцию open), можно выполнять ввод информации в файл или вывод из файла.

```
#include "iostream.h"
#include "fstream.h"
#include "string.h"


class string
{  char *st;
   int size;
public :
  string(char *ST,int SIZE) : size(SIZE)
  { st=new char[size];
    strcpy(st,ST);
  }
  ~string() {delete [] st;}
  string(const string &s)    // копирующий конструктор (необходим, так как
  { st=new char[s.size];     // при перегрузке << в функцию operator переда-
    strcpy(st,s.st);         // ется объект, содержащий указатель на строку, а
  }                          // в конце вызовется деструктор для объекта obj
  friend ostream &operator<<(ostream &,const string);
  friend istream &operator>>(istream &,string &);
};

ostream &operator<<(ostream &out,const string obj)
{ out << obj.st << endl;
  return out;
}

istream &operator>>(istream &in,string &obj)
{ in >> obj.st;
```

```
  return in;
}
main()
{ string s("asgg",10),ss("aaa",10);
  int state;
  ofstream out("file");
  if (!out)
  { cout<<"ошибка открытия файла"<<endl;
    return 1;      // или exit(1)
  }
  out<<"123"<<endl;
  out<<s<<ss<<endl;      //  запись в файл
  ifstream in("file");
  while(in >> ss)            //  чтение из файла
  {cout<<ss<<endl;}
  in.close();
  out.close();
  return 0;
}
```

В приведенном примере в классе содержится копирующий конструктор, так как в функцию operator передается объект obj, компонентой которого является строка. В инструкции cout <<s<<ss копирующий конструктор вызывается вначале для объекта ss, затем для s, после этого выполняется перегрузка в порядке, показанном ранее. При завершении каждой из функций operator (вначале для s, затем для ss) будет вызван деструктор.

В операторе if (!out) вызывается (как и ранее для потоков) функция ios::operator! для определения, успешно ли открылся файл.

Условие в заголовке оператора while автоматически вызывает функцию класса ios перегрузки операции void *:

```
operator void *() const { if(state&(badbit|failbit) ) return 0; return (void *)this;
}
```

то есть выполняется проверка; если для потока устанавливается failbit или badbit, то возвращается 0.

Организация файла последовательного доступа

В C++ файлу не предписывается никакая структура. Для последовательного поиска данных в файле программа обычно начинает считывать данные с начала файла до тех пор, пока не будут считаны требуемые данные. При поиске новых данных этот процесс вновь повторяется.

Данные, содержащиеся в файле последовательного доступа, не могут быть модифицированы без риска разрушения других данных в этом файле. Например, если в файле содержится информация:

Коля 12 Александр 52

то при модификации имени Коля на имя Николай может получиться следующее:

НиколайАлександр 52

Аналогично в последовательности целых чисел 12 –1 132 32554 7 для хранения каждого из них отводится sizeof(int) байт. А при форматированном выводе их в файл они занимают различное число байт. Следовательно, такая модель ввода-вывода неприменима для модификации информации на месте. Эта проблема может быть решена перезаписью (с одновременной модификацией) в новый файл информации из старого. Это сопряжено с проблемой обработки всей информации при модификации только одной записи.

Следующая программа выполняет перезапись информации из одного файла в два других, при этом в первый файл из исходного переписываются только числа, а во второй вся остальная информация.

```
#include "fstream.h"
#include "stdlib.h"
#include "math.h"

void error(char *s1,char *s2="")  // вывод сообщения об ошибке
{ cerr<<s1<<"  "<<s2<<endl;      // при открытии потока для файла
  exit(1);
}

main(int argc,char **argv)
{ char *buf=new char[20];
  int i;
  ifstream f1;                    // входной поток
  ofstream f2,f3;                 // выходные потоки
  f1.open(argv[1],ios::in);       // открытие исходного файла
  if(!f1)                         // проверка состояния потока
  error("ошибка открытия файла",argv[1]);
  f2.open(argv[2],ios::out);     // открытие 1 выходного файла
  if(!f2) error("ошибка открытия файла",argv[1]);
  f3.open(argv[3],ios::out);     // открытие 2 выходного файла
  if(!f3) error("ошибка открытия файла",argv[1]);
  f1.seekg(0);                  // установить текущий указатель в начало потока
  while(f1.getline(buf,20,' '))  // считывание в буфер до 20 символов
  { if(int n=f1.gcount())        // число реально считанных символов
    buf[n-1]='\0';
                                 // проверка на только цифровую строку
    for(i=0;*(buf+i)&&(*(buf+i)>='0' && *(buf+i)<='9');i++);
    if(!*(buf+i)) f2 <<::atoi(buf)<<' ';     // преобразование в число и запись
                                             // в файл f2
```

```
    else f3<<buf<<' ';                    // просто выгрузка буфера в файл f3
   }
  delete [] buf;
  f1.close();                    // закрытие файлов
  f2.close();
  f3.close();
}
```

В программе для ввода имен файлов использована командная строка, первый параметр – имя файла источника (входного), а два других – имена файлов приемников (выходных). Для работы с файлами использованы функции – open, close, seekg, getline и gcount. Более подробное описание функций приведено ниже.

