Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное
образовательное учреждение
высшего профессионального образования
«Пермский национальный исследовательский
политехнический университет»

А.М. Ноткин

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ: ООП на языке C++

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия

Издательство
Пермского национального исследовательского
политехнического университета
2013

УДК 681.3
Н85

Рецензенты:
канд. техн. наук, ст. науч. сотрудник
ТиИС ИМСС УрО РАН *Г. Ф. Масич*;
канд. техн. наук, доцент кафедры информационных
технологий и автоматизированных систем
Пермского национального исследовательского
политехнического университета
О. А. Полякова

Ноткин, А.М.

Н85 Объектно-ориентированное программирование : ООП на языке С++ : учебное пособие / А.М. Ноткин. – Пермь : Изд-во Перм. нац. исслед. политехн. ун-та, 2013. – 230 с.

ISBN 978-5-398-00966-8

Пособие является первой частью многотомного издания, посвященного технологии объектно-ориентированного программирования на языке С++. Следующие тома будут посвящены ООП на языках Java, C# NET, Pyton и Ruby.

Даны основные понятия ООП и технология объектно-ориентированного программирования на языке С++. Подробно рассматриваются синтаксис, семантика и техника программирования. Приведено большое количество примеров, иллюстрирующих возможности и особенности применения языка С++ для создания объектно-ориентированных программ.

Предназначено для студентов направления «Информатика и вычислительная техника» как для самостоятельной работы, так и для аудиторных занятий.

УДК 681.3

ISBN 978-5-398-00966-8

ОГЛАВЛЕНИЕ

1. КЛАССЫ C++

1.1. Новый тип данных – класс

Целью введения концепции классов в C++ является предоставление программисту средств создания новых типов, которые настолько же удобны в использовании, как и встроенные типы. Тип является конкретным представлением некоторой концепции. Например, встроенный тип C++ *float* вместе с операциями +, –,* и другими. является воплощением математической концепции вещественного числа. Класс – это определенный пользователем тип. Создаем новый тип для определения концепции, невыражаемой непосредственно встроенными типами. Например, можно ввести тип *TrunkLine* (междугородная линия) в программе, имеющей отношение к телефонии, тип *Depositir* (вкладчик) в программе управления банком или тип *Pretator* (хищник) в программе экологического моделирования.

Класс – фундаментальное понятие C++ и лежит в основе многих свойств C++. Класс предоставляет механизм для создания объектов. В классе отражены важнейшие концепции объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм.

С точки зрения синтаксиса класс в C++ – это структурированный тип, образованный на основе уже существующих типов.

В этом смысле класс является расширением понятия структуры. В простейшем случае класс можно определить с помощью конструкции:

тип_класса имя_класса{список_членов_класса};

где *тип_класса* – одно из служебных слов **class, struct, union**;

имя_класса – идентификатор;

список_членов_класса – определения и описания типизированных данных и принадлежащих классу функций.

Функции – это методы класса, определяющие операции над объектом.

Данные – это поля объекта, образующие его структуру. Значения полей определяют состояние объекта.

Будем называть члены класса компонентами класса, различая компонентные данные и компонентные функции.

Пример 1.1

```
struct date                          //дата
{int month,day,year;                 // поля: месяц, день, год
  void set(int,int,int);             // метод – установить дату
  void get(int*,int*,int*);          // метод – получить дату
  void next();                       // метод – установить следующую дату
  void print();                      //  метод – вывести дату
};
```

Пример 1.2

```
struct complex     // комплексное число
{double re,im;
  double real(){return(re);}
  double imag(){return(im);}
  void set(double x,double y){re = x; im = y;}
  void print(){cout<<"re = "<<re; cout<<"im = "<<im;}
};
```

Для описания объекта класса (экземпляра класса) используется конструкция

имя_класса имя_объекта

```
date today,my_birthday;
date *point = &today;            //указатель на объект типа date
date clim[30];                   // массив объектов
date &name = my_birthday;    //ссылка на объект
```

В определяемые объекты входят данные, соответствующие членам – данным класса. Функции – члены класса позволяют обрабатывать данные конкретных объектов класса. Обра-

щаться к данным объекта и вызывать функции для объекта можно двумя способами. Во-первых, с помощью «квалифицированных» имен:

имя_объекта.имя_класса : : имя_данного
имя_объекта.имя_класса : : имя_функции
Имя класса может быть опущено
имя_объекта.имя_данного
имя_объекта.имя_функции
Например:
класс “комплексное число”
```
complex x1,x2;
x1.re = 1.24;
x1.im = 2.3;
x2.set(5.1,1.7);
x1.print();
```

Второй способ доступа использует указатель на объект:
указатель_на_объект–>имя_компонента
```
complex *point = &x1;  //  или  point = new complex;
point –>re = 1.24;
point –>im = 2.3;
point –>print();
```

Пример 1.3
Класс “товары”
```
int percent=12; // наценка
struct goods
{char name[40];
  float price;
 void Input()
 {cout<<“наименование: ”;
 cin>>name;
 cout<<“цена: ”;
 cin>>price;}
```

```
void print()
{cout<<“\n”<<name;
cout<<“, цена: ”;
cout<<long(price*(1.0+percent*0.01));}
};
void main(void)
{  goods wares[5];
int k = 5;
for(int i = 0; i < k; i++) wares[i].Input();
cout<<“\nСписок товаров при наценке ”<<percent<<“ % ”;
for(i = 0; i < k; i++) wares[i].print();
percent = 10;
cout<<“\nСписок товаров при наценке ”<< percent<<” % ”;
goods *pGoods = wares;
for(i = 0; i < k; i++) pGoods++–>print();
}
```

1.2. Доступность компонентов класса

В рассмотренных ранее примерах классов компоненты классов являются общедоступными. В любом месте программы, где «видно» определение класса, можно получить доступ к компонентам объекта класса. Тем самым не выполняется основной принцип абстракции данных – инкапсуляция (сокрытие) данных внутри объекта. Для изменения видимости компонентов в определении класса можно использовать спецификаторы доступа: **public, private, protected**.

Общедоступные (public) компоненты доступны в любой части программы. Они могут быть использованы любой функцией как внутри класса, так и вне его. Доступ извне осуществляется через имя объекта:

имя_объекта.имя_члена_класса;
ссылка_на_объект.имя_члена_класса;
указатель_на_объект->имя_члена_класса;

Собственные (private) компоненты локализованы в классе и не доступны извне. Они могут использоваться функциями – членами данного класса и функциями – «друзьями» того класса, в котором они описаны.

Защищенные (protected) компоненты доступны внутри класса и в производных классах. Защищенные компоненты нужны только в случае построения иерархии классов. Они используются так же, как и private-члены, но дополнительно могут использоваться функциями – членами и функциями – «друзьями» классов, производных от описанного класса.

Изменить статус доступа к компонентам класса можно и с помощью использования в определении класса ключевого слова **class**. В этом случае все компоненты класса по умолчанию являются собственными.

Пример 1.4

```
class complex
{
  double re, im;                  // private по умолчанию
public:
  double real(){return re;}
  double imag(){return im;}
  void set(double x,double y){re = x; im = y;}
};
```

Современный стиль программирования рекомендует для определения класса использовать ключевое слово class.

1.3. Конструктор и деструктор

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо было вызвать функцию типа set (как для класса complex) либо явным образом присваивать значения данным объекта. Однако для инициализации

объектов класса в его определение можно явно включить специальную компонентную функцию, называемую **конструктором**. Формат определения конструктора следующий:

имя_класса(список_форм_параметров)
{операторы_тела_конструктора};

Имя этой компонентной функции по правилам языка C++ должно совпадать с именем класса. Такая функция автоматически вызывается при определении или размещении в памяти с помощью оператора new каждого объекта класса.

Пример 1.5

complex(double re1 = 0.0,double im1 = 0.0){re = re1; im = im1;}

Конструктор выделяет память для объекта и инициализирует данные – члены класса.

Конструктор имеет ряд особенностей:

– Для конструктора не определяется тип возвращаемого значения. Даже тип void не допустим.

– Указатель на конструктор не может быть определен и, соответственно, нельзя получить адрес конструктора.

– Конструкторы не наследуются.

– Конструкторы не могут быть описаны с ключевыми словами virtual, static, const, mutuable, valatile.

Конструктор всегда существует для любого класса, причем если он не определен явно, он создается автоматически. По умолчанию создается конструктор без параметров и конструктор копирования. Если конструктор описан явно, то конструктор по умолчанию не создается. По умолчанию конструкторы создаются общедоступными (public).

В классе может быть несколько конструкторов, но только один с умалчиваемыми значениями параметров. Перегрузка чаще всего используется для передачи конструктору аргументов, предназначенных для инициализации данных – членов класса. Параметром конструктора не может быть его собственный класс, но может быть ссылка на него (T&). Без явного указания программиста конструктор всегда автоматически вызывается

при определении (создании) объекта. В этом случае вызывается конструктор без параметров. Для явного вызова конструктора используются две формы:

имя_класса имя_объекта(фактические_параметры);

имя_класса(фактические_параметры);

Первая форма допускается только при непустом списке фактических параметров. Она предусматривает вызов конструктора при определении нового объекта данного класса:

complex ss(5.9,0.15);

Вторая форма вызова приводит к созданию объекта без имени:

complex ss = complex(5.9,0.15);

Существуют два способа инициализации данных объекта с помощью конструктора. Ранее мы рассматривали первый способ, а именно передачу значений параметров в тело конструктора. Второй способ предусматривает применение списка инициализаторов данного класса. Этот список помещается между списком параметров и телом конструктора. Каждый инициализатор списка относится к конкретному компоненту и имеет вид

имя_данного(выражение)

Пример 1.6

```
class A
{
  int i; float e; char c;
public:
  A(int ii,float ee,char cc) : i(8),e( i * ee + ii ),c(cc){}
  . . .
};
```

Пример 1.7. Класс "символьная строка”.

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

```
  char *ch; // указатель на текстовую строку
  int len;      // длина текстовой строки
public:
  // конструкторы
  // создает объект – пустая строка
  string(int N = 80): len(0){ch = new char[N+1]; ch[0] = '\0';}
  // создает объект по заданной строке
  string(const char *arch){len = strlen(arch);
      ch = new char[len+1];
      strcpy(ch,arch);}
  // компоненты-функции
  // возвращает ссылку на длину строки
  int& len_str(void){return len;}

  // возвращает указатель на строку
  char *str(void){return ch;}
  . . .
};
```

Здесь у класса string два конструктора – перегружаемые функции.

По умолчанию создается также конструктор копирования вида T::T(const T&), где T – имя класса. Конструктор копирования вызывается всякий раз, когда выполняется копирование объектов, принадлежащих классу. В частности, он вызывается:

а) когда объект передается функции по значению;

б) при построении временного объекта как возвращаемого значения функции;

в) при использовании объекта для инициализации другого объекта.

Если класс не содержит явным образом определенного конструктора копирования, то при возникновении одной из этих трех ситуаций производится побитовое копирование объекта. Побитовое копирование не во всех случаях является адекватным. Именно для таких случаев и необходимо определить соб-

ственный конструктор копирования. Например, создадим два объекта типа string:

```
string s1("это строка");
string s2=s1;
```

Здесь объект s2 инициализируется объектом s1 путем вызова конструктора копирования, созданного компилятором по умолчанию. В результате эти объекты имеют одинаковое значение в полях ch, то есть эти поля указывают на одну и ту же область памяти. В результате при удалении объекта s1 будет освобождаться и область, занятая строкой, но она еще нужна объекту s2. Чтобы не возникало подобных ошибок, определим собственный конструктор копирования.

```
string(const string& st)
{len=strlen(st.len);
ch=new char[len+1];
strcpy(ch,st.ch);  }
```

Конструктор с одним аргументом может выполнять неявное преобразование типа своего аргумента в тип класса конструктора.

Например:

```
class complex
{double re,im;
complex(double r):re(r),im(0){ }
...
};
```

Этот конструктор реализует представление вещественной оси в комплексной плоскости.

Вызвать этот конструктор можно традиционным способом:

```
complex b(5);
```

Но можно вызвать его и так:

```
complex b=5;
```

Здесь необходимо преобразование скалярной величины (типа аргумента конструктора) в тип complex. Это осуществляется вызовом конструктора с одним параметром. Поэтому кон-

структор, имеющий один аргумент, не нужно вызывать явно, а можно просто записать complex b=5, что означает complex b = complex(3).

Преобразование, определяемое пользователем, неявно применяется в том случае, если оно уникально. Например,

```
class demo{
demo(char);
demo(long);
demo(char*);
demo(int*);
...}
```

Здесь в

```
demo a=3;
```

неоднозначность: вызов demo(char)? или demo(long)?

А в demo a=0; также неоднозначность: вызов demo(char*) или demo(int*), или demo(char), или demo(int)?

В некоторых случаях необходимо задать конструктор, который можно вызвать только явно. Например,

```
class string{
char * ch;
int len;
public:
string(int size){
len=size; ch=new[len+1]; ch[0]='\0';
};
```

В этом случае неявное преобразование может привести к ошибке. В случае string s='a'; создается строка длиной int('a'). Вряд ли это то, что мы хотели.

Неявное преобразование можно подавить, объявив конструктор с модификатором **explicit.** В этом случае конструктор будет вызываться только явно. В частности, там, где конструктор копирования в принципе необходим, explicit-конструктор не будет вызываться неявно. Например:

```
class string{
char * ch;
int len;
public:
explicit string(int size);
string(const char* ch);
};
string s1='a';  // Ошибка,
// нет явного преобразования char в string
string s2(10);  // Правильно,
//строка для хранения 10 символов-
//явный вызов конструктора
string s3=10;   //Ошибка, нет явного преобразования int в string
string s4=string(10); // Правильно,
// конструктор вызывается явно
string s5="строка";// Правильно,
// неявный вызов конструктора s5=string("строка")
```

Можно создавать массив объектов, однако при этом соответствующий класс должен иметь конструктор по умолчанию (без параметров).

Это связано с тем, что при объявлении массива объектов невозможно определить параметры для конструкторов этих объектов, и единственная возможность вызова конструкторов – это передача им параметров, заданных по умолчанию.

Массив объектов может инициализироваться либо автоматически конструктором по умолчанию, либо явным присваиванием значений каждому элементу массива.

```
class demo{
int x;
public:
demo(){x=0;}
demo(int i){x=i;}
};
void main(){
```

class demo a[20]; //вызов конструктора без параметров (по умолчанию)

class demo b[2]={demo(10),demo(100)};//явное присваивание

При объявлении массива объектов невозможно определить параметры для конструкторов этих объектов и единственная возможность вызова конструкторов – это передача им параметров, заданных по умолчанию. Таким образом, для того чтобы создавать массив объектов, соответствующий класс должен иметь заданный по умолчанию конструктор. Можно уменьшить количество конструкторов, если задать конструктор с аргументами по умолчанию. Он же будет конструктором по умолчанию.

Динамическое выделение памяти для объекта создает необходимость освобождения этой памяти при уничтожении объекта. Например, если объект формируется как локальный внутри блока, то целесообразно, чтобы при выходе из блока, когда уже объект перестает существовать, выделенная для него память была возвращена. Желательно, чтобы освобождение памяти происходило автоматически. Такую возможность обеспечивает специальный компонент класса – **деструктор** класса. Его формат

~имя_класса(){операторы_тела_деструктора};

Имя деструктора совпадает с именем его класса, но предваряется символом “~” (тильда).

Деструктор не имеет параметров и возвращаемого значения. Вызов деструктора выполняется неявно (автоматически), как только объект класса уничтожается.

Например, при выходе за область определения или при вызове оператора delete для указателя на объект.

string *p=new string(“строка”);

delete p;

Если в классе деструктор не определен явно, то компилятор генерирует деструктор по умолчанию, который просто освобождает память занятую данными объекта. В тех случаях, когда требуется выполнить освобождение и других объектов памяти, на-

пример, область, на которую указывает ch в объекте string, необходимо определить деструктор явно: ~string(){delete []ch;}

Так же, как и для конструктора, не может быть определен указатель на деструктор.

1.4. Компоненты-данные и компоненты-функции

1.4.1. Данные – члены класса

Определение данных класса внешне аналогично описанию переменных базовых или производных типов. Однако при описании данных класса не допускается их инициализация. Для их инициализации должен использоваться автоматический или явно вызываемый конструктор. Принадлежащие классу функции имеют полный доступ к его данным. Для доступа к элементам-данным из операторов, выполняемых вне определения класса, нужно использовать операции выбора компонентов класса (“ . ” или ”–>”). Данные класса необязательно должны быть определены или описаны до их первого использования в принадлежащих классу функциях. Все компоненты класса “видны” во всех операторах его тела. Область доступа к компонентам-данным регулируется модификатором доступа (см. п. 1.2).

Компоненты-данные могут быть описаны как **const**. В этом случае после инициализации они не могут быть изменены.

Компоненты-данные могут быть описаны как **mutuable**. В этом случае они являются изменяемыми, даже если объект, содержащий их, описан как const.

1.4.2. Функции – члены класса

Компонентная функция должна быть обязательно описана в теле класса. При определении классов их компонентные функции могут быть специфицированы как **подставляемые** (**inline**). Кроме явного использования слова inline для этого используются следующие соглашения. Если определение функции полно-

стью размещено в теле класса, то эта функция по умолчанию считается подставляемой. Если в теле класса помещен только прототип функции, а ее определение – вне класса, то для того, чтобы функция была подставляемой, ее надо явно специфицировать словом inline. При внешнем определении функции в теле класса помещается прототип функции

тип имя_функции(спецификция_и_инициализация_параметров);

Вне тела класса функция определяется так

тип имя_класса : : имя_функции(спецификция_формальных_параметров)
{тело_функции}

1.4.3. Константные компоненты-функции

Функции – члены класса могут быть описаны как **const.** В этом случае они не не могут изменять значения данных – членов класса и могут возвращать указатель или ссылку только на данные–члены класса, описанные как **const**. Они являются единственными функциями, которые могут вызываться для объекта-константы.

Например, в классе conplex:

```
class complex{
double re,im;
public:
//...
double real()const{return re;}
double imag()const{return im;}
};
```

Объявление функций real() и imag() как const гарантирует, что они не изменяют состояние объекта complex. Компилятор обнаружит случайные попытки нарушить это условие.

Когда константная функция определяется вне класса, указывать const надо обязательно:

```
double complex::real()const{return re:}
```

Константную функцию-член можно вызвать как для константного, так и для неконстантного объекта, в то время как неконстантную функцию-член можно вызвать только для объекта, не являющегося константой.

1.4.4. Статические члены класса

Каждый объект одного и того же класса имеет собственную копию данных класса. Это не всегда соответствует требованиям решаемой задачи. Например, счетчик объектов, указатели на первый и последний объект в списке объектов одного класса или наценка в классе *goods* в примере 1.1.3. Эти данные должны быть компонентами класса, но иметь их нужно только в единственном числе. Такие компоненты должны быть определены в классе как **статические** (**static**). Статические данные классов не дублируются при создании объектов, т.е. каждый статический компонент существует в единственном экземпляре. Доступ к статическому компоненту возможен только после его инициализации. Для инициализации используется конструкция

тип имя_класса : : имя_данного инициализатор;

Например, int goods : : percent = 12;

Это предложение должно быть размещено в глобальной области после определения класса. Только при инициализации статическое данное класса получает память и становится доступным. Обращаться к статическому данному класса можно обычным образом через имя объекта

имя_объекта.имя_компонента

Но к статическим компонентам можно обращаться и тогда, когда объект класса еще не существует. Доступ к статическим компонентам возможен не только через имя объекта, но и через имя класса

имя_класса : : имя_компонента

Однако так можно обращаться только к *public* компонентам.

А как обратиться к *private* статической компоненте извне определения объекта? С помощью компонента-функции этого класса. Однако при вызове такой функции необходимо указывать

имя объекта, а объект может еще не существовать. Эту проблему решают **статические компоненты-функции**. Эти функции можно вызвать через имя класса.

имя_класса : : имя_статической_функции

Пример 1.8

```
#include <iostream.h>
class TPoint
{
  double x,y;
// статический компонент-данное : количество точек
static int N;
public:
  // конструктор
  TPoint(double x1 = 0.0,double y1 = 0.0){N++; x = x1; y = y1;}
// статический компонент-функция
static int& count(){return N;}
};
//инициализация статического компонента-данного
int TPoint : : N = 0;
void main(void)
{TPoint A(1.0,2.0);
  TPoint B(4.0,5.0);
  TPoint C(7.0,8.0);
  cout<<“\nОпределены ”<<TPoint : : count()<<“точки.”;
}
```

Пример 1.9. Моделирование списка

```
class list
{int x;        //информационное поле
list *next;   //указатель на следующий элемент
static list *begin // начало списка
public:
list(int x1);
```

```
~list();
add();  //объект добавляет себя в список
static show(); //статическая функция для просмотра списка
};
list * list : : begin = NULL;
void main()
{list* p;
// создаем первый объект и добавляем его в список
p=new list(5); p->add();
// создаем второй объект и добавляем его в список
p=new list(8); p->add();
// создаем третий объект и добавляем его в список
p=new list(35); p->add();
list::show();                    // показываем весь список
}
```

Пример 1.10. Другая реализация класса goods – см. пример 1.3.

```
#include <iostream.h>
// Класс “товары”
class goods
{char name[40];
  float price;
  static int percent;  // наценка
public:
void Input(){cout<<“наименование: ”;
 cin>>name;
 cout<<“цена: ”;
 cin>>price;}
void print(){cout<<“\n”<<name;
cout<<“, цена: ”;
cout<<long(price*(1.0+goods : : percent*0.01));}
};
int goods : : percent = 12;
```

```
void main(void)
{
  goods wares[5];
  int k = 5;
  for(int i = 0; i < k; i++) wares[i].Input();
  cout<<“\nСписок товаров при наценке“
<<wares[0].percent<<“% ”;
  for(i = 0; i < k; i++) wares[i].print();
  goods : : percent = 10;
  cout<<“\nСписок товаров при наценке ”
<<goods : : percent<<” % ”;
  goods *pGoods = wares;
  for(i = 0; i < k; i++) pGoods++–>print();
}
```

1.5. Указатели на компоненты класса

1.5.1. Указатели на компоненты-данные

Можно определить указатель на компоненты-данные:

*тип_данных(имя_класса : :*имя_указателя)*

В определении указателя можно включить его инициализатор:

&имя_класса : : имя_компонента

Пример: double(complex : :*pdat) = &complex : : re;

Естественно, что в этом случае данные-члены должны иметь статус открытых(pubic).

После инициализации указателя его можно использовать для доступа к данным объекта.

```
complex c(10.2,3.6);
c/*pdat=22.2;      //изменилось значение поля re объекта c.
```

Указатель на компонент класса можно использовать в качестве фактического параметра при вызове функции.

Если определены указатели на объект и на компонент, то доступ к компоненту с помощью операции ‘ –>* ’.

*указатель_на_объект –>*указатель_на_компонент*

Пример 1.11

```
double(complex : :*pdat) = &complex : : re;
complex C(10.2,3.6);
complex *pcom = &C;
pcom –>*pdat = 22.2;
```

Можно определить тип указателя на компоненты-данные класса:

```
typedef  double(complex::*PDAT);
void f(complex c, PDAT pdat) {c.*pdat=0;}
complex c;
PDAT pdat=&complex::re; f(c,pdat);
pdat=&complex::im; f(c,pdat);
```

1.5.2. Указатели на компоненты-функции

Можно определить указатель на компоненты-функции.

тип_возвр_значения(имя_класса::
**имя_указателя_на_функцию)(специф_параметров_функции);*

Пример 1.12

```
// Определение указателя на функцию-член класса
double(complex : :*ptcom)();
// Настройка указателя
ptcom = &complex : : real;
// Теперь для объекта  A
complex A(5.2,2.7);
// можно вызвать его функцию
cout<<(A.*ptcom)();
// Если метод real определить типа ссылки
double& real(void){return re;}
```

```
// то, используя этот метод, можно изменить поле re
(A.*ptcom)() = 7.9;
// При этом указатель определяется так:
double&(complex : :*ptcom)();
```

Можно определить также тип указателя на функцию:

```
typedef  double&(complex::*PF)();
```

а затем определить и сам указатель:

```
PF ptcom=&complex::real;
```

1.6. Указатель this

Когда функция-член класса вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет имя **this** и неявно определен в каждой функции класса следующим образом:

*имя_класса *const this = адрес_объекта*

Указатель this является дополнительным скрытым параметром каждой нестатической компонентной функции. При входе в тело принадлежащей классу функции this инициализируется значением адреса того объекта, для которого вызвана функция. В результате этого объект становится доступным внутри этой функции.

В большинстве случаев использование **this** является неявным. В частности, каждое обращение к нестатической функции-члену класса неявно использует this для доступа к члену соответствующего объекта. Например, функцию add в классе complex можно определить эквивалентным, хотя и более пространным способом:

```
void complex add(complex ob)
{this->re=this->re+ob.re;
// или *this.re=*this.re+ob.re
this->im=this->im+ob.im;}
```

Если функция возвращает объект, который ее вызвал, используется указатель this.

Например, пусть функция add возвращает ссылку на объект. Тогда

```
complex& complex add(complex& ob)
{re=re+ob.re;
im=im+ob.im;
return *this;
}
```

Примером широко распространенного использования this являются операции со связанными списками.

Пример. 1.13. Связанный список.

```
#include <iostream.h>
//Определение класса
class item
{
  static item *begin;
  item *next;
  char symbol;
public:
  item (char ch){symbol = ch;}  // конструктор
  void add(void);  // добавить в начало
  static void print(void);
};
//Реализация класса
void item : : add(void)
{
  this –>next = begin;
  begin = this;
}
void item : : print(void)
{
  item *p;
```

```
  p = begin;
  while(p != NULL )
  {
    cout<<p –>symbol<<“ \t ”;
    p = p –>next;
  }
}
//Создание и просмотр списка
item *item : : begin = NULL; // инициализация статического компонента
void main()
{
  item A(‘a’); item B(‘b’); item C(‘c’);
  // включение объектов в список
  A.add(); B.add(); C.add();
  // просмотр списка в обратном порядке
  item : : print();
}
```

1.7. Друзья классов

1.7.1. Дружественная функция

Дружественная функция – это функция, которая, не являясь компонентом класса, имеет доступ к его защищенным и собственным компонентам. Такая функция должна быть описана в теле класса со спецификатором **friend**.

Пример 1.14

```
class myclass
{
  int x,y;
  friend void set(myclass*,int,int);
public:
```

```
  myclass(int x1,int y1){x = x1; y = y1;}
  int sum(void){return (x+y);}
};
void set(myclass *p,int x1,int y1){p–>x = x1; p–>y = y1;}
void main(void)
{
  myclass A(5,6);
  myclass B(7,8);
  cout<<A.sum();
  cout<<B.sum();
  set(&A,9,10);
  set(&B,11,12);
  cout<<A.sum();
  cout<<B.sum();
}
```

Функция set описана в классе myclass как дружественная и определена как обычная глобальная функция (вне класса, без указания его имени, без операции ‘ : : ‘ и без спецификатора friend).

Дружественная функция при вызове не получает указателя this. Объекты класса должны передаваться дружественной функции только через параметр.

Итак, дружественная функция:

– не может быть компонентной функцией того класса, по отношению к которому определяется как дружественная;

– может быть глобальной функцией;

– может быть компонентной функцией другого ранее определенного класса.

Например,

```
class CLASS1
{. . .
  int f(. . .);
  . . .
};
```

```
class CLASS2
{. . .
  friend int CLASS1 : : f(. . .);
  . . .
};
```

// В этом примере класс CLASS1 с помощью своей компонентной функции f( )

// получает доступ к компонентам класса CLASS2.

– может быть дружественной по отношению к нескольким классам.

Например,

```
// предварительное неполное определение класса
class CL2;
class CL1
{friend void f(CL1,CL2);
  . . .
};
class CL2
{friend void f(CL1,CL2);
  . . .
};
```

// В этом примере функция f имеет доступ к компонентам классов CL1 и CL2.

1.7.2. Дружественный класс

Класс может быть дружественным другому классу. Это означает, что все компонентные функции класса являются дружественными для другого класса. Дружественный класс должен быть определен вне тела класса, «предоставляющего дружбу».

Например,

```
class X2{friend class X1; . . .};
class X1
{. . .
```

```
  void f1(. . .);
  void f2(. . .);
  . . .
};
```

// В этом примере функции f1 и f2 класса X1 являются друзьями класса X2, хотя они

// описываются без спецификатора friend.

Пример 1.15

Рассмотрим класс **point** – точка в *n*-мерном пространстве и дружественный ему класс **vector** – радиус-вектор точки («вектор с началом в начале координат n-мерного пространства»). В классе vector определим функцию для определения нормы вектора, который вычисляется как сумма квадратов координат его конца.

```
class point
{int N;          // размерность
  double *x;   // указатель на массив координат
  friend class vector;
public:
  point(int n,double d = 0.0);
};
point : : point(int n,double d)
{N = n;
  x = new double[N];
  for(int i = 0; i < N; i++) x[i] = d;
}
class vector
{double *xv;
  int N;
public:
  vector(point,point);
  double norma();
};
vector : : vector(point begin,point end)
```

```
{N = begin.N;
 xv = new double[N];
 for(int i = 0; i < N; i++) xv[i] = end.x[i]–begin.x[i];
}
double vector : : norma()
{double dd = 0.0;
 for(int i = 0; i < N; i++) dd += xv[i]*xv[i];
 return dd;
}
void main(void)
{point A(2,4.0);
 point B(2,2.0);
 vector V(A,B);
 cout<<V.norma();
}
// Будет выведено – 8.
```

Недостатком предложенного класса point является то, что значения всех координат точки x[i] одинаковы. Чтобы они были произвольными и разными, необходимо определить конструктор как функцию с переменным числом параметров, например, так:

```
point : : point(int n,double d,. . .)
{
 N = n;
 x = new double[N];
 double *p = &d;
 for(int i = 0; i < N; i++){x[i] = *p; p++;}
}
```

1.8. Определение классов и методов классов

Определение классов обычно помещают в заголовочный файл.

Пример 1.16

```
// POINT.H
#ifndef POINTH
#define POINTH 1
class point
{
  int x,y;
public:
  point(int x1 = 0,int y1 = 0);
  int& getx(void);
  int& gety(void);
  . . .
};
#endif
```

Поскольку описание класса point в дальнейшем планируется включать в другие классы, то для предотвращения недопустимого дублирования описаний в текст включена условная препроцессорная директива #ifndef POINTH. Таким образом, текст описания класса point может появляться в компилируемом файле только однократно, несмотря на возможность неоднократного появления директив #include “point.h”.

Определить методы можно следующим образом:

```
// POINT.CPP
#ifndef POINTCPP
#define POINTCPP 1
#include “point.h”
point : : point(int x1,int y1){x = x1; y = y1;}
int& point : : getx(void){return x;}
int& point : : gety(void){return y;}
. . .
#endif
```

В программе, использующей объекты класса

```
#include “point.cpp”
. . .
void main(void)
```

{ point A(5,6);
  point B(7,8);
 . . .}

Внешнее определение методов класса дает возможность, не меняя интерфейс объектов класса с другими частями программ, по-разному реализовать компонентные функции.

Пример программы, разрабатываемой и выполняемой в среде *Microsoft Visual Studio*

Задание: создать фабрику роботов.

Для этого мы создаем два класса: класс «робот» и дружественный ему класс «фабрика». Поскольку роботов можно создавать только на фабрике, все поля класса «робот» закрыты (private).

Класс «фабрика» имеет метод add(), с помощью которого создается робот и сохраняется в массиве.

Ниже представлены файлы программы на C++, разработанные в среде Visual C++ 6.0

Файл “robot.h” Определение классов (рис.1.1).

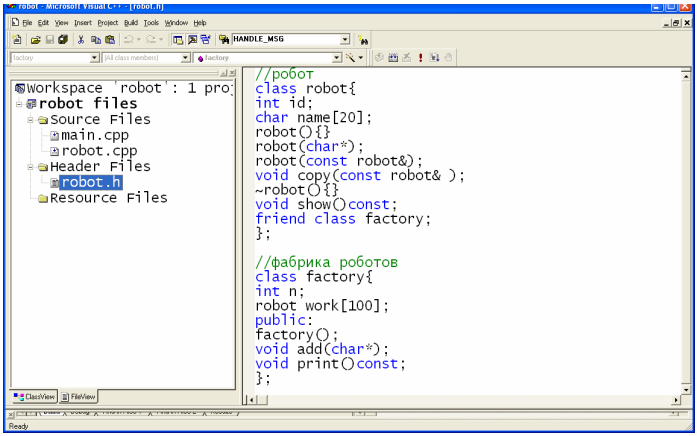


Рис. 1.1

//класс «робот»

class robot{

int id;

```
char name[20];
robot(){}
robot(char*);
robot(const robot&);
void copy(const robot& );
~robot(){}
void show()const;
friend class factory;
};

//класс «фабрика роботов»
class factory{
int n;
robot work[100];
public:
factory();
void add(char*);
void print()const;
};
```

Файл “robot.cpp” Определение методов классов (рис.1.2).

```
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
#include "robot.h"
robot::robot(char*NAME)
{id=rand();
strcpy(name,NAME);}
robot::robot(const robot&ob)
{id=ob.id;
strcpy(name,ob.name);}
void robot::copy(const robot& ob)
{id=ob.id;
strcpy(name,ob.name);}
```

void robot::show()const
{cout<<id<<" "<<name<<endl;}
factory::factory()
{n=0;}
void factory::add(char* name)
{robot temp(name);
work[n].copy(temp);
n++;}
void factory::print()const
{for(int i=0;i<n;i++)work[i].show();
}

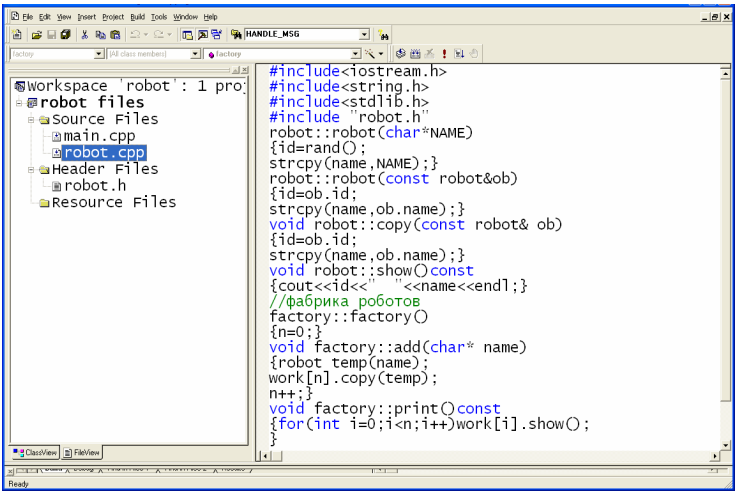


Рис. 1.2

Файл “main.cpp” Демонстрация работы программы (рис.1.3).

