МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра механики, мехатроники и робототехники

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В МЕХАТРОНИКЕ

Методические указания к проведению практических занятий по дисциплине «Объектно-ориентированное программирование в мехатронике» по направлению 221000.62 - «Мехатроника и робототехника»

Курск 2015

УДК 004.432

Составители: С.Ф. Яцун, П.А. Безмен

Рецензент
Кандидат технических наук, доцент кафедры механики, мехатроники
и робототехники
В.Я. Мищенко

Объектно-ориентированное программирование в мехатронике: методические указания к проведению практических занятий по дисциплине «Объектно-ориентированное программирование в мехатронике» по направлению 221000.62 - «Мехатроника и робототехника» / Юго-Зап. гос. ун-т; сост.: С.Ф. Яцун, П.А. Безмен; Курск, 2015. 40 с. Библиогр.: с. 40.

Содержат сведения по принципам описания классов на языке C++: управление доступом к членам класса, описание конструкторов и деструкторов, дружественные функции, перегрузка операторов, шаблоны класса, наследование, обработка исключительных ситуаций.

Методические указания соответствуют требованиям программы, утверждённой учебно-методическим объединением (УМО).

Предназначены для студентов направления 221000.62 - «Мехатроника и робототехника» всех форм обучения.

Текст печатается в авторской редакции

Подписано в печать . Формат 60x84 1/16.
Усл. печ. л. . Уч.-изд. л. .Тираж 100 экз. Заказ. Бесплатно.
Юго-Западный государственный университет.
305040, Курск, ул. 50 лет Октября, 94.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4

1. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ 6

2.ПРИМЕРЫ АБСТРАКТНЫХ ТИПОВ ДАННЫХ 9

3. ОБЗОР СРЕДСТВ ОПИСАНИЯ КЛАССОВ 11

3.1. Классы и члены 11

3.2. Конструктор и деструктор 12

3.3. Встроенные (inline) функции 14

3.4. Класс vector 17

3.5. Перегрузка операторов 19

3.6. Указатель this 21

3.7.Дружественные функции-операции 22

3.8. Класс очередь que. Шаблон класса 22

3.9. Наследование. Производные классы 24

4. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ 27

4.1. Необходимость обработки исключений 27

4.2. Общие принципы обработки исключительных ситуаций в C++ 29

4.3. Возбуждение ситуации 30

4.4. Обработка исключений 32

5. РЕАЛИЗАЦИЯ ПРИМЕРОВ 34

5.1. Класс полином 34

5.2. Класс таблица 36

6. ЗАДАНИЯ ДЛЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ 39

ЛИТЕРАТУРА 41

ВВЕДЕНИЕ

Язык С++ предоставляет гибкие и эффективные средства определения новых типов. Используя определения новых типов, отвечающих понятиям проблемной области (приложениям), программист может разбивать разрабатываемую программу на части, легко поддающиеся контролю.

Новый тип создается пользователем для определения понятия, которому среди встроенных типов нет соответствия. Гораздо легче понять программу, в которой создаются типы, отвечающие понятиям проблемной области. Хорошо определенные типы делают программу ясной и короткой. Кроме того, компилятор может обнаружить недопустимое использование данных.

Такой метод построения программ называют абстракцией данных. Информация о типах содержится в объектах, тип которых определяется пользователем. Программирование с применением объектов называют объектно-ориентированным [1,2].

Язык С++ является языком объектно-ориентированного программирования (ООП). Автор языка создавал его с целью поддержки абстракции данных и объектно-ориентированного программирования. Родоначальниками языков такого типа являются языки Smalltalk и Simula 67.

Объектно-ориентированный язык – это язык программирования, на котором программа задается описанием поведения совокупности взаимосвязанных объектов. Объектно-ориентированное программирование имеет дело с объектами.

Объектно-ориентированные языки включают следующие основные черты: инкапсуляция данных, полиморфизм, наследование.

Объекты могут включать закрытые (private) данные и правила их обработки, доступные только объекту, защищенные (protected), доступные объекту и его наследникам, а также общие (public) данные и правила, которые доступны объектам и модулям в других частях программы. Важной чертой ООП является наследование, то есть возможность создавать иерархическую последовательность объектов от более общих к более специфическим объектам.

Методические указания содержат четыре основных раздела.

В первом разделе обсуждается взаимосвязь абстрактных типов данных и объектно-ориентированного программирования.

Во втором разделе приводится список примеров, в которых вводится понятие, являющееся абстракцией.

В третьем разделе дается краткий обзор средств языка C++, необходимых для реализации абстрактного понятия, причем вводимые средства показываются на конкретных примерах. Для определения нового (пользовательского) типа вводится понятие класс. С понятием класс непосредственно связаны понятия конструктор и деструктор. В этом же разделе рассматриваются простейшие средства для реализации производных классов.

Четвертый раздел посвящен обработке исключительных ситуаций, возникающих в процессе работы программ.

Пятый раздел представляет собой реализацию конкретных абстрактных понятий, а именно, реализацию полинома и таблицы.

В шестом разделе представлены задания для практических занятий.

1. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В языках высокого уровня можно выделить следующие категории средств определения данных:

– встроенные типы данных (integer, real, character, boolean);

– средства структурирования составных объектов (array, record, union);

– средства определения новых типов данных – это абстракция данных;

– средства отображения иерархии типов данных.

Понятие абстрактного типа [1] пришло в программирование из математики. Абстрактный объект – это то, что является предметом математического рассуждения, состоящего из определений, допущений (или постулатов) и утверждений, выводимых из определений и допущений по общепонятным правилам логического вывода (например, хорошо знакомые математикам: множества, функции, последовательности и рекурсивные структуры).

Организация данных в языках программирования базируется на понятии *типа*. К особенностям понятия типа следует отнести следующие:

1) тип определяет класс значений, которые могут принимать переменная или выражение;

2) каждое значение принадлежит одному и только одному типу;

3) тип значения константы, переменной или выражения можно вывести либо из контекста, либо из вида самого операнда, не обращаясь к значениям, вычисляемым во время работы программы;

4) каждой операции соответствует некоторый фиксированный тип ее операндов и некоторый фиксированный тип результата;

5) для каждого типа свойства значений и элементарных операций над значениями задаются с помощью аксиом;

6) при работе с языком высокого уровня знание типа позволяет обнаруживать в программе бессмысленные конструкции и решать вопрос о методе представления данных и преобразования их в вычислительной машине.

Абстракция – это инструмент познавательной деятельности человека, позволяющий лучше отразить суть дела и приводящий к абстрактным понятиям. Поэтому имеет смысл говорить о средствах абстракции в языках программирования, которые сами являются средством понимания и построения алгоритмов и обмена мыслями и результатами между программистами.

Абстрактный тип данных в языках программирования – это определение некоторого понятия в виде класса объектов с некоторыми свойствами и операциями. Такое определение оформляется как специальная синтаксическая конструкция, которая называется *классом* в языке C++.

В определение абстрактного типа данных входят следующие четыре части.

1) Внешность, содержащая имя определяемого типа (понятия), имена операций с указанием типов их аргументов и значений.

2) Абстрактное описание операций и объектов, с которыми они работают, средствами некоторого языка спецификаций.

3) Конкретное описание операций и объектов средствами языка программирования.

4) Описание связи между (2) и (3), объясняющее, в каком смысле часть (3) корректно представляет часть (2).

Внешность – это видимая часть определения, его интерфейс. Отметим, что спецификация определяет то, что важно для пользователя. А для пользо-

вателя существенным является «поведение», то есть «то, что делается», а несущественным – «то, как это делается» [1].

Главное достоинство абстракции через спецификацию состоит в несущественности способа реализации, что позволяет изменять реализацию без внесения изменений в программу.

Основные особенности средства абстракции данных:

1) инкапсуляция описания и представления объектов определяемого типа и описания операций над объектами;

2) защита инкапсулированной информации, так что детали представления не доступны вне определения абстрактного типа.

Средства абстракции данных в языке C++ включают классы, механизм управления доступом, конструкторы и деструкторы, совместное использование операций (перегрузка операций), преобразование типов, полиморфизм, обработку исключительных ситуаций.

Средства отображения иерархии классов непосредственно связаны с объектно-ориентированным программированием, для поддержки которого в языке C++ введены производные классы и виртуальные функции.

2.ПРИМЕРЫ АБСТРАКТНЫХ ТИПОВ ДАННЫХ

Спецификация абстрактных типов данных может быть выполнена на специальном формализованном языке [1]. Приведем примеры абстрактных типов данных, используя неформальный язык.

Пример 1. Опишем понятие стек (stack). Стек представляет собой последовательность элементов некоторого типа, которая характеризуется свойством: "последний вошел – первый вышел" (Last In - First Out) и над которой выполняются следующие операции:

– создать стек;

– положить в стек;

– проверить, пуст ли стек;

– проверить, переполнен ли стек;

– взять элемент, находящийся в вершине стека, удаляя его из стека;

– показать элемент, находящийся в вершине стека, т.е. взять элемент, находящийся в вершине стека, не удаляя его из стека.

Пример 2. Опишем понятие очередь (Query). Очередь представляет собой последовательность элементов некоторого типа, которая характеризуется свойством: "первый вошел – первый вышел" (First In - First Out) и над которой выполняются следующие операции:

– создать очередь;

– послать в очередь;

– проверить, пуста ли очередь;

– проверить, переполнена ли очередь;

– взять элемент.

Пример 3. Опишем понятие вектор (vector). Вектор представляет собой последовательность из n элементов со следующими операциями:

– создать вектор с заданным количеством элементов;

– добавить вектор;

– вычесть вектор;

– присвоить вектор вектору;

– показать вектор.

Пример 4. Опишем понятие полином (polynom). Полином – это алгебраическое выражение вида

$$y = a_0 \cdot x^n + a_1 \cdot x^{n-1} + \ldots + a_{n-1} \cdot x + a_n$$

Это значит, что полином определяется степенью n и последовательностью из n+1 коэффициентов. Над полиномом предусматриваются следующие операции:

– создать полином по заданной степени и коэффициентам;

– прибавить заданный полином;

– вычесть заданный полином;

– взять наибольшую степень полинома;

– взять коэффициент полинома при заданной степени;

– вычислить полином по заданному значению аргумента;

– выдать полином.