Ниже приведена программа, выполняющая ввод символов в файл с их одновременным упорядочиванием (при вводе) по алфавиту. Использование функций seekg, tellg позволяет позиционировать текущую позицию в файле, то есть осуществлять прямой доступ в файл. Обмен информацией с файлом осуществляется посредством функций get и put.

```
#include "fstream.h"
#include "stdlib.h"

void error(char *s1,char *s2="")
{ cerr<<s1<<"  "<<s2<<endl;
  exit(1);
}

main()
{ char c,cc;
   int n;
   fstream f;                    // выходной поток
   streampos p,pp;
   f.open("aaaa",ios::in|ios::out); // открытие выходного файла
   if(!f) error("ошибка открытия файла","aaaa");
   f.seekp(0);                    // установить текущий указатель в начало потока
   while(1)
   { cin>>c;
     if (c=='q' || f.bad()) break;
     f.seekg(0,ios::beg);
     while(1)
     { if(((cc=f.get())>=c) || (f.eof()))
        { if(f.eof())
          { f.clear(0);
            p=f.tellg();
          }
          else
```

```
            { p=f.tellg()-1;
              f.seekg(-1,ios::end);
              pp=f.tellg();
              while(p<=pp)
              { cc=f.get();
                f.put(cc);
                f.seekg(--pp);
              }
            }
           f.seekg(p);
           f.put(c);
           break;
        }
     }
     }
     f.close();
     return 1;
    }
```

Каждому объекту класса istream соответствует указатель get (указывающий на очередной вводимый из потока байт) и указатель put (соответственно на позицию для вывода байта в поток). Классы istream и ostream содержат по 2 перегруженных компонента-функции для перемещения указателя в требуемую позицию в потоке (связанном с ним файле). Такими функциями являются seekg (переместить указатель для извлечения из потока) и seekp (переместить указатель для помещения в поток)

istream& seekg(streampos pos);
istream& seekg(streamoff off, ios::seek_dir dir);

Сказанное выше справедливо и для функций seekp. Первая функция перемещает указатель в позицию pos относительно начала потока. Вторая перемещает соответствующий указатель на off (целое число) байт в трех возможных направлениях: ios::beg (от начала потока), ios::cur (от текущей позиции) и ios::end (от конца потока). Кроме рассмотренных функций в этих классах имеются еще функции tellg и tellp, возвращающие текущее положение указателей get и put соответственно

streampos tellg();

Создание файла произвольного доступа

Организация хранения информации в файле прямого доступа предполагает доступ к ней не последовательно от начала файла по некоторому ключу, а непосредственно, например, по их порядковому номеру. Для этого требуется, чтобы все записи в файле были бы одинаковой длины.

Наиболее удобными для организации произвольного доступа при вводе-выводе информации являются следующие компоненты-функции:

istream& istream::read(E *s, streamsize n);
ostream& ostream::write(E *s, streamsize n);

при этом, так как функция write (read) ожидает первый аргумент типа const char * (char *), то для требуемого приведения типов используется оператор явного преобразования типов:

istream& istream::read(reinterpret_cast<char *>(&s), streamsize n);
ostream& ostream::write(reinterpret_cast<const char *>(&s),
streamsize n);

Ниже приведен текст программы организации работы с файлом произвольного доступа на примере удаления и добавления в файл информации о банковских реквизитах клиента (структура inf).

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

struct inf
{ char cl[10];    // клиент
  int pk;          // пин-код
  double sm;      // сумма на счете
} cldata;

class File
{  char filename[80];       // имя файла
   fstream *fstr;            // указатель на поток
   int maxpos;               // число записей в файле
 public:
   File(char *filename);
  ~File();
   int Open();
   const char* GetName();
   int Read(inf &);
   void Remote();
   void Add(inf);
   int Del(int pos);
   friend ostream& operator << (ostream &out, File &obj)
   { inf p;
     out << "File " << obj.GetName() << endl;
     obj.Remote();
     while(obj.Read(p))
     out<<"\nКлиент -> "<<p.cl<<' '<<p.pk<<' '<<p.sm;
     return out;
   }
```