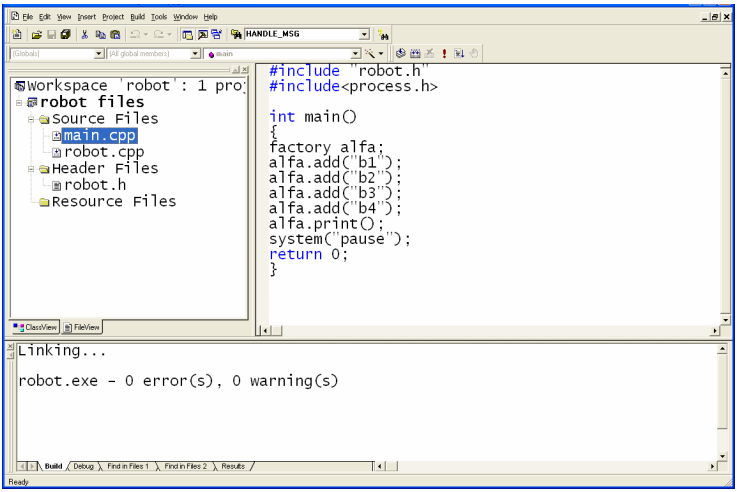


Рис. 1.3

Создаются четыре робота и просматриваются

```
#include "robot.h"
int main()
{
factory alfa;
alfa.add("b1");
alfa.add("b2");
alfa.add("b3");
alfa.add("b4");
alfa.print();
return 0;}
```

2. НАСЛЕДОВАНИЕ

2.1. Определение производного класса

Наследование – это механизм получения нового класса на основе уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.

Существующие классы называются *базовыми*, а новые – *производными*. Производный класс наследует описание базового класса; затем он может быть изменен добавлением новых членов, изменением существующих функций-членов и изменением прав доступа. Таким образом, наследование позволяет повторно использовать уже разработанный код, что повышает производительность программиста и уменьшает вероятность ошибок. С помощью наследования может быть создана иерархия классов, которые совместно используют код и интерфейсы.

Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически и незаметно для программиста в базовом классе.

При наследовании некоторые имена методов и данных базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компоненты базового класса становятся недоступными из производного класса. Для доступа к ним используется операция указания области видимости **‘ : : ’**.

В иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов (*public, protected*).

Допускается множественное наследование – возможность для некоторого класса наследовать компоненты нескольких, никак не связанных между собой базовых классов. В иерархии классов соглашение относительно доступности компонентов класса следующее:

private – член класса может использоваться только функциями- членами данного класса и функциями-«друзьями» своего класса. В производном классе он недоступен;

protected – то же, что и ***private,*** но дополнительно член класса с данным атрибутом доступа может использоваться функциями–членами и функциями-«друзьями» классов, производных от данного;

public – член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к ***public***-членам возможен доступ извне через имя объекта.

Следует иметь в виду, что объявление friend не является атрибутом доступа и не наследуется.

Синтаксис определение производного класса:

class имя_класса : список_базовых_классов

{список_компонентов_класса};

В производном классе унаследованные компоненты получают статус доступа **private**, если новый класс определен с помощью ключевого слова **class**, и статус **public**, если с помощью **struct**.

Например.

а) class S : X,Y,Z{. . .};

S – производный класс;

X,Y,Z – базовые классы.

Здесь все унаследованные компоненты классов X,Y,Z в классе A получают статус доступа **private**;

б) struct S : X,Y,Z{. . .};

S – производный класс;

X,Y,Z – базовые классы.

Здесь все унаследованные компоненты классов X,Y,Z в классе A получают статус доступа **public.**

Явно изменить умалчиваемый статус доступа при наследовании можно с помощью атрибутов доступа – *private, protected и public*, которые указываются непосредственно перед именами базовых классов. Как изменяются при этом атрибуты доступа в производном классе, показано в табл. 2.1.

Таблица 2.1

Атрибут, указанный при наследовании	Атрибут в базовом классе	Атрибут, полученный в производном классе
public	public protected	public protected
protected	public protected	protected protected
private	public protected	private private

Пример 2.1

```
class B
{protected: int t;
  public: char u;
  private: int x;
};
```

struct S : B{ }; // наследуемые члены t, u имеют атрибут доступа public

```
class E : B{ };  // t, u  имеют атрибут доступа  private
class M : protected B{ };  // t, u – protected.
class D : public B{ };  // t – protected, u – public
class P : private B{ };  // t, u – private
```

Таким образом, можно только сузить область доступа, но не расширить.

Таким образом, внутри производного класса существуют четыре уровня, для которых определяется атрибут доступа:

– для членов базового класса;

– для членов производного класса;

– для процесса наследования;

– для изменения атрибутов при наследовании.

Рассмотрим, как при этом регулируется доступ к членам класса извне класса и внутри класса.

Доступ извне

Доступными являются лишь элементы с атрибутом *public.*

♦ Собственные члены класса.

Доступ регулируется только атрибутом доступа, указанным при описании класса.

♦ Наследуемые члены класса.

Доступ определяется атрибутом доступа базового класса, ограничивается атрибутом доступа при наследовании и изменяется явным указанием атрибута доступа в производном классе.

Пример 2.2

```
class Basis{
public:
int a,b;
protected:
int c;};
class Derived:public Basis{
public:
Basis::a;};
void main(){
Basis ob;
Derived od;
ob.a;        //правильно
ob.c;        //ошибка
od.a;       //правильно
od.b;       //ошибка
```

Для членов класса действуют следующие права доступа:

♦ Собственные члены класса.

Доступ извне возможен только для *public*-членов класса.

private и *protected*-члены класса могут быть использованы только функциями-членами данного класса.

♦ Наследуемые члены класса.

private-члены класса могут использоваться только собственными функциями-членами базового класса, но не функциями членами производного класса.

protected или *public*-члены класса доступны для всех функций-членов. Подразделение на *public, protected* и *private* относится при этом к описаниям, приведенным в базовом классе, независимо от формы наследования.

Пример 2.3

```
class Basis{
int a;
public b;
void f1(int i){a=i;b=i;}
class Derived:private Basis{
public:
void f2(int i){
a=i;  //ошибка
b=i;}  // //правильно
};
```

2.2. Конструкторы и деструкторы производных классов

Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные-члены должны инициализироваться конструктором базового класса. Конструктор базового класса вызывается автоматически и выполняется до конструктора производного класса. Если наследуется несколько базовых классов, то их конструкторы выполняются в той последовательности, в которой перечислены базовые классы в определении производного класса. Конструктор производного класса вызывается по окончании работы конструкторов

базовых классов. Параметры конструктора базового класса указываются в определении конструктора производного класса. Таким образом происходит передача аргументов от конструктора производного класса конструктору базового класса.

Например:

```
class Basis
{ int a,b;
public:
Basis(int x,int y){a=x;b=y;}
};
class Inherit:public Basis
{int sum;
public:
Inherit(int x,int y, int s):Basis(x,y){sum=s;}
};
```

Запомните, что конструктор базового класса вызывается автоматически, и мы указываем его в определении конструктора производного класса только для передачи ему аргументов.

Объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они имеются), а потом сам производный класс. Таким образом объект производного класса содержит в качестве подобъекта объект базового класса.

Уничтожаются объекты в обратном порядке: сначала производный, потом его компоненты-объекты, а потом базовый объект.

Как мы знаем, объект уничтожается при завершении программы или при выходе из области действия определения объектов, и эти действия выполняет деструктор. Статус деструктора по умолчанию **public**. Деструкторы не наследуется, поэтому даже при отсутствии в производном классе деструктора он не передается из базового, а формируется компилятором как умалчиваемый. Классы, входящие в иерархию, должны иметь в своем распоряжении виртуальные деструкторы. Деструкторы могут переопределяться, но не перегружаться.

В любом классе могут быть в качестве компонентов определены другие классы. В этих классах могут быть свои деструкторы, которые при уничтожении объекта охватывающего (внешнего) класса выполняются после деструктора охватывающего класса. Деструкторы базовых классов выполняются в порядке, обратном перечислению классов в определении производного класса. Таким образом, порядок уничтожения объекта противоположен по отношению к порядку его конструирования.

Пример 2.4

// Определение класса базового класса ТОЧКА и производного класса ПЯТНО.

```
#include <graphics.h> // используем графику
#include <conio.h>
class point // Определение класса ТОЧКА
{
 protected:
 int x,y;
 public:
 point(int x1=0,int y1=0);
 int& getx(void);
 int& gety(void);
 void show(void);
 void move(int x1=0,int y1=0);
private:
void hide();
};
class spot : public point  // Определение класса ПЯТНО
{protected:
   int r;  // радиус
   int vis;  // признак видимости
   int tag;// признак сохранения образа объекта в памяти
  spot *pspot; // указатель на область памяти для образа
public:
```

```
    spot(int ,int ,int );
    void show();
    void hide();
    void move(int ,int );
    void change (float d)  // изменить размер
  };
// Определение функций – членов класса ТОЧКА
point : : point(int x1,int y1){x = x1; y = y1;}
int& point : : getx(void){return x;}
int& point : : gety(void){return y;}
void point : : show(void){putpixel(x, y, getcolor());}
void point : : hide(void){putpixel(x,y,getbkcolor());}
void point : : move(int x1,int y1)
{
  hide();
  x = x1; y = y1;
  show();
}
// Определение функций – членов класса ПЯТНО
spot::spot(int x1,int y1,int r1) : point(x1,y1)
{int size;
 vis = 0; tag = 0; r = r1;
size = imagesize(x1–r,y1–r,x1+r,y1+r);
pspot = (spot*)new char[size];}
spor::~spot(){hide(); tag = 0; delete pspot;}
void show()
{if(tag = = 0)
{circle(x,y,r);
  floodfill(x,y,getcolor());
  getimage(x–r,y–r,x+r,y+r,pspot);
 tag = 1; }
else putimage(x–r,y–r,pspot,XOR_PUT);
vis = 1;}
void spot::hide()
```

```
{if(vis = = 0) return;
putimage(x–r,y–r,pspot,XOR_PUT);
vis = 0;}
void spot::move(int x1,int y1)
{hide();
 x = x1; y = y1;
 show();}
 void spot::change(float d)
{float a; int size; hide(); tag = 0;
 delete pspot;
 a = d*r;
 if(a<=0)  r = 0;
 else r = (int)a;
 size = imagesize(x–r,y–r,x+r,y+r);
 pspot = (spot*)new char[size];
 show();}
 int& spot::getr(void){return r;}
};
// Создаются  два объекта, показываются, затем один
//перемещается, а другой изменяет размеры
void main()
{
//инициализация графики
int dr=DETECT,mod;
initgraph(&dr,&mod,“C : \ tc \ bgi”);
{
 spot A(200,50,20);
 spot B(500,200,30);
 A.show(); getch();
 B.show(); getch();
 A.move(50,60); getch();
 B.change(3); getch();
}
closegraph();
}
```

В этом примере в объекте spot точка создается как безымянный объект класса point. Особенностью функции main в примере является наличие внутреннего блока для работы с объектами spot. Это связано с наличием в классе spot деструктора, при выполнении которого вызывается метод hide(), требующий графического режима. Если построить программу без внутреннего блока, то деструктор будет вызываться при окончании программы, когда графический режим закрыт.

Эту проблему можно также решить путем явного вызова деструктора, например:

```
…
B.change(3); getch();
A.spot : : ~spot();
getch();
B.spot : : ~spot();
closegraph();
```

2.3. Виртуальные функции

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы, включающие такие функции, называются **полиморфными** и играют особую роль в ООП.

Рассмотрим, как ведут себя при наследовании невиртуальные компонентные функции с одинаковыми именами, типами и сигнатурами параметров.

Пример 2.5

```
class base
{
public :
  void print(){cout<<“\nbase”;}
```

```
};
class dir : public base
{
public:
  void print(){cout<<“\ndir”;}
};
void main()
{
  base B,*bp = &B;
  dir D,*dp = &D;
  base *p = &D;
  bp –>print(); // base
  dp –>print(); // dir
  p –>print(); // base
}
```

В последнем случае вызывается функция print базового класса, хотя указатель p настроен на объект производного класса. Дело в том, что выбор нужной функции выполняется при компиляции программы и определяется типом указателя, а не его значением. Такой режим называется **ранним** или **статическим связыванием**.

Пример 2.6

```
//птицы
class bird{
//...
public:
void fly()const{cout<<"fly"<<endl;}          //может летать
};
//пингвин
class penguin:public bird{
//...
public:
void fly()const{cout<<"nofly"<<endl;}        //не летает
```

```
};
int main()
{bird b; penguin p;
b.fly();                                    //летит
p.fly();                                    //не летит
return 0;
}
```

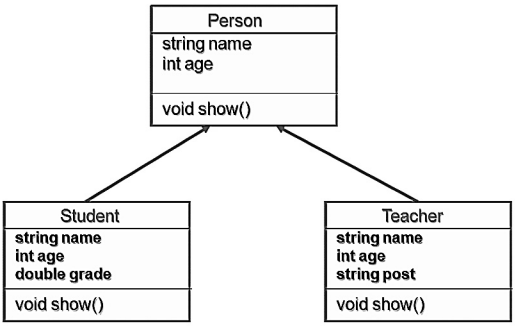
Выполним эту программу. Все нормально: птицы летают, но пингвин не летает.

Усложним задание. Добавим функцию alarm() – сигнал «тревога», который поступает всем птицам, в том числе и пингвину. Реакция птиц на сигнал «тревога» : убежать, улететь.

```
//Сигнал тревоги
void alarm(const bird& b)
{
b.fly();
}
int main()
{
bird b;
penguin p;
b.fly();            //летит
p.fly();            //не летит
alarm(b);           //полетела
alarm(p);           //как ни странно, и пингвин полетел ???
return 0;
}
```

Выполним эту программу. Видим, что все птицы, в том числе и пингвин, летят.

Рассмотрим, почему так происходит. На рис. 2.1 приведена мерархия классов.



21

Рис. 2.1

```
void person::show()
{
cout<<name<<“ “<<age<<endl;
}
void student::show()
{
cout<<name<<“ “<<age<<“ “<<grade<<endl;
}
void teacher::show()
{
cout<<name<<“ “<<age<<“ “<<post<<endl;
}
void main()
{
person* p;
p=new person(“Иванов”,35);
p->show();  //  Что будет выведено?
p=new student(“Петров”,21,75.8);
p->show();  // Что будет выведено?
p=new teacher(“Поляков”,51,”Декан”);
p->show();  // Что будет выведено?
```

```
return 0;
}
```
Ожидаем следующее:
```
void main()
{
person* p;
p=new person(“Иванов”,35);
p->show();  //  Иванов 25
p=new student(“Петров”,21,75.8);
p->show();  //  Петров 21 75.5
p=new teacher(“Поляков”,51,”Декан”);
p->show();  //  Поляков 51 Декан
return 0;
}
```
А на самом деле выведено
```
void main()
{
person* p;
p=new person(“Иванов”,35);
p->show();  //  Иванов 25
p=new student(“Петров”,21,75.8);
p->show();  //  Петров 21
p=new teacher(“Поляков”,51,”Декан”);
p->show();  //  Поляков 51
return 0;
}
```
Это происходит потому, что вызывается функция show() базового класса, хотя указатель «p» настроен на объект производного класса.
```
void alarm(const bird& b)
{ b.fly(); }
person* p;
p->show();
```

Дело в том, что выбор нужной функции выполняется при компиляции программы и определяется типом указателя, а не его значением.

Как мы уже знаем, такой режим называется ранним или статическим связыванием.

Большую гибкость обеспечивает **позднее** (**отложенное**) или **динамическое связывание**, которое предоставляется механизмом **виртуальных функций**. Любая нестатическая функция базового класса может быть сделана виртуальной, для чего используется ключевое слово **virtual**.

Пример 2.7

```
class base
{
public:
  virtual void print(){cout<<“\nbase”;}
. . .
};
// и так далее – см. предыдущий пример.
```

В этом случае будет напечатано

```
base
dir
dir
```

Пример 2.8

```
class person
{
protected:
string name;
int age;
public:
person(string  Name=“Noname”,int Age=0):
name(Name),age(Age){}
virtual void show();
```

```
};
class student:public person
{
protected:
double grade;
public:
student(string Name=“Noname”, int Age=0, double
Grade=0.0):
person(Name,Age),grade(Grade){}
virtual void show();
};
```

Таким образом, интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя, т.е. от типа объекта, для которого выполняется вызов.

Виртуальные функции – это функции, объявленные в базовом классе и переопределенные в производных классах. Иерархия классов, которая определена открытым наследованием, создает родственный набор пользовательских типов, на все объекты которых может указывать указатель базового класса. Выбор того, какую виртуальную функцию вызвать, будет зависеть от типа объекта, на который фактически (в момент выполнения программы) направлен указатель, а не от типа указателя.

Виртуальными могут быть только нестатические функции-члены.

Виртуальность наследуется. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор virtual может не использоваться.

Конструкторы не могут быть виртуальными в отличие от деструкторов. Практически каждый класс, имеющий виртуальную функцию, должен иметь виртуальный деструктор.

Если в производном классе ввести функцию с тем же именем и типом, но с другой сигнатурой параметров, то эта функция производного класса не будет виртуальной.

Виртуальная функция может быть дружественной в другом классе.

Механизм виртуального вызова может быть подавлен с помощью явного использования полного квалифицированного имени (рис. 2.2).

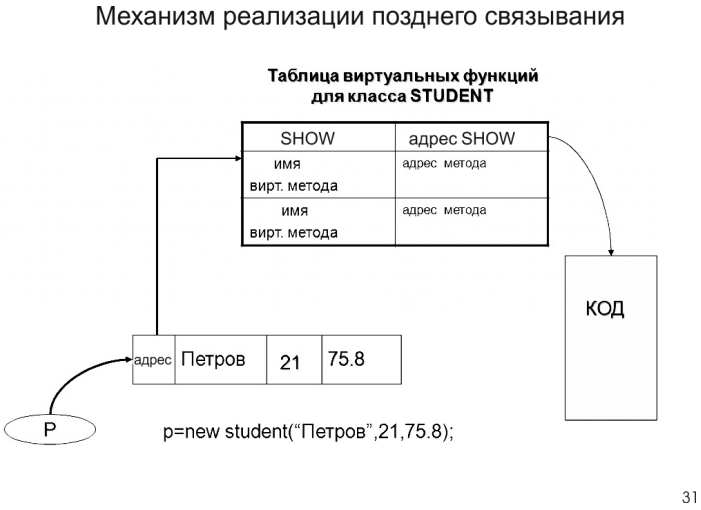


Рис. 2.2

В Visual Studio C++ .NET для переопределения виртуальной функции вы можете использовать ключевое слово **override**

```
class person
{
public:
virtual void show();
};
class student:public person
```

```
{
public:
void show()override;
};
```

2.4. Абстрактные классы

Абстрактным называется класс, в котором есть хотя бы одна чистая (пустая) виртуальная функция.

Чистой виртуальной называется компонентная функция, которая имеет следующее определение:

virtual тип имя_функции(список_формальных_параметров) = 0;

Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах. Абстрактный класс может использоваться только в качестве базового для производных классов.

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. При этом построение иерархии классов выполняется по следующей схеме. Во главе иерархии стоит абстрактный базовый класс. Он используется для наследования интерфейса. Производные классы будут конкретизировать и реализовать этот интерфейс. В абстрактном классе объявлены чистые виртуальные функции, которые ,по сути, есть **абстрактные методы**.

Пример 2.9

```
class Base{
public:
Base();                  // конструктор по умолчанию
Base(const Base&);  // конструктор копирования
virtual ~Base();         // виртуальный деструктор
```

```
virtual void Show()=0;     //  чистая виртуальная функция
// другие чистые виртуальные функции
protected:
// защищенные члены класса
private:

};
class Derived: virtual public Base{
public:
Derived();                      // конструктор по умолчанию
Derived(const Derived&);  // конструктор копирования
Derived(параметры);         // конструктор с параметрами
virtual ~Derived();             // виртуальный деструктор
void  Show();                   // переопределенная виртуальная
функция
// другие переопределенные виртуальные функции
// перегруженная операция присваивания
Derived& operator=(const Derived&);
// ее смысл будет понятен после прочтения главы 3
// другие перегруженные операции
protected:
// используется вместо private, если ожидается наследование
private:
// используется для деталей реализации
};
```

По сравнению с обычными классами абстрактные классы пользуются «ограниченными правами». А именно:

– невозможно создать объект абстрактного класса;

– абстрактный класс нельзя употреблять для задания типа параметра функции или типа возвращаемого функцией значения;

– абстрактный класс нельзя использовать при явном приведении типов; в то же время можно определить указатели и ссылки на абстрактный класс.

Объект абстрактного класса не может быть формальным параметром функции, однако формальным параметром может быть указатель на абстрактный класс. В этом случае появляется возможность передавать в вызываемую функцию в качестве фактического параметра значение указателя на производный объект, заменяя им указатель на абстрактный базовый класс. Таким образом мы получаем **полиморфные объекты**.

В *Visual Studio C++ .NET* для определения абстрактных функций и классов вы можете использовать ключевое слово abctract.

```
class person  abstract
{
protected:
string name;
int age;
public:
person(string Name,int Age): name(Name), age(Age){}
virtual void show()const abstract;
};
```

2.5. Включение объектов

Есть два варианта включения объекта типа X в класс A:

1) Объявить в классе A член типа X;

```
class A{
X x;
//...
};
```

2) Объявить в классе A член типа X* или X&.

```
class A{
X* p;
X& r;
};
```

Предпочтительно включать собственно объект, как в первом случае. Это эффективнее и меньше подвержено ошибкам, так как связь между содержащимся и содержащим объектами описывается правилами конструирования и уничтожения.

Например,

```
// Персона
class person{
char* name:
public:
person(char*);
//...
};
//Школа
class school{
person head;  //директор
public:
school(char* name):head(name){}
//...
};
```

Второй вариант с указателем можно применять тогда, когда за время жизни «содержащего» объекта нужно изменить указатель на «содержащийся» объект.

Например,

```
class school{
person* head;  //директор
public:
school(char* name):head(new person(name)){}
~school{delete head;}
person* change(char * newname){
person* temp=head;
head=new person(newname);
return temp;}
//...
};
```

Второй вариант можно использовать, когда требуется задавать «содержащийся» объект в качестве аргумента.

Например,

```
class school{
person* head;  //директор
public:
school(person* q):head(q){}
//...
};
```

Имея объекты, включающие другие объекты, мы создаем **иерархию объектов.** Она является альтернативой и дополнением к иерархии классов. А как быть в том случае, когда количество включаемых объектов заранее неизвестно и (или) может изменяться за время жизни «содержащего» объекта. Например, если объект school содержит учеников, то их количество может меняться.

Существуют два способа решения этой проблемы. *Первый* состоит в том, что организуется связанный список включенных объектов, а «содержащий» объект имеет член-указатель на начало этого списка.

Например,

```
class person{
char* name;
person* next;
...
};
class school{
person* head;     // указатель на директора школы
person* begin;     // указатель на начало списка учеников
public:
shool(char* name):head(new person(name)),begin(NULL){}
~shool();
void add(person* ob);
//...
};
```

В этом случае при создании объекта school создается пустой список включенных объектов. Для включения объекта вызывается метод add(), которому в качестве параметра передается указатель на включаемый объект. Деструктор последовательно удаляет все включенные объекты. Объект person содержит поле next, которое позволяет связать объекты в список. Законченная программа, демонстрирующая этот способ включения, приведена в упражнении.

Второй способ заключается в использовании специального объекта-контейнера.

Контейнерный класс предназначен для хранения объектов и представляет собой простые и удобные способы доступа к ним.

```
class school{
person* head;
container pupil;
...
};
```

Здесь pupil – контейнер, содержащий учеников. Все, что необходимо для добавления, удаления, просмотра и т.д. включенных объектов, должно содержаться в методах класса container. Примером могут служить контейнеры стандартной библиотеки шаблонов (STL) C++.

Рассмотрим отношения между наследованием и включением.

2.6. Включение и наследование

Пусть класс B есть производный класс от класса D.

```
class B{...};
class D:public B{...};
```

Слово public в заголовке класса D говорит об открытом наследовании. Открытое наследование означает, что производный класс D является подтипом класса B, т.е. объект D является и объектом B. Такое наследование является отношением **is-a**

или говорят, что D **есть разновидность** B. Иногда его называют также **интерфейсным наследованием.** При открытом наследовании переменная производного класса может рассматривается как переменная типа базового класса. Указатель, тип которого – «указатель на базовый класс», может указывать на объекты, имеющие тип производного класса. Используя наследование, мы строим **иерархию классов.**

Рассмотрим следующую иерархию классов:

```
class person{
protected:
char* name;
int age;
public:
person(char*,int);
virtual void show() const;
//...
};
class employee:public person{
protected:
int work;
public:
employee)char*,int,int);
void show() const;
//...
};
class teacher:public employee{
protected:
int teacher_work;
public:
teacher(char*,int,int,int);
void show() const;
//...
};
```

Определим указатели на объекты этих классов.

```
person* pp;
teacher* pt;
```

Создадим объекты этих классов.

```
person a(“Петров”,25);
employee b(“Королев”,30.10);
pt=new teacher(“Тимофеев”,45.23,15);
```

Просмотрим эти объекты.

```
pp=&a;
pp->show(); //вызывает person::show для объекта a
pp=&b;
pp->show(); // вызывает employee::show для объекта b
pp=pt;
pp->show(); // вызывает teacher::show для объекта *pt
```

Здесь указатель базового класса pp указывает на объекты производных классов employee, teacher, т.е. он совместим по присваиванию с указателями на объекты этих классов. При вызове функции show с помощью указателя pp вызывается функция show того класса, на объект которого фактически указывает pp. Это достигается за счет объявления функции show виртуальной, в результате чего мы имеем позднее связывание.

Пусть теперь класс D имеет член класса B.

```
class D{
public:
B b;
//...
};
```

В свою очередь, класс B имеет член класса C.

```
class B{
public:
C c;
//...
};
```

Такое включение называют отношением **has-a**. Используя включение, мы строим **иерархию объектов.**

На практике возникает проблема выбора между наследованием и включением. Рассмотрим классы «Самолет» и «Двигатель». Новичкам часто приходит в голову сделать «Самолет» производным от «Двигатель». Это неверно, поскольку самолет **не является** двигателем, он **имеет** двигатель. Один из способов увидеть это – задуматься, может ли самолет иметь несколько двигателей? Поскольку это возможно, нам следует использовать включение, а не наследование.

Рассмотрим следующий пример:

```
class B{
public:
virtual void f();
void g(); };
class D{
public:
B b;
void f(); };
void h(D* pd){
B* pb;
pb=pd;            // #1  Ошибка
pb->g();  // #2   вызывается  B::g()
pd->g();  // #3  Ошибка
pd->b.g();          // #4   вызывается  B::g()
pb->f();   // #5   вызывается  B::f()
pd->f();   // #6   вызывается  D::f()
}
```

Почему в строках #1 и #3 ошибки ?

В строке #1 нет преобразования D* в B*.

В строке #3 D не имеет члена g().

В отличие от открытого наследования не существует неявного преобразования из класса в один из его членов, и класс, содержащий член другого класса, не замещает виртуальных функций того класса.

Если для класса D использовать открытое наследование

```
class D:public B{
public:
void f();};
```

то функция

```
void h(D* pd){
B* pb=pd;
pb->g();  // вызывается  B::g()
pd->g();  // вызывается  B::g()
pb->f();   // вызывается  D::f()
pd->f();   // вызывается  D::f()
}
```

не содержит ошибок.

Поскольку D является производным классом от B, то выполняется неявное преобразование из D в B. Следствием является возросшая зависимость между B и D.

Существуют случаи, когда вам нравится наследование, но вы не можете позволить таких преобразований.

Например, мы хотим повторно использовать код базового класса, но не предполагаем рассматривать объекты производного класса как экземпляры базового. Все, что мы хотим от наследования – это повторное использование кода. Решением здесь является **закрытое** наследование. Закрытое наследование не носит характера отношения подтипов или отношения is-a. Мы будем называть его отношением **like-a**(подобный) или **наследованием реализации** в противоположность наследованию интерфейса. Закрытое (так же, как и защищенное) наследование не создает иерархии типов.

С точки зрения проектирования закрытое наследование равносильно включению, если не считать вопроса с замещением функций. Важное применение такого подхода – открытое наследование из абстрактного класса и одновременно закрытое (или защищенное) наследование от конкретного класса для представления реализации.

Пример 2.10. Бинарное дерево поиска

```
//Файл  tree.h
// Обобщенное дерево
typedef void* Tp; //тип обобщенного указателя
int comp(Tp a,Tp b);
class node{ //узел
private:
friend class tree;
node* left;
node* right;
Tp data;
int count;
node(Tp d,Tp* l,Tp*r):data(d),left(l),right(r),count(1){}
friend void print(node* n);
};
class tree{//дерево
public:
tree(){root=0;}
void insert(Tp d);
Tp find(Tp d) const{return(find(root,d));}
void print() const{print(root);}
protected:
node* root; //корень
Tp find(node* r,Tp d) const;
void print(node* r) const;
};
```

Узлы двоичного дерева содержат обобщенный указатель на данные data. Он будет соответствовать типу указателя в производном классе. Поле count содержит число повторяющихся вхождений данных. Для конкретного производного класса мы должны написать функцию comp для сравнения значений конкретного производного типа. Функция insert() помещает узлы в дерево.

```
void tree::insert(TP d)
{node* temp=root;
node* old;
if(root==0){root=new node(d,0,0);return;}
while(temp!=0){
old=temp;
if(comp(temp->data,d)==0){(temp->count)++;return;}
if(comp(temp->data,d)>0)temp=temp->left;
 else temp=temp->right;}
it(comp(old->data,d)>0)old->left=new(d,0,0);
 else old->right=new node(d,0,0);
}
```

Функция Tp find(node* r,Tp d) ищет в поддереве с корнем r информацию, представленную d.

```
Tp tree::find(node* r,Tp d)const
{if(r==0) return 0;
   else if(comp(r->data,d)==0)return(r->data);
         else if (comp(r->data,d)>0)return(find(r->left,d));
               else return(find(r->right,d));
}
```

Функция print() – стандартная рекурсия для обхода бинарного дерева

```
void tree::print(node* r) const
{if(r!=0){
  print(r->left);
  ::print(r);
  print(r->right);
}
```

В каждом узле применяется внешняя функция ::print().

Теперь создадим производный класс, который в качестве членов данных хранит указатели на char.

```
//Файл s_tree.cpp
#include “tree.h”
#include <string.h>
```

```
class s_tree:private tree{
public:
s_tree(){}
void insert(char* d){tree::insert(d);}
char* find(char* d) const {return(char*)tree::find(d));
voif print() const{tree::print();}
};
```

В классе s_tree функция insert использует неявное преобразование char* к void*.