Пример 5. Опишем понятие таблицы. Таблица – это последовательность элементов заданной структуры со следующими операциями:

– создать таблицу с заданным количеством элементов;

– выбрать элементы таблицы по условию;

– сортировать таблицу по различным полям;

– показать таблицу.

3. ОБЗОР СРЕДСТВ ОПИСАНИЯ КЛАССОВ

В определении нового типа главное – отделить внешность, которая существенна для его правильного использования, от конкретной реализации, связанной с форматом данных для хранения объекта типа и операций над объектом. Доступ к таким типам ограничен заданным множеством операций (функциями доступа).

Средством определения нового типа в C++ является понятие класса.

3.1. Классы и члены

Класс – это определяемый пользователем тип. Класс специфицирует данные, необходимые для представления объекта этого типа, и множество операций для работы с объектами.

Определение класса имеет две части: закрытую часть (private), содержащую информацию для разработчика, и открытую часть (public), являющуюся интерфейсом для пользователя.

Доступ к объектам класса ограничивается множеством функций, которые описываются в интерфейсной части класса. Такие функции называются *функциями – членами***.** Пользователь может обращаться только к этим функциям.

Опишем тип стек (stack) (пример 1 в разделе 2). Стек будем располагать в динамической памяти. Поэтому элемент стека – это структура, состоящая из информационной части и ссылки на следующий элемент.

```
class  STACK
{  private:
        struct sp
```

```
            {  int inf;
               struct sp *next;
            };
     sp *s;
    public:
CREATE();           //  создать стек
int top();          //  показать элемент
push(int x);        //  добавить элемент
int  pop();         //  взять элемент
int empty();        //  проверить, пуст ли стек
int full();         //  проверить, переполнен  ли стек
};
```

Для того чтобы пользоваться объектом стек, его нужно описать:

```
STACK  st;
```

Объект st создаётся и инициализируется функцией-членом CREATE(), специально описанной для этой цели. При этом пользователь должен не забыть вызвать функцию CREATE():

```
st.CREATE();
```

Но в языке C++ есть более удобный подход, а именно, определяется специальная функция, предназначенная для инициализации объекта. Эта функции называется *конструктором*.

3.2. Конструктор и деструктор

Конструктор – это предписание, как создавать значение данного типа. Конструктор распознается по тому, что имеет то же имя, что и сам класс. Поэтому в примере роль операции создать стек CREATE() выполняет специальная функция STACK(). Функция вызывается автоматически при объявлении объекта типа STACK.

Определяемый пользователем тип всегда имеет конструктор, так как необходимы выделение памяти и инициализация объектов.

Для обеспечения уничтожения объектов используется специальная функция класса, называемая *деструктором*. Когда закончена работа с объектом, автоматически вызывается деструктор.

Деструктор имеет такое же имя, как и класс, только перед ним ставится знак тильды (~).

Для класса STACK деструктором является ~STACK() .

Опишем объект стек с использованием конструктора и деструктора, а также выполним реализацию функций-членов.

Вариант 1.

```
class  STACK
{  private:
     struct sp
        {   int inf;
            struct sp *next;
        };
     sp *s;
    public:
       STACK()  { s=0;}
      ~ STACK()
        { while (s!=0) delete s;}
       int top()
        { return s->inf;
        }

       push(int x)
        { sp *p;
          p=new sp;  p->inf=x; p->next=s;
          s=p;
         }

       int  pop()
       { sp *p;   p=s;
         int r=p->inf; s=p->next;
         delete p;
         return r;
       }
      int empty()
```

```
        { return s==0;
        }
       int full()
        {  sp *p;
           p=new sp;
           if (p!=0)
              {  delete p; return 1; }
             else return 0;
        }
};
```

3.3. Встроенные (inline) функции

Функции-члены, реализованные в классе, являются встроенными (inline). Это значит, что после компиляции в объектном коде программы пользователя вместо вызовов функций-членов будут подставлены уже преобразованные тела функций, т.е. произойдёт макроподстановка. Другими словами нет затрат на вызов функции.

Удобно реализацию функций-членов выполнять вне класса. Но в этом случае описатель inline должен быть указан явно. Следует отметить, что описатель inline эффективно использовать для коротких операций (таких, как в нашем случае со стеком).

Для того чтобы вне класса был доступ к членам класса, используется операция :: разрешения области видимости. Операнд в левой части операции :: должен быть именем класса.

Вариант 2.

```
class  STACK
{  private:
     struct sp
       { int inf;
         struct sp *next;
       };
     sp *s;
```

```
    public:
  STACK()  { s=0;}
  ~STACK()
    { if (s!=0) delete s;}
  int top();
  push(int x);
  int  pop();
  int empty();
  int full();
};

inline int STACK :: top()
 { return s->inf;
 }

STACK :: push(int x)
 { sp *p;
   p=new sp;  p->inf=x; p->next=s;
   s=p;
 }

inline int STACK :: pop()
{ sp *p;
  p=s;
  int r=p->inf;s=p->next;
  delete p;
  return r;
}

inline int STACK :: empty()
{ return s == 0;
};

inline int STACK :: full()
 {  sp *p;
    p=new sp;
    if (p!=0)
       {  delete p; return 1; }
    else return 0;
  };
```

Конструктор и деструктор тоже можно определять вне класса.