```
};
File::File(char *_filename)             // конструктор
{  strncpy(filename,_filename,80);
   fstr = new fstream();
}
File::~File()                           // деструктор
{  fstr->close(); }
int File::Open()   // функция открывает файл для ввода-вывода бинарный
{  fstr->open(filename, ios::in | ios::out | ios::binary);
   if (!fstr->is_open()) return -1;
   return 0;
}
int File::Read(inf &p)     // функция чтения из потока fstr в объект   p
{  if(!fstr->eof() &&              // если не конец файла
   fstr->read(reinterpret_cast<char*>(&p),sizeof(inf)))
   return 1;                       //
   fstr->clear();
   return 0;
}
void File::Remote()      // сдвиг указателей get и put в начало потока
{  fstr->seekg(0,ios_base::beg); // сдвиг указателя на get на начало
   fstr->seekp(0,ios_base::beg); // сдвиг указателя на put на начало
   fstr->clear();                // чистка состояния потока
}
const char* File::GetName()      // функция возвращает имя файла
{  return this->filename;  }
void File::Add(inf cldata)
{  fstr->seekp(0,ios_base::end);
  fstr->write(reinterpret_cast<char*>(&cldata),sizeof(inf));
  fstr->flush();
}
int File::Del(int pos) // функция удаления из потока записи с номером pos
{  Remote();
  fstr->seekp(0,ios::end);  // для вычисления maxpos
  maxpos = fstr->tellp();   // позиция указателя put
  maxpos/=sizeof(inf);
  if(maxpos<pos) return -1;

  fstr->seekg(pos*sizeof(inf),ios::beg); // сдвиг на позицию
                                         // следующую за pos
  while(pos<maxpos)
```

```
    { fstr->read(reinterpret_cast<char*>(&cldata),sizeof(inf));
      fstr->seekp(-2*sizeof(inf), ios::cur);
      fstr->write(reinterpret_cast<char*>(&cldata),sizeof(inf));
      fstr->seekg(sizeof(inf),ios::cur);
      pos++;
    }
    strcpy(cldata.cl,"");  // для занесения пустой записи в
    cldata.pk=0;           // конец файла
    cldata.sm=0;
    fstr->seekp(-sizeof(inf), ios::end);
    fstr->write(reinterpret_cast<char*>(&cldata),sizeof(inf));
    fstr->flush();         // выгрузка выходного потока в файл
}
int main(void)
{ int n;
    File myfile("file");
    if(myfile.Open() == -1)
    { cout << "Can't open the file\n";
      return -1;
    }
    cin>>cldata.cl>>cldata.pk>>cldata.sm;
    myfile.Add(cldata);
    cout << myfile << endl;      // просмотр файла
    cout << "Введите номер клиента для удаления ";
    cin>>n;
    if(myfile.Del(n) == -1)
    cout << "Клиент с номеном "<<n<<" вне файла\n";
    cout << myfile << endl;      // просмотр файла
}
```

Порядок выполнения работы

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

Варианты заданий

1. Вы являетесь владельцем склада металлических изделий и нуждаетесь в инвентаризации, которая сказала бы вам, сколько всего различных изделий вы имеете, какое количество каждого из них у вас на руках и стоимость каждого из них. Напишите программу, которая бы создавала файл произвольного доступа,

позволяла бы вводить данные по каждому изделию, давала бы вам возможность получать список всех изделий, удалять записи по изделиям, которых у вас уже нет, и позволяла бы обновлять любую информацию в файле. Ключом должен быть идентификационный номер изделия.

2. Имеются данные о пользователе, состоящие из имени, отчества, фамилии и номера работника. Осуществите форматированный вывод в объект ofstream с помощью оператора вставки << . Когда пользователь сообщит об окончании ввода, закройте объект ofstream, откройте объект ifstream, прочитайте и выведите на экран все данные из файла.

3. Создайте класс name, включающий в себя данные (имя, фамилия, отчество и номер работника). Создайте методы этого класса, ощуществляющие файловый ввод/вывод данных указанного класса (с использованием ofstream и ifstream). Используйте форматирование данных (операторы << и >>). Функции чтения и записи должны быть независимыми: в них необходимо внести выражения для открытия соответствующего потока, а также чтения и записи. Функция записи может просто добавлять записи в конец файла. Функции чтения потребуется некоторое условие выборки.

4. Реализуйте класс, для которого оператор [] перегружен, чтобы выполнять чтение символов из указанной позиции файла.

5. Реализуйте класс как в задаче 4, но пусть оператор [] индексирует объекты произвольного типа, а не только символы.

Литература

1. Бьерн Страуструп. Язык программирования C++. Пер. с англ.- М.: «Издательство БИНОМ», 2004. – 1098 с.
2. Скляров В.А. Язык C++ и объектно-ориентированное программирование. Мн.: Выш.шк., 1997г. – 478 с.: ил.
3. Дейтел Х., Дейтел П. Как программировать на C++. Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2001. – 1152 с.: ил.
4. Лафоре Р. Объектно-ориентированное программирование в C++.Питер, 2003. – 923 с.
5. Луцик Ю.А., Ковальчук А.М., Лукьянова И.В. Объектно-ориентированное программирование на языке C++. Мн.: БГУИР, 2003.- 203 с.: ил.