Функция сравнения comp выглядит следующим образом:

```
int comp(Tp a,Tp b)
{return(strcmp((char*)a,(char*)b));}
```

Для вывода значений, хранящихся в узле, используется внешняя функция:

```
print(node* n){
cout<<(char*)(n->data)<<endl;
cout<<n->cout<<endl;
}
```

Здесь для явного приведения типа void* к char* мы используем операцию приведения типа *(имя_типа)выражение.* Более надежным является использование оператора **static_cast**<char*>(Tp)

2.7. Множественное наследование

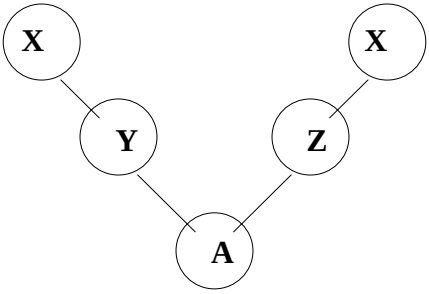
Класс может иметь несколько непосредственных базовых классов:

```
class A1{. . .};
class A2{. . .};
class A3{. . .};
class B : public A1,public A2,public A3{. . .};
```

Такое наследование называется **множественным**. При множественном наследовании никакой класс не может больше одного раза использоваться в качестве непосредственного базового. Однако класс может больше одного раза быть непрямым базовым классом.

```
class X{. . . f(); . . .};
class Y : public X{. . .};
class Z : public X{. . .};
class A : public Y,public Z{. . .};
```

Имеем следующую иерархию классов (и объектов):



Такое дублирование класса соответствует включению в производный объект нескольких объектов базового класса. В этом примере существуют два объекта класса X. Для устранения возможных неоднозначностей нужно обращаться к конкретному компоненту класса X, используя полную квалификацию

Y : : X : : f() *или* Z : : X : : f)

Пример

```
class circ  // окружность
{
  int x,y,r;
public:
  circ(int x1,int y1,int r1){x = x1; y = y1; r = r1;}
  void show();
. . .
};
class square // квадрат
{
  int x,y,l;
  // x, y – координаты центра
  // l – длина стороны
```

```
public:
  square(int x1,int y1,int l1){x = x1; y = y1; l = l1;}
  void show();
  . . .
};
```

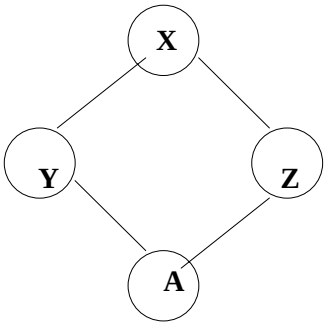
class **cirlsqrt** : public circ,public square // окружность, вписанная в квадрат

```
{
public:
  cirlsqrt(int x1,int y1,int r1) : circ(x1,y1,r1),square(x1,y1,2*r1){. . .}
  void show()
     {circ : : show();
       square : : show();
     }
  . . .
};
```

Чтобы устранить дублирование объектов непрямого базового класса при множественном наследовании, этот базовый класс объявляют **виртуальным**.

```
class X{. . .};
class Y : virtual public X{. . .};
class Z : virtual public X{. . .};
class A : public Y,public Z{. . .};
```

Теперь класс A будет включать только один экземпляр X, доступ к которому равноправно имеют классы Y и Z.



Пример 2.11

```
class base
  {int x; char c,v[10]; . . .};
class abase : public virtual base
  {double y; . . .};
class bbase : public virtual base
  {float f; . . .};
class top : public abase, public bbase
  {long t; . . .};
void main()
{cout<<sizeof(base)<< endl;
  cout<<sizeof(abase)<< endl;
  cout<<sizeof(bbase)<<endl;
  cout<<sizeof(top)<< endl;
}
```

Здесь

♦ объект класса **base** занимает в памяти 13 байт:
2 байта – поле int;
1 байт – поле char;
10 байт – поле char[10].

♦ объект класса abase занимает в памяти 23 байта:
8 байт – поле double;
13 байт – поля базового класса base;
2 байта – для связи в иерархии виртуальных классов;

♦ объект класса bbase занимает в памяти 19 байт:
4 байта – поле float;
13 байт – поля базового класса base;
2 байта для связи в иерархии виртуальных классов;

♦ объект класса top занимает в памяти 33 байта:
4 байта поле long;
10 байт – данные и связи abase;
6 байт – данные и связи bbase;
13 байт – поля базового класса base.

Если при наследовании base в классах abase и bbase базовый класс сделать невиртуальным, то результаты будут такими:

♦ объект класса **base** занимает в памяти 13 байт;

♦ объект класса abase занимает в памяти 21 байт (нет 2 байт для связи);

♦ объект класса bbase занимает в памяти 17 байт (нет 2 байт для связи);

♦ объект класса top занимает в памяти 42 байта (объект base входит дважды).

2.8. Локальные и вложенные классы

Класс может быть объявлен внутри блока, например внутри определения функции. Такой класс называется **локальным**. Локализация класса предполагает недоступность его компонентов вне области определения класса (вне блока).

Локальный класс не может иметь статических данных, так как компоненты локального класса не могут быть определены вне текста класса.

Внутри локального класса разрешено использовать из объемлющей его области только имена типов, статические (static) переменные, внешние (extern) переменные, внешние функции и элементы перечислений. Из того, что запрещено, важно отметить переменные автоматической памяти. Существует еще одно важное ограничение для локальных классов – их компонентные функции могут быть только inline.

Внутри класса разрешается определять типы, следовательно, один класс может быть описан внутри другого. Такой класс называется **вложенным**. Вложенный класс является локальным для класса, в рамках которого он описан, и на него распространяются те правила использования локального класса, о которых говорилось выше. Следует особо сказать, что вложенный класс не имеет никакого особого права доступа к членам охватывающего класса, то есть он может обращаться к ним

только через объект типа этого класса (так же, как и охватывающий класс не имеет каких-либо особых прав доступа к вложенному классу).

Пример 2.12

```
int i;
class global{
static int n;
public:
int i;
static float f;
 class intern{
 void func(global& glob)
  {i=3;      // Ошибка: используется имя нестатического данного
             // из охватывающего класса
    f=3.5;  // Правильно: f-статическая функция
    ::i=3;   // Правильно: i-внешняя (по отношению к классу)
                        // переменная
    glob.i=3;// Правильно: обращение к членам охватывающего
                    // класса через объект этого класса
    n=3;     // Ошибка: обращение к private-члену охватывающего
           //класса
  } }; };
```

Пример 2.13. Класс «ПРЯМОУГОЛЬНИК»

Определим класс «прямоугольник». Внутри этого класса определим класс как вложенный класс «отрезок». Прямоугольник будет строится из отрезков.

```
#include <conio.h>
#include <graphics.h>
// точка
class point{
protected:
int x,y;
```

```
public:
point(int x1=0,int y1=0):x(x1),y(y1){}
int& getx(){return x;}
int& gety(){return y;}
};
// прямоугольник
class rect{
// вложенный клас “отрезок”
class segment{
point a,b; //начало и конец отрезка
public:
segment(point a1=point(0,0),point b1=point(0,0))
{a.getx()=a1.getx();
 a.gety()=a1.gety();
 b.getx()=b1.getx();
 b.gety()=b1.gety();}
 point& beg(){return a;}
 point& end(){return b;}
 void Show()        //показать отрезок
 {line(a.getx(),a.gety(),b.getx(),b.gety());}
};   //конец определения класса segment
segment ab,bc,cd,da;   //стороны прямоугольника
public:
rect(point c1=point(0,0),int d1=0,int d2=0)
{point a,b,c,d;   //координаты вершин
 a.getx()=c1.getx();
 a.gety()=c1.gety();
 b.getx()=c1.getx()+d1;
 b.gety()=c1.gety();
 c.getx()=c1.getx()+d1;
 c.gety()=c1.gety()+d2;
 d.getx()=c1.getx();
 d.gety()=c1.gety()+d2;
//граничные точки отрезков
```

```
ab.beg()=a;  ab.end()=b;
bc.beg()=b;  bc.end()=c;
cd.beg()=c;  cd.end()=d;
da.beg()=d;  da.end()=a;}
void Show()     //пока прямоугольник
{ab.Show();
 bc.Show();
 cd.Show();
 da.Show();}
};   //конец определения класса rect

void main()
{int dr=DETECT,mod;
 initgraph(&dr,&mod,"C:\\tc3\\bgi");
 point p1(120,80);
 point p2(250,240);
 rect A(p1,80,30);
 rect B(p2,100,200);
 A.Show();getch();
 B.Show();getch();
 closegraph();
}
```

Используя эту методику, можно определить любую геометрическую фигуру, состоящую из отрезков прямых.

Пример 2.14. Класс «СТРОКА»

Класс **string** хранит строку в виде массива символов с завершающим нулем в стиле Си и использует механизм подсчета ссылок для минимизации операций копирования.

Класс string пользуется тремя вспомогательными классами:

– **srep**, который позволяет разделять действительное представление между несколькими объектами типа string с одинаковыми значениями;

– **range** – для генерации исключения в случае выхода за пределы диапазона;

– **cref** – для реализации операции индексирования, который различает операции *чтения* и *записи.*

```
class string{
struct srep;
srep* rep;
public:
class cref; //ссылка на char
class range{};
//…
};
```

Так же, как и другие члены, вложенный класс может быть объявлен в самом классе, а определен позднее:

```
struct string::srep{
char *s; //указатель на элементы
int sz;   //количество символов
int n;    //количество обращений
srep(const char *p)
{n=1;
sz=strlen(p);
s=new char[sz+1];
strcpy(s,p);
}
~srep(){delete[]s;}
srep *get_copy()   //сделать копию,если необходимо
{if(n==1)return this;
n--;
return new srep(s);}
void assign(const char *p)
{if(strlen(p)!=sz){delete[]s;
sz=strlen(p);
s=new char[sz+1];}
strcpy(s,p);}
```

private: //предохраняет от копирования
sper(const srep&);
srep& **operator=**(const srep&);
}

2.9. Пример программы для Microsoft Visual Studio

Задание: создать иерархию классов – персона, студент, преподаватель и класс «список», хранящий объекты этих классов.

Файл list.h – Определение классов

Абстрактный класс Person содержит абстрактные (чистые виртуальные) функции Show() и Input(), которые должны быть определены в производных классах. В классе Person объявлен дружественный ему класс List (рис. 2.3).

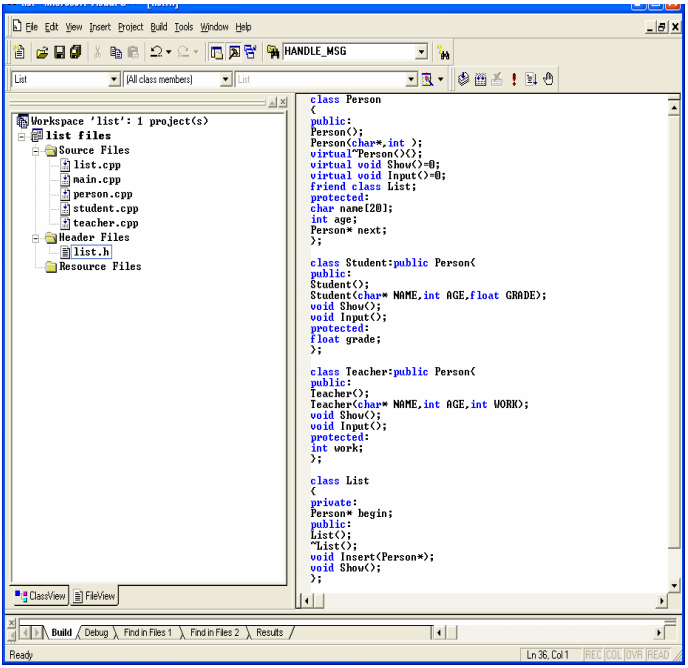


Рис. 2.3

```
class Person
{
public:
Person();
Person(char*,int );
virtual~Person(){};
virtual void Show()=0; //показать объект
virtual void Input()=0;  //ввести значения данных объекта
friend class List;
protected:
char name[20];  //имя персоны
int age;            //возраст персоны
Person* next;    //указатель на следующий объект в списке
};

class Student:public Person{
public:
Student();
Student(char* NAME,int AGE,float GRADE);
void Show();
void Input();
protected:
float grade; //рейтинг
};

class Teacher:public Person{
public:
Teacher();
Teacher(char* NAME,int AGE,int WORK);
void Show();
void Input();
protected:
int work; //рабочий стаж
};
class List
```

```
{
private:
Person* begin; //указатель на начало списка
public:
List();
~List();
void Insert(Person*); //вставить в список объект
void Show(); //показать весь список
};
```

Файл person.cpp – Определение функций класс Person *(рис. 2.4)*

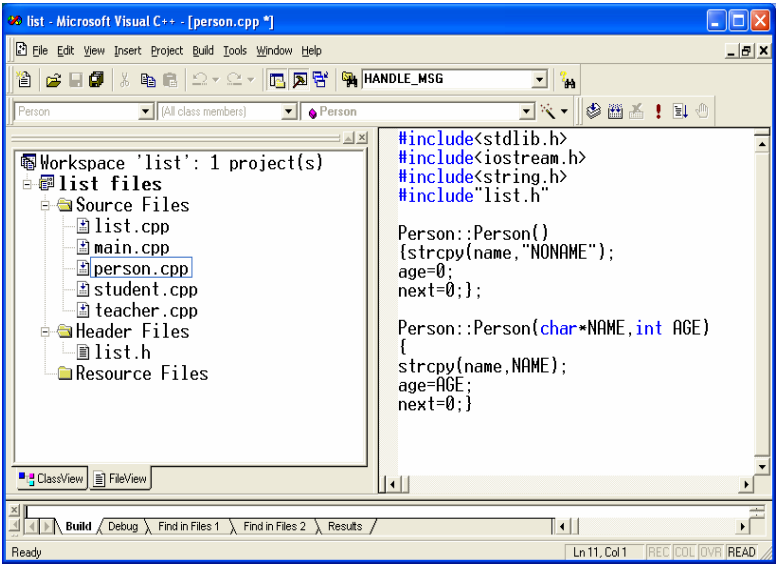


Рис. 2.4

```
#include<stdlib.h>
#include<iostream.h>
#include<string.h>
#include"list.h"
```

```
Person::Person()
{strcpy(name,"NONAME");
age=0;
next=0;};

Person::Person(char*NAME,int AGE)
{
strcpy(name,NAME);
age=AGE;
next=0;}
```

Файл student.cpp – Определение функций класса Student *(рис. 2.5)*

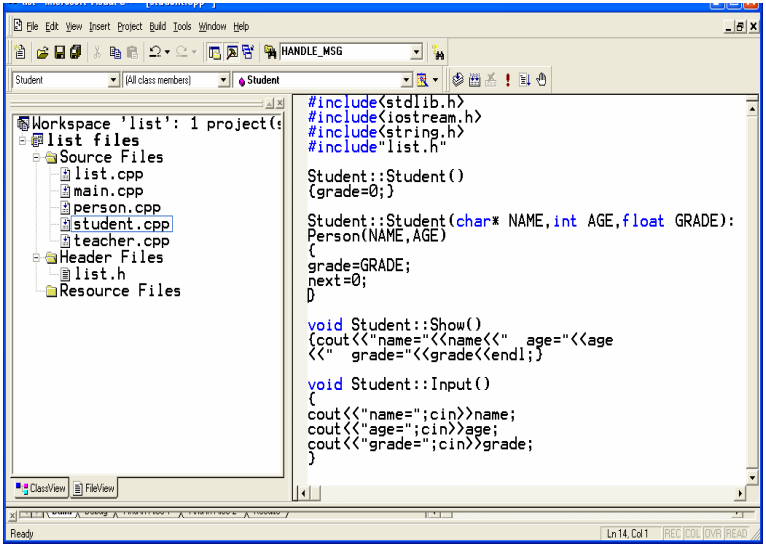


Рис. 2.5

```
#include<stdlib.h>
#include<iostream.h>
#include<string.h>
#include"list.h"
```

```
Student::Student()
{grade=0;}

Student::Student(char* NAME,int AGE,float GRADE)
:Person(NAME,AGE)
{
grade=GRADE;
next=0;
}

void Student::Show(){cout<<"name="<<name<<"
age="<<age<<"  grade="<<grade<<endl;}

void Student::Input()
{
cout<<"name=";cin>>name;
cout<<"age=";cin>>age;
cout<<"grade=";cin>>grade;
}
```

Файл teacher.cpp– Определение функций класса Teacher *(рис. 2.6)*

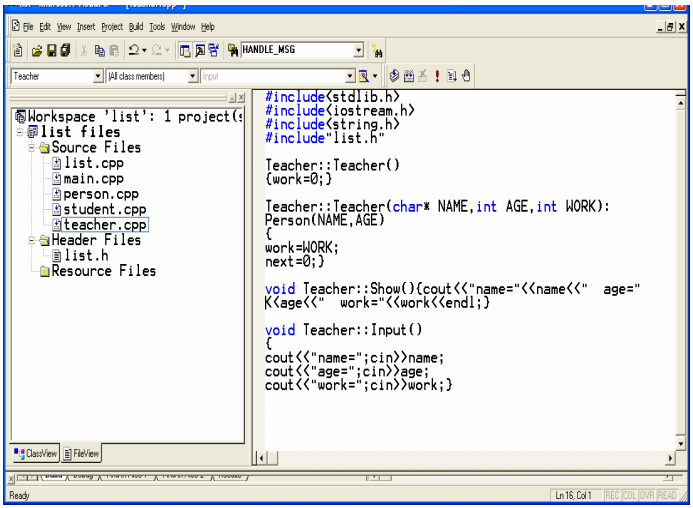


Рис. 2.6

```
#include<stdlib.h>
#include<iostream.h>
#include<string.h>
#include"list.h"

Teacher::Teacher()
{work=0;}

Teacher::Teacher(char* NAME,int AGE,int
WORK):Person(NAME,AGE)
{
work=WORK;
next=0;
}

void Teacher::Show(){cout<<"name="<<name<<"
age="<<age<<"  work="<<work<<endl;}
void Teacher::Input()
{
cout<<"name=";cin>>name;
cout<<"age=";cin>>age;
cout<<"work=";cin>>work;
}
```

Файл list.cpp – Определение функций класса List *(рис. 2.7)*

```
#include<stdlib.h>
#include<iostream.h>
#include<string.h>
#include"list.h"
List::List(){begin=0;} //создается пустой список
List::~List(){
Person*r;
//проходим по списку и удаляем каждый объект
```

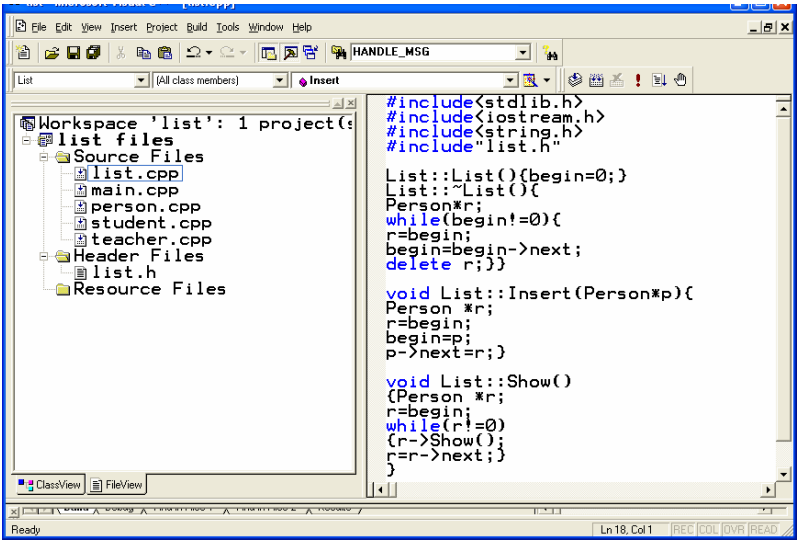


Рис. 2.7

```
while(begin!=0){
r=begin;
begin=begin->next;
delete r;}}

void List::Insert(Person*p){
Person *r;
//вставляем объект в начало списка
r=begin;
begin=p;
p->next=r;}

void List::Show()
{Person *r;
r=begin;
//проходим по списку и вызываем для каждого объекта
//его виртуальную  функцию Show
```

```
while(r!=0)
{r->Show();
r=r->next;}
}
```

Файл main.cpp – Демонстрационная программа *(рис. 2.8)*

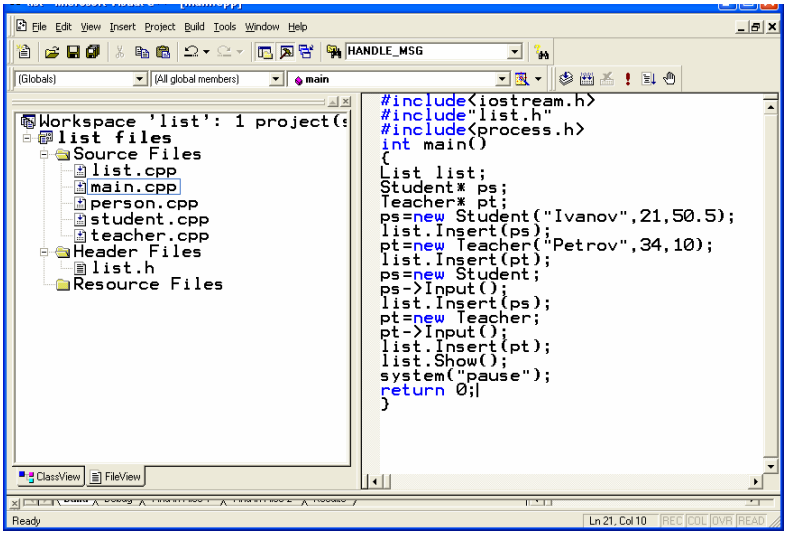


Рис. 2.8

```
#include<iostream.h>
#include"list.h"
#include<process.h>
int main()
{
List list;
Student* ps;
Teacher* pt;
ps=new Student("Ivanov",21,50.5);
list.Insert(ps);
pt=new Teacher("Petrov",34,10);
```

```
list.Insert(pt);
ps=new Student;
ps->Input();
list.Insert(ps);
pt=new Teacher;
pt->Input();
list.Insert(pt);
list.Show();
system("pause");
return 0;
}
```

2.10. Упражнения

Упражнение 1. В этом упражнении мы покажем, что функции-члены базового и производного классов не составляют множества перегруженных функций, т.е. функция set класса CDerived скрывает видимость функции-члена CBase::set, а не перегружает ее. Вызов функции-члена базового класса из производного в этом случае приводит к ошибке при компиляции.

В среде Microsoft Visual C++ создайте следующую программу и попытайтесь ее откомпилировать:

```
#include<iostream.h>
class CBase
{
protected:
  int x;
public:
  CBase(int);
  void print();
  void set(int);
};
```

```
CBase::CBase(int X):x(X){}
void CBase::print(){cout<<x<<endl;}
void CBase::set(int X){x=X;}

class CDerived:public CBase
{
protected:
  int y;
public:
  CDerived(int,int);
  void print();
  void set(int,int);
};
CDerived::CDerived(int X,int Y):CBase(X),y(Y){}
void CDerived::print(){cout<<x<<' '<<y<<endl;}
void CDerived::set(int X,int Y){x=X;y=Y;}

int main()
{
CDerived ob(5,6);
ob.set(7,8);
ob.print();
ob.set(9);// Ошибка компиляции
ob.print();
ob.CBase::print();
return 0;
}
```

При компиляции выделенной строки вы получите сообщение:

```
Compiling...
error C2660: 'set' : function does not take 1 parameters
Error executing cl.exe.
```

Замените ошибочную строку строкой b.set(9);

на ob.CBase::set(9); откомпилируйте и выполните программу, и вы увидите, что она выполняется без ошибки.

Результат выполнения посмотрите здесь inh1.exe

Другой способ решения этой проблемы – написать в производном классе небольшую встроенную функцию заглушку для вызова функции базового класса.

```
class CDerived:public CBase
{
protected:
  int y;
public:
  CDerived(int,int);
  void print();
  void set(int,int);
  void bset(int i){CBase::set(i);}  //Функция-заглушка
};
```

Выполнив

```
int main()
{
CDerived ob(5,6);
ob.set(7,8);
ob.print();
ob.bset(9); //
ob.print();
ob.CBase::print();
system("pause");
return 0;
}
```

Вызов ob.bset(9); дает тот же результат, что и ob.CBase::set(9);

Упражнение 2. В этом упражнении мы покажем использование виртуальных функций.

Создадим иерархию классов: «птицы» имеют свойство «летать» и наследуемый класс «пингвин», который летать не умеет.

```
#include<iostream.h>
//птицы
class bird{
//...
public:
void fly()const{cout<<"fly"<<endl;} //может летать
//Если не const, то при вызове в функции alarm()  b.fly();
// сообщение
//cannot convert 'this' pointer from 'const class bird' to 'class bird &'
};
//пингвин
class penguin:public bird{
//...
  public:
  // Переопределяем, чтобы пингвин не летал
  void fly()const{cout<<"nofly"<<endl;}
};

int main()
{
bird b;
penguin p;
b.fly();//летит
p.fly();//не летит
return 0;
}
```

Создадим и выполним эту программу. Все нормально: птицы летают, но пингвин не летает.

Усложним задание. Добавим функцию alarm() – сигнал «тревога», который поступает всем птицам, в том числе и пингвину. Реакция птиц на сигнал «тревога» : убежать, улететь.

```
#include<iostream.h>
//птицы
class bird{
//...
public:
void fly()const{cout<<"fly"<<endl; }
};
//пингвин
class penguin:public bird{
//...
  public:
  void fly()const{cout<<"nofly"<<endl;}
};
//Сигнал тревоги
void alarm(const bird& b)
{
b.fly();
}

int main()
{
bird b;
penguin p;
b.fly();//летит
p.fly();//не летит
alarm(b);//полетела
alarm(p);//как ни странно, и пингвин полетел
return 0;
}
```

Создадим и выполним эту программу. Видим, что все птицы, в том числе и пингвин, летят.

Сделаем функцию fly() виртуальной.

```
virtual    void fly()const{cout<<"fly"<<endl;
```

Создадим и выполним эту программу. Все нормально: птицы летают, но пингвин не летает.

```
int main()
{
bird b;
penguin p;
b.fly();//летит
p.fly();//не летит
alarm(b);//полетела
alarm(p);//не летит, а только бежит
return 0;
}
```

Более грамотным решением будет создание абстрактного класса «птицы» с абстрактной виртуальной функцией fly(). В конкретном классе птиц эта функция переопределяется должным образом.

```
#include<iostream.h>
// Абстрактный класс-птицы
class bird{
//...
public:
virtual     void fly()const=0;//Абстрактный метод-
//должен быть переопределен в производных классах
};
//ворона
class crow:public bird{
//...
public:
 //Ворона летает
  void fly()const{cout<<"fly"<<endl;}
```

```
};

//пингвин
class penguin:public bird{
//...
  public:
  //Пингвин не летает
  void fly()const{cout<<"nofly"<<endl;}
};
//Сигнал тревоги
void alarm(const bird& b)
{
b.fly();
}

int main()
{
crow c;
penguin p;
c.fly();       // летит
p.fly();      //не летит
alarm(c);   //полетела
alarm(p);  //не летит, а бежит
 return 0;
}
```

3. ПЕРЕГРУЗКА ОПЕРАЦИЙ

В языке C++ определены множества операций над переменными стандартных типов, такие как +, *, / и т.д. Каждую операцию можно применить к операндам определенного типа.

К сожалению, лишь ограниченное число типов непосредственно поддерживается любым языком программирования. Например, C и C++ не позволяют выполнять операции с комплексными числами, матрицами, строками, множествами. Однако все эти операции можно выполнить 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 объединения (+) и пересечения (*) множеств.

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

A*B = {a3}.

Можно определить класс Set – «множество» и определить операции над объектами этого класса, выразив их с помощью знаков операций, которые уже есть в языке C++, например, + и *. В результате операции + и * можно будет использовать, как и раньше, а также снабдить их дополнительными функциями (объединения и пересечения). Как определить, какую функцию должен выполнять оператор: старую или новую? Очень просто – по типу операндов. А как быть с приоритетом операций? Сохраняется определенный ранее приоритет операций. Для распространения действия операции на новые типы данных надо определить специальную функцию, называемую «операция-функция» (operator-function). Ее формат:

тип_возвр_значения operator *знак_операции*
(специф_параметров) {операторы_тела_функции}

При необходимости может добавляться и прототип:

тип_возвр_значения operator знак_операции (специф_параметров)

Если принять, что конструкция *operator знак_операции* есть имя некоторой функции, то прототип и определение операции-функции подобны прототипу и определению обычной функции языка С++. Определенная таким образом операция называется перегруженной (overload).

Чтобы была обеспечена явная связь с классом, операция-функция должна быть либо компонентом класса, либо она должна быть определена в классе как дружественная, и у нее должен быть хотя бы один параметр типа класс (или ссылка на класс). Вызов операции-функции осуществляется так же, как и любой другой функции С++: $operator\oplus$.Однако разрешается использовать сокращенную форму ее вызова: $a \oplus b$, где $\oplus$ – знак операции.

3.1. Перегрузка унарных операций

♦ Любая унарная операция ⊕ может быть определена двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно, дружественная) функция с одним параметром. В первом случае выражение ⊕ Z означает вызов Z.operator⊕(), во втором – вызов operator⊕(Z).

♦ Унарные операции, перегружаемые в рамках определенного класса, могут перегружаться только через нестатическую компонентную функцию без параметров. Вызываемый объект класса автоматически воспринимается как операнд.

♦ Унарные операции, перегружаемые вне области класса (как глобальные функции), должны иметь один параметр типа класса. Передаваемый через этот параметр объект воспринимается как операнд.

Синтаксис:
а) в первом случае (описание в области класса):
тип_возвр_значения operator знак_операции
б) во втором случае (описание вне области класса):
тип_возвр_значения operator
знак_операции(идентификатор_типа)

Примеры
1)
```
class person
{int  age;
//другие поля
public:
 //конструкторы, деструктор и другие методы
void operator++(){ ++age;}
};
void main()
{
class person jon;
 ++jon;
}
```

2)
```
class person
{int age;
//другие поля
public:
//конструкторы, деструктор и другие методы
friend void operator++(person&);
};

void operator++(person&ob)
{++ob.age;}
```

```
void main()
{
class person jon;
 ++jon;
}
```

3.2. Перегрузка бинарных операций

♦ Любая бинарная операция ⊕ может быть определена двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае **x⊕y** означает вызов **x.operator⊕(y)**, во втором – вызов **operator⊕(x,y).**

♦ Операции, перегружаемые внутри класса, могут перегружаться только нестатическими компонентными функциями с параметрами. Вызываемый объект класса автоматически воспринимается в качестве первого операнда.

♦ Операции, перегружаемые вне области класса, должны иметь два операнда, один из которых должен иметь тип класса.

Примеры.

```
1)class person{…};
class adresbook
{
// содержит в качестве компонентных данных
// множество объектов типа person, представляемых как
//динамический массив, список или дерево
public:
//конструкторы, деструктор и другие методы
person&  operator[](int);  //доступ к i-му объекту
};
person& adresbook : : operator[](int i)
{/*реализация метода*/}
void main()
{class adresbook persons;
```

```
  class person record;
  record = persons[3];
}

2) class person{…};
class adresbook
{ // содержит в качестве компонентных данных
// множество объектов типа person, представляемых как
//динамический массив, список или дерево
public:
//конструкторы, деструктор и другие методы
 friend person&  operator[](const adresbook&,int);  //доступ к
//i-му объекту
};
 person&  operator[](const adresbook& ob ,int i)
{ /*реализация метода*/}

void main()
{class adresbook persons;
  class person record;
  record = persons[3];
}
```

3.3. Перегрузка операций ++ и --

Унарные операции инкремента ++ и декремента – существуют в двух формах: префиксной и постфиксной. В современной спецификации C++ определен способ, по которому компилятор может различить эти две формы. В соответствии с этим способом задаются две версии функции operator++() и operator—(). Они определены следующим образом:

– префиксная форма:

```
operator++();
operator—();
```

– постфиксная форма:

operator++(int);

operator—(int);

Указание параметра **int** для постфиксной формы не специфицирует второй операнд, а используется только для отличия от префиксной формы.

Пример

```
class person
{ int age;
//другие поля
public:
 //конструкторы, деструктор и другие методы
  void operator++(){ ++age;}//префиксная форма
  void operator++(int){ age++;}  //  постфиксная форма
  };
void main()
{class person jon;
  ++jon;   jon++}
```

3.4. Перегрузка операции вызова функции

Это операция ‘()’. Она является бинарной операцией. Первым операндом обычно является объект класса, вторым – список параметров.

Пример

```
class matriza // двумерный массив вещественных чисел
{
// поля  класса
public:
//конструкторы, деструктор и другие методы
double operator()(int,int); //доступ к элементам матрицы по индексам
};
double matriza::operator()(int i,int j)
```

```
{/*реализация метода*/}
void main()
{
class matriza a(7,8);//создание матрицы 7*8
double k;
k:=a(5,6);// k получает значение элемента матрицы
//с индексами 5 и 6
}
```

3.5. Перегрузка операции присваивания

Операция отличается тремя особенностями:

– операция не наследуется;

– операция определена по умолчанию для каждого класса в качестве операции поразрядного копирования объекта, стоящего справа от знака операции, в объект, стоящий слева;

– операция может перегружаться только в области определения класса. Это гарантирует, что первым операндом всегда будет леводопустимое выражение.

Если вас устраивает поразрядное копирование, нет смысла создавать собственную функцию operator=(). Однако бывают случаи, когда поразрядное копирование нежелательно. Например, использование предопределенной операции присваивания для классов, содержащих указатели в качестве компонентных данных, чаще всего приводит к ошибкам. Покажем это на примере.

Пользовательский класс – строка **string:**

```
class string
{
char *p;       //указатель на строку
int len;       //текущая длина строки
public:
string(char *);
~string();
void show();
```

```
};
string::string(char*ptr)
{len=strlen(ptr);
p=new chat[len+1];
if(!p){cout<<”Ошибка выделения памяти\n”);
  exit(1);}
strcpy(p,ptr);}
string::~string()
{delete[]p;}
void string::show()
{cout<<*p<<”\n”;}

void main()
string s1(“Это первая строка”),
          s2(“А это вторая строка”);
s1.show; s2.show;
s2=s1;   // Это ошибка
s1.show; s2.show;
}
```

В чем здесь ошибка? Когда объект s1 присваивается объекту s2, указатель p объекта s2 начинает указывать на ту же самую область памяти, что и указатель p объекта s1. Таким образом, когда эти объекты удаляются, память, на которую указывает указатель p объекта s1, освобождается дважды, а память, на которую до присваивания указывал указатель p объекта s2, не освобождается вообще.

Хотя в данном примере эта ошибка и неопасна, в реальных программах с динамическим распределением памяти она может вызвать крах программы.