Следует отметить, что конструктор не может иметь тип возвращаемого значения. Нельзя вызывать функцию–конструктор в явном виде.

Приведем программу, в которой используется класс стек для решения следующей задачи: проверить правильность расстановки круглых скобок в арифметическом выражении.

Алгоритм анализа скобок заключается в следующем:

– если встретилась левая скобка, то она заносится в стек;

– если встретилась правая скобка, то она выталкивает из стека левую скобку в случае, если стек оказался не пуст, иначе устанавливается ошибочная ситуация;

– если по окончании просмотра арифметического выражения стек пуст, то проверка прошла успешно, иначе устанавливается ошибочная ситуация.

Будем предполагать, что класс STACK находится в заголовочном файле stack.h. Описание этого класса соответствует варианту 1 или варианту 2 с той лишь разницей, что информация в стеке будет символьного типа.

Программа:

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include “stack.h”
void main()
{ STACK  a;
 char x;
 int n, i, B;
 char *s;
 cout << "\n vvod  str";
 cin >> s;
 n=strlen(s);
 i=0;  B=1;
while (B && (i<n))
```

```
    {  if (s[i]=='(')  a.push(s[i]);
       if (s[i]==')')
            if (a.empty()) B=0; else x=a.pop();
       i++;
    }
 if  ((B) && (i=n) && (a.empty()) )   cout <<"\n yes";
   else cout <<"\n no";
  getch();
}
```

3.4. Класс vector

Конструктор аналогично другим функциям может иметь параметры. Кроме того, класс может иметь несколько конструкторов.

Рассмотрим класс vector с точки зрения использования различных конструкторов.

Когда объект создается, бывает необходимо присваивать элементам объекта конкретные значения. Это достигается передачей значений в качестве параметров. В классе STACK использовался конструктор без параметра. Конструктор без параметра называется конструктором по умолчанию, конструктор с параметром называется конструктором инициализации. Конструктор с параметром, который имеет тип класса называется копирующим конструктором (*copy–конструктор*). В классе vector используются конструкторы с одним параметром, двумя параметрами и *copy– конструктор*.

```
class vector
  { private:
      int *v;   int s;
    public:
      vector(int s1)
        { s=s1;   v=new int[s];  }
      vector(int _n,int *n);
      vector(const vector &y);
```

```
      ~vector()
          { delete  [ ]v; };
      void   vv_vect();
      void show();
   };

vector::vector(int s1,int *n)
{ s=s1;
  v=new  int[s];
  for(int i=0;i<s;i++)
       v[i]=n[i];
};

vector::vector(const vector &y)
{ s=y.s;
  v=new  int[s];
  for(int i=0;i<s;i++)
   v[i]=y.v[i];
}

void vector::vv_vect()
{ cout<<"\nVvod vectora\n";
  for (int i=0;i<s;i++)
      cin>>v[i];
  cout<<"\nVector vveden\n";
};

void vector::show()
{  for (int i=0;i<s;i++)
          cout<<v[i]<<" ";
};
```

Приведём примеры описания объекта класса vector.

Описание: vector a(n);

К моменту этого описания значение n должно быть определено. Автоматически будет вызван конструктор

vector(int s1);

и вектору a будет выделена память для n значений целого типа.

Описание: vector d(m,y);

Автоматически будет вызван конструктор

vector(int s1,int *n)

и в этом случае вектору d будет выделена память для m значений целого типа и скопированы n значений из области y.

Описание: vector c(a);

Автоматически будет вызван *copy–конструктор*

vector(vector &y)

и элементам вектора c будут присвоены значения элементов вектора a.

Приведем фрагменты использования класса vector.

```
int n,m;
int y[]={6,3,6,5,9};
cout <<"\n  введите  n,m " <<"  ";
cin >> n >> m;
vector  a(n);
a.vv_vect();
a.show();


vector d(m,y);
d.show();
vector c(a);
a.show();
```

3.5. Перегрузка операторов

Определим в классе vector операторы (операции): добавление вектора (символ +), вычитание вектора (символ -) и присваивание вектора (символ =).

```
class vector
    { private:
        int *v;int s;
      public:
```

```
        vector(int s1)
          { s=s1;   v=new int[s];
          }
        vector(int _n,int *n);
        vector( vector &y);
        ~vector()
          { delete []v; };
        void vv_vect();
        void show();
        vector operator+(vector n);
        vector operator-(vector n);
        vector operator=(vector n);

   };
```

Вне класса определим реализацию введённых операторов.

```
vector vector::operator+(vector n)
{
  vector t(s);
  for (int i=0;i<s;i++) t.v[i]=v[i]+n.v[i];
  return t;
};



vector vector::operator-(vector n)
{
  vector t(s);
  for (int i=0;i<s;i++) t.v[i]=v[i]-n.v[i];
  return t;
};

vector vector::operator=(vector n)
{
  if (n.s<s)  s=n.s;
  for (int i=0;i<s;i++) v[i]=n.v[i];
  return *this;
};
```

Приведём фрагмент программы, использующий описанные операторы.