В этом случае необходимо самим перегружать операцию присваивания. Покажем, как это сделать для нашего класса string.

```
class string
{
char *p;       //указатель на строку
int len;       //текущая длина строки
```

```
public:
...
string& operator=(string& );
};
string& string::operator=(string& ob);
{if(this==&ob) return *this;
if(len<ob.len){
 //требуется выделить дополнительную память
delete[]p;
p=new char[ob.len+1];
if(!p){cout<<”Ошибка выделения памяти\n”);
  exit(1);}
len=ob.len;
strcpy(p,ob.p);
return *this;}
```

В этом примере выясняется, не происходит ли самоприсваивание (типа ob=ob). Если имеет место самоприсваивание, то просто возвращается ссылка на объект.

Затем проверяется, достаточно ли памяти в объекте, стоящем слева от знака присваивания, для объекта, стоящего справа от знака присваивания. Если недостаточно, то память освобождается и выделяется новая, требуемого размера. Затем строка копируется в эту память.

Отметим две важные особенности функции operanor=. Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, передаваемого через параметр по значению. В случае создания копии она удаляется вызовом деструктора при завершении работы функции. Но деструктор освобождает память, на которую указывает p. Однако эта память все еще необходима объекту, который является аргументом. Параметр-ссылка помогает решить эту проблему.

Во-вторых, функция operator=() возвращает не объект, а ссылку на него. Смысл этого тот же, что и при использовании парамет-

ра-ссылки. Функция возвращает временный объект, который удаляется после завершения ее работы. Это означает, что для временной переменной будет вызван деструктор, который освобождает память по адресу p. Но она необходима для присваивания значения объекту. Поэтому, чтобы избежать создания временного объекта, в качестве возвращаемого значения используется ссылка.

Другой путь решения проблем, описанных выше, – это создание конструктора копирования. Но конструктор копирования может оказаться не столь эффективным решением, как ссылка в качестве параметра и ссылка в качестве возвращаемого значения функции. Это происходит потому, что использование ссылки исключает затраты ресурсов, связанных с копированием объектов в каждом из двух указанных случаев.

3.6. Перегрузка операции new

Операция **new**, заданная по умолчанию, может быть в двух формах:

1) new тип <инициализирующее выражение>

2) new тип[];

Первая форма используется не для массивов, вторая – для массивов.

Перегруженную операцию new можно определить в следующих формах, соответственно для немассивов и для массивов:

void* operator new(size_t t[,остальные аргументы]);

void* operator new[](size_t t[,остальные аргументы]);

Первый и единственный обязательный аргумент t всегда должен иметь тип size_t. Если аргумент имеет тип size_t, то в операцию-функцию new автоматически подставляется аргумент sizeof(t), т.е. она получает значение, равное размеру объекта t в байтах.

Например, пусть задана следующая функция:

void* operator new(size_t t,int n){return new char[t*n];}

и она вызывается следующим образом:

double *d=new(5)double;

Здесь t=double, n=5.

В результате после вызова значение t в теле функции будет равно sizeof(double).

При перегрузке операции new появляется несколько глобальных операций new, одна из которых определена в самом языке по умолчанию, а другие являются перегруженными. Возникает вопрос: как различить такие операции? Это делается путем изменения числа и типов их аргументов. При изменении только типов аргументов может возникнуть неоднозначность, являющаяся следствием возможных преобразований этих типов друг к другу. При возникновении такой неоднозначности следует при обращении к new задать тип явно, например:

new((double)5)double;

Одна из причин, по которой перегружается операция new, состоит в стремлении придать ей дополнительную семантику, например, обеспечение диагностической информацией или устойчивости к сбоям. Кроме того, класс может обеспечить более эффективную схему распределения памяти, чем та, которую предоставляет система.

В соответствии со стандартом C++ в заголовочном файле <new> определены следующие функции-операции new, позволяющие передавать наряду с обязательным первым size_t аргументом и другие:

void* operatop new(size_t t)throw(bad_alloc);

void* operatop new(size_t t,void* p)throw();

void* operatop new(size_t t,const nothrow&)throw();

void* operatop new(size_t t,allocator& a);

void* operatop new[](size_t t)throw(bad_alloc);

void* operatop new[](size_t t,void* p)throw();

void* operatop new[](size_t t,const nothrow&)throw();

Эти функции используют генерацию исключений (throw) и собственный распределитель памяти (allocator).

Версия с nothrow выделяет память, как обычно, но если выделение заканчивается неудачей, возвращается 0, а не генерируется bad_alloc. Это позволяет нам для выделения памяти использовать стратегию обработки ошибок до генерации исключения.

Правила использования операции new

1. Объекты, организованные с помощью new, имеют неограниченное время жизни. Поэтому область памяти должна освобождаться оператором delete.

2. Если резервируется память для массива, то операция new возвращает указатель на первый элемент массива.

3. При резервировании памяти для массива все размерности должны быть выражены положительными величинами.

4. Массивы нельзя инициализировать.

5. Объекты классов могут организовываться с помощью операции new, если класс имеет конструктор по умолчанию.

6. Ссылки не могут организовываться с помощью операции new, так как для них не выделяется память.

7. Операция new самостоятельно вычисляет потребность в памяти для организуемого типа данных, поэтому первый параметр операции всегда имеет тип size_t.

Обработка ошибок операции new

Обработка ошибок операции new происходит в два этапа:

1. Устанавливается, какие предусмотрены функции для обработки ошибок. Собственные функции должны иметь тип **new_handler** и создаются с помощью функции **set_new_handler.** В файле new.h объявлены

typedef void(*new_handler)();

new_handler set_new_handler(new_handler new_p);

2. Вызывается соответствующая new_handler функция. Эта функция должна:

– либо вызвать **bad_alloc** исключение;

– либо закончить программу;

– либо освободить память и попытаться распределить ее заново.

Диагностический класс **bad_alloc** объявлен в new.h.

В реализации BC++ включена специальная глобальная переменная **_new_handler**, значением которой является указатель на new_handler функцию, которая выполняется при неудачном завершении new. По умолчанию, если операция new не может выделить требуемое количество памяти, формируется исключение bad_alloc. Изначально это исключение называлось **xalloc** и определялось в файле except.h. Исключение xalloc продолжает использоваться во многих компиляторах. Тем не менее оно вытесняется определенным в стандарте C++ именем bad_alloc.

Рассмотрим несколько примеров.

Пример 1. В примере использование блока **try...catch** дает возможность проконтролировать неудачную попытку выделения памяти.

```
#include <iostream>
#include <new>
void main()
{double *p;
try{
p=new double[1000];
cout<<"Память выделилась успешно"<<endl;
}
catch(bad_alloc xa)
{cout<<"Ошибка выделения памяти\n";
cout<<xa.what();  return;} }
```

Пример 2. Поскольку в предыдущем примере при работе в нормальных условиях ошибка выделения памяти маловероятна, в этом примере ошибка выделения памяти достигается принудительно. Процесс выделения памяти длится до тех пор, пока не произойдет ошибка.

```
#include <iostream>
```

```
#include <new>
void main()
{double *p;
do{
try{
p=new double[1000];
//cout<<"Память выделилась успешно"<<endl;
}
catch(bad_alloc xa)
{cout<<"Ошибка выделения памяти\n";
cout<<xa.what();
 return;}
}while(p);
}
```

Пример 3. Демонстрируется перегруженная форма операции new-операция new(nothow).

```
#include <iostream>
#include <new>
void main()
{double *p;
struct nothrow noth_ob;
do{
p=new(noth_ob) double[1000];
if(!p) cout <<"Ошибка выделения памяти\n";
else cout<<"Память выделилась успешно\n";
}while(p);
}
```

Пример 4. Демонстрируются различные формы перегрузки операции **new.**

```
#include<iostream.h>
#include<new.h>
double *p,*q,**pp;
class demo
```

```
{ int value;
public:
  demo(){value=0;}
  demo(int i){value = 1;}
  void* operator new(size_t t,int,int);
  void* operator new(size_t t,int);
  void* operator new(size_t t,char*);
  };

void* demo :: operator new(size_t t,int i, int j)

{
 if(j)  return new(i)demo;
 else return NULL;
}
void* demo :: operator new(size_t t,int i)
{demo* p= ::new demo;
  (*p).value=i;
  return p;
}
void* demo::operator new(size_t t,char* z)
{
return ::new(z)demo;
}

void main()
{ class demo *p_ob1,*p_ob2;
//  struct nothrow noth_ob;
  p=new double;
  pp=new double*;
  p=new double(1.2);  //инициализация
  q=new double[3];    //массив
  p_ob1=new demo[10]; //массив объектов demo
  void(**f_ptr)(int); //указатель на указатель на функцию
```

```
f_ptr=new(void(*[3])(int)); //массив указателей на функцию
char z[sizeof(demo)]; //резервируется память в соответствии с величиной //demo
p_ob2=new(z)demo;  //организуется demo-объект в области памяти на //которую указывает переменная z
p_ob2=new(3)demo;  //demo-объект с инициализацией
p_ob1=new(3,0)demo; //возвращает указатель NULL
// p_ob2=new(noth_ob)demo[5];//массив demo-объектов,
// в случае ошибки возвращает NULL
}
```

3.7. Перегрузка операции delete

Операция-функция delete бывает двух видов:

– void operator delete(void*);

– void operator delete(void*,size_t);

Вторая форма включает аргумент типа size_t, передаваемый вызову delete. Он передается компилятору как размер объекта, на который указывает p.

Особенностью перегрузки операции delete является то, что глобальные операции delete не могут быть перегружены. Их можно перегрузить только по отношению к классу.

В заключение сформулируем основные правила перегрузки операций.

3.8. Основные правила перегрузки операций

1. Вводить собственные обозначения для операций, не совпадающие со стандартными операциями языка C++ , нельзя.

2. Не все операции языка C++ могут быть перегружены. Нельзя перегрузить следующие операции:

‘**.**’ – прямой выбор компонента,

‘**.***’ – обращение к компоненту через указатель на него,

‘**?:**’ – условная операция,

‘**::**’ – операция указания области видимости,

‘sizeof’,

‘#’, ‘##’– препроцессорные операции.

3. Каждая операция, заданная в языке, имеет определенное число операндов, свой приоритет и ассоциативность. Все эти правила, установленные для операций в языке, сохраняются и для ее перегрузки, т.е. изменить их нельзя.

4. Любая унарная операция ⊕ определяется двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром. Выражение ⊕**z** означает в первом случае вызов **z.operator⊕(),** во втором – вызов **operator⊕(z).**

5. Любая бинарная операция ⊕ определяется также двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае **x⊕y** означает вызов **x.operator⊕(y)**, во втором – вызов **operator⊕(x,y).**

6. Перегруженная операция не может иметь аргументы (операнды), заданные по умолчанию.

7. В языке C++ установлена идентичность некоторых операций, например, **++z** – это то же, что и **z+=1**. Эта идентичность теряется для перегруженных операций.

8. Функцию **operator** можно вызвать по ее имени, например, **z=operator*(x,y)** или **z=x.operator*(y)**. В первом случае вызывается глобальная функция, во втором – компонентная функция класса **X** , и **x** – это объект класса **X**. Однако чаще всего функция **operator** вызывается косвенно, например **z=x*y**.

9. За исключением перегрузки операций **new** и **delete** функция **operator** должна быть либо нестатической компонентной функцией, либо иметь, как минимум, один аргумент (операнд) типа «класс» или «ссылка на класс» (если это глобальная функция).

10. Операции ‘=’, ‘**[]**’, ‘**–>**’ можно перегружать только с помощью нестатической компонентной функции **operator⊕**. Это гарантирует, что первыми операндами будут леводопустимые выражения.

11. Операция ‘**[]**’ рассматривается как бинарная. Пусть **a** – объект класса **A**, в котором перегружена операция ‘**[]**’. Тогда выражение **a[i]** интерпретируется как **a.operator[](i)**.

12. Операция ‘**()**’ вызова функции рассматривается как бинарная. Пусть **a** – объект класса **A**, в котором перегружена операция ‘**()**’. Тогда выражение **a(x1,x2,x3,x4)** интерпретируется как **a.operator()(x1,x2,x3,x4)**.

13. Операция ‘**–>**’ доступа к компоненту класса через указатель на объект этого класса рассматривается как унарная. Пусть **a** – объект класса **A**, в котором перегружена операция ‘**–>**’. Тогда выражение **a–>m** интерпретируется как **(a.operator–>())–>m**. Это означает, что функция **operator–>()** должна возвращать указатель на класс **A**, или объект класса **A**, или ссылку на класс **A**.

14. Перегрузка операций ‘++’ и ‘--‘, записываемых после операнда (z++, z--), отличается добавлением в функцию **operator** фиктивного параметра **int**, который используется только как признак отличия операций **z++** и **z--** от операций **++z** и **--z**.

15. Глобальные операции **new** можно перегрузить, и в общем случае они могут не иметь аргументов (операндов) типа «класс». В результате разрешается иметь несколько глобальных операций **new**, которые различаются путем изменения числа и (или) типов аргументов.

16. Глобальные операции **delete** не могут быть перегружены. Их можно перегрузить только по отношению к классу.

17. Заданные в самом языке глобальные операции **new** и **delete** можно изменить, т.е. заменить версию, заданную в языке по умолчанию, на свою версию.

18. Локальные функции operator new() и operator delete() являются статическими компонентами класса, в котором они определены, независимо от того, использовался или нет специ-

фикатор static (это, в частности, означает, что они не могут быть виртуальными).

19. Для правильного освобождения динамической памяти под базовый и производный объекты следует использовать виртуальный деструктор.

20. Если для класса X операция “=” не была перегружена явно и *x* и *y* – это объекты класса X, то выражение *x* = *y* задает по умолчанию побайтовое копирование данных объекта *y* в данные объекта *x*.

21. Функция **operator** вида **operator type()** без возвращаемого значения, определенная в классе **A**, задает преобразование типа **A** к типу **type**.

22. За исключением операции присваивания ‘=’ все операции, перегруженные в классе **X**, наследуются в любом производном классе **Y**.

23. Пусть **X** – базовый класс, **Y** – производный класс. Тогда локально перегруженная операция для класса **X** может быть далее повторно перегружена в классе **Y**.

3.9. Примеры программ

Программа 1

Задание: определить и реализовать класс «complex» – комплексное число. Для сложения чисел определить в классе функцию и перегруженную операцию. Предусмотреть счетчик созданных в программе чисел (рис. 3.1).

Файл «complex.h»

```
class complex
{
private:
double re,im;
static int n; //счетчик объектов
public:
```

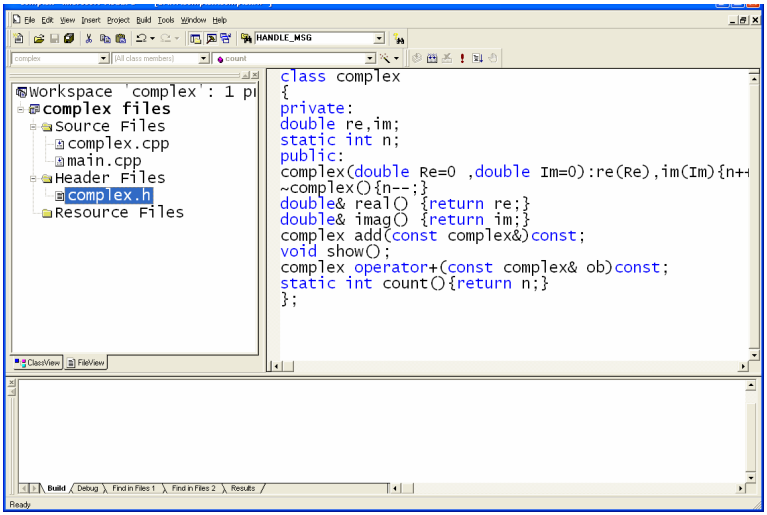


Рис. 3.1

```
complex(double Re=0 ,double Im=0):re(Re),im(Im){n++;};
~complex(){n--;}
double& real() {return re;}
double& imag() {return im;}
//функции для сложения чисел
complex add(const complex&)const;
//функции для вывода числа
void show();
//перегруженная операция сложения чисел
complex operator+(const complex& ob)const;
static int count(){return n;}
};
```

Файл «complex.cpp» (рис. 3.2).

```
#include<iostream.h>
#include"complex.h"
 void complex::show()
{
```

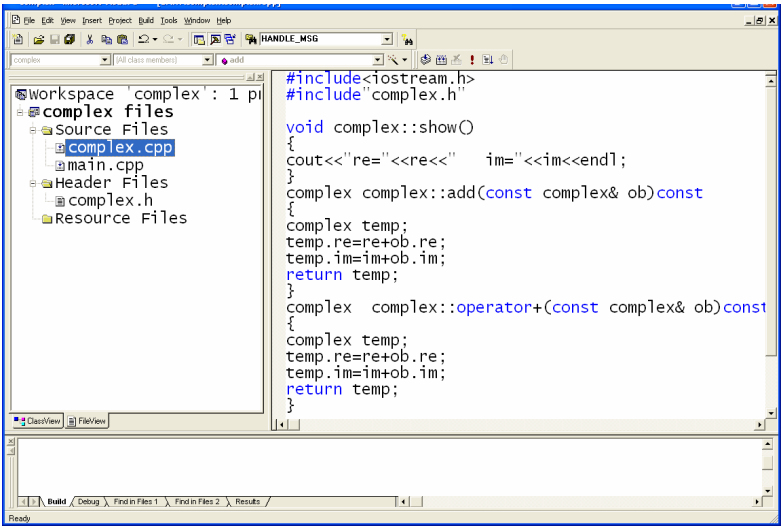


Рис. 3.2

```
cout<<"re="<<re<<"  im="<<im<<endl;
}
complex complex::add(const complex& ob)const
{
complex temp;
temp.re=re+ob.re;
temp.im=im+ob.im;
return temp;
}
complex  complex::operator+(const complex& ob)const
{
complex temp;
temp.re=re+ob.re;
temp.im=im+ob.im;
return temp;
}
```

Файл «main.cpp» (рис. 3.3).

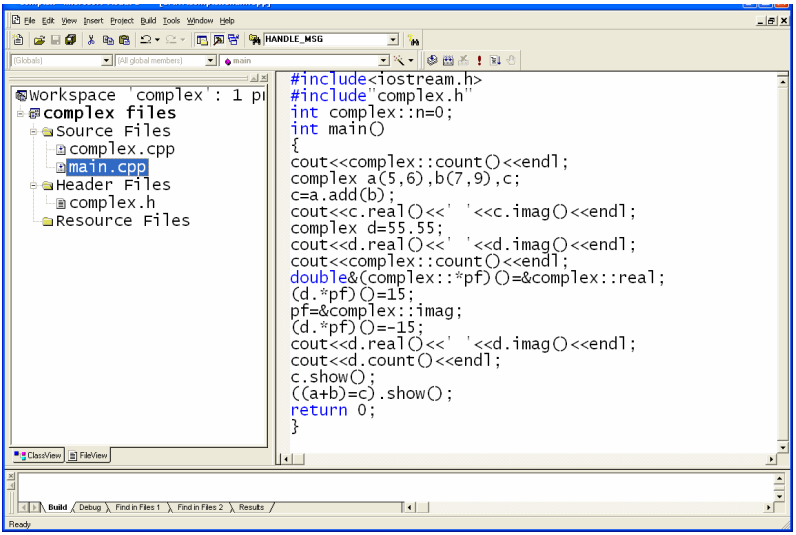


Рис. 3.3

```
#include<iostream.h>
#include"complex.h"
int complex::n=0;//
int main()
{
//Чисел еще нет, статическая функция возвращает 0
cout<<complex::count()<<endl;
complex a(5,6),b(7,9),c;

//сложение чисел a и b, результат в c
//используем автоматически созданную операцию –
функцию
//присваивания
c=a.add(b);
cout<<c.real()<<' '<<c.imag()<<endl;
complex d=55.55;
cout<<d.real()<<' '<<d.imag()<<endl;
```

```
//Сейчас имеется три числа – функция возвращает 3
cout<<complex::count()<<endl;
//вводим указатель на функцию и инициализируем его
double&(complex::*pf)()=&complex::real;
//используя указатель на функцию , изменяем
// действительную часть числа d
(d.*pf)()=15;
//изменяем значение указателя на функцию
//сейчас он указывает на функцию imag
pf=&complex::imag;
//используя указатель на функцию , изменяем
// мнимую часть числа d
 (d.*pf)()=-15;
cout<<d.real()<<' '<<d.imag()<<endl;
cout<<d.count()<<endl;
//показываем число c
c.show();
//демонстрируем, что перегруженные операции сложения
//и присваивания возвращают объект
((a+b)=c).show();
return 0;
}
```

4. ШАБЛОНЫ ФУНКЦИЙ И КЛАССОВ

4.1. Шаблоны функций

Шаблоны являются одой из важных особенностей C++.

Причина этого заключается в том, что шаблоны фундаментально изменяют внешнюю сторону программирования. Используя шаблон, можно создать обобщенные спецификации для функций и для классов, которые называются **параметризованными функциями** (generic functions) и **параметризованными классами** (generic classes). Параметризованная функция определяет общую процедуру, которая может быть применена к данным различных типов. Параметризованный класс определяет общий класс, который может применяться к различным типам данных. В обоих случаях конкретный тип данных, над которыми выполняется операция, передается в качестве параметра.

Причина использования параметризованных функций заключается в следующем.

Многие алгоритмы не зависят от типа данных, которыми они манипулируют. Например, обмен значениями:

```
TData temp,x,y;
// . . .
temp=x; x=y; y=temp;
```

Этот алгоритм работает вне зависимости от фактического значения типа данных TData. Однако в большинстве языков программирования для обмена данными каждого типа требуется написание новой версии подпрограммы, несмотря на то, что лежащий в ее основе алгоритм остается неизменным.

Многие алгоритмы допускают отделение метода от данных. При использовании таких алгоритмов большим преимуществом была бы возможность однократного определения и отладки логики алгоритма и последующее применение алгоритма

к различным типам данных без необходимости перепрограммирования. Это не только позволяет экономить усилия и время, но и страхует от ошибок.

Итак, обобщенные параметризованные процедуры предоставляют очевидные преимущества. Поэтому программисты всегда пытались их использовать. Однако до изобретения шаблонов такие попытки имели лишь частичный успех. Появились два далеких от совершенствования метода.

Первый заключается в построении параметризированных функций через использование **макросов**. Например, макрос создает «родовую функцию», возвращающую абсолютное значение числа.

```
# define ABS(a)
((a<0)?–(a):(a))
. . .
int x;
float f;
x=ABS(-10);
f=ABS(15.25);
```

Однако добиться работы макросов с типами данных, определенными пользователем, достаточно сложно. Кроме того, поскольку этот макрос не выполняет никакой проверки типов, возможна ситуация, когда он будет ошибочно использован с такими типами данных, для которых не определены используемые в нем операции. Также не для всех алгоритмов можно написать макрос.

Второй метод построения параметризованной функции заключается в добавлении одного или нескольких параметров, предназначенных для определения типов данных, над которыми функция выполняет операции. Например, распространенным подходом была передача функции указателя на данные в качестве одного параметра и размера этих данных в байтах – в качестве другого (например, указатель типа void*).

Примером может служить библиотечная функция qsort().

```
void qsort(void* buf,size_t num,size_t size,int(*comp)(const void*,const void*));
```

Но поскольку параметры передаются через стек, передача каждого параметра генерирует несколько инструкций в машинном коде, что уменьшает эффективность кода, увеличивая время, требующееся для вызова функции.

Так, функция qsort() обычно реализуется как рекурсивная функция, и поэтому наличие дополнительных параметров приводит к значительному снижению ее производительности.

В C++ параметризированная функция создается с помощью ключевого слова **template**. Шаблон определяет общий набор операций (алгоритм), которые будут применяться к данным различных типов. При этом тип данных, над которыми функция должна выполнять операции, передается ей в виде параметра на стадии компиляции. Формат функции-шаблона:

template <class тип_данных> тип_возвр_значения имя_функции(список_параметров)

{тело_функции}

Параметр *тип_данных* обозначает тип данных, используемых функцией. Это имя может использоваться в пределах действия определения функции. Когда компилятор будет создавать конкретную версию этой функции, он автоматически заменит этот параметр конкретным типом данных. Можно определить несколько родовых типов данных, которые в списке должны отделяться друг от друга запятыми. Каждый элемент данного списка предваряется ключевым словом **class**, которое в данном контексте ссылается не на конкретный тип данных **class**, а на любой тип данных, фактически передаваемый при вызове функции (встроенный либо определенный программистом). Это так называемые параметры – шаблоны.

Пример 4.1. Шаблон функции для обмена значениями.

```
template <class Stype> void swap(Stype& x,Stype& y)
{Stype temp;
 temp=x; x=y; y=temp;}
```

Хотя функция-шаблон по мере надобности может перегружать себя сама, можно выполнять ее явную перегрузку. Если параметризованная функция перегружается явно, то эта перегруженная функция «скрывает» параметризованную функцию по отношению к конкретной версии.

Пример 4.2. Сортировка методом обмена

```
# include <iostream.h>
# include <string.h>
template <class Stype> void bubble(Stype* item,int count);
void main()
{
  // сортировка массива символов
  char str[]=”dcab”;
  bubble(str,strlen(str));//здесь компилятор построит функцию
 //сортировки для данных типа char,
 // т.е. Stype заменится на char
  cout<<str<<endl;
  // сортировка массива целых чисел
  int nums[]={5,7,3,9,5,1,8};
  int i;
  bubble(nums,7);//а здесь компилятор построит
 //функцию сортировки для данных типа int
  for(i=0;i<7;i++) cout<<nums[i]<<” “;
  cout<<endl;
}
// Определение параметризованной функции
template <class Stype> void bubble(Stype* item,int count)
{register int i,j;
  Stype temp;
  for(i=1;i<count;i++)
    for(j=count-1;j>=i;--j)
      if(item[j-1]>item[j])
        {temp=item[j-1];
```

```
            item[j-1]=item[j];
            item[j]=temp;
          }
    }
```

Можно считать, что параметры шаблона функции являются его формальными параметрами, а типы тех параметров, которые используются в конкретных обращениях к функции, служат фактическими параметрами шаблона.

При первом вызове функции с конкретными типами параметров компилятор построит функцию для параметров этого типа. Естественно, операции, используемые в функции, должны быть определены для этих типов.

Перечислим основные свойства параметров шаблона функции:

Имена параметров шаблона должны быть уникальными во всем определении шаблона.

1. Список параметров шаблона не может быть пустым.

2. В списке параметров шаблона может быть несколько параметров, и каждому из них должно предшествовать ключевое слово **class**.

3. Имя параметра шаблона имеет все права имени типа в определенной шаблоном функции.

4. Все параметры шаблона должны быть обязательно использованы в спецификациях параметров определения функции. Например, будет ошибочным такой шаблон:

```
template <class A,class B,class C> C func(A x,B y)
{C temp; . . .}
```

Здесь остался неиспользованным параметр шаблона с именем **C**.

5. Определенная с помощью шаблона функция может иметь любое количество непараметризованных формальных параметров. Может быть не параметризовано и возвращаемое функцией значение.

6. В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона.

7. При конкретизации параметризованной функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковы.

8. При использовании шаблонов функций возможна перегрузка как шаблонов, так и функций. Могут быть шаблоны с одинаковыми именами, но разными параметрами. Или с помощью шаблона может создаваться функция с таким же именем, что и явно определенная функция. В обоих случаях «распознавание» конкретного вызова выполняется по сигнатуре, т.е. по типам, порядку и количеству фактических параметров.

4.2. Шаблоны классов

Шаблон класса используется для построения родового класса. При создании родового класса создается целое семейство родственных классов, которые можно применять к любому типу данных. Таким образом, тип данных, которым оперирует класс, указывается в качестве параметра при создании объекта, принадлежащего к этому классу. Принципиальное преимущество параметризованного класса заключается в том, что он позволяет определить члены класса один раз, но применять класс к данным любых типов. Подобно тому, как класс определяет правила построения и формат отдельных объектов, шаблон класса определяет способ построения отдельных классов. В определении класса, входящего в шаблон, имя класса является не именем отдельного класса, а параметризованным именем семейства классов.

Наиболее широкое применение шаблоны классов находят при создании контейнерных классов. Фактически создание контейнеров является одной из основных причин, по которым были введены в употребление шаблоны.

Контейнерными классами (контейнерами) называются классы, в которых хранятся организованные данные. Например, массивы и связные списки. Преимущество, даваемое определением параметризованных контейнерных классов, заключается в том, что, как только логика, необходимая для поддержки контейнера, определена, он может применяться к любым типам данных без необходимости его переписывания. Например, параметризованный контейнер связного списка можно использовать для построения списков, содержащих почтовые адреса, заглавия книг, названия автомобилей.

Общая форма объявления параметризованного класса:

template <class тип_данных> class имя_класса { . . . };

Здесь *тип_данных* представляет собой имя типа шаблона, которое в каждом случае конкретизации будет замещаться фактическим типом данных. При необходимости можно использовать более одного параметризованного типа данных, используя список с разделителем – запятой. В пределах определения класса имя *тип_данных* можно использовать в любом месте.

Создав параметризованный класс, можно создать конкретную реализацию этого класса, используя следующую общую форму:

имя_класса <тип> имя_объекта;

Здесь *тип* представляет собой имя типа данных, над которыми фактически оперирует класс, и заменяет собой переменную *тип_данных*.

Перечислим основные свойства шаблонов классов:

1. Компонентные функции параметризованного класса автоматически являются параметризованными. Их необязательно объявлять как параметризованные с помощью template.

2. Дружественные функции, которые описываются в параметризованном классе, не являются автоматически параметризованными функциями, т.е. по умолчанию такие функции являются дружественными для всех классов, которые организуются по данному шаблону.

3. Если friend-функция содержит в своем описании параметр типа параметризованного класса, то для каждого созданного по данному шаблону класса имеется собственная friend-функция.

4. В рамках параметризованного класса нельзя определить friend-шаблоны (дружественные параметризованные классы).

5. С одной стороны, шаблоны могут быть производными (наследоваться) как от шаблонов, так и от обычных классов, с другой стороны, они могут использоваться в качестве базовых для других шаблонов или классов.

6. Определенные пользователем имена в описании шаблона по умолчанию рассматриваются как идентификаторы переменных. Чтобы имя рассматривалось как идентификатор типа, оно должно быть определено внутри шаблона или в окружающей области определения через ключевое слово **typename**.

7. Шаблоны функций, которые являются членами классов, нельзя описывать как virtual.

8. Локальные классы не могут содержать шаблоны в качестве своих элементов.

4.3. Компонентные функции параметризованных классов

Реализация компонентной функции шаблона класса, которая находится вне определения шаблона класса, должна включать дополнительно следующие два элемента:

1. Определение должно начинаться с ключевого слова template, за которым следует такой же *список_параметров_типов* в угловых скобках, какой указан в определении шаблона класса.

2. За *именем_класса*, предшествующим операции области видимости (**::**), должен следовать *список_имен_параметров* шаблона.

template <список_типов> тип_возвр_значения
имя_класса<список_имен_параметров>::

имя_функции(список_параметров)
{тело функции };

Пример 4.3

```
template <class A,class B> class myclass
{A x;
  B y;
public:
  A func();
};
template <class A,class B> A myclass<A,B>: : func()
{return A;}
```

Пример 4.4. «Защищенный» массив

В C++ во время выполнения можно выйти за границу массива без генерации сообщения об ошибке. Хотя эта возможность позволяет генерировать исключительно быстрый исполняемый код, но одновременно служит источником ошибок. Решить эту проблему можно, если создать класс, который содержит массив, и разрешить доступ к массиву через перегруженную операцию **[]**. В функции **operator[]()** можно перехватывать индекс, выходящий за рамки диапазона массива.

```
# include <iostream.h>
# include <stdlib.h>
template <class ARRAY> class array
{ARRAY *a;
  int length;
public:
  array(int size);
  ~array(){delete[]a;}
  ARRAY& operator[](int i);
};
template <class ARRAY> array<ARRAY>: : array(int size)
{register int i;
  length=size;
```

```
  a=new ARRAY[size];
 //Проверка, распределена ли память?
  if(!a){cout<<”Ошибка!”; exit(1);}
  for(i=0;i<size;i++) a[i]=0;
 }
 template <class ARRAY> ARRAY& array<ARRAY>: : operator[](int i)
 {
 //Проверка, не вышел ли индекс за границы
 if((i<0)||(i>length-1)){cout<<”Ошибка!”; exit(1);}
  return a[i];
 }
 void main()
 {array<int> masint(20);
  array<double> masdouble(10);
  int i;
  for(i=0;i<20;i++){masint[i]=i; cout<<masint[i]<<” ”;}
  cout<<endl;
  for(i=0;i<10;i++){masdouble[i]=(double)i*3.14;
cout<<masdouble[i]<<” “;}
  cout<<endl;
  masint[45]=100; //ошибка, недопустимый индекс
  array<char> C(5);
  array<int> X(5);
  for(i=0;i<5;i++){X[i]=i; C[i]=’A’+i;}
  for(i=0;i<5;i++) cout<<X[i]<<C[i]<<’ ’;
 }
```

В списке параметров шаблона могут присутствовать формальные параметры, тип которых фиксирован.

Пример 4.5. Размер массива задается по умолчанию

```
#include<iostream.h>
template <class ARRAY, int size=64> class array
{
```

```
  ARRAY *a;
  int len;
public:
  array(){len=size; a=new ARRAY[len];}
  int setlen(){return len;}

};
int main()
{
array<int,5> x; //создается массив типа int размером 5
cout<<x.setlen()<<endl; //будет выведено 5
array<int> y; //создается массив типа int размером 64

cout<<y.setlen()<<endl; //будет выведено 64
return 0;
 }
```

Этот пример показывает возможность использования для создания объекта конструктора без параметров с заданием параметров объекта через параметры шаблона.

4.4. Примеры программ

Задание: написать программу, использующую так называемый «умный» указатель. «Интеллектуальный» или «умный» указатель– это указатель, который автоматически уничтожает объект, когда уничтожается последняя ссылка на него, т.е. память освобождается, когда на объект нет больше ссылок.

Создайте и выполните в Microsoft Visual C++ следующую программу.

```
#include<iostream>      //Для поддержки нового типа string
#include <string>        //Тип string библиотеки STL
using namespace std;   //использовать пространство имен std
```

```
//класс Ref хранит исходный указатель и счетчик ссылок
template<class T> struct Ref
{
T* realPtr; //исходный указатель на объект
int count;   //счетчик ссылок
};
//шаблон класса «умного» указателя
template<class T> class SmartPtr
{
Ref<T>* refPtr;
public:
SmartPtr(T*ptr=NULL);
SmartPtr(const SmartPtr& s);
~SmartPtr();
//Перегруженная операция присваивания
SmartPtr& operator=(const SmartPtr& s);
// Перегруженная операция доступа к членам класса че-
рез ука//затель
T* operator->()const;
// Перегруженная операция разыменовывания
T& operator*()const;
// Перегруженная операция преобразования типа
operator int(){return refPtr->count;}
};

template<class T> SmartPtr<T>::SmartPtr(T*ptr)
{
if(!ptr)refPtr=NULL;
else
{
 refPtr=new Ref<T>;
 refPtr->realPtr=ptr;
 refPtr->count=1;
```

```
}
}

template<class T> SmartPtr<T>::~SmartPtr()
{
if(refPtr)
{
refPtr->count--;
 if(refPtr->count<=0)//если ссылок нет, освободить память
 {
  delete refPtr->realPtr;
  delete refPtr;
  refPtr=NULL;
  }
}
}

template<class T> T* SmartPtr<T>::operator->()const
{
if(refPtr)return (refPtr->realPtr);
else return NULL;
}

template<class T> T& SmartPtr<T>::operator*()const
{
if(refPtr)return *(refPtr->realPtr);
else throw ; //генерировать исключение
 }

template<class T> SmartPtr<T>::SmartPtr(const SmartPtr& s)
{
refPtr=s.refPtr;
if(refPtr)refPtr->count++;
}
/*
```

При выполнении операции присваивания прежде всего нужно отсоединиться от имеющегося объекта, а затем присоединиться к новому, подобно тому, как это сделано в конструкторе копирования

```
*/
template<class T> SmartPtr<T>& SmartPtr<T>::operator=(const SmartPtr& s)
{
if(refPtr)
{
refPtr->count--;
 if(refPtr->count<=0) //если ссылок нет, освободить память
 {
  delete refPtr->realPtr;
  delete refPtr;
 }
}
refPtr=s.refPtr;
if(refPtr)refPtr->count++;//если есть ссылки
return *this;
}

//Типы объектов, с которыми будет работать указатель SmartPtr
struct A
{
int x,y;
double d;
string s;
};

struct B
{
int x,y,z;
```

```
string s1,s2;
};

int main()
{
SmartPtr<A> aPtr(new A);
//С объектами класса SmartPtr можно обращаться, как с обычными указателями на объект типа T
{//начало блока
(aPtr->x)=5;
cout<<"aPtr->x="<<aPtr->x<<endl;
(*aPtr).y=3;
cout<<"aPtr->y="<<aPtr->y<<endl;
(aPtr->d)=45.67;
cout<<"aPtr->d="<<aPtr->d<<endl;
(aPtr->s)="new string";
cout<<"aPtr->s="<<aPtr->s<<endl;
cout<<"count="<<(int)aPtr<<endl;//Количество ссылок=1;
{//Объекты SmartPtr размещаются в автоматической памяти
SmartPtr<A> aPtr1=aPtr;
cout<<"count="<<(int)aPtr<<endl;// количество ссылок=2;
SmartPtr<A> aPtr2=aPtr;
cout<<"count="<<(int)aPtr<<endl;//количество ссылок=3;
SmartPtr<A> aPtr3;
aPtr3=aPtr;
cout<<"count="<<(int)aPtr<<endl;// количество ссылок=4;
(aPtr2->x)=15;    //Изменяется для всех связанных объектов
cout<<"aPtr->x="<<aPtr->x<<endl;//Выводится 15
} //конец блока - освобождаются 3 ссылки на объект
cout<<"count="<<(int)aPtr<<endl;//Количество ссылок=1;
//Теперь будем размещать объекты в динамический памяти
 SmartPtr<A>*p,*p1,*p2;
 p=new SmartPtr<A>(new A);
 (*p)->x=155;
```

```
cout<<"count="<<((int)(*p))<<endl;//Количество ссылок =1;
p1=new SmartPtr<A>(*p);
cout<<"count="<<((int)(*p))<<endl;//Количество ссылок =2
p2=new SmartPtr<A>(new A);
//количество ссылок не изменилось, т.к. p2 указывает на
новый //объект
cout<<"count="<<((int)(*p))<<endl;//Количество ссылок =2
(*p2)=(*p); //теперь p2 указывает на тот же объект
cout<<"count="<<((int)(*p))<<endl;//Количество ссылок =3
delete p;
cout<<"count="<<((int)(*p1))<<endl;//Количество ссылок =2
delete p1;
cout<<"count="<<((int)(*p2))<<endl;//Количество ссылок =1
cout<<(*p2)->x<<endl;
//Будем обрабатывать ошибки через исключения
//Контролируемый блок
try
{
delete p2;
//Здесь будет ошибка, т.к. ссылок больше нет и память ос-
во - //бождена
cout<<"count="<<(int)(*p2)<<endl;
}
catch (...){cout<<"No reference!"<<endl;}
// «Умный» указатель можно использовать для
//объектов любого типа
//Создаем указатели на объект типа B
SmartPtr<B> bPtr(new B); //Количество ссылок =1
{
SmartPtr<B> bPtr1=bPtr; //Количество ссылок =2
SmartPtr<B> bPtr2=bPtr; //Количество ссылок =3
}//Здесь две ссылки освобождаются
cout<<"count="<<(int)bPtr<<endl; //Количество ссылок =1
return 0;
  }
```

5. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

5.1. Механизм обработки исключений в C++

Рассмотрим программы, которые используют **библиотеки классов**. Разработчик библиотеки может предложить методы выявления ошибок, возникающих на этапе выполнения программы. Например, в классе **array** перегружена операция **[]** для проверки принадлежности индекса диапазону. Выявить такие ошибки можно только на этапе выполнения программы, и разработчик библиотеки знает, как это сделать. Однако он не знает, что делать дальше, поскольку об этом знает только разработчик программы, использующий эту библиотеку. С другой стороны, разработчик программы не знает, как найти эти ошибки, а даже если и знает, то это требует введения в программу специальных фрагментов, осуществляющих поиск ошибок, что приводит к ее усложнению и ухудшению читабельности.

В языке C++ вводится понятие **исключения** (**exception**), которое использует специальный механизм для выявления и устранения ошибок рассмотренного типа.

Для реализации механизма обработки исключений в языке C++ введены следующие три ключевых слова:

try (попытка контролировать),

catch (ловить),

throw (бросать, кидать, генерировать).

Служебное слово **try** позволяет выделить в любом месте программы так называемый контролируемый блок:

try{операторы}

Среди операторов могут быть описания, определения, обычные операторы C++ и специальные операторы генерации исключения:

throw выражение_генерации_исключения

Когда выполняется такой оператор, то с помощью выражения после throw формируется специальный объект, называемый **исключением**. Исключение создается как статический объект, тип которого определяется типом выражения. После формирования исключения throw передает исключение и управление непосредственно за границы контролируемого блока. В этом месте (за закрывающей фигурной скобкой) обязательно находится один или несколько обработчиков исключений:

catch(тип_исключения имя){операторы}

Обработчик исключений похож на определение функции с одним параметром. Когда обработчиков несколько, они должны отличаться друг от друга типами исключений. Это похоже на перегрузку функций.

Механизм обработки исключений является весьма общим средством управления программой. Он может использоваться не только при обработке аварийных ситуаций, но и при любых других состояниях в программе, которые почему-либо выделил программист.

Для этого достаточно, чтобы та часть программы, где планируется возникновение исключений, была оформлена в виде контролируемого блока, в котором выполнялись бы операторы генерации исключений при обнаружении заранее запланированных ситуаций.

Общую идею обработки исключений неформально можно выразить следующим образом:

```
а) программа пользователя
try{ // try-блок
/* Попытайтесь делать что-то, и если не возникают какие-то особые ситуации, то продолжайте в том же духе. Если возникает особая ситуация, то выполнение операций прерывается и осуществляется автоматический переход к catch-блоку */
}
catch(…){
/* Обработка исключительной ситуации */
}
```

б) библиотечная программа (библиотечный класс)

Обычное выполнение программы, если не возникает особая ситуация. Если возникает заранее определенная особая ситуация, то выполняется переход **throw** к обработке этой ситуации, вид которой определяется именем класса после слова throw.

В целом механизм обработки исключений представляет собой альтернативу традиционным методам, таким как:

– завершение программы;

– возвращение значения, указывающего на ошибку;

– установление некоторого кода ошибки (глобальная переменная **errno** в C);

– вызов некоторой функции, которая обрабатывает ошибку.

Но эти методы работают хуже.

Техника обработки исключений позволяет выделить в программе отдельные независимые части: собственно программу (try-блок) и фрагмент для обработки ошибок (catch-блок). Это делает программу более читабельной и регулярной (нехаотичной). Этот механизм поддерживает хороший стиль программирования, так как плохая программа будет периодически прерываться (завершаться), а не давать ошибочные результаты.

Следует указать, что рассмотренный механизм применим только для выявления синхронных исключений, т.е. таких, которые можно сопоставить с выполнением различных действий в самой программе. Асинхронные исключения, генерируемые, например, аппаратно (сигналы от клавиатуры, таймера и т.д.), нельзя непосредственно обработать этим способом.

Пример 5.1. Содержит локальный класс – индикатор исключения range.

```
#include <iostream.h>
#include<windows.h>
#define ss 10
class string
{char *p;
  int size;
```

```
public:
  string(int SIZE){p=new char[size=SIZE];}
  ~string(){delete[] p;}
//локальный класс – класс индикатор исключения
 class range{/*здесь могут быть поля и методы */};
 char& operator[](int); //опреция «доступ по индексу
};
char& string : : operator[](int j)
{
/*
```

проверяем, не вышел ли индекс за границы и если вышел, то генерируем исключение–создаем объект класса исключения

```
*/
if((j<0)||(j>=size)) throw range();
  return p[j];
}
void main()
{
SetConsoleOutputCP(1251); //Меняем кодовую страницу
int index;
  string str(ss);
//заполняем строку символами
  for(int i=0;i<ss;i++) str[i]='A'+i;
  try
    {for(;;) //бесконечный цикл
       {cout<<"\nВведите индекс: ";
         cin>>index;
         cout<<str[index];
       }
    }
  catch(string : : range){cout<<"\nОшибка!";}
}
```

Здесь обработчик исключения позволяет завершить при необходимости или при ошибке бесконечный цикл.

Пример 5.2. Содержит глобальный класс – индикатор исключения range.

```
#include <iostream.h>
#include<windows.h>
class range{};
class string
{char *p;
  int size;
public:
  string(int SIZE){p=new char[size=SIZE];}
  ~string(){delete[] p;}
   char& operator[](int);
};
char& string : : operator[](int j)
{if((j<0)||(j>=size)) throw ::range();
  return p[j];
}
void main()
{
  void main()
{
SetConsoleOutputCP(1251);
int index;
string str(ss);
//заполняем строку символами
  for(int i=0;i<ss;i++) str[i]='A'+i;
  try
    {for(;;) //бесконечный цикл
       {cout<<"\nВведите индекс: ";
         cin>>index;
         cout<<str[index];
       }
    }
  catch(range){cout<<"\nОшибка!";}
}
}
```

Пример 5.3

Используем два обработчика исключений и два индикатора исключений, один из которых глобальный, а другой – локальный.

Исключение типа локального класса range генерируется, когда индекс выходит за границу.

Исключение типа глобального класса range генерируется, когда количество ошибок больше заданного (в нашем примере больше 2).

```
#include <iostream.h>
#include <stdlib.h>
#include<windows.h>
#define ss 10
class range{}; // глобальный класс – индикатор исключения range.
class string
{char *p;
  int size;
  int i; //счетчик ошибок
public:
  string(int SIZE):i(0){ p=new char[size=SIZE];}
  class range{}; //локальный класс–индикатор исключения range.
   ~string(){delete[] p;}
   char& operator[](int);}; //конец определения string
char& string : : operator[](int j)
{
 if((j<0)||(j>=size)){i++; if(i>2) throw : : range(); else
 throw range();}
  return p[j];
}
void main()
{
SetConsoleOutputCP(1251);
 int index;
```

```
string str(ss);
//заполняем строку символами
  for(int i=0;i<ss;i++) str[i]=’A’+i;
   for(;;)
     {try{for(;;){for(;;) //бесконечный цикл
        {
        cout<<”\nВведите индекс: ”;
          cin>>index;
          cout<<str[index];
}}}
       catch(range) //первый обработчик.
         {cout<<”\nСлишком много ошибок!”;
           exit(0);
         }
       catch(string : : range) //второй обработчик.
         {cout<<”\nИндекс за границей!”;
           cout<<”\nПопытайтесь снова”<<endl;
         }
     }
}
```

Здесь при активизации первого обработчика исключений осуществляется завершение программы по exit. При активизации второго обработчика после его завершения выполняется очередная итерация бесконечного цикла.

Пример 5.4

Покажем возможный вызов в обработчике исключения функции, содержащей блоки try и catch. Это можно делать, например, когда возникшая ошибка не является опасной. В результате ее можно исправить и повторить соответствующие операции.

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

```
#define ss 20
class string
{
char *p;
int size;
public:
  string(int SIZE){p=new char[size=SIZE];}
  ~string(){delete[] p;}
 class range{/*здесь могут быть поля и методы */};
 char& operator[](int);
};
char& string::operator[](int j)
{
 if((j<0)||(j>=size)) throw range();
 return p[j];
}
void f() //рекурсивная функция
{
 string str(ss);
  for(int i=0;i<ss;i++) str[i]='A'+i;
  int index;
  try{
    cout<<"\nВведите индекс: ";
    cin>>index;
          cout<<str[index];
      }
  catch(string::range)
    {cout<<"\nОшибка!";
      f(); //новая попытка
    }
}
int main()
{
SetConsoleOutputCP(1251);
```

```
int k;
char s[]="\nПродолжить? Да-1,Нет-0  ";
while(true)
{
f();
cout<<s;
cin>>k;
if(k==0){system("pause"); return 0;}
}
```

5.2. Получение дополнительной информации об исключении

Передача такой информации производится через дополнительные поля, которые указываются в классе-индикаторе исключения. Вся необходимая информация передается в обработчик исключения с помощью соответствующего объекта (например, объекта класса range).

Пример 5.5

Добавим в класс string вложенный класс range, поле которого i будет хранить значение ошибочного индекса.

```
struct range
  {
  int i;
  range(int s):i(s){};
  };
```

При ошибке создается объект range,

```
char& string :: operator[](int s)
{if((s<0)||(s>size)) throw range(s); return p[s];}
```

Теперь в выражении throw range(s) в поле i объекта класса range будет записано значение ошибочного индекса s.

Для того чтобы в обработчике исключения проанализировать значение ошибочного индекса, необходимо задать имя объекту-индикатору исключения:

try{/* контролируемый код*/}
catch(string : : range ob)
{cerr<<”Неверный индекс: ”<<ob.i<<endl;}

Выражение (string : : range ob) объявляет тип исключения (string : : range) и может дополнительно задавать имя объекта исключения (ob).

Полностью программа выглядит так:

```
#include <iostream.h>
#include<stdio.h>
#include<windows.h>
#include<process.h>
#define ss 20
class string
{
  char *p;
  int size;
public:
   struct range
  {int i;
  range(int s):i(s){};
  };// конец класса range
string(int SIZE){p=new char[size=SIZE];}
  ~string(){delete[] p;}
 char& operator[](int s);
};//конец класса string
char& string :: operator[](int s)
{if((s<0)||(s>size)) throw range(s);
  return p[s];}
int main()
{
SetConsoleOutputCP(1251);
int index;
string str(ss);
for(int i=0;i<ss;i++) str[i]='A'+i;
```

```
try
{
for(;;) //бесконечный цикл
 {cout<<"\nВведите индекс: ";
   cin>>index;
   cout<<str[index];}
}
catch(string ::range ob)
{cerr<<"Неверный индекс: "<<ob.i<<endl;
}
system("pause");
return 0;
}
```

Пример 5.6

В классе range задается переменная i с атрибутом **private**, получающая значение ошибочного индекса и две public функции get_index() и index_range(), которые соответственно возвращают значение ошибочного индекса и диапазон допустимых индексов.

```
class range
    {int i;
     int s;
     public:
      range(int I,int S):i(I),s(S){}
      int get_index(){return i;}
      void index_range(){cout<<"Допустимый диапазон:0 -
"<<s<<endl;}
    };
```

При ошибке создается объект range, содержащий уже два значения: ошибочный индекс и максимальное значение индекса.

```
char& string::operator[](int j)
{if((j<0)||(j>=size)) throw range(j,size);  return p[j];}
```

Таким образом, поскольку исключение – это класс, он может содержать любые поля и методы.

5.3. Определение типа исключения

Во время выполнения программы в ней могут возникать различные ошибки, которые можно сопоставить с исключениями различных типов. Например, для класса string можно рассмотреть два типа возможных ошибок: неправильное значение индекса и невозможность конструирования объекта из-за того, что задан очень большой размер строки.

```
class string
{…
public:
  class range{};
  class error_size{};
  string(int);
  ~string();
  char& operator[](int);
};
```

Мы хотим использовать исключение для того, чтобы установить, правильно ли сконструирован объект. Мы знаем, что конструктор не возвращает никаких значений, поэтому нельзя использовать традиционные подходы для проверки правильности конструирования объекта. Но эту задачу можно решить другим способом.

```
string : : string(int s)
{if((s<0)||(s>max)) throw error_size();
  p=new char[size=s];
}
```

или таким образом:

```
string : : string(int s)
{p=new char[size=s];
  if(!p) throw error_size();
}
```

Тогда прикладная программа выглядит следующим образом:

```
void main()
{…
 try
{
```

описание и выполнение каких-либо операций с объектами класса string

```
}
 catch(string : : range){…}
 catch(string : : error_size){…}
}
```

При наличии ошибки активизируется соответствующий обработчик исключений.

```
void f()
{…
 try{…}
 catch(string : : range)
   {исправление ошибки и повторный вызов функции  f();
   }
 catch(string : : error_size)
   {выдача сообщений и завершение программы
    exit(1);
   }
}
```

Здесь записана часть функции, которая будет выполнена, если либо не было исключений, либо была достигнута закрывающая фигурная скобка исключения range. Функция f может перехватывать лишь некоторые исключения. Например, можно рассмотреть такие две функции f1 и f2, что f1 перехватывает только error_size, а f2 – только range. В этом случае после блока try первой функции записывается один блок catch(string : : error_size), а после блока try второй – один блок catch(string : : range).

Также можно в программе использовать два блока try, один из которых позволяет выявлять ошибки конструирования объекта, а второй – ошибки индекса.

5.4. Иерархия исключений

Исключения можно группировать так, что в каждую группу включается некоторое подмножество близких по сути исключений.

Например:

– ошибки распределения памяти;

– ошибки ввода-вывода;

– ошибки при работе с файлами;

– ошибки при выполнении арифметических операций.

Если проанализировать все группы, то можно построить некоторую **иерархию исключений** с использованием принципов **наследования**. Можно рассмотреть базовый класс и производные классы, каждый из которых соответствует либо некоторой группе исключений, либо конкретному исключению. В качестве главного базового класса рассматривают некоторый пустой класс. Это позволяет построить иерархическую структуру с единым корнем. Тогда блоки try и catch будут иметь такой вид:

```
try{…}
catch(group_N)
{
//Обработка конкретного исключения нижнего уровня N
}
catch(group_N-1)
{
//Обработка исключений следующего уровня (класс group_N-1
// является базовым для  класса group_N
}
catch(…)
{
//Обработка всех возможных исключений
}
```

Здесь в выражении catch(…) три точки означают «любой аргумент». Поэтому обработчик catch(…) перехватывает любое

исключение. Последовательность, в которой записаны различные обработчики исключений, является существенной, поскольку производные исключения могут быть перехвачены несколькими обработчиками.

Например, если в цепочке обработчиков встретится catch(…), то все последующие обработчики будут заблокированы (они никогда не будут выполняться). Иногда обработчик исключения активизируется, но не знает, как устранить возникшую ошибку, более того, возможно, не знает и о том, что делать дальше. В этом случае он может активизировать то же исключение снова в надежде на то, что какой-то другой обработчик знает, как решить эту проблему. Повторная активизация осуществляется инструкцией throw без аргументов.

```
try{…}
catch(класс-индикатор)
  {if(знает как обработать исключение){обработка исключения}
    else throw;
  }
```

5.5. Спецификация функций, обрабатывающих исключения

Если некоторая функция содержит инструкцию throw для генерации исключения, то ее можно специфицировать (объявить) как функцию, генерирующую исключения, например:

void f(void) throw(x,y,z);

Здесь объявляется функция f, генерирующая исключения z, y, z и исключения, которые являются производными от x, y, z. Другие исключения функция не генерирует. В этом случае при возникновении исключений из указанного списка функция f должна установить индикатор для соответствующего обработчика. Если же возникает другое исключение, то f должна завершить свое выполнение некоторым выходом по критической (не-

исправимой) ошибке (например, с помощью библиотечной функции **abort**).

Важным здесь является то, что объявление функции дает пользователю информацию о том, как она взаимодействует с внешней средой. Пользователь получает представление о возможности использования этой функции для обработки тех или иных ошибок.

void f(void) – эта функция может генерировать любое исключение.

int f(void) throw() – эта функция не генерирует исключения.

Рассмотрим некоторую функцию

void f(void) throw(A);

Предположим, что функция генерирует исключение B, которое не является производным от A (т.е. исключение, которое не в списке). В этом случае автоматически вызывается библиотечная функция **unexpected** ()

По умолчанию функция unexpected вызывает другую библиотечную функцию **terminate()**

По умолчанию terminate вызывает функцию **abort()**, которая выдает сообщение:

Abnormal program termination

и завершает программу. Пользователь может заменить функцию unexpected и terminate своими. Для этого надо:

1) описать новую версию этой функции

void my_unexpected(void){описание новых действий}

void my_terminate(void){описание новых действий}

2) установить (зарегистрировать) новую функцию с помощью функций **set_unexpected()** и **set_terminate()**;

Функция terminate() будет вызвана автоматически и тогда, когда генерируется исключение и не находится соответствующий обработчик.

6. ПОТОКОВЫЕ КЛАССЫ

6.1. Библиотека потоковых классов

Потоковые классы представляют объектно-ориентированный вариант библиотечных функций ввода-вывода С. Поток данных между источником и приемником при этом обладает следующими свойствами:

Источник или приемник данных определяется объектом потокового класса.

Потоки используются для ввода-вывода высокого уровня.

Общепринятые стандартные С-функции ввода/вывода разработаны как методы потоковых классов, чтобы облегчить переход от С-функций к С++ классам.

Потоковые классы опредены как шаблоны и делятся на три группы :

♦ basic_istream, basic_ostream – общие потоковые классы, которые могут быть связаны с любым буферным объектом;

♦ basic_ifstream, basic_iostream – потоковые классы для считывания и записи файлов;

♦ basic_istringstream, basic_ostringstream – потоковые классы для объектов-строк.

Каждый потоковый класс поддерживает буферный объект, который предоставляет память для передаваемых данных, а также важнейшие функции ввода/вывода низкого уровня для их обработки.

Базовым шаблоном классов basic-ios (для потоковых классов) и basic-streambuf (для буферных классов) передаются по два параметра шаблона:

♦ первый параметр (charT) определяет символьный тип;

♦ второй параметр (traits) – объект типа ios-traits (шаблон класса), в котором заданы тип и функции, специфичные для используемого символьного типа;

♦ для типов char и wchar_t образованы соответствующие объекты типа ios_traits и потоковые классы.

Например, шаблон класса для потокового ввода определен следующим образом:

template <class charT, class traits = ios_traits <charT>> class basic_istream: virtual public basic_ios <charT, traits>;

6.2. Ввод-вывод в языке C++

Ввод-вывод – одна из самых сложных областей любого языка. Она тесно связана и с операционной системой, и с оборудованием.

Система ввода-вывода, принятая в ANSI, хороша, но не идеальна. Она не учитывает огромной разницы между устройствами, с точки зрения программиста. Все устройства можно разделить на блочные и символьные. Например, диски и ленты – это блочные устройства, тогда как терминалы – символьные. Блочные устройства для эффективного чтения-записи требуют буферизации , а символьные в ней не нуждаются. Устройства ввода-вывода имеют и много других различий. В библиотеку C (stdio.h) были введены некоторые исключения для обработки специальных устройств. В конце 60-х годов обращение к устройствам, как к файлам, было новинкой. Однако в большинстве случаев программист не работает с устройствами, как с файлами. Это приводит к добавлению в C новых функций, поскольку иногда программист должен знать, работает он с дисковым файлом или с устройством.

Существует множество примеров того, что функции, применяемые к устройствам, а не к нормальным файлам, ведут себя по-разному, а иногда и некорректно.

C++ – это не первый язык, работающий с потоками. ANSI C использует так называемые потоки для указания на некоторый абстрактный порт, через который могут передаваться неструктурированные данные. В ANSI C поток рассматривается как

конструкция ввода-вывода низкого уровня, на которую наслаивается более структурированная файловая система.

В C++ потоки реализованы через целую иерархию классов, которая является единственной основной иерархией, поставляемой со всеми компиляторами языка C++.

В C++ потоки составляют некую оболочку, которая придает операциям ввода-вывода полиморфные и объектно-ориентированные свойства. Несмотря на то, что потоки часто ассоциируются с вводом-выводом, реально они являются абстракцией передачи данных от одного объекта к другому с помощью механизма буферизации. Это означает, что любая функция, используемая для передачи данных из одного места в памяти в другое (с изменением данных или без такового), может рассматриваться как потоковая операция.

Библиотека потоковых классов C++ построена на основе двух базовых классов: ios и streambuf .

Класс streambuf и его потомки обеспечивают организацию и взаимосвязь буферов ввода-вывода, размещаемых в памяти, с физическими устройствами ввода-вывода. Методы и данные класса streambuf программист явно обычно не использует. Этот класс нужен другим классам библиотеки ввода-вывода. Он доступен и программисту для создания новых классов на основе уже существующих. На рис. 6.1 показана иерархия буферных классов.

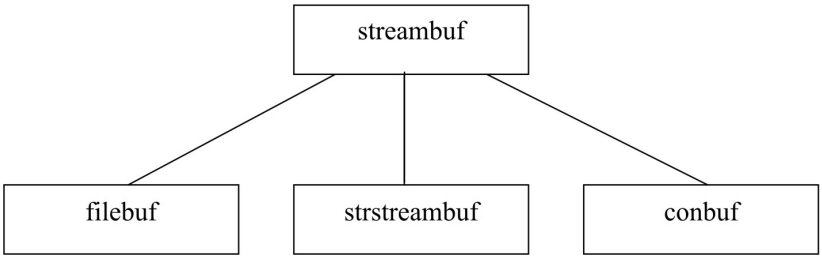


Рис. 6.1

Класс ios и его потомки содержат средства для форматированного ввода-вывода и проверки ошибок. Иерархия классов ввода-вывода представлена на рис. 6.2.

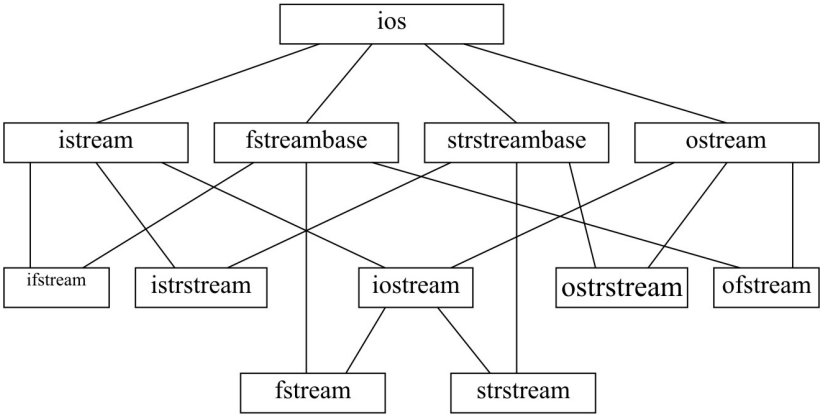


Рис. 6.2

Потоковые классы, их методы и данные становятся доступными в программе, если в неё включен нужный заголовочный файл.

Итак, потоки C++ обеспечивают:

– буферизацию при обменах с внешними устройствами;

– независимость программы от файловой системы конкретной ЭВМ;

– контроль типов передаваемых данных;

– возможность удобного обмена для типов, определенных пользователем.

6.3. Стандартные потоки ввода-вывода

В файле iostream.h определены следующие объекты, связанные со стандартными потоками ввода-вывода.

cin – объект класса istream, связанный со стандартным буферизированным входным потоком.

cout – объект класса ostream, связанный со стандартным буферизированным выходным потоком.

cerr – объект класса ostream, связанный со стандартным небуферизированным выходным потоком для сообщения об ошибках.

clog – объект класса ostream, связанный со стандартным буферизированным выходным потоком для сообщения об ошибках.

Для класса istream перегружена операция >>, а для класса ostream <<. Операции << и >> имеют два операнда. Левым операндом является объект класса istream (ostream), а правым – данное, тип которого задан в языке.

Для того чтобы использовать операции << и >> для всех стандартных типов данных, используется соответствующее число перегруженных функций operator<< и operator>>. При выполнении операций ввода-вывода в зависимости от типа правого операнда вызывается та или иная перегруженная функция operator.

Поддерживаются следующие типы данных: целые, вещественные, строки (char*), для вывода – void* (все указатели, отличные от char*, автоматически переводятся к void*).

Функции operator<< и operator>> возвращают ссылку на тот потоковый объект, который указан слева от знака операции. Таким образом, можно формировать «цепочки» операций.

```
cout << a << b << c;
cin >> i >> j >> k;
```

При записи цепочек операторов вывода нужно не забывать о приоритете операций.

Зависимость от компилятора результатов выполнения цепочки операций вывода и необходимость аккуратно учитывать приоритеты операций приводят к следующей рекомендации: ***изменяемая переменная не должна появляться в цепочке вывода более одного раза.***

Извлечение данных из потока начинается только после нажатия клавиши «Enter». При этом при извлечении числовых данных (без буфера) игнорируются начальные пробельные символы.

Чтение начинается с первого непробельного символа и заканчивается при появлении нечислового символа.

Значения указателей (т.е. адреса) выводятся в стандартный поток в шестнадцатеричном виде.

Ввод-вывод массивов и символьных массивов-строк – это различные процедуры.

```
char st[] = ”строка”;
char* pst = st;
cout << st;
cout << pst;
```

В обоих случаях будет выведено – **строка**. Операция <<, настроенная на операнд char*, всегда выводит строку, а не значение указателя. Чтобы вывести указатель, необходимо явное приведение типа.

При вводе строки с клавиатуры набираются любые символы до тех пор, пока не будет нажата клавиша Enter. Система ввода-вывода переносит эту последовательность в буфер входного потока, а из буфера при выполнении каждой операции >> извлечение происходит до ближайшего пробела.

6.4. Форматирование

Непосредственное применение операций ввода << и вывода >> к стандартным потокам cout, cin, cerr, clog для данных базовых типов приводит к использованию «умалчиваемых» форматов внешнего представления пересылаемых значений.

Форматы представления выводимой информации и правила восприятия данных при вводе могут быть изменены программистом с помощью флагов форматирования. Эти флаги унаследованы всеми потоками из базового класса ios.

Флаги форматирования определены в public перечислении класса ios следующим образом:

```
enum {
skipws = 0X0001;
```

```
left      = 0X0002;
right     = 0X0004;
// и т.д.
stdio     = 0X4000;
};
```

Таким образом, флаги форматирования реализованы в виде отдельных фиксированных битов и хранятся в protected компоненте класса long x_flags.

Открытая функция long flags() возвращает значение переменной x_flags. Функция long flags(long f) устанавливает x_flags в новое значение f и возвращает предыдущее значение x_flags.

Функция long setf(long s, long f) устанавливает те биты в x_flags, которые в s и f имеют значение 1. Все биты, которые в s имеют значение 0, а в f – 1, сбрасываются. Возвращает предыдущее значение x_flags, логически умноженное на f.

Функция long setf(long s) устанавливает те биты в x_flags, которые в s равны 1. Те биты, которые в s равны 0, остаются без изменения. Возвращает предыдущее значение x_flags, логически умноженное на s.

Функция long unsetf(long s) сбрасывает те биты, которые в s равны 1. Биты, которые в s равны 0, остаются без изменений. Возвращает предыдущее значение x_flags, логически умноженное на s.