```
a=a-d;
cout<<"\n vect a=a-d  "<<" " ;  a.show();
a=a+d;
cout<<"\n vect a=a+d  "<<" " ;  a.show();
```

Возможно, что для объекта vector естественней было бы использовать вызовы функций, а именно, для операции вычесть a.minus(d), добавить a.add(d).

3.6. Указатель this

Каждой функции–члену передается указатель на объект, для которого она вызвана. Таким указателем является служебное слово this. Доступ к объекту внутри функции – члена производится с помощью этого указателя this.

Служебное слово this используется при перегрузке операций.

Операции сложения и вычитания реализуем, используя указатель this:

```
vector vector :: operator+(vector n)
{ if (n.s<s)  s=n.s;
  for (int i=0;i<s;i++) v[i]= v[i]+n.v[i];
  return *this;
};
vector vector :: operator-(vector n)
{
 if (n.s<s)  s=n.s;
 for (int i=0;i<s;i++) v[i]=v[i]-n.v[i];
 return *this;
};
```

3.7.Дружественные функции-операции

Функции–операции могут быть членами класса или дружественными функциями класса. При объявлении дружественной функции должны передаваться два параметра для бинарных операций и один для унарных операций. Дружественными функциями не могут перегружаться операции присваивания =, индексирования [ ], а также () и –>.

Рассмотрим перегрузку операции сложения + для класса vector с помощью дружественной функции. Описание в классе будет иметь вид:

```
friend vector operator + (vector v1, vector v2);
```

Реализация этой операции вне класса:

```
vector operator + (vector v1, vector v2)
 { int s;
   if  (v1.s<v2.s)   s=v1.s ;   else s=v2.s;
    vector t(s);
    for (int i=0;i<s;i++) t.v[i]=v1.v[i]+v2.v[i];
    return t;
 }
```

3.8. Класс очередь que. Шаблон класса

Средством реализации полиморфизма в языке С++ является шаблон класса. На примере объекта очередь покажем использование шаблона класса: template <class T>.

Это описание указывает на то, что описывается параметризованный тип, имеющий параметром тип Т.

```
template <class T>
```

```
class QUE
{  private:
   struct link
   { T inf;
     link *next;
   };
   link *start,*end;
    public:
   QUE ();
   ~QUE ();
   void add(T elem);
   T get();
   int empty();
};
```

Приведем реализацию введенных операций.

```
template <class T>
QUE <T>:: QUE ()
{  start=end=0;
};
template <class T>
void  QUE <T> :: add(T elem)
{  link *p;
   p=new link;
   p->inf=elem;
   p->next=0;
   if (start==0 && end==0) start=p;
    else end->next=p;
   end=p;
};
template <class T>
T  QUE <T> :: get()
{  link *p;
   T buf;
   p=start;
   buf=p->inf;
   start=start->next;
   delete p;
   return buf;
};
template <class T>
QUE <T>::~ QUE ()
```

```
{  link *p;
   while (start!=0)
      {
       p=start;
       start=start->next;
       delete p;
      };
};
```

Приведем фрагменты программы, использующие класс очередь.

```
QUE <int> ocher;
int x;
scanf("%d",&x);
ocher.add(x);
printf("%d\n",ocher.get());
……
QUE <float> ocher1;
int m;
float y;
scanf("%f",&y);
ocher1.add(y);
printf("%f\n",ocher1.get());
```

Аналогично можно описать параметризованный класс стек или класс вектор.

3.9. Наследование. Производные классы

Наследование является важной особенностью объектно-ориентированных языков. Для того чтобы отобразить иерархические связи, выражающие общность между классами, вводится понятие производного класса.

В языке C++ наследуемый класс называют базовым классом, наследующий класс – производным классом. Производный класс наследует свой-

ства базового класса. Поэтому отношение базовый–производный между классами называется наследованием.

Покажем наследование классов на примере объекта очередь. Пусть нам нужно над последовательностью элементов очереди выполнить дополнительно следующую операцию: определить, есть ли среди элементов нулевые. Основные операции над объектом определены в классе QUE. Это операции: добавить элемент

```
void add(T elem);
```

и взять элемент

```
T get();
```

Для простоты опишем класс очередь QUERY без использования шаблона класса.

```
class QUERY
{  private:
   struct link
   { int inf;
     link *next;
   };
   link *start,*end;
   public:
   QUERY ();
   ~QUERY ();
   void add(int elem);
   int get();
   int empty();
};
```

Для реализации новой операции можно ввести производный класс, ссылаясь на класс QUERY как базовый. Назовем производный класс QUE_0.