Кроме флагов форматирования используются следующие protected компонентные данные класса ios:

int x_width; – минимальная ширина поля вывода;

int x_precision; – точность представления вещественных чисел (количество цифр дробной части) при выводе;

int x_fill; – символ заполнитель при выводе, по умолчанию – пробел.

Для получения (установки) значений этих полей используются следующие public компонентные функции:

```
int width();
int width(int);
```

```
int precision();
int precision(int);
char fill();
char fill(char);
```

6.5. Манипуляторы

Несмотря на гибкость и большие возможности управления форматами с помощью компонентных функций класса ios, их применение достаточно громоздко. Более простой способ изменения параметров и флагов форматирования обеспечивают манипуляторы.

Манипуляторами называются специальные функции, позволяющие модифицировать работу потока. Особенность манипуляторов состоит в том, что их можно использовать в качестве правого операнда операции >> или <<. В качестве левого операнда, как обычно, используется поток (ссылка на поток), и именно на этот поток воздействует манипулятор.

Для обеспечения работы с манипуляторами в классах istream и ostream имеются следующие перегруженные функции operator:

```
istream& operator>>(istream&(*_f)( istream&));
ostream& operator<<(ostream&(*_f)(ostream&));
```

При использовании манипуляторов следует включить заголовочный файл <iomanip.h>.

Приведем список встроенных манипуляторов:

dec – устанавливает десятичную систему счисления;

hex – шестнадцатеричную;

oct – восьмеричную;

endl – вставляет в поток вывода символ новой строки и затем сбрасывает поток;

ends – вставляет ‘\∅’ в поток вывода;

flush – сбрасывает поток вывода;

ws – выбирает из потока ввода пробельные символы, поток будет читаться до появления символа, отличного от пробельного;

setbase(int) – устанавливает основание системы счисления;

resetiosflags(long f) – сбрасывает флаги форматирования, для которых в f установлен бит в 1;

setiosflags(long f) – устанавливает те флаги форматирования, для которых в f соответствующие биты установлены в 1;

setfill(int c) – устанавливает символ-заполнитель (c – код символа-заполнителя);

setprecision(int n) – устанавливает точность при выводе вещественных чисел;

setw(int n) – устанавливает минимальную ширину поля вывода.

6.6. Ввод-вывод объектов пользовательских классов

Чтобы использовать операции >> и << с данными типов, определяемых пользователем, необходимо расширить действие этих операций, введя новые операции-функции. Первым параметром операции-функции должна быть ссылка на объект потокового типа, вторым – ссылка или объект пользовательского типа.

Пример 6.1

```
#include <iostream.h>
//Точка в трехмерном пространстве
class Point{
float x,y,z;
public:
Point(float i,float j,float k):x(i),y(j),z(k){};
friend ostreeam operator<<(ostream& stream,Point& p);
friend istreeam operator>>(istream& stream,Point& p);
};

ostream& operator<<(ostream& stream,Point& p)
{
```

```
return stream<<"\nx="<<p.x<<"  y="<<p.y<<"  z="<<p.z<<endl;
}
istream& operator>>(istream& stream,Point& p)
{cout<<"\nx="; stream>>p.x;
cout<<"y="; stream>>p.y;
cout<<"z="; stream>>p.z;
return stream;
}
```

При перегрузке операторов ввода-вывода для пользовательских типов можно предусмотреть определенный контроль, например, читать только некоторые символы из потока и пропускать все остальные. Это позволит вводить данные в более гибком формате. Например, для класса Point приемлемы следующие входные потоки:

```
10 20 30
(10, 20, 30)
(10
20
30)
```

Пример 6.2

```
#include<ctype.h>
#include <iostream.h>
#include<process.h>
#include<windows.h>
//Точка в трехмерном пространстве
class Point{
float x,y,z;
public:
Point(float i,float j,float k):x(i),y(j),z(k){}
friend ostream& operator<<(ostream& os,Point& p);
friend istream& operator>>(istream& is,Point& p);
};
```

```
ostream& operator<<(ostream& os,Point& p)
{return os<<"\nx="<<p.x<<"   y="<<p.y<<"   z="<<p.z<<endl;}
// Функция помещает в входной поток только символы чисел
void SkipNoDigits(istream& is)
{char c;
for(;;)
{is>>c;
if(isdigit(c)||c=='-')
{
is.putback(c);  //поместить символ в входной поток
return;}
}}

//Перегрузка операции ввода
istream& operator>>(istream& is,Point& p)
{SkipNoDigits(is);
is>>p.x;
SkipNoDigits(is);
is>>p.y;
SkipNoDigits(is);
is>>p.z;
return is;
}

void main()
{
SetConsoleOutputCP(1251);
Point a(1.1,2.2,3.3);
cout<<"Старые координаты"<<a;
cout<<"Введите новые координаты:";
cin>>a;
cout<<"Новые координаты"<<a;
system("pause");
}
```

Ниже показан пример выполнения программы (рис. 6.3).

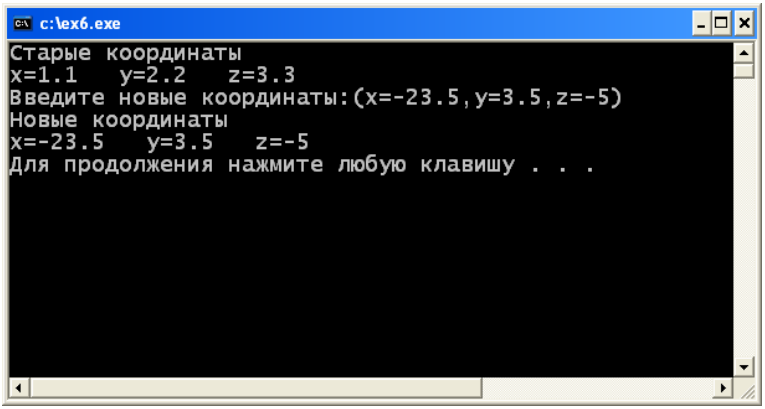


Рис. 6.3

6.7. Определение пользовательских манипуляторов

В ситуации, когда нужно повторно определять одну и ту же команду форматирования, пользовательский манипулятор обеспечивает простую запись, нечто вроде макро.

Манипулятор без параметра создается следующим образом. Определяется класс с функцией operator<<(), которая, в свою очередь, вызывает определенные функции форматирования.

Пример 6.3

Определим класс манипулятора

```
class my_f{};
```

Перегрузим функцию-оператор operator<< для объекта класса манипулятора. Манипулятор позволит выполнить выравнивание по правому краю. По умолчанию выравнивание по левому краю.

```
ostream& operator<<(ostream& out,my_f)
{
out.width(12);
```

```
out.fill('# ');
cout.setf(ios:: right,ios::adjustfield);
 return out;
}
void main()
{my_f MF;
cout<<52.3456<<endl<<MF<<52.3456<<endl;
}
```

Синтаксис языка разрешает параметры без имен, если последние не используются в теле функции. В стандарте языка C++ отмечается, что использование специального параметра без имени «полезно для резервирования места в списке параметров». В дальнейшем этот параметр может быть введен в функции без изменения интерфейса, т.е. без изменения вызывающей программы. Такая возможность бывает удобной при развитии уже существующей программы за счет изменения входящих в нее функций.

Поскольку манипулятор – это функция, принимающая и возвращающая ссылку на поток, то лучше определить манипулятор MF, как показано ниже.

Пример 6.4

```
ostream& MF(ostream& out)
{
return out<<resetiosflags(ios::right)<<setiosflags(ios::right);
}
```

Примеры 6.3 и 6.4 показывают два различных подхода к созданию пользовательских манипуляторов. В первом случае мы, по сути, не создавали манипулятор, а перегрузили операцию >> для класса my_f. Во втором случае мы создали собственно функцию-манипулятор. Вызов её обеспечивает перегруженная операция << в классе ostream.

6.8. Пользовательские манипуляторы с параметрами

Пусть, например, необходимо создать новый манипулятор с параметрами wp(n,m), где n – ширина поля вывода, m – точность.

Пример 6.5

```
//Класс манипулятора
class my_manip {
int n,m;
ostream&(*f)(ostream&,int,int);
public:
//конструктор
my_manip(ostream&(*F)(ostream&,int,int),int N, int M):
f(F),n(N),m(M){}
//здесь f,F – указатели на функцию
friend ostream& operator<<(ostream& s,my_manip& my)
{return my.f(s,my.n,my.m);}
};
ostream& wi_pre(ostream& s,int n,int m)
{s.width(n);
s.flags(ios::fixed);
s.precision(m);
return s;
}
my_manip wp(int n,int m)
{return my_manip(wi_pre,n,m);
}
void main()
{cout<<52.3456<<endl<<wp(10,5)<<52.3456<<endl;
}
```

Рассмотрим этот пример подробнее. Конструктор класса my_manip имеет первый аргумент ostream&(*F)(ostream&,int,int). Здесь F – указатель на функцию, имеющую три аргумента типов

ostream&, int и int и возвращающую ссылку на класс ostream. Теперь конкретную функцию можно задавать для класса my_manip через указатель при конструировании объекта этого класса. Мы задали функцию wi_pre. При выполнении cout<<wp(10,5) фактически вызывается следующая функция:

osteram& operator<<(cout,wp(10,5))

{return wi_pre(cout,10,5);}

Действительно, функция wp возвращает объект класса my_manip с передачей в его конструктор параметров: wi_pre, n и m.

Сформулируем последовательность создания пользовательского манипулятора с параметрами:

1. Определить класс (my_manip) с полями: параметры манипулятора + поле – указатель на функцию типа

ostream& (*f)(ostream&,<параметры манипулятора>);

2. Определить конструктор этого класса (my_manip) с инициализацией полей.

3. Определить в этом классе дружественную функцию – operator<<. Эта функция в качестве правого аргумента принимает объект класса (my_manip), левого аргумента (операнда) поток ostream и возвращает поток ostream как результат выполнения функции f.

4. Определить функцию типа *f(wi_pre), принимающую поток и параметры манипулятора и возвращающую поток. Эта функция выполняет форматирование.

5. Определить собственно манипулятор (wp) как функцию, принимающую параметры манипулятора и возвращающую объект my_manip, поле f которого содержит указатель на функцию wi_pre.

При выполнении манипулятора cout<<wp(10,5); вызывается функция wp, которая создает и возвращает объект my_manip, в поле f которого содержится указатель на функцию wi_pre.

Затем вызывается функция operator<<, которой передается объект my_manip, созданный функцией wp.

Наконец, функция operator<< выполняет функцию *f, т.е. wi_pre, и возвращает поток.

6.9. Использование макросов для создания манипуляторов

Для создания манипуляторов можно использовать макросы OMANIP(int), IMANIP(int), IOMANIP(int).

Эти макросы содержатся в файле <iomanip.h>

Ниже приведены примеры использования этих макросов:

а) манипулятор с одним параметром

```
ostream& w(ostream& out,int n)
{out.width(n);
//out<<setw(n);
return out;
}
OMANIP(int) w(int n)
{return OMANIP(int)(w,n);
}
```

Здесь две функции с именем w, вторая на первый взгляд кажется ненужной. Однако OMANIP(int) – это макрос, позволяющий определить манипулятор, который принимает параметры. Обнаружив в операции << манипулятор, компилятор использует вторую функцию для вызова первой;

б) манипулятор с двумя параметрами.

Проблема здесь в том, что макросы принимают только один параметр, так что для передачи нескольких параметров придется использовать структуру или класс. Например, для двух параметров:

```
struct Point
{
int n;
int m;
};
```

Перед использованием в манипуляторе Point нужно вызвать IOMANIPdeclare(Point). IOMANIPdeclare(Type) принимает только идентификатор, так что передача указателей или ссылок требует использования typedef.

```
#include<iostream.h>
#include<iomanip.h>
struct Point
{int x,y;};
IOMANIPdeclare(Point);
ostream& wp(ostream& out,Point p)
{out.width(p.x);
out.flags(ios::fixed);
out.precision(p.y);
return out;
}
OMANIP(Point) wp(int x,int y)
{Point p;
p.x=x;
p.y=y;
return OMANIP(Point)(wp,p);
}
```

6.10. Состояние потока

Каждый поток имеет связанное с ним состояние. Состояния потока описываются в классе ios в виде перечисления enum.

```
public:
enum io_state
{
goodbit,//нет ошибки 0X00
eofbit,//конец файла 0X01
failbit,//последняя операция не выполнилась 0X02
badbit,//попытка использования недопустимой операции 0X04
hardfail   //фатальная ошибка 0X08
};
```

Флаги, определяющие результат последней операции с объектом ios, содержатся в переменной state. Получить значение этой переменной можно с помощью функции int rdstate();

Кроме того, проверить состояние потока можно следующими функциями:

int bad(); возвращает 1, если badbit или hardfail

int eof(); возвращает, если eofbit

int fail(); возвращает, если failbit, badbit или hardfail

int good();возвращает, если goodbit

Если попытаться выполнить ввод из потока, который находится не в состоянии good(), то возвращается значение NULL. Если осуществляется чтение в переменную и происходит ошибка, то значение переменной не изменяется (предполагается, что переменная имеет стандартный тип, определенный в самом языке). Если операция >> используется для новых типов данных, то при её перегрузке необходимо предусмотреть соответствующие проверки.

Установить состояние потока в заданное значение можно функцией void clear(int s=0);

Эта функция устанавливает флаги ios::state в значение, заданное параметром s, не изменив при этом бита hardfail.

Пример 6.6

```
#include<iostream.h>
#include<stdlib.h>
void main()
{int flags;
int k;
cin>>k;
flags=cin.rdstate();
if(flags) //если ошибка
  if(flags& ios:badbit)
    {cout<<”Ошибка\n”;
    cin.clear(0);}
  else{cerr<<”Эту ошибку нельзя исправить\n”;
    abort();}
cout<<”Ввод без ошибок\n”;
cout<<k<<endl;
}
```

Пример 6.7

```
#include<iostream.h>
#include<stdlib.h>
void main()
{int k;
cin>>k;
if(!cin)
  if(cin.bad())
    {cout<<”Ошибка\n”;
    cin.clear(0);}
  else
    {cerr<<”Неисправимая ошибка\n”;
    abort();}
cout<<”Ввод без ошибок\n”;
cout<<k<<endl;
}
```

Здесь используется перегруженная int operator!();

Результат операции – значение функции ios::fail().

Есть еще operator void*(), который возвращает 0, когда fail() возвращает 1, в противном случае возвращает указатель this на объект ios.

6.11. Неформатированный ввод-вывод

В классе **istream** определены следующие функции:

1) istream& get(char* buffer,int size,char delimiter=’\n’);

Эта функция извлекает символы из istream и копирует их в буфер. Операция прекращается при достижении конца файла, либо когда будет скопировано size символов, либо при обнаружении указанного разделителя. Сам разделитель не копируется и остается в streambuf. Последовательность прочитанных символов всегда завершается нулевым символом;

2) istream& read(char* buffer,int size);

Не поддерживает разделителей, и считанные в буфер символы не завершаются нулевым символом. Количество считанных символов запоминается в istream::gcount_ (private);

3) istream& getline(char* buffer,int size, char delimiter='\n');

Разделитель извлекается из потока, но в буфер не заносится. Это основная функция для извлечения строк из потока. Считанные символы завершаются нулевым символом;

4) istream& get(streambuf& s,char delimiter='\n');

Копирует данные из istream в streambuf до тех пор, пока не обнаружит конец файла или символ-разделитель. Последний не извлекается из istream. В «s» нулевой символ не записывается;

5) istream get (char& C);

Читает символ из istream в C. В случае ошибки C принимает значение 0XFF;

6) int get();

Извлекает из istream очередной символ. При обнаружении конца файла возвращает EOF;

7) int peek();

Возвращает очередной символ из istream, не извлекая его из istream;

8) int gcount();

Возвращает количество символов, считанных во время последней операции неформатированного ввода;

9) istream& putback(C)

Если в области get объекта streambuf есть свободное пространство, то туда помещается символ C;

10) istream& ignore(int count=1,int target=EOF);

Извлекает символ из istream, пока не произойдет следующего:

– функция не извлечет count символов;

– не будет обнаружен символ target;

– не будет достигнуто конца файла.

В классе **ostream** определены следующие функции:

1) ostream& put(char C);

Помещает в ostream символ C;

2) ostream& write(const char* buffer,int size);

Записывает в ostream содержимое буфера. Символы копируются до тех пор, пока не возникнет ошибка или не будет скопировано size символов. Буфер записывается без форматирования. Обработка нулевых символов ничем не отличается от обработки других. Данная функция осуществляет передачу необработанных данных (бинарных или текстовых) в ostream;

3) ostream& flush();

Сбрасывает буфер streambuf.

Следующие функции позволяют использовать прямой доступ:

1) istream& seekg(long p);

Устанавливает указатель потока get (не путать с функцией) со смещением «p» от начала потока;

2) istream& seekg(long p,seek_dir point);

Указывается начальная точка перемещения:

enum seek_dir{beg,curr,end}

Положительное значение «p» перемещает указатель get вперед (к концу потока), отрицательное значение «p» – назад (к началу потока);

3) long tellg();

Возвращает текущее положение указателя get.

4) ostream& seekp(long p);

Перемещает указатель put в streambuf на позицию «p» от начала буфера streambuf;

5) ostream& seekp(long p,seek_dir point);

Указывается начальная точка перемещения;

6) long tellp();

Возвращает текущее положение указателя.

Пример 6.8. Использование istream с filebuf для чтения

```
#include<fstream.h>
void main()
{char buffer[80];
filebuf dir_file;
```

```
//присоединение файла к объекту filebuf
if(!dir_file.open(“dir.txt”,ios::in))
{cout<<“Ошибка\n”;
return;}
//создание потока ввода с присоединенным файлом
istream dir_stream(&dir_file);
for(;;);
{//считывание очередной строки текста
dir_stream.getline(buffer,sizeof(buffer));
if(dir_stream.eof())break;
//печать строки текста
cout<<“\n”<<buffer;
}
```

Нет необходимости закрывать файл явно: деструктор для dir_file сделает это автоматически.

Пример 6.9. Чтение текстового файла

```
#include<fstream.h>
void main()
{ifstream fin(“c:\\user\\my.txt”);
if(!fin){cout<<“\nОшибка”; return;}
while(fin)
{char buffer[80];
fin.getline(buffer,sizeof(buffer));
cout<<“\n”<<buffer;}
}
```

Пример 6.10.Чтение текстового файла в бинарном режиме

```
#include<fstream.h>
void main()
{ifstream fin(“имя”,ios::binary);
if(!fiin){cout<<“\nОшибка”; return;}
while(fin)
{char c;
```

```
fin.get(c);
cout<<c;}
}
```

Выходной поток можно прочитать и с помощью операции >>

Например, dir_stream >>buffer;

Для копирования всего файла используется и более быстрый способ, который связан с применением

istream::operator>>(streambuf*)

Эта функция выполняет все сразу без необходимости использования цикла и проверки конца файла. Она копирует символы из istream в streambuf, начиная с текущей позиции потока. Копирование прекращается при достижении конца файла.

Цикл for в нашем примере можно заменить одной строкой ввода dir_stream>>cout.rdbuf();

Выражение cout.rdbuf() возвращает указатель на streambuf, используемый стандартным потоком вывода.

ofstream::filebuf* rdbuf();

Затем вызывается функция istream::operator>>(strstreambuf*), которая копирует символы потока в буфер до тех пор, пока не будет достигнут конец файла или не возникнут какие-либо ошибки.

Пример 6.11. Запоминание текущего каталога в файле

```
#include<fstream.h>
#include<dir.h>
void main()
{char buffer[80];
filebuf dir_file;
if(!dir_file.open("dir.txt",ios::out)){cout<<"Ошибка\n"; return;}
//создание потока вывода с присоединенным файлом
ostream dir_stream(&dir_file);
//определение пути к текущему каталогу
if(getcwd(buffer,sizeof(buffer))==0){cout<<"Ошибка\n"; return;}
//сохранение пути
dir_stream<<"\nДиректорий: "<<buffer<<"\n\n";
```

```
//сохранение каталога в файле
struct ffblk file_block;
if(findfirst“*.*”,&fileblock,0))
{dir_stream<<“Ошибка доступа к каталогу\n”; return;}
do dir_stream<< file_block.ff_name<<“\n”;
while(!findnext(&file_block));
}
```

Пример 6.12. Запись в поток текстового файла

```
#include<fstream.h>
void main()
{ifstream fin(“имя”);
ofstream fout(“имя”);
if(!fin){cout<<“\nОшибка”; return;}
if(!fout){cout<<“\nОшибка”; return;}
//копирование пяти строк файла fin
int count=0;
while(fin && fout){
char buffer[80];fout<<buffer<<“\n”;
if(++count==5)break;}
}
```

В этих примерах используется следующая схема создания потока:

1. Создается объект класса filebuf:

```
filebuf dir_file;
```

2. Объект filebuf связывается с устройством (в нашем примере с дисковым файлом, который открывается либо в режиме in, либо out):

```
dir_file.open(“dir.txt”,ios::in);
```

При успешном выполнении функции возвращает указатель на собственный объект класса, в противном случае – нуль.

3. Создается поток как объект класса istream или ostream и связывается с объектом типа filebuf:

```
istream dir_stream(&dir_file);
```

Конструктор создает объект dir_strcom и связывает его с dir_file.

6.12. Файловый ввод-вывод

Потоки для работы с файлами создаются как объекты следующих классов:

ofstream – запись в файл;

ifstream – чтение из файла;

fstream – чтение/запись.

Для создания потоков имеются следующие конструкторы:

1) fstream();

создает поток, не присоединяя его ни к какому файлу;

2) fstream(int file_descriptor);

создает поток и присоединяет его к уже открытому файлу;

3) fstream(int fd,char* buffer,int size);

Этот конструктор получает буфер buffer размером size и использует его для создания внутреннего объекта filebuf. Поток присоединяется к уже открытому файлу;

4) fstream(const char* name,int mode,int p=filebuf::openprot);

создает поток, присоединяет его к файлу с именем name, предварительно открыв файл, устанавливает для него режим mode и уровень защиты p. Если файл не существует, то он создается. Для m=ios::out, если файл существует, то его размер будет усечен до нуля. Флаги режима имеют следующие значения:

enum ios::open_mode{

in=0X01,//для чтения

out=0X02, //для записи

ate=0X04, /*индекс потока помещен в конец файла. Чтение больше недопустимо, //выводные данные записываются в конец файла. При открытии файла //ищется конец файла. Далее можно указать seekp(k);*/

app=0X08, /*поток открыт для добавления данных в конец. Независимо от seekp будет писаться в конец */

trunc=0X10, //усечение существующего потока до нуля

nocreate=0X20, /*команда открытия потока будет завершена неудачно, если файл не существует*/

noreplace=0X40, /*команда открытия потока будет завершена неудачно, если файл существует*/

binary=0X80, //поток открывается для двоичного обмена

};

static const int filebuf::openprot указывает на тип защиты доступа к файлам. Система DOS предлагает ограниченный набор защитных механизмов, поэтому по умолчанию защита файлов установлена в(S_IREAD|S_IWRITE). Системы UNIX могут использовать openprot для предоставления определенным категориям пользователей прав на чтение, запись или выполнение файла.

Если при создании потока он не присоединен к файлу, то присоединить существующий поток к файлу можно функцией.

void open(const char* name,int mode,int p=filebuf::openprot);

Функция void fstreambase::close(); сбрасывает буфер потока, отсоединяет поток от файла и закрывает файл. Эту функцию необходимо явно вызвать при изменении режима работы с потоком. Автоматически она вызывается только при завершении программы.

Пример 6.13. Запись в бинарные файлы

```
#include<iostream.h>
#include<fstream.h>
struct Person{
char name[10];
int age;
float salary;};
void main
{//открытие файла
ofstream fout(“person.dat”,ios::binary);
//проверка на ошибки открытия
if(!fout){cout<<“Ошибка\n”; return;}
```

```
Person employee;
int k=0;
while(k<10 && fout)
{cout<<“name=”;
cin>> employee.name;
cout<<“age=”;
cin>> employee.age;
cout<<“salary=”;
cin>> employee.salary;
fout.write((char*)&employee,sizeof(employee));
k++;}
fout.seekp(0,ios::end);
long size=fout.tellp();
cout<<“записано  ”<<size<<“  байт\n”;
}
```

Пример 6.14. Чтение бинарного файла

```
#include<iostream.h>
#include<fstream.h>
struct Person;
void main()
{ifstream fin(“person.dat”,ios::binary);
if(!fin){cout<<“Ошибка\n”; return;}
fin.seekg(0,ios::end);
long size=fin.tellg();
cout<<“В файле  ”<<size<<“  байт\n”;
fin.seekg(0,ios::beg);
Person employee;
int k=0;
fin.read((char*)&employee,sizeof(employee));
while(fin)
{cout<<“name=”<<employee.name<<“  age=”<<employee.age
   <<“  salary=”<<employee.salary<<endl;
k++;
```

```
fin.read((char*)&employee,sizeof(employee));}
cout<<“прочитано ”<<k<<“ записей\n”;
}
```

Пример 6.15. Запись в бинарный файл структуры перегруженной операцией <<

```
#include<fstream.h>
struct Person;
ofstream& operator<<(ofstream& out, Person& item)
{out.write((char*)& item,sizeof(item));}
void main()
{ofstream fout(“person.dat”,ios::binary);
if(!fout){…}
int k=0;
TPerson employee;
while(k<10 && fout)
{//сформировать employee
fout<<employee;
k++;}
fout.seekp(0,ios::end);
long size=fout.tellp();
cout<<“записано ”<<size<<“ байт\n”;
}
```

Общая схема записи в файл:

1) создать потоковый объект

```
fstream f;
```

2) открыть для записи

```
f.open(“имя”,ios::out|ios::binary);
```

3) выполнять в цикле запись из переменной в файл

```
f.write((char*)&zap,sizeof(zap));
```

4) закрыть файл

```
f.close();
```

Общая схема чтения из файла:

1) создать потоковый объект

fstream f;

2) открыть для чтения

f.open(“имя”,ios::in|ios::binary);

3) выполнить в цикле чтение из файла в переменную

f.read((char*)&zap,sizeof(zap));

Открыть файл для изменений (чтения/записи) следует следующим образом:

f.open(“имя”,ios::binary|ios::in|ios::out);

Если указать режим только out без in, то файл усекается до нуля.

Если требуется только писать в файл (в любую позицию), то следует открыть его следующим образом:

f.open(“имя”,ios::binary|ios::out|ios::ate);

Если указать только ate без out, то файл не откроется. Если вместо ate указать app, то данные будут писаться всегда в конец файла независимо от seekp().

Таким образом, открытие файла на запись в режиме ios::out приведет к усечению существующего файла до нуля. Для того чтобы это предотвратить, нужно использовать режим ios::ate. При этом после открытия индекс файла переместится в конец. Затем, если необходимо, перемещение по файлу осуществляет seekp(). Если при открытии был использован режим ios::app, то все данные будут записываться в конец файла, независимо от того, какие команды seekp() выдавались. Однако команда seekg() все же влияет на операцию чтения.

Команда открытия файла выполняется успешно, независимо от того, существует файл или нет. Чтобы предотвратить открытие уже существующего файла, нужно воспользоваться режимом открытия ios::noreplace. Чтобы возникла ошибка, когда файла нет, нужно воспользоваться режимом ios::nocreate.

Как проверить файловый поток на ошибки?

Пусть имеется следующий поток fstream stream. Тогда:

if(!stream) //ошибка была

if(stream) //ошибки не было

Более подробную информацию об ошибке можно получить с помощью следующих функций:

if(stream.bad()) //были ошибки

if(stream.eof()) //конец файла

if(stream.good()) //ошибок не было

Если произошла ошибка потока, то дальнейшая работа с ним блокируется. Для продолжения работы необходимо очистить все биты ошибок потока, выполнив функцию clear(), например stream.clear();

Если установлен бит ошибки при чтении за концом файла, то для продолжения работы с файлом следует сбросить бит ios::eofbit и переместить указатель get в начало файла.

stream.clear();stream.seekg(0);

Если требуется проверить, открыт ли файл, то следует использовать функцию is_open(), которая возвращает true, если файл открыт, например,

if(!stream.is_open()) //файл не открыт

В заключение перечислим возможные способы открытия файла для операций ввода/вывода:

1. Создается объект filebuf

filebuf fbuf;

Объект filebuf связывается с устройством файлом, который открывается в требуемом режиме

fbuf.open(“имя”,ios::in);

Создается поток и связывается с filebuf

istream stream(&fbuf);

2. Создается поток fstream (ifstream, ofstream)

fstream stream;

Открывается файл, который связывается через filebuf с потоком

stream.open(“имя”,ios::in);

3. Создается объект fstream, одновременно открывается файл, который связывается с потоком

fstream stream(“имя”,ios::in);

4. Файл открывается функцией ANSI C

FILE* f;

f=fopen(“имя”,”r”);

Создается поток fstream (ifstream, ofstream) и связывается с файлом

fstream stream(*f.fd);

7. НОВЫЕ ВОЗМОЖНОСТИ ЯЗЫКА C++

7.1. Пространство имен

Пространства имен предназначены для локализации имен идентификаторов во избежание конфликтов имен. До введения понятия пространств имен все имена используемых в среде программирования C++ переменных, функций и классов находились в одном глобальном пространстве имен, и возникало множество конфликтов. Конфликты могут быть двух типов.

1. Вы работаете в группе разработчиков и ввели в своей программе класс Dog. Кому-то из ваших коллег тоже понадобилось ввести класс Dog. Само по себе это не так уж страшно, поскольку если вы попытаетесь объединить ваши программы, то компилятор обнаружит ошибку дублирования имен. Гораздо хуже, если конфликт имен возникает из-за наличия одинакового имени где-нибудь внутри библиотеки классов, которую вы купили для использования в вашем проекте. Особенно вероятна такая ситуация, когда в одной и той же программе используются библиотеки функций и классов разных производителей. Если библиотеки поставляются без исходных текстов, а только с заголовочными файлами, вы окажетесь в очень затруднительной ситуации.

2. Конфликт второго рода связан с сокрытием переменных. Например, имеется следующий текст:

```
void main()
{int x;
…
{int x;
…}
…}
```

В момент входа во внутренний блок локальная переменная **x** скрывает внешнюю **x** из main().

Сокрытие переменных в C++ является на первый взгляд достоинством языка, а не недостатком. Оно облегчает инкапсуляцию. С другой стороны, также нетрудно вообразить случай, когда программист полагает, что он использует одну переменную, в то время как в действительности использует совершенно другую. Множество различных определений переменных, вложенных одно в другое и составленных пирамидой благодаря наследованию, может привести к ситуации, когда происходит непреднамеренное сокрытие переменных и в конце концов неправильное их использование.

Ясно, что необходим некоторый механизм, который позволил бы квалифицировать (уточнять) переменные и тем самым избавлять программы от возможных двусмысленностей.

Этим механизмом и является понятие пространств имен и ключевое слово **namespace**.

Основная форма объявления пространства имен

namespace имя**{**

//объявления

}

Все, что определено внутри инструкции namespace, находится внутри области видимости данного пространства имен.

Пример 7.1

```
namespace MyNameSpace{
int i,k;
void myfunc(int j);
class myclass{
int x;
public:
…};
//определение функций
…
}
```

Здесь имена переменных i и k, функции myfunc и класса myclass находятся в области видимости, определенной пространством имен MyNameSpace.

К идентификаторам, объявленным в пространстве имен, внутри этого пространства можно обращаться напрямую. Однако при обращении извне пространства имен к объектам, объявленным внутри этого пространства, следует указывать оператор расширения области видимости

MyNameSpace::i=10;

Если имя часто используется вне своего пространства имен, довольно утомительно писать его каждый раз с квалификатором. Эту проблему можно решить с помощью объявления using.

using имя_ пространства_имен::идентификатор;

Такое объявление делает видимым и, соответственно, доступным в текущем пространстве имен указанный идентификатор.

using MyNameSpace::i;

i=10;

Для того чтобы сделать доступными в текущем пространстве имен все имена из заданного пространства имен, необходимо использовать директиву using.

using namespace имя_ пространства_имен;

using namespace MyNameSpace;

i=10;

k=5;

myclass ob;

Имеется возможность объявить более одного пространства имен с одним и тем же именем. Это позволяет разделить пространство имен на несколько файлов или внутри одного файла.

namespace NS{

int i;

…}

…

namespace NS{

int j;

...}

Здесь пространство имен NS разделено на две части. Несмотря на это, содержимое каждой части по-прежнему остается в одном и том же пространстве имен NS.

Пространство имен должно объявляться вне всех остальных областей видимости, за исключением того, что одно пространство имен может быть вложено в другое. Следовательно, вложенным пространство имен может быть только в другое пространство имен, но не в какую бы то ни было иную область видимости. Это означает, что нельзя объявить пространство имен, например, внутри функции.

Пространство имен может быть объявлено без имени

namespace{

//объявления

}

Это позволяет создавать идентификаторы, являющиеся уникальными внутри некоторого файла. Вне файла, содержащего безымянное пространство имен, члены этого пространства не видимы.

Одна из проблем с namespace состоит в том, что короткие названия пространств имен могут войти в конфликт друг с другом. С другой стороны, весьма неудобно применять длинные имена.

Решение данной проблемы состоит в использовании **псевдонимов** для namespace.

namespace Lib=Foundation_library_classes;

Перегрузка компонентных функций.

Одним из достоинств пространства имен является возможность вводить в свои программы библиотеки, написанные другими, не заботясь о вероятном совпадении имен.