```
class   QUE_0 : public  QUERY
 {    public:
      int def_0 ();
 };
```

Вне класса опишем новую операцию:

```
int QUE_0 ::def_0 ()
  { link *p=start;
    while  (p!= end)  and  (p->inf !=0 )
       p=p->next;
    if (p->inf == 0 ) return 1;
    else return 0;
  }
```

Когда один класс наследует другой класс, все элементы, определенные как private в базовом классе, не имеют доступа в производном классе.

Для того чтобы в классе QUE_0 был доступ к данным класса QUERY, необходимо использовать доступ protected. Описание класса QUERY будет иметь вид:

```
class QUERY
{  protected:
   struct link
   { int inf;
     link *next;
   };
   link *start,*end;

   public:
   QUERY();
   ~QUERY();
   void add(int elem);
   int get();
   int empty();
};
```

В нашем примере конструктор базового класса представлен без параметра. В этом случае производный класс может не иметь конструктор.

4. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

4.1. Необходимость обработки исключений

Во время своей работы программа иногда сталкивается с нештатными ситуациями. Например, вызванный метод некоторого объекта может обнаружить у себя внутренние проблемы (неверные значения полей, в результате чего может произойти, например, деление на ноль) или найти ошибки в других объектах или входных данных. Следовательно, необходим некоторый механизм обнаружения и обработки нештатных ситуаций (исключений) в программе.

Когда программа конструируется из раздельных модулей и особенно когда эти модули находятся в независимо разработанных библиотеках, обработка ошибок должна быть разделена на две части:

1) генерация информации о возникновении ошибочной ситуации, которая не может быть разрешена локально;

2) обработка ошибок, обнаруженных в других местах.

Автор библиотеки может обнаружить ошибки времени выполнения, но, как правило, не знает, что делать в этом случае. Пользователь библиотеки может знать, как поступить в случае возникновения ошибок, но не в состоянии их обнаружить (иначе их поиск не перепоручался бы библиотеке). Для помощи в решении подобных проблем в языке C++ введено понятие *исключения*. Фундаментальная идея состоит в том, что функция (метод), обнаружившая проблему, но не знающая, как ее решать, *возбуждает* (throw) исключение в надежде, что вызвавшая её функция (прямо или косвенно) может решить проблему. В функции, которая хочет решать проблемы данного типа, можно указать, что она *перехватывает* (catch) такие исключения.

Такой стиль обработка ошибок предпочтительнее многих традиционных методов. Рассмотрим альтернативы. При обнаружении проблемы, которая не может быть решена локально, программа может:

а) прекратить выполнение,

б) возвратить специальное «ошибочное» значение,

в) возвратить какое-то допустимое значение и оставить программу вненормальном состоянии,

г) вызвать функцию, предназначенную для обработки ошибки.

Вариант а) – прекратить выполнение – это то, что происходит по умолчанию, когда не перехватывается исключение. Библиотечная функция, безусловно завершающая выполнение, не может использоваться в программе, первое требование к которой – надежность.

Вариант б) – возвратить специальное «ошибочное» значение – не всегда выполним, т.к. часто просто нет приемлемого ошибочного значения (например, при возврате целого любое из них может быть приемлемо). Даже когда такой подход применим, он часто неудобен, потому что он вынуждает программиста каждый раз проверять результат на ошибочное значение. Это легко может увеличить размер программы.

Вариант в) – возвратить допустимое значение и оставить программу в ненормальном состоянии – имеет тот недостаток, что вызывающая функция может не заметить, что программа находится в ненормальном состоянии. Например, многие стандартные функции библиотеки С устанавливают значение глобальной переменной errno для индикации ошибки. Однако программы в большинстве случаев не проверяют эту переменную достаточно систематически, чтобы избежать последующих ошибок. Более того, использование глобальных переменных для записи информации об ошибках работает неудовлетворительно при наличии параллельных процессов.

Вариант г) – вызвать функцию, предназначенную для обработки ошибки – возлагает на вызванную функцию решение проблемы одним из перечисленных выше способов.

Механизм обработки исключений предоставляет альтернативу традиционным методам. Он позволяет отделить код обработки ошибок от собственного кода алгоритма, делая программу более понятной и более «чистой». В результате получаем более регулярный способ обработки ошибок, что упрощает взаимодействие между отдельно написанными фрагментами программы.

4.2. Общие принципы обработки исключительных ситуаций в C++

Механизм обработки ситуаций дает способ передачи управления из точки выполнения программы в расположенную выше по управлению точку, в которой определен *обработчик исключений* (exception handler). Главная идея состоит в том, что функция, сталкивающаяся с неразрешимой проблемой, объявит исключительную ситуацию, в надежде на то, что вызвавшая ее (прямо или косвенно) функция может решить проблему. Обработчик ситуации будет вызван только в случае исполнения *выражения-возбуждения-ситуации* внутри так называемого *блока-с-контролем* или в функциях, вызванных из этого блока. В C++ синтаксис возбуждения ситуации выглядит следующим образом:

выражение-возбуждения-ситуации:

throw выражение

Выражение-возбуждения-ситуации в C++ является выражением некоторого типа. *Выражение-возбуждения-ситуации* иногда еще называют точкой возникновения (возбуждения) ситуации (throw-point). О части програм-

мы, в которой исполнилось *выражение-возбуждения-ситуации,* говорят, что в ней возникла ситуация (она возбудила ситуацию).

Рассмотрим, каким образом в C++ можно определить и обработать ошибки диапазона, возникающие в классе Vector.