Пример 7.2

Файл lib.h.

```
#ifndef LIB_H
#define LIB_H
namespace Other_lib
```

```
{
#include<head1.h>
#include< head2.h >
#include< head3.h >
}
namespace LIB=Other_lib;
#endif

//использование другой библиотеки
#include<lib.h>
…
LIB::func();
```

Для небольших и средних по объему программ маловероятна необходимость создания своих пространств имен. Однако если вы собираетесь создавать библиотеки функций или классов, предназначенных для многократного использования, или хотите гарантировать своим программам широкую переносимость, вам следует рассмотреть возможность размещения своих кодов внутри некоторого пространства имен. В *Microsoft Visual C++* библиотека классов определена в пространстве имен **std**.

using namespace std;

7.2. Динамическая идентификация типов

Динамическая идентификация типа (RTTI–Run-Time Type Identification) характерна для языков, в которых поддерживается полиморфизм. В этих языках возможны ситуации, в которых тип объекта на этапе компиляции неизвестен.

В C++ полиморфизм поддерживается через иерархии классов, виртуальные функции и указатели базовых классов. При этом указатель базового класса может использоваться либо для указания на объект базового класса, либо для указания на объект любого класса, производного от этого базового.

Информацию о типе объекта получают с помощью оператора **typeid**, при использовании которого следует подключить заголовочный файл <typeinfo.h>

Оператор typeid имеет две формы:

1) typeid (объект),

2) typeid (имя_типа).

Оператор typeid возвращает ссылку на объект типа **type_info**.

В классе type_info определены следующие открытые члены:

bool operator==(const type_info&ob)const;

bool operator!=(const type_info&ob)const;

const char* name()const;

Перегруженные операции == и != обеспечивают сравнение типов.

Функция name() возвращает указатель на имя типа.

Оператор typeid имеет одно ограничение. Он работает корректно только с объектами, у которых определены виртуальные функции. Когда оператор typeid применяют к неполиморфному классу (в классе нет виртуальной функции), получают указатель или ссылку базового типа.

Пример 7.3

```
#include<iostream.h>
#include<typeinfo.h>
class Base{
virtual void f(){};
//…
};
class Derived: public Base{
//…
};
void main()
{int i;
Base ob,*p;
Derived ob1;
```

```
cout<<typeid(i).name(); //Выводится int
p=&ob1;
cout<<typeid(*p).name(); //Выводится Derived
}
```

Пример 7.4

```
#include<iostream.h>
#include<typeinfo.h>
class Base{
virtual void f(){};
//…
};
class Derived: public Base{
//…
};
void WhatType(Base& ob)
{cout<< typeid(ob).name()<<endl;
}
void main()
{
Base ob;
Derived ob1;
WhatType(ob);   // Выводится Base
WhatType(ob1); // Выводится Derived
}
```

Пример 7.5

```
class X{
virtual void f(){};
};
class Y{
virtual void f(){};
};
void main()
```

```
{X x1,x2;
Y y1;
if(typeid(x1)==typeid(x2))cout<<“тип одинаков”;
else cout<<“тип не одинаков”;
if(typeid(x1)!=typeid(y1)) cout<<“тип неодинаков”;
…
if(typeid(x1)==typeid(X)) cout<<“x1 имеет тип X”;
…
}
```

При использовании указателей на объект, например typeid(*p), если p==NULL, то возникает исключительная ситуация bad_typeid.

7.3. Безопасное приведение типа

Приведение типа вида **(имя_типа) выражение,** определенное в стандарте языка С, может привести к ошибкам. Ошибка может произойти, когда вы попытаетесь привести один объект к другому при их несовместимости.

Пример 7.6

```
struct A
{int i};
struct B
{char* s;}
void f(A* x)
{B* p=(B*)x;
cout<<p–>s;}
void main()
{B b;
A a,*r;
b.s=new char[5];
b.s=“abcd”;
```

```
a.i=15;
r=&a;
f(r);//Здесь ошибка
r=(A*)(&b);
f(r);//А так ошибки нет
}
```

Для решения проблемы безопасного приведения типов в C++ введены операторы:

dynamic_cast, static_cast, const_cast, reinterpret_cast.

Оператор dynamic_cast

Оператор dynamic_cast предназначен для безопасного приведения типа одного указателя или ссылки в другой. Основное назначение оператора dynamic_cast заключается в реализации приведения полиморфных типов.

Синтаксис оператора dynamic_cast:

dynamic_cast<целевой_тип>(выражение)

Здесь «целевой_тип» – это тип, которым должен стать тип параметра «выражение» после выполнения приведения типа. Оператор выполняется успешно, когда указатель (или ссылка) после приведения типа становится указателем на объект целевого типа либо на объект производного от целевого типа. При невозможности приведения результатом является либо NULL, если приводятся указатели, либо возбуждается исключительная ситуация bad_cast, если приводятся ссылки.

Пример 7.7

```
Base* bp,b;
Derived* dp,d;
bp=&d;
dp= dynamic_cast<Derived*>(bp);
if(dp)cout<<“приведение прошло успешно”;
bp=&b;
dp=dynamic_cast<Derived*>(bp);
```

if(!dp)cout<<“приведение невозможно”;

Оператор dynamic_cast в некоторых случаях можно использовать вместо typeid.

Base* bp;

Derived* dp;

Приведение типов можно выполнить так:

if (typeid(*bp)==typeid(Derived))dp=(Derived*)bp;

А можно и короче:

dp=dynamic_cast<Derived*>(bp);

Оператор const_cast

const_cast<целевой_тип>(выражение)

Используется в том случае, когда вам необходимо удалить атрибут const из объекта.

void f(const int* p)

{int* v;

v=const_cast<int*>(p);

v=2(*v);

}

Оператор static_cast

static_cast<целевой_тип> (выражение)

Предназначен для выполнения операции приведения типов над объектами неполиморфных классов. Например, его можно использовать для приведения типа указателя базового класса к типу указателя производного класса. Кроме этого, он подойдет и для выполнения любой стандартной операции преобразования, но только не в динамическом режиме.

int i;

float f;

f=199.22;

i=static_cast<int>(f);

Оператор reinterpret_cast

reinterpret_cast<целевой_тип>(выражение)

Дает возможность преобразовать указатель одного типа в указатель совершенно другого типа. Он также позволяет

приводить указатель к типу целого или целое к типу указателя. Это преобразование также опасно, как и старый С-стиль преобразования.

```
int i;
char* p=“Это строка”;
i=reinterpret_cast<int>(p);
cout<<i;
```

8. СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

8.1. Введение в STL

Создание стандартной библиотеки шаблонов (Standard Template Library, STL) является результатом многолетних исследований под руководством Александра Степанова и Менга Ли из компании «Hewlett-Packard» и Девида Мюссера из «Rensselaer Polytechnic Institute».

Одна из наиболее необычных идей STL – это **обобщенные алгоритмы**. Обобщенные алгоритмы в STL напоминают классы-шаблоны. Но есть принципиальная разница между принципами ООП и принципами, лежащими в основе STL.

В ООП хорошо разработанный объект старается инкапсулировать всё состояние и поведение, необходимые для выполнения задачи и в то же время скрывает как можно больше деталей внутреннего устройства. Во многих предшествующих объектно-ориентированных библиотеках этот подход воплощался в контейнерных классах, обладающих широкой функциональностью и богатым интерфейсом.

Разработчики STL пошли в другом направлении. Поведение, обеспечиваемое их стандартными компонентами, является минимальным. Вместо этого каждый компонент предназначен для функционирования совместно с большим набором **обобщенных** алгоритмов (шаблонов), имеющихся в библиотеке. Эти алгоритмы не зависят от контейнеров и поэтому могут работать с многими различными типами.

Отделяя функционирование алгоритмов от контейнерных классов, библиотека STL много выигрывает в размере – как в объёме самой библиотеки, так и в генерируемом коде. Вместо того, чтобы дублировать алгоритмы для многих контейнерных классов, одно-единственное описание библиотечной функции

может использоваться с любым контейнером. Более того, описание этих функций является настолько общим, что они могут применяться с обычными массивами и указателями и с другими типами данных.

Парадигму обобщенного программирования можно сформулировать следующим образом: реши, какие требуются алгоритмы; параметризуй их так, чтобы они могли работать со множеством подходящих типов и структур данных.

STL обеспечивает общецелевые, стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных. Поскольку STL строится на основе шаблонов классов, входящие в неё алгоритмы и структуры применимы почти ко всем типам данных

Ядро библиотеки образуют три элемента: **контейнеры**, **алгоритмы** и **итераторы**.

Контейнеры (containers) – это объекты, предназначенные для хранения других элементов: вектор, линейный список, множество.

Ассоциативные контейнеры (associative containers) позволяют с помощью ключей получить быстрый доступ к хранящимся в них значениям.

В каждом классе-контейнере определен набор функций для работы с ними. Например, список содержит функции для вставки, удаления и слияния элементов.

Алгоритмы (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) – это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива.

Вдобавок к контейнерам, алгоритмам и итераторам в STL поддерживается ещё несколько стандартных компонентов. Главными среди них являются **распределители памяти**, **предикаты** и **функции сравнения**.

У каждого контейнера имеется определенный для него распределитель памяти (**allocator**), который управляет процессом выделения памяти для контейнера.

По умолчанию распределителем памяти является объект класса **allocator**. Можно определить собственный распределитель.

В некоторых алгоритмах и контейнерах используется функция особого типа, называемая **предикатом**. Предикат может быть унарным и бинарным. Возвращаемое значение: истина либо ложь. Точные условия получения того или иного значения определяются программистом. Тип унарных предикатов – **UnPred**, бинарных – **BinPred**. Тип аргументов соответствует типу хранящихся в контейнере объектов.

Специальный тип бинарного предиката для сравнения двух элементов называется функцией сравнения (comparison function). Функция сравнения возвращает истину, если первый элемент меньше второго. Типом функции является тип **Comp**.

Для поддержки контейнерных классов STL включает так называемые **классы-утилиты** (Utility-классы). Заголовочные файлы: <utility.h> и <function.h>. Например, шаблон класса pair (пара) для хранения пары объектов.

Шаблоны из заголовочного файла <function.h> помогают создавать объекты, определяющие оператор-функцию operator(). Эти объекты называются объектами-функциями (function objects) и во многих случаях могут использоваться вместо указателей на функции, что позволяет генерировать более эффективный код. Например, класс-функция **less** (меньше), который позволяет определить, является значение одного объекта меньше, чем другого. Это предикат.

template<class T> struct less: public binary_function<T,T,bool>
{bool operator()(const T& x,const T& y)const
//возвращает результат сравнения x<y
{return x<y;}
};

Для поддержки единого определения типов аргументов и типа возвращаемого значения для различных объектов-функций с двумя аргументам в STL введен вспомогательный базовый класс:

template<class Arg1,class Arg2,class Result> struct binary_function{}
typedef Arg1 first_argument_type;
typedef Arg2 second_argument_type;
typedef Result result_type;

8.2. Итераторы

Итератор – это специальный вспомогательный объект, который поставляется разработчиком контейнерного класса. Единственное назначение такого объекта – обеспечить доступ к элементам контейнера без показа внутренней структуры контейнера (один элемент за одно обращение).

Обычно итератор содержит указатель, с которым производятся различные манипуляции.

В STL итератор – это фундамент, на котором основано использование контейнерных классов и алгоритмов.

Итераторы применяются для различных целей:

– итератор может обозначать конкретное значение;

– пара итераторов может задавать диапазон значений. При этом второе значение (второй итератор) рассматривается не как часть определяемого диапазона, но как запредельный элемент, описывающий значение, следующее за последним значением из заданного диапазона.

Основное действие, которое модифицирует итератор, – это операции: инкремент ++ , декремент -- и увеличение +.

Для доступа к данным, определяемым итератором, используется оператор разыменования *.

Итератор является шаблоном, параметром которого является контейнерный класс. Существует пять типов итераторов.

Итераторы ввода (input_iterator) поддерживают операции равенства, разыменования и автоинкремента. Итераторы, отвечающие этим условиям, могут использоваться для однопроходных алгоритмов, которые читают значения данных в одном направлении. Специальным случаем итератора ввода является istream_iterator.

==, !=, *i, ++i, i++, *i++.

Итераторы вывода (output_iterator) поддерживают разыменование, допустимое только с левой стороны присваивания, и инкремент:

++i, i++, *i=t, *i++=t

Однонаправленные итераторы (forward_iterator) поддерживают все операции итераторов ввода-вывода и, кроме того, позволяют без ограничений применять присваивание. Для них из i==j следует ++i==++j, что не всегда истинно для итераторов ввода, т.е. forward-итераторы сохраняют позицию внутри структуры данных (контейнера) при многократных проходах. Таким образом, их можно использовать в алгоритмах с многократным проходом.

==, !=, =, *i, ++i, i++, *i++.

Двунаправленные итераторы (biderectional_iterator) обладают всеми свойствами forward-итераторов, а также возможностью прохода контейнера в обоих направлениях, т.е. имеют дополнительная операции –декремент:

--i, i--, *i--.

Итераторы произвольного доступа (random_access_ iterator) обладают всеми свойствами двунаправленных итераторов, а также поддерживают операции сравнения и адресной арифметики, т.е. непосредственный доступ по индексу.

i+=n, a+n, i-=n, a-n, i-j, i[n], i<j, i<=j, i>j, i>=j.

Таким образом, с их помощью можно написать такие алгоритмы, как алгоритмы быстрой сортировки, которые требуют эффективного произвольного доступа.

С итераторами можно работать так же, как с указателями. К ним можно применить операции *, инкремента, декремента. Типом итератора объявляется тип **iterator**, который определен соответствующим образом в различных контейнерах. В STL также поддерживаются **обратные итераторы** (reverse iterators). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении. Следовательно, если обратный итератор указывает на последний элемент последовательности, то инкремент этого итератора приведет к тому, что он будет указывать на предпоследний элемент.

8.3. Классы-контейнеры

В STL определены два типа контейнеров – последовательности и ассоциативные контейнеры.

Встроенные массивы, строки **string**, векторы **valarray** (векторы, оптимизированные для численных расчетов) и битовые поля **bitset** содержат элементы и, следовательно, могут считаться контейнерами. Однако эти типы не являются полностью разработанными стандартными контейнерами. Если бы они являлись таковыми, это помешало бы их основному назначению.

Ниже перечислены основные классы-контейнеры с указанием заголовочных файлов, которые необходимо подключить при их использовании:

bitset	множество битов <bitset>,
deque	двусторонняя очередь <deque>,
list	линейный список <list>,
map	ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано одно значение <map>,

multimap	с каждым ключом связаны два или более значений <map>,
multiset	множество, в котором каждый элемент не обязательно уникален <set>,
priority_queue	очередь с приоритетом <queue>,
queue	очередь <queue>,
set	множество <set>,
stack	стек <stack>,
vector	динамический массив <vector>.

Ключевая идея для стандартных контейнеров заключается в том, что, когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться **map** (ассоциативным массивом). С другой стороны, если преобладают операции, характерные для списков, можно воспользоваться контейнером **list**. Если добавление и удаление элементов часто производятся в концы контейнера, следует подумать об использовании очереди **queue**, очереди с двумя концами **deque**, стека **stack**. По умолчанию пользователь должен использовать vector; он реализован, чтобы хорошо работать для самого широкого диапазона задач. Кроме того, пользователь может разработать дополнительные контейнеры, вписывающиеся в рамки стандартных контейнеров.

Идея обращения с различными видами контейнеров – и в общем случае со всеми видами источников информации – унифицированным способом ведет к понятию обобщенного программирования, о чем излагалось выше.

Поскольку имена типов элементов, входящих в объявление класса-шаблона, могут быть самыми разными, в классах-контейнерах с помощью ключевого слова typedef объявляются

некоторые согласованные версии этих типов. Эта операция позволяет конкретизировать имена типов. Ниже перечислены имена этих типов:

size_type интегральный тип, эквивалентный типу size_t, являющемуся беззнаковым целочисленным типом, представляющим результат операции sizeof,

reference тип ссылки на элемент контейнера,

const_reference тип константной ссылки на элемент контейнера,

iterator тип итератора,

const_iterator тип константного итератора,

reverse_iterator тип обратного итератора,

const_reverse_iterator тип константного обратного итератора

value_type тип хранящегося в контейнере значения,

allocator_type тип распределителя памяти,

key_type тип ключа в ассоциативных контейнерах,

mapped_type тип отображенного значения в ассоциативных контейнерах,

key_compare тип функции, которая сравнивает два ключа,

value_compare тип функции, которая сравнивает два значения.

Следующие компонентные функции контейнеров используют итераторы в качестве возвращаемого значения:

iterator begin() указывает на первый элемент,

iterator end() указывает на элемент, следующий за последним,

reverse_iterator begin() указывает на первый элемент в обратной последовательности,

reverse_iterator rend() указывает на элемент, следующий за последним в обратной последовательности.

Следующие компонентные функции контейнеров используют итераторы в качестве возвращаемого ссылки:

reference front() ссылка на первый элемент,
reference back() ссылка на последний элемент,
reference operator[]() доступ по индексу без проверки,
reference at() доступ по индексу с проверкой.

Следующие компонентные функции контейнеров включают элементы в контейнер:

insert() включает элемент (последовательность элементов) в контейнер,
push_back() добавление *x* в конец,
push_front() добавление нового первого элемента (только для списков и очередей с двумя концами).

Следующие компонентные функции контейнров удаляют элелементы из контейнера:

pop_back() удаление последнего элемента,
pop_front() удаление первого элемента (только для списков и очередей с двумя концами),
erase() удаление заданной последовательности элементов,
clear() удаление всех элементов контейнера.

Операция доступа к элементу контейнера по индексу:

reference **operator[]**(size_type pos);

Операции для ассоциативных контейнеров:

T& operator[](const key_type& k)

доступ к элементу с ключом k,

iterator find(const key_type&k)

находит элемент с ключом k,

iterator lower_bound(const key_type&k)

находит первый элемент с ключом k,

iterator upper_bound(const key_type&k)

находит первый элемент с ключом большим k,

pair<iterator,iterator>equal_range(const key_type&k)

находит lower_bound (нижнюю границу) и upper_bound (верхнюю границу) элементов с ключом k.

Другие операции для контейнеров:

size_type size()const	возвращает число элементов в контейнер,
bool empty()const	проверяет, пуст ли контейнер,
size_type capacity()const	возвращает размер памяти, выделенной под контейнер-вектор,
void reserve(size_type n)	выделяет память под вектор,
void resize(size_type n,T obj=T())	изменяет размер контейнера (только для векторов, списков и очередей с двумя концами),
swap(x)	обмен местами двух контейнеров,
==, **!**=, <	операции сравнения контейнеров.

Для создания контейнеров имеются следующие конструкторы:

container()	создается пустой контейнер,
container(n)	создается контейнер, содержащий *n* элементов со значением по умолчанию,
container(n,x)	создается контейнер, содержащий *n* копий элементов *x*,
container(first,last)	создается контейнер, содержащий элементы из диапазона [first:last],
container(x)	конструктор копирования.

8.4. Контейнер vector

В качестве примера подробнее рассмотрим контейнер vector.

Контейнер вектор представляет собой динамический массив с доступом к его элементам по индексу. Возможность доступа к элементам по индексу обеспечивает поддерживаемый контейнером итератор произвольного доступа. Шаблон класса vector определен следующим образом:

template<class T, class Allocator=allocator<T> >

class std:: vector{/*члены класса*/};

T – тип предназначенных для хранения данных.

Allocator задает распределитель памяти, который по умолчанию является стандартным.

В классе определены следующие конструкторы.

explicit **vector**(const Allocator& a=Allocator());

explicit **vector**(size_type число,const T&значение=T(),const Allocator&a=Allocator());

vector(const vector<T,Allocator>&объект);

template<class InIter>**vector**(InIter начало,InIter конец, const Allocator&a=Allocator());

Описатель explicit подавляет неявное преобразование типов из типа аргумента в тип класса конструктора.

Первая форма представляет собой конструктор пустого вектора.

Во второй форме конструктора вектора число элементов – это ***число***, а каждый элемент равен значению ***значение***. Параметр ***значение*** может быть значением по умолчанию.

Третья форма конструктора вектор – это конструктор копирования.

Четвертая форма – это конструктор вектора, содержащего диапазон элементов, заданный итераторами ***начало*** и ***конец***.

Пример 8.1

```
vector<int> a;
vector<double> x(5);
vector<char> c(5,'*');
vector<int> b(a);
```

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме того, для объекта должны быть определены операторы < и ==.

Для класса вектор определены следующие операторы сравнения

==, <, <=, !=, >, >=.

Кроме этого, для класса vector определяется оператор доступа по индексу [].

Новые элементы могут включаться в контейнер с помощью функций:

1) iterator **insert**(iterator i,const T&значение=T());

Вставляет параметр ***значение*** перед элементом, заданным итератором i. Возвращает итератор элемента,

2) void **insert**(iterator i,size_type число,const T&значение);

Вставляет ***число*** копий параметра ***значение*** перед элементом, заданным итератором i,

3) template<class InIter>void **insert**(iterator i,InIter начало,InIter конец);

Вставляет последовательность, определенную между итераторами ***начало*** и ***конец***, перед элементом, заданным итератором i,

4) void **push_back**(const T&значение);

Добавляет в конец вектора элемент, значение которого равно параметру ***значение***,

5) void **resize**(size_type число,T значение=T());

Изменяет размер вектора в соответствии с параметром ***число***. Если при этом вектор удлиняется, то добавление в конец вектора. Элементы получают значение, заданное параметром ***значение***;

6)template<class InIter>void assign(InIter начало,InIter конец);

Присваивает вектору последовательность, определенную итераторами ***начало*** и ***конец***.

Существующие элементы могут удаляться с помощью функций:

1) iterator **erase**(iterator i);

Удаляется элемент заданный итератором i,

2) iterator **erase**(iterator ***начало***, iterator ***конец***);

Удаляет элементы последовательности, определенной итераторами ***начало*** и ***конец***,

3) void **pop_back**();

Удаляет последний элемент,

4) void **clear**() const;

Удаляет все элементы. Контейнер становится пустым,

5) void **resize**(size_type число,T значение=T());

Изменяет размер вектора в соответствии с параметром ***число***. Если размер уменьшается, последние элементы удаляются.

Доступ к отдельным элементам осуществляется с помощью итераторов:

1) iterator **begin**();

Возвращает итератор на первый элемент,

2) iterator **end**();

Возвращает итератор на конец последовательности,

3) reference **operator[]**(size_type pos);

Возвращает ссылку на элемент в позиции pos без контроля,

4) reference **at**(size_type pos);

Возвращает ссылку на элемент в позиции pos с контролем. При выходе индекса за границу генерируется исключение out_of_range,

5) reference **front**();

Возвращает ссылку на первый элемент.

6) reference **back**();

Возвращает ссылку на последний элемент.

Манипулирование контейнером: сортировка, поиск в нем и тому подобное возможны с помощью глобальных функций файла-заголовка <algorithm.h>.

Два вектора *x* и *y* считаются равными (==), если x.size()==y.size() и x[i]=y[i] для любого допустимого индекса *i*.

Вектор *x* меньше вектора *y* (<), если первый x[i], не равный соответствующему y[i], меньше, чем y[i], или x.size()<y.size() при равенстве всех *x*[i] соответствующим *y*[i].

Пример 8.2. Создание массива целых чисел и заполнение его числами от 0 до 10.

```
#include<iostream.h>
#include<vector.h>
using namespace std;
int main()
{vector<int> v;
```

```
int i;
for(i=0;i<10;i++)v.push_back(i);
cout<<“size=”<<v.size()<<“\n”;
for(i=0;i<10;i++)cout<<v[i]<<“ ”;
cout<<endl;
for(i=0;i<10;i++)v[i]=v[i]+v[i];
for(i=0;i<v.size();i++)cout<<v[i]<<“ ”;
cout<<endl;
return 0;
}
```

Пример 8.3. То же, что и предыдущий, но используется итератор

```
int main()
{vector<int> v;
int i;
for(i=0;i<10;i++)v.push_back(i);
//доступ к вектору через итератор
vector<int>::iterator p=v.begin();
while(p!=v.end()){cout<<*p<<“ ”; p++;}
return 0;
}
```

Пример 8.4

```
int main()
{vector<int> v(5,1);
vector<int>::iterator p=v.begin();
while(p!=v.end()){cout<<*p<<“ ”; p++;}
 p=v.begin();
p+=2;//указывает на третий элемент
/*вставка в вектор v на то место, куда указывает итератор p, десять новых элементов, каждый из которых равен 9*/
v.insert(p,10,9);
//вывод
```

```
p=v.begin();
while(p!=v.end()){cout<<*p<<“ ”; p++;}
//удаление вставленных элементов
p=v.begin();
p+=2;
v.erase(p,p+10);
//вывод
p=v.begin();
while(p!=v.end()){cout<<*p<<“ ”; p++;}
return 0;
}
```

Пример 8.5. Массив объектов пользовательского класса

```
#include<iostream.h>
#include “student.h”
using namespace std;
int main()
{vector<STUDENT> v(3);
int i;
v[0]= STUDENT(“Иванов”,21);
v[1]= STUDENT(“Петров”,19);
v[2]= STUDENT(“Попов”,20);
for(i=0;i<3;i++)cout<<v[i];
return 0;
}
```

8.5. Многомерные массивы

Многомерный массив можно представить как вектор с компонентами типа вектор:

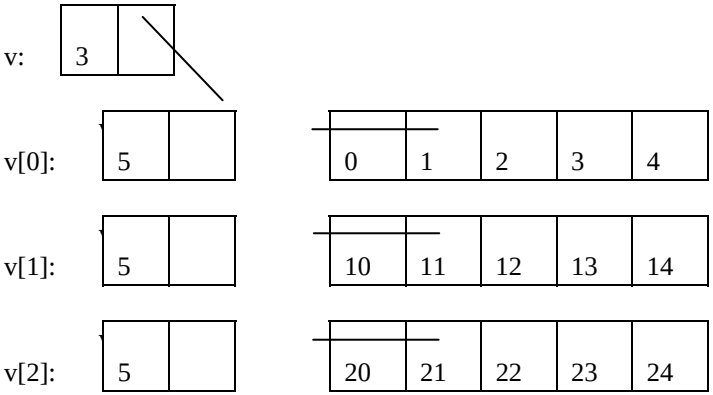
```
vector<vector<int>> v;
```

Так создаётся вектор векторов с целыми элементами, который в начале не содержит ни одного элемента. Проинициализируем его в матрицу 3×5.

v.resize(3);//теперь вектор содержит три пустых вектора
for(int i=0;i<v.size();i++)
{v[i].resize(5);//каждый из векторов содержит пять элементов
//проинициализируем их
for(int j=0;j<v[i].size();j++)v[i][j]=10*i+j;}

Вектора vector<int> в векторе vector<vector<int>> не обязаны иметь один и тот же размер.

Каждая реализация шаблона vector содержит указатель на его элементы плюс число элементов. Как правило, элементы содержатся в массиве.



Доступ к элементам осуществляется путем двойного индексирования.

for(int i=0;i<v.size();i++){
for(int j=0;j<v[i].size();j++)cout<<v[i][j]<<“ ”;
Можно перегрузить операции << и >> для вектора.
ostream& **operator<<**(ostream& out,const vector<int>& v){
vector<int>::iterator p;
for(p=v.begin();p!=v.end();++p)out<<*p<<’\t’;
out<<endl;
return out;}
istream& **operator>>**(istream& in,vector<int>& v){

```
vector<int>::iterator p;
for(p=v.begin();p!=v.end();++p)in>>*p;
return in;}
```

Можно написать также функцию суммирования элементов вектора:

```
int sum(vector<int>::iterator first,vector<int>::iterator last,
int initial_val)
{vector<int>::iterator p;
int sum=initial_val;
for(p=first;p!=last;p++)sum+=*p;
return sum;
}
```

Программа 8.1. Напишите функцию быстрой сортировки массива целых чисел.

Функцию быстрой сортировки сложно запрограммировать из-за многочисленных индексов, которые отслеживаются в традиционной реализации. Мы будем использовать контейнер vector и заменим индексирование итераторами.

Вначале запишем рекурсивную функцию сортировки:

```
void quicksort(vector<int>::iterator from,vector<int>::iterator to)
{
vector<int>::iterator mid;
if(from<(to-1)){
mid=partition(from,to);
quicksort(from,mid);
quicksort(mid+1,to);}
}
```

Здесь используется функция partition, которая в соответствиии с алгоритмом разделяет массив на две части.

Запишем эту функцию.

```
vector<int>::iterator
partition(vector<int> :: iterator from, vector <int> :: iterator to)
{
```

```
vector<int>::iterator front=from+1;
vector<int>::iterator back=to-1;
int compare=*from;
while(front<back){
while((front<back)&&(compare>*front))++front;
while((front<back)&&(compare<=*back))--back;
swap(*front,*back);}
if(compare>*front)
{swap(*from,*front);
return front;}
else{
swap(*from,*(front-1));
return front-1;}
}
```

Используемая здесь функция swap() меняет местами два элемента.

```
inline void swap(int& i,int& j)
{int temp=i;
i=j;
j=temp;}
```

Напишем также функцию print() для вывода массива:

```
void print(vector<int> v)
{
for(int i=0;i<v.size();i++)cout<<v[i]<<' ';
cout<<endl;
}
```

И,наконец, запишем функцию main():

```
int main()
{
srand( (unsigned)time( NULL ) );
vector<int> v(10);
for(int i=0;i<v.size();i++)v[i]=rand()/10;
print(v);
quicksort(v.begin(),v.end());
```

```
print(v);
return 0;
}
```

Не забудьте подключить заголовочные файлы и указать пространство имен:

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

В программе определяется вектор, заполняется случайными числами и сортируется.

В качестве самостоятельной работы напишите шаблон функции быстрой сортировки последовательностей любого типа.

8.6. Ассоциативные контейнеры

Ассоциативные контейнеры – это обобщение понятия ассоциативного массива.

Ассоциативный массив – это один из самых полезных и универсальных типов, определяемых пользователем. Фактически в языках, занимающихся главным образом обработкой текстов и символов, это зачастую встроенный тип .

Ассоциативный массив, часто называемый **отображением** (map), а иногда **словарем** (dictionary), содержит пары значений. Зная одно значение, называемое **ключом** (key), мы можем получить доступ к другому, называемому **отображенным значением** (mapped value).

Ассоциативный массив можно представить как массив, для которого индекс необязательно должен иметь целочисленный тип:

```
template<class K,class V> class Assoc{
public:
```

V& operator[](const K&);

//…

}

Здесь operator[] возвращает ссылку на элемент V, соответствующий ключу K.

STL содержит два вида контейнеров, построенных как ассоциативные массивы: **map** и **multimap, set** и **multiset**, причем **set** и **multiset** можно рассматривать как вырожденные ассоциативные массивы, в которых ключу не соответствует никакое значение (т.е. set содержит одни ключи).

Контейнеры map и multimap

Это последовательность пар (ключ, значение), которая обеспечивает быстрое получение значения по ключу. Контейнер map предоставляет двунаправленные итераторы.

Контейнер map требует, чтобы для типов ключа существовала операция “<”. Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

Спецификация шаблона для класса map:

template<class Key,classT,class Comp=less<Key>,

class Allocator=allocator<pair>> class std::map

В классе map определены следующие конструкторы:

explicit **map**(const Comp& c=Comp(),

const Allocator& a=Allocator());

map(const map<Key,T,Comp,Allocator>& ob);

template<class InIter> **map**(InIter first,InIter last,const Comp& c=Comp(),const Allocator& a=Allocator());

Первая форма представляет собой конструктор пустого ассоциативного контейнера, вторая – конструктор копии, третья – конструктор ассоциативного контейнера, содержащего диапазон элементов.

В классе определена операция присваивания:

map& **operator=**(const map&);

Определены также следующие операции сравнения: ==, <, <=, !=, >, >=.

Копирование контейнера требует выделения памяти для элементов и создания копии каждого элемента. Это может оказаться очень дорого, и делать это нужно лишь при необходимости. Поэтому такие контейнеры часто передаются по ссылке.

Пары ключ/значение хранятся в контейнере в виде объектов типа **pair**. Тип pair – это класс, точнее, шаблон класса.

```
template<class Key,class V> struct pair{
typedef Key TFirst;//тип ключа
typedef V TSecond;//тип значения
Key first;//ключ
V second;//значение
pair():first(Key()),second(V()){}
pair(const Key& x,const V& y):first(x),second(y){}
template<class A,class B> pair(const pair<A,B>& ob): first(ob.first),second(ob.second){}
};
```

Последний конструктор существует для того, чтобы позволить преобразование в инициализаторе. Например:

```
pair<int,double> f(char c,int i)
{return pair<int,double>(c,i);}
```

Создавать пары ключ/значение можно не только с помощью конструкторов класса pair, но и с помощью функции **make_pair**, которая создает объекты типа pair, используя типы данных в качестве параметров.

```
template<class T1,classT2> pair<T1,T2>
std::make_pair(const T1& t1,const T2& t2){
return pair<T1,T2>(t1,t2);}
```

Преимущество этой функции в том, что она дает возможность компилятору автоматически распознавать типы предназначенных для хранения объектов, и вам не нужно указывать их явно. Таким образом, map – это последовательность.

Что касается реализации контейнера map, то он, скорее всего, реализован с использованием какой-либо формы дерева, и итераторы для map обеспечивают некоторый способ прохода по

дереву. Такая реализация обеспечивает быстрый поиск значения по ключу.