```
class Vector {
  int* p;
  int* sz;
public:
  class Range { }; // класс ситуаций

  int& operator[] (int i);
  // . . .
};
```

Объекты класса Range предназначены для использования в качестве исключений и возбуждать последние следующим образом:

```
int& Vector::operator[] (int i)
{
  if (0<=i && i<sz) return p[i];
  throw Range();
}
```

В общем случае исключения новых типов следует создавать тогда, когда программист хочет обрабатывать ошибки одного типа и пропускать ошибки других типов.

4.3. Возбуждение ситуации

При возбуждении ситуации управление передается на один из обработчиков ситуации. При этом посылается объект, и тип этого объекта определяет, какой обработчик может перехватить данную ситуацию.

Например, ситуация

```
throw "Help!";
```

может быть перехвачена некоторым обработчиком типа char*:

```
try {
  // . . .
}   catch (const char* p) {
// здесь реакция на ситуацию типа символьной строки
}
```

а ситуация

```
class Overflow {
  // . . .
public: Overflow(char, double, double);
};
void f(double x) {
  // . . .
  throw Overflow('+', x, 3.45e107);
}
```

может быть перехвачена обработчиком

```
try {
  // . . .
  f(1.2);
  // . . .
}   catch (Overflow& oo) {
  // здесь обработчик ситуации типа Overflow
}
```

При возникновении ситуации управление передается на ближайший обработчик соответствующего типа; "ближайший" – тот, в чей *блок-с-контролем* управление попало в последний раз; "соответствующий тип" определяется ниже.

Выражение-возбуждения-ситуации создает временный объект, тип которого статически определяется операндом операции throw, и затем инициализирует им переменную, указанную в описании обработчика.

В C++ *выражение-возбуждения-ситуации* без операнда повторно возбуждает обрабатываемую ситуацию. *Выражение-возбуждения-ситуации* без операнда может появиться только в обработчике исключения или в функции, явно или неявно вызванной из него. Например, код, который нужно выполнить при возникновении какой-либо ситуации, не обрабатывая ее полностью, мог бы быть написан так:

```
  try {
    // . . .
  }
  catch (...) { // перехват ситуации
                // частичная обработка ситуации
    throw;
// передача ситуации некоторому другому обработчику
  }
```

4.4. Обработка исключений

Функция, нуждающаяся в обнаружении возбужденной ситуации, должна поместить вызов функции в *блок-с-контролем* с реакцией на ситуацию. Такой блок имеет следующий синтаксис в C++:

блок-с-контролем:

try { … }

catch (*объявлении-ситуации* 1) { реакция 1 }

catch (*объявлении-ситуации* 2) { реакция 2 }

catch (...) { реакция n }

Например:

```
void f(Vector& v)
 {
  // . . .
  try
  {
    do_something(v);
  }
  catch (Vector::Range)
  {
    // реакция на исключение типа Vector::Range
    // так как do_something() вызвало ситуацию,
    // делаем что-то другое,
    // попадаем сюда только, если обращение к
    // do_something() приводит
    // к вызову Vector::operator[](i) с плохим
    // индексом
  }
    // . . .
}
```

Конструкция

```
  catch (/* . . . */) {  }
```

представляет собой обработчик ситуации. Он может использоваться только сразу за *блоком-с-контролем* или непосредственно за другим обработчиком (если есть несколько обработчиков разных типов).

Многоточие (. . .) в *объявлении-ситуации* дает отождествление для любой ситуации. Обработчик с многоточием, если он есть, должен быть последним в своем *блоке-с-контролем*.

5. РЕАЛИЗАЦИЯ ПРИМЕРОВ

5.1. Класс полином

```
class stp       // Базовый класс
 { protected:
   int n;
   stp(){};
   stp(int st)
     {n=st;}
 };
```

Класс полином poli является производным классом.

```
 class poli: public stp
 { public:
    int *a;
    poli(){};
    poli(int s){ n=s;};
    ~poli()    { delete []a;};
    void cr_poli();
    poli(poli &y);
   int  operator()(int );
   friend poli   operator+ (poli  bb, poli cc);
   poli & operator= (poli bb);
   void prosm();
 };

 poli & poli::  operator= (poli bb)
 { n=bb.n;
   for (int i=0;i<=n;i=i+1) a[i]=bb.a[i];
   return *this;
 };
poli::poli(poli &y)
 { n=y.n;   a=new int[n+1];
    for(int i=0;i<=n;i++)
       a[i]=y.a[i];
 };
```

```
void poli::cr_poli()
 { a=new int[n+1];
   cout<<"введите коэф-ты  "<<n;
    for (int i=0;i<=n;i++)
      cin>>a[i];
  }

int poli:: operator()(int x)
  { int k=n;
    float r;
    r=a[0];
    for (int i=1;i<=k;i=i+1)
      r=r*x+a[i];
    return r;
   }

poli  operator+ (poli bb,poli cc)
  {  int i,k,l;
     k=bb.n; l=cc.n;
     if (k>=l)
      { poli r(k);
        for (i=0;i<=k;i++) r.a[i]=bb.a[i];
        for (i=0;i<=l;i++)
          r.a[k-l+i]=r.a[k-l+i]+cc.a[i];
        return r;
       };
     poli t(l);
     for (i=0;i<=l;i++) t.a[i]=cc.a[i];
     for (i=0;i<=k;i++)
          t.a[l-k+i]=t.a[l-k+i]+bb.a[i];
     return t;
 }
void poli:: prosm()
   { int i;
     for (i=0;i<=n;i++)
        cout <<a[i]<<"  ";
     cout<<"\n";
   } ;
```

5.2. Класс таблица