Операции с ассоциативными контейнерами

Типичная операция – это ассоциативный поиск при помощи операции индексации ([]).

mapped_type& **operator[]**(const key_type& K);

Операция индексации должна найти ключ. Когда ключ не находится, добавляется элемент по умолчанию. Таким образом, индексация может использоваться, если mapped_type имеет значение по умолчанию. Это является следствием того, что pair по умолчанию инициализируется значениями по умолчанию для типов её элементов. Элементы встроенных типов инициализируются нулями, а строки string – пустыми строками.

Пример 8.1

map<string,int> m; Создан пустой контейнер

int x=m[“Иванов”]; Добавляется элемент с ключом “Иванов” и знечение 0.

m[“Петров”]=7; Добавляется элемент с ключом “Петров” и знечение 7.

int y=m[“Иванов”]; y будет иметь значение 0

int z=m[“Петров”]; z будет иметь тзначение 7

m[“Петров”]=9; Изменено значение элемента с ключем “Петров”

Пример 8.2. Подсчитать общую стоимость предметов, представленных в виде пар (название предмета,стоимость):

```
void InitMap(map<string,int,less<string>>& m)
{string name;
int cost=0;
while(cin>>name>>cost)
//выход по ^z(Ctrl+z)
m[name]+=cost;
}
```

Если вводятся несколько предметов с одинаковым названием, то стоимость суммируется. Это видно в последней строке.

```
void main(){
map<string,int,less<string>> ware;
InitMap(ware);
int total=0;
typedef map<string,int,less<string>>::const_iterator pware;
for(pware p=ware.begin();p!=ware.end();p++){
total+=(*p).second;
cout<<(*p).first<<'\t'<<(*p).second<<'\n';}
cout<<"----------\n";
cout<<"total\t"<<total<<'\n'
}
```

В этом примере используется определенная в STL функция сравнения less.

Для доступа к элементам контейнера map можно использовать также функции **find**(k), **lower_bound**(k), **upper_bound**(k), которые возвращают итератор, соответствующий элементу с ключом k или начало/конец последовательности элементов контейнера, имеющих ключ k.

Обычно конец последовательности – это итератор, указывающий на элемент, следующий за последним в последовательности. Если элемента с ключом k нет, возвращается итератор end().

Вводить значения в ассоциативный контейнер принято простым присваиванием с использованием индекса (см. пример 8.1):

```
m["Петров"]=7;
```

Это гарантирует, что будет занесена соответствующая запись.

Также можно вставить элемент функцией **insert()** и удалить функцией **erase()**. Функция insert() пытается добавить в map пару типа <Key,T>. Вставка производится, только если в map не существует элемента с таким ключом. Возвращаемое значение – пара **pair<iterator,bool>**. Переменная типа bool принимает значение true, если элемент вставлен. Итератор указывает на элемент с заданным ключом.

Примеры 8.3. Типы пар ключ/значение указаны явно в конструкции pair<char,int>.

```
#include<iostream>
#include<map>
using namespace std;
void main(){
map<char,int> m;
int i;
for(i=0;i<10;i++)m.insert(pair<char,int>('A'+i,i));
char ch;
cin>>ch;
map<char,int>::iterator p;
//поиск
p=m.find(ch);
if(p!=m.end())cout<<(*p).second;
else cout<<"Не найден\n";
}
```

Примеры 8.4. Используем функцию make_pair, которая создает пары объектов на основе типов данных своих параметров.

```
void main(){
map<char,int> m;
int i;
for(i=0;i<10;i++)m.insert(make_pair(char('A'+i),i));
//дальше также, как в примере 8.6.3
```

Так же, как и в других контейнерах, в ассоциативных контейнерах можно хранить создаваемые пользователем типы данных. Например, создадим map для хранения слов с соответствующими словами антонимами.

Примеры 8.5

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

```
using namespace std;
class word{
string str;
public:
word(){str="";}
word(string s){str=s;}
string get(){return str;}
};
bool operator<(word a,word b){return (a.str<b.str);}
class opposite{
string str;
public:
opposite(){str="";}
opposite (string s){str=s;}
string get(){return str;}
};
void main(){
map<word,opposite> m;
m.insert(pair<word,opposite>(word("хорошо"),opposite(
"плохо")));
//и т.д.
//поиск антонима по слову
string ss;
cin>>ss;
map<word,opposite>::iterator p;
p=m.find(word(ss));
if(p!=m.end())cout<<(*p).second.get();
else cout<<"Такого слова нет\n";
}
```

Примеры 8.6. Используется контейнер multimap для хранения группы студентов

```
typedef multimap<string,STUDENT,less<string>> TGroup;
typedef TGroup::value_type TItem;
void PrintStudent(const TItem& s,bool printCourse=true)
```

```
{if(printCourse)cout<<s.first;
cout<<s.second<<endl;
}
void main(){
TGroup curs; //студенты курса
Включить всех студентов в контейнер, так как показано ниже
curs.insert(TItem(“АСУ-07-1”,STUDENT(“Иванов”,19,0)));

//Распечатать всех студентов курса
for(TGroup::iterator i=curs.begin(); i!=curs.end(); i++)
{PrintStudent(*i);}
//Распечатаь студентов только заданной группы
for(TGroup::iterator i=curs.lower_bound(“АСУ-07-1”);
(i!=curs.upper_bound(“АСУ-07-1”))&&(i!=curs.end());i++)
PrintStudent(*i,false);
}
```

Контейнеры set и multiset

Множества можно рассматривать как ассоциативные массивы, в которых значения не играют роли, так что мы отслеживаем только ключи. Шаблон класса контейнера set

```
template<class T,class Cmp=less<T>,
class Allocator =allocator <T> > class std::set{…};
```

Множество, как и ассоциативный массив, требует, чтобы для типа T существовала операция «меньше» (<). Оно хранит свои элементы отсортированными, так что перебор происходит по порядку.

Примеры 8.7. Множество объектов пользовательского класса

Пользовательский класс, объекты которого будут сохраняться в множестве

```
class STUDENT{
string name;
float grade;
public:
```

STUDENT(const string& name1="",
float grade1=-1.0):name(name1), grade(grade1){}
STUDENT(const STUDENT& Student){*this=Student;}
const string& **GetName**()const{return name;}
float **GetGrade**()const{return grade;}
void **SetName**(string& name1){name=name1;}
void **SetGrade**(float grade1){grade=grade1;}
STUDENT& **operator=**(const STUDENT& student)
{if(this!=&student){name=student.name;
grade=student.grade;}
return *this;}
bool **operator==**(const STUDENT& student)const
{return(name==student.name);}
bool **operator<**(const STUDENT& student)const
{return(name<student.name;}
friend ostream& **operator<<**(ostream& os,const STUDENT& s)
{os<<s.GetName()<<"grade="<<s.GetGrade()<<endl;
return os;
}
};
typedef set<STUDENT> **STUDENT_SET1**;

Класс с перегруженной операцией для сравнения объектов STUTENT по полю grade-рейтинг. Таким образом, студенты в контейнере будут упорядочены по рейтингу.

struct GRADE_COMPARE{
bool **operator()**(const STUDENT& s1,const STUDENT& s2)const
{return(s1.GetGrade()<s2.GetGrade();}
};
typedef multiset<STUDENT,GRADE_COMPARE>
STUDENT_SET2;
void main(){
Создается два множества
STUDENT_SET1 Set1;

```
STUDENT_SET2 Set2;
Помещаются студенты в контейнер Set1
Set1.insert(STUDENT(“Иванов”,35.5));
//и т.д.
```

Просматриваем контейнер Set1 и помещаем студентов в контейнер Set2,

```
for(STUDENT_SET1::iterator
i=Set1.begin();i!=Set1.end();i++){
cout<<*i;
Set2.insert(*i);}
Просматриваем контейнер Set2
for(STUDENT_SET2::iterator
i=Set2.begin;i!=Set2.end();i++)cout<<*i;
}
```

8.7. Объекты-функции

Объект-функция – это экземпляр класса, в котором перегружена операция «круглые скобки» (). В ряде случаев удобно заменить функции сравнения на объекты-функции. Когда объект-функция используется в качестве функции, то для её вызова используется operator().

Примеры 8.8. Сравнение двух целых

```
class less{
public:
bool operator()(int x,int y){return x<y;}
};
```

Можно сделать шаблон класса для сравнения данных любых типов.

```
template<class T> class less{
public:
bool operator()(const T& x,const T& y)const{return x<y;}};
```

Следует иметь в виду, что для типа T должна быть определена операция меньше(<).

В STL определены вспомогательные базовые классы, поддерживающие единое определение типов аргументов и типа возвращаемого значения для различных объектов функций с одним и двумя аргументами. Это шаблоны **unary functions** и **binary_function**, которые доступны при подключении заголовочного файла< functional >**.**

template<class Arg, class Result>struct **unary_function**

{typedef Arg **argument_type**;typedef Result **result_type**;};

Этот шаблон служит базовым для классов, в которых операция «круглые скобки» определена в форме

result_type operator()(argument_type)

```
template<class Arg1, class Arg2, class Result>
   struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
   };
```

Этот шаблон служит базовым для классов, в которых операция «круглые скобки» определена в форме

result_type operator()(first_argument_type,

second_argument_type)

В STL также определен шаблон **less.**

template<class T> struct **less**: public binary_function<T,T,bool>;

Этот шаблон используется для создания класса функции, проверяющей, меньше ли первый операнд второго.

8.8. Алгоритмы

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, алгоритм может работать с очень разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор,

как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировку, поиск, вставку и удаление элементов. Алгоритмы доступны при включении заголовочного файла <algorithm>

Ниже приведены имена некоторых наиболее часто используемых функций – алгоритмов STL.

I. Немодифицирующие операции

for_earch() – выполняет операции для каждого элемента последовательности,

find() – находит первое вхождение значения в последовательность,

find_if() – находит первое соответствие предикату в последовательности,

count() – подсчитывает количество вхождений значения в последовательность,

count_if() – подсчитывает количество выполнений предиката в последовательности,

search() – находит первое вхождение последовательности как подпоследовательности,

search_n() – находит *n*-е вхождение значения в последовательность.

II. Модифицирующие операции

copy() – копирует последовательность, начиная с первого элемента,

swap() – меняет местами два элемента,

replace() – заменяет элементы с указанным значением,

replace_if() – заменяет элементы при выполнении предиката,

replace_copy() – копирует последовательность, заменяя элементы с указанным значением,

replace_copy_if() – копирует последовательность, заменяя элементы при выполнении предиката,

fill() – заменяет все элементы данным значением

remove() – удаляет элементы с данным значением,

remove_if() – удаляет элементы при выполнении предиката,

remove_copy() – копирует последовательность, удаляя элементы с указанным значением,

remove_copy_if() – копирует последовательность, удаляя элементы при выполнении предиката,

reverse() – меняет порядок следования элементов на обратный,

random_shuffle() – перемещает элементы согласно случайному равномерному распределению («тасует» последовательность),

transform() – выполняет заданную операцию над каждым элементом последовательности,

unique() – удаляет равные соседние элементы,

unique_copy() – копирует последовательность, удаляя равные соседние элементы.

III. Сортировка

sort() – сортирует последовательность с хорошей средней эффективностью,

partial_sort() – сортирует часть последовательности,

stable_sort() – сортирует последовательность, сохраняя порядок следования равных элементов,

lower_bound() – находит первое вхождение значения в отсортированной последовательности,

upper_bound() – находит первый элемент, больший, чем заданное значение,

binary_search() – определяет, есть ли данный элемент в отсортированной последовательности,
merge() – сливает две отсортированные последовательности.

IV. Работа с множествами

includes() – проверка на вхождение,
set_union() – объединение множеств,
set_intersection() – пересечение множеств,
set_difference() – разность множеств.

V. Минимумы и максимумы

min() – меньшее из двух,
max() – большее из двух,
min_element() – наименьшее значение в последовательности,
max_element() – наибольшее значение в последовательности.

VI. Перестановки

next_permutation() – следующая перестановка в лексикографическом порядке,
pred_permutation() – предыдущая перестановка в лексикографическом порядке.

Пример 8.9. Сортировка массива данных встроенных типов
Алгоритм **sort()** имеет следующие основные формы:

```
template<class RandIter>
 void sort(RandIter начало,RandIter конец)
template<class RandIter,class Comp>
void sort(RandIter начало,RandIter конец,
Comp функция_сравнения)
```

В примере создается и сортируется массив символов.

```
#include<iostream>
#include<vector>
#include<cstdlib>
```

```
#include<algorithm>
using namespace std;
void main()
{vector<char> v;
int i,k;
cin>>k;
//создание векторов из случайных символов
for(i=0;i<k;i++)
v.push_back('A'+(rand()%26));
//исходный массив
for(i=0;i<v.size();i++)cout<<v[i];
cout<<endl;
//сортировка вектора
sort(v.begin(),v.end());
//отсортированный массив
for(i=0;i<v.size();i++)cout<<v[i];
cout<<endl;
}
```

Пример 8.10. Сортировка массива данных пользовательских типов

```
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>
#include <functional>
#include"student.h" //Определение класса STUDENT
using namespace std;
```

Определим функцию для сравнения студентов по рейтингу:

```
class pred : public binary_function< STUDENT, STUDENT,bool>
{
public:
bool operator()(const STUDENT& st1, STUDENT& s2)const
```

```
{return s1.GetGrade()<s2.GetGrade();}
};
```

В функции main() запишем:

```
vector< STUDENT> v;
```

Затем надо заполнить контейнер v, например так:

```
v1.push_back(STUDENT ("Иванов",19,54.5));
```

Если сейчас отсортировать контейнер следующим образом:

```
sort(v.begin(),v.end());
```

то студенты будут отсортированы в зависимости от того, как определена в классе STUDENT операция <.

Если мы хотим отсортировать контейнер нужным для нас образом, следует использовать объект-функцию. Например, определим функцию для сравнения студентов по рейтингу:

```
class pred : public binary_function< STUDENT, STUDENT,bool>
{
public:
bool operator()(const STUDENT& st1, STUDENT& s2)const
{return s1.GetGrade()<s2.GetGrade();}
};
```

Сортируем контейнер

```
sort(v.begin(),v.end(),pred());
```

При передаче алгоритму sort третьего параметра создается объект класса pred (вызовом контейнера без конструктора) и вызывается перегруженная операция ().

Пример 8.11. Поиск в контейнере

Создадим фунциональный класса для поиска по имени:

```
class pred1:public unary_function< STUDENT,bool>
{
string s;
public:
  explicit pred1(const string& ss):s(ss){}
  bool operator()(const STUDENT & ob)const
```

```
  {return ob.Get_Name()==s;}
};
```

После этого для поиска можно использовать алгоритм find_if:

```
vector<STUDENTt>::iterator it;
it= find_if(v1.begin(),v1.end(),pred1("Котов"));
```

После выполнения find_if следует проверить, найден ли объект. Вспомним, что в случае неудачи алгоритм возвращает значение функции end().

```
if(it!=v1.end())cout<<endl<<*p<<endl;
 else cout<<"Такого объекта нет"<<endl;
```

Создание C++ приложений в среде Microsoft Visual Studio

Концепция решений и проектов

Сеанс работы в *Microsoft Visual Studio.Net* начинается с открытия существующего или создания нового **решения (solution).** Решение – это синоним **рабочего пространства (workspace)** в *Microsoft Visual C++ 6.0*. Файлы решений имеют расширение **sln** и используются IDE для хранения настроек и начальных установок конкретных решений. Концепция решений помогает объединить проекты и другие элементы в одном рабочем пространстве. Рабочее пространство может содержать несколько проектов, быть пустым или содержать файлы, которые имеют смысл и вне контекста решений. **Проект** как часть решения состоит из отдельных компонентов, например, файлов ресурсов(rc-файл), файлов с исходными кодами(.cpp, .h). Настройки проектов хранятся в специальных файлах проектов. Они могут иметь разные расширения, так как в одном пространстве можно объединять проекты разных типов. Например, проект Win32 Aplication хранит свои установки в файле с расширением **vcproj.**

Создание нового проекта

После запуска *Visual Studio* появится следующее окно (рис. П 1).

В этом окне выберем в меню File команду New->Project либо нажмем Ctrl+Shift+N.

В появившемся окне New Project в списке Project type выберем Visual C++/Win32, а в списке Templates – Win32 Console Application (рис. П2). В строке name введем имя проекта, а в строке Location выберем расположение проекта.

Если вы хотите создать папку для решения (solution), то следует отметить «Create directory for solution» и ввести имя решения в строке «Solution Name».

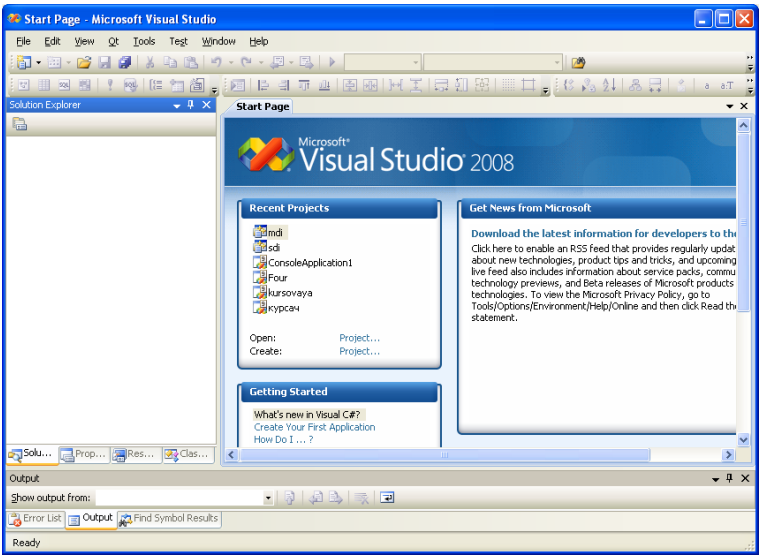

Start Page - Microsoft Visual Studio
File Edit View Qt Tools Test Window Help
Solution Explorer
Start Page
Microsoft Visual Studio 2008
Recent Projects
mdi
sdi
ConsoleApplication1
Four
kursovaya
курсач
Open: Project...
Create: Project...
Get News from Microsoft
Download the latest information for developers to th
Click here to enable an RSS feed that provides regularly updat
about new technologies, product tips and tricks, and upcoming
live feed also includes information about service packs, commu
technology previews, and Beta releases of Microsoft products
technologies. To view the Microsoft Privacy Policy, go to
Tools/Options/Environment/Help/Online and then click Read th
statement.
Getting Started
What's new in Visual C#?
Create Your First Application
How Do I ... ?
Output
Show output from:
Error List
Output
Find Symbol Results
Ready

Рис. П1

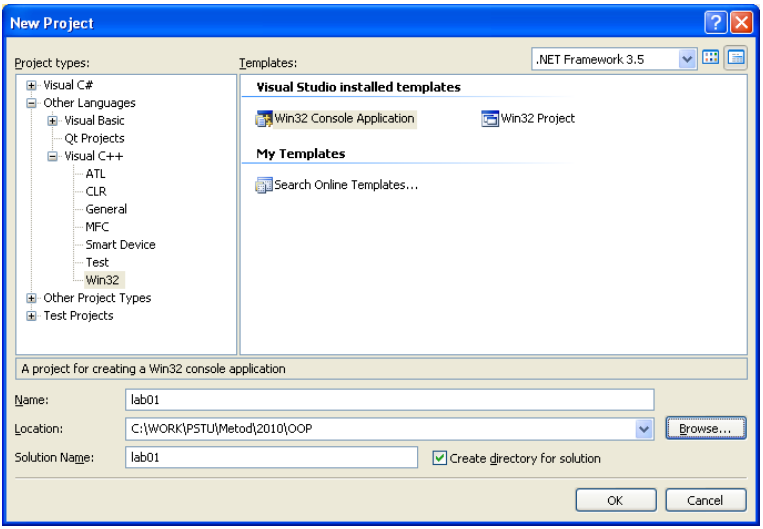

New Project
Project types:
Templates:
.NET Framework 3.5
Visual C#
Other Languages
Visual Basic
Qt Projects
Visual C++
ATL
CLR
General
MFC
Smart Device
Test
Win32
Other Project Types
Test Projects
Visual Studio installed templates
Win32 Console Application
Win32 Project
My Templates
Search Online Templates...
A project for creating a Win32 console application
Name: lab01
Location: C:\WORK\PSTU\Metod\2010\OOP
Browse...
Solution Name: lab01
Create directory for solution
OK
Cancel

Рис. П2

Нажав кнопку OK, перейдем в мастер-приложения (рис. П3).

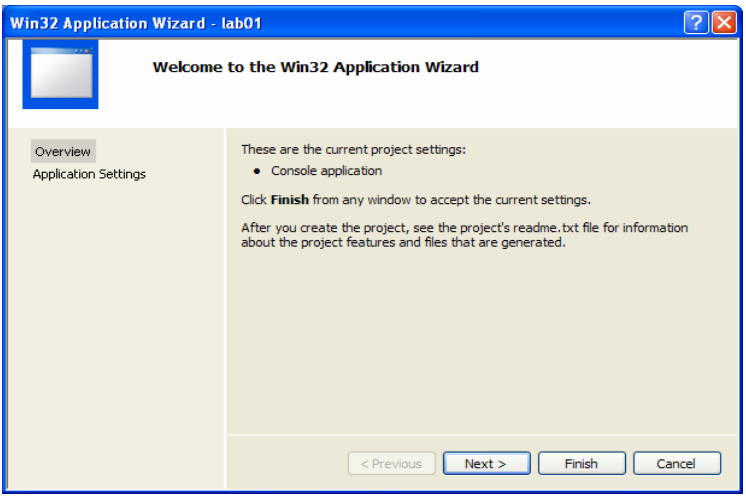


Рис. П3

Нажмем кнопку **“Next”** и перейдем в окно установки параметров проекта (рис. П4).

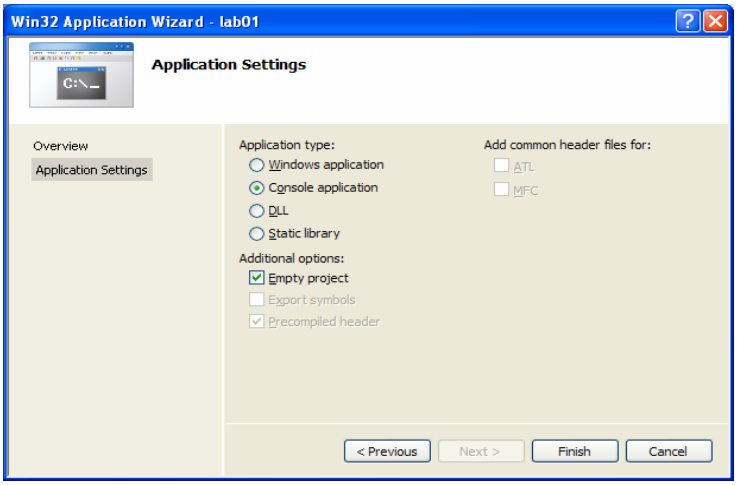


Рис. П4

В этом окне выберем **Console application** и **Empty project**. Нажмем кнопку **“Finish”.** Мастер создаст проект – консольное приложение.

В окне проекта (рис. П5) мы видим три раздела: Object Browser, Solution Explorer и Resourse View. В окне Solution Explorer можно осуществлять навигацию по файлам проекта. В этом окне мы видим имя проекта (lab1) и три группы файлов (Header Files, Resource Files и Source Files). Пока проект не содержит файлов. Создадим необходимые файлы. Для этого в окне Solution Explorer щелкнем правой кнопкой мыши по проекту и в появившемся контекстном меню выберем Add/New Item… .

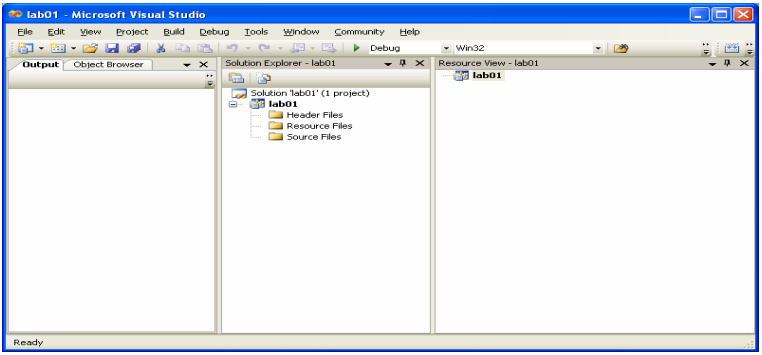


Рис. П5

В появившемся окне **Add New Item** (рис. П6) в списке **Categories** выберем **Code,** а в списке **Templates** – тип добавляемого в проект файла.

Создадим три файла: country.h – определение класса, country.cpp – реализация класса и main.cpp – демонстрационная программа, содержащая функцию main(). Пока файлы пустые. Щелкнув по имени файла, мы перейдем в редактор кода, где и запишем соответствующий код (рис. П7).

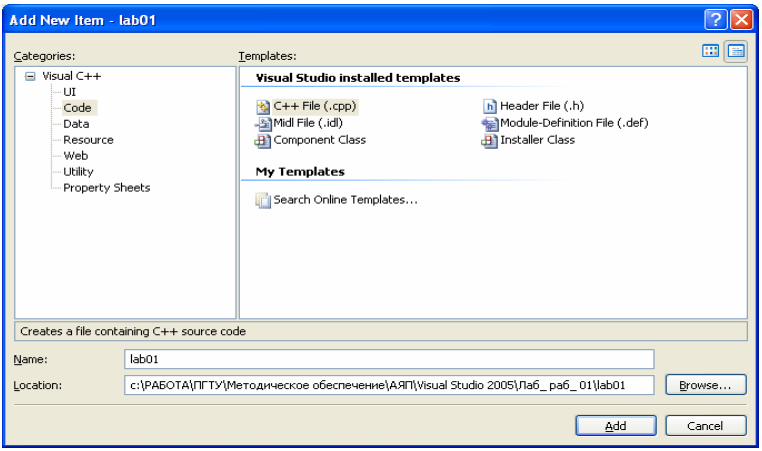


Рис. П6

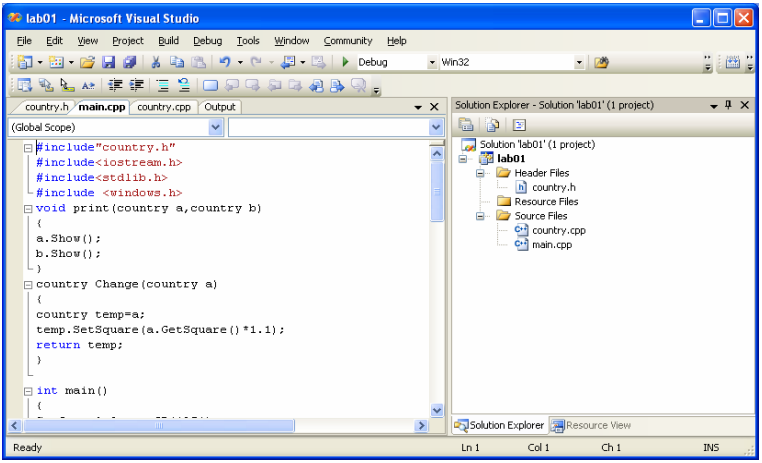


Рис. П7

Для того чтобы откомпилировать и собрать проект (создать выполняемый exe-файл), щелкните правой кнопкой мыши по проекту и в появившемся контекстном меню выберите **Build, Rebuild** или **Debug/Start new instance.** То же самое можно сделать через меню **Build** и **Debug**, или нажав Ctrl+F5. Результаты

выполнения проекта увидим на вкладке **Output** (рис. П8). Если программа содержит ошибки, получим сообщение, показанное на рисунке.

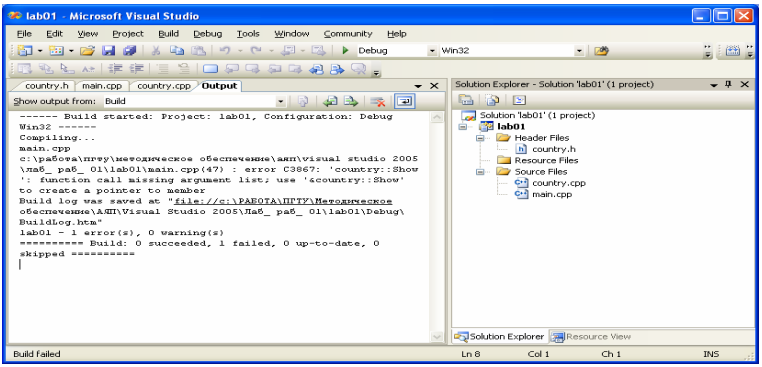


Рис. П8

Щелкнув по сообщению об ошибке (рис. П9), мы перейдем к месту ошибки в коде программы (рис. П10).

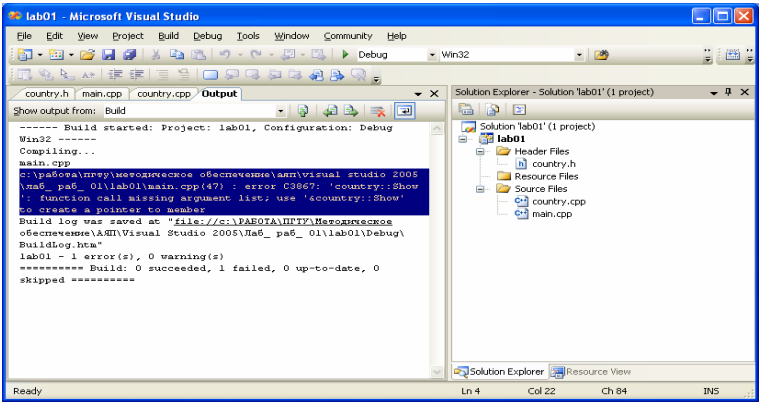


Рис. П9

На рис. П11 показаны сообщения при успешной компиляции и компоновке программы.

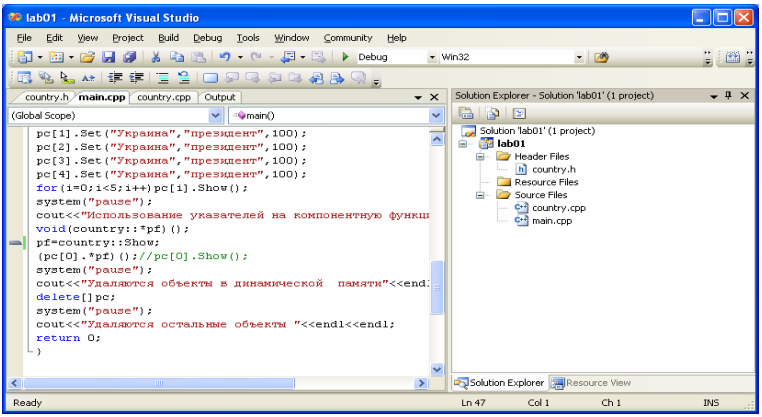


Рис. П10

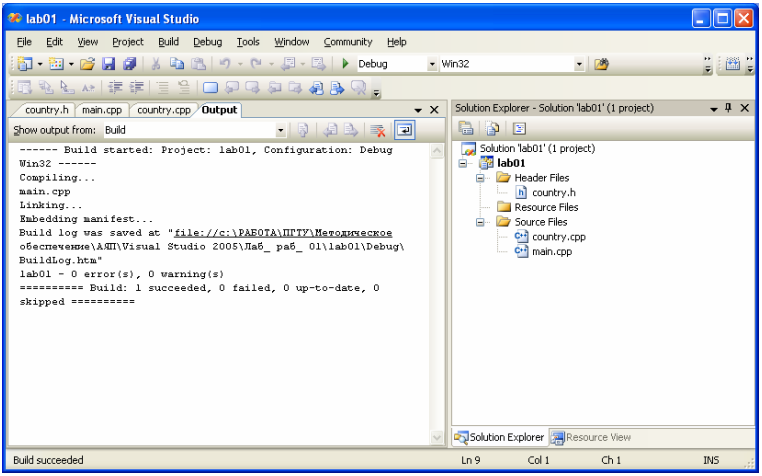


Рис. П11

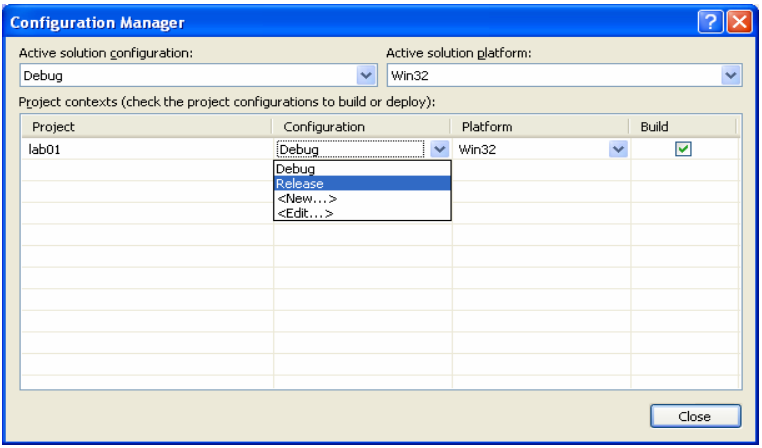


Рис. П12

Установить конфигурацию проекта (Debug или Release) можно через меню Build/Configuration Menager(рис. П12).

Учебное издание

Ноткин Аркадий Михайлович

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ: ООП на языке C++

Учебное пособие

Редактор и корректор *И.Н. Жеганина*

Подписано в печать 14.01.2013. Формат 60×90/16.
Усл. печ. л. 14,5. Тираж 100 экз. Заказ № 2/2013.

Издательство
Пермского национального исследовательского
политехнического университета.
Адрес: 614990, г. Пермь, Комсомольский проспект, 29, к. 113.
Тел. (342) 219-80-33.