```
class zap      // Базовый класс
 { protected:
   char fio[31];
   char pr[10];
   int a1;
   public:
   zap(){};
   zap(zap &z)
   { strcpy(fio,z.fio);
     strcpy(pr,z.pr);    a1=z.a1;
   };
    int get(){ return a1;};
    char* get_f()
      { char *s; s=strdup(fio);
        return s;
      };
  void input();
  void output();
 };
 class table : public zap
 {  public:
    zap *a;  int n;
    table(){};
    table(int s){ n=s;};
    ~table()    { delete []a;};
    void cr_table();
    void sort(int t);
    void sort(char t);
    void show();
  };

  void zap:: input()
  {  cout<<"\nVvod fio\n";     cin>>fio;
     cout<<"\nVvod pred\n";    cin>>pr;
     cout<<"\nVvod ball\n";    cin>>a1;
  };

  void zap::output()
  { cout<< fio <<"  "<<pr<<"  "<<a1<<"  ";
```

```
 };
void table :: cr_table()
 { a=new zap [n];
   for(int i=0;i<n;i++)
    a[i].input();
  cout <<"\n таблица создана \n";
 };
void table :: show()
{ int i;
  for(i=0;i<n;i++)
  { a[i].output();
    cout <<"\n ";
  };
};
int SRAVN(int a,int b)
 {  if(a>b) return 1;
    else return 0;
 };
int SRAVN(char a[],char b[])
 { char *u,*v;
   u=strdup(a);v=strdup(b);
   if (strcmp(u,v)>0) return 1;
   else return 0;
 };
void table ::  sort(char t)
{ zap s;  int i,j;
  for (i=n-1;i>0;i--)
  { int k=0;
    for (j=1;j<i;j++)
      if (SRAVN(a[j].get_f(),a[k].get_f()))   k=j;
    s=a[k];a[k]=a[i];a[i]=s;
  }
};
void table ::  sort(int t)
{ zap s;
  int i,j;
  for (i=n-1;i>0;i--)
   { int k=0;
    for (j=1;j<i;j++)
       if (SRAVN(a[j].get(),a[k].get()))   k=j;
    s=a[k];a[k]=a[i];a[i]=s;
   }
```

```
}
```

6. ЗАДАНИЯ ДЛЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

1) Описать класс «Дробное число со знаком». Число должно быть представлено двумя полями: целая часть – длинное целое число со знаком, дробная часть – беззнаковое короткое целое число. Реализовать арифметические операции сложения, вычитания, умножения, деления и операцию сравнения дробных чисел. Написать текст программы на языке C++, демонстрирующей функционал объектов, порожденных от класса.

2) Описать класс «Деньги» для работы с денежными суммами. Денежная сумма должна быть представлена двумя полями: длинное целое число для рублей и беззнаковое короткое целое число – для копеек. Дробная часть денежной суммы (копейки) при выводе на экран должна быть отделена от целой части запятой. Реализовать сложение сумм, вычитание сумм, деление сумм, деление суммы на дробное число, умножение суммы на дробное число, операцию сравнения сумм и преобразование денежной суммы в символьную строку. Написать текст программы на языке C++, демонстрирующей функционал объектов, порожденных от класса.

3) Описать класс «Трапеция», поля класса – координаты четырех точек на плоскости: координаты – числа с плавающей запятой. Предусмотреть в классе конструктор и методы: проверка, является ли фигура, образованная четырьмя точками трапецией; проверка, является ли фигура, образованная четырьмя точками равнобедренной трапецией, вычисление длины каждой стороны, вычисление периметра трапеции, вычисление площади трапеции. Написать текст программы на языке C++, демонстрирующей функционал объекта, порожденного от класса.

4) Описать класс «Комплексное число», поля класса - действительная и мнимая части комплексного числа являются числами с плавающей запятой и находятся в закрытом разделе класса. Предусмотреть в классе конструктор и методы: присваивание комплексному числу значения, сложение комплексных чисел, вычитание комплексных чисел, умножение комплексных чисел, деление комплексных чисел и преобразование комплексного числа в символьную строку. Написать текст программы на языке C++, демонстрирующей функционал объектов, порожденных от класса.

ЛИТЕРАТУРА

1. Невская, Е.С. Искусство программирования [Текст]: Учеб. пособие для вузов / Е.С. Невская, А.А. Чекулаева, М.И. Чердынцева – Москва: Вузовская книга, 2002. - 207 с.

2. Страуструп, Б. Язык программирования C++ для профессионалов [Текст] / Б. Страуструп. – Москва: Интернет-Университет Информационных Технологий, 2006. – 568 с.