Содержание:

1. Статические элементы класса
2. Виртуальные функции
3. Абстрактные классы
4. Виртуальные классы
5. Шаблоны
6. Обработка исключительных ситуаций в C++
7. **Статические элементы класса**

В классе можно определять элементы данных и функции со спецификатором static. В этом случае их можно рассматривать как глобальные переменные или функции в пределах класса, и доступны они не одному объекту, а разделяются между всем классом, не ассоциируются с отдельными представителями класса.

class Name

{ **…**

public:

**…**

static int count; }

Для элементов, которые объявлены в открытой секции класса, вызов, например, будет выглядеть следующим образом:

Name::count+=2;

Если элементы определены в закрытой секции, то такое обращение некорректно.

Если мы объявили такой элемент в закрытой секции класса, то мы должны определить функции для работы с этими элементами. Обращение к таким функциям выглядит так: Name::Add1();

ПРАВИЛО:

1) Статические функции и элементы класса не ассоциируются с отдельными представителями класса. Обращение к ним производится выражением вида:

имя класса:: имя элемента

2) Им не передается указатель this, т.к. он ассоциируется с отдельным представителем класса.

3) Статические функции могут вызываться независимо от того, существует ли хотя бы один представитель класса.

4) Статические функции не могут быть виртуальными.

#### Виртуальные функции

Виртуальные функции – это функции-члены класса, вызов которых осуществляется во время выполнения программы (то есть динамически) по типу объекта, с которым она используется.

Например, базовый класс может описывать фигуру на экране без конкретизации ее вида, а производные классы (окружность, эллипс и т.п.) однозначно определяют ее формы и размеры. Если в базовом классе ввести функцию для изображения фигуры на экране, то выполнение этой функции будет возможно только для объектов каждого из производных классов, определяющих конкретные изображения.

Объявление виртуальной функции в классе:

virtual прототип\_функции;

Отметим, что классы, включающие такие функции, играют особую роль в объектно-ориентированном программировании. Именно поэтому они носят специальное название – полиморфные.

Вернемся к упомянутому выше примеру с фигурами. Рассмотрим класс Point (производный от Location).

Пусть в этом классе определена компонентная функция void Show(). Так как внешний вид фигуры, для которой будет использоваться данная функция, в базовом классе еще не определен, то в каждый из производных классов нужно включить свою функцию void Show() для формирования изображения на экране. Это не очень удобно, поэтому в таких случаях используют механизм виртуальных функций. Любая нестатическая функция базового класса может быть сделана виртуальной, если в ее объявлении использовать спецификатор virtual.

class Point: public Location{

protected:

Boolean vis;

public:

Point (int nx,int ny);

virtual void Show();

virtual void Hide();

virtual void Drag(int by);

Boolean Isvis() { return vis;}

void MoveTo (int nx ,int ny); };

Виртуальными могут быть не любые функции, а только нестатические компонентные функции какого-либо класса. 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.

В производном классе нельзя определять функцию с тем же именем и с той же сигнатурой параметров, но с другим типом возвращаемого значения, чем у виртуальной функции базового класса. Это приводит к ошибке на этапе компиляции.

Если в производном классе ввести функцию с тем же именем и типом возвращаемого значения, что и виртуальная функция базового класса, но с другой сигнатурой параметров, то эта функция производного класса не будет виртуальной. В этом случае с помощью указателя на базовый класс при любом значении этого указателя выполняется обращение к функции базового класса (несмотря на спецификатор virtual и присутствие в производном классе похожей функции).

Рассмотрим, как будет выглядеть вызов виртуальной функции void Show() в производном классе:

class Circle : public Point {

protected:

int R;

public:

Circle (int nx,int ny, int nr);

void Show();

void Hide();

void Expand(int by);

void Contract(int by); };

Как мы видим, спецификатор virtual можно уже не указывать.

Механизм виртуального вызова может быть подавлен с помощью явного использования полного квалифицированного имени. Таким образом, при необходимости вызова из производного класса виртуального метода (компонентной функции) базового класса употребляется полное имя.

Все выше сказанное можно объединить в ПРАВИЛА:

1. не объявлять static;

2) Объявления виртуальных функций в производных классах должны иметь ту же сигнатуру, что и в базовом. Указывать спецификатор virtual в этом случае не обязательно.

3) Виртуальная функция должна быть обязательно определена или быть чистой виртуальной функцией.

Чистой виртуальной функцией называется компонентная функция, которая имеет следующее определение:

virtual тип имя\_функции (список формальных параметров)=0;

В этой записи конструкция “=0” называется “чистый спецификатор”. Пример описания чистой виртуальной функции:

virtual void F()=0;

Чистая виртуальная функция в определении класса показывает, что ее определение откладывается до производных классов. Чистая виртуальная функция “ничего не делает” и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах и показывать, что данный класс абстрактный.

1. **Абстрактные классы**

Абстрактные классы могут использоваться только в качестве базовых для других классов. Предполагается, что абстрактный класс содержит одну или несколько чистых виртуальных функций.

Определять объекты абстрактного класса нельзя, но можно определять ссылку или указатель на абстрактный класс. Нельзя использовать абстрактный класс в качестве параметра функции. Производный от абстрактного класса также считается абстрактным, если в нем не определена хотя бы одна чистая виртуальная функция.

Как всякий класс, абстрактный класс может иметь явно определенный конструктор. Из конструктора возможен вызов методов класса, но любые прямые или опосредованные обращения из конструктора к чистым виртуальным функциям приведут к ошибкам во время выполнения программы.

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. Эти общие понятия обычно невозможно использовать непосредственно, но на их основе можно, как на базе, построить частные производные классы, пригодные для описания конкретных объектов.

Таким образом, механизм абстрактных классов используется при создании сложных иерархий наследования.

В качестве иллюстрации выше сказанного рассмотрим пример для расчета площади треугольника и прямоугольника:

// Создание абстрактного класса

# include <iostream.h>

class area

{double dim1, dim2; //размеры фигуры

public:

void setarea(double dim1,double dim2)

{dim1=d1;

dim2=d2;

}

void getdim(double &dim1,double &dim2)

{d1=dim1;

d2=dim2;

}

virtual double getarea()=0; //чистая виртуальная функция

};

class rectangle: public area

{ public:

double getarea()

{double d1,d2;

getdim(d1,d2);

return d1\*d2;

}

};

class triangle: public area

{ public:

double getarea()

{double d1,d2;

getdim(d1,d2);

return 0.5\*d1\*d2;

}

};

int main()

{area \*p;

rectangle r;

triangle t;

r.setarea(3.3,4.5);

t.setarea(4.0,5.0);

p=&r;

cout<< “Площадь прямоугольника:”<<p->getarea()<<’\n’;

p=&t;

cout<< “Площадь треугольника:”<<p->getarea()<<’\n’;

return 0;

}

Теперь то, что функция getarea() является чистой виртуальной, гарантирует ее обязательную подмену в каждом производном классе.

##### Виртуальные классы

При таком наследовании может возникнуть проблема в наследовании двух экземпляров полей класса A в классе D через B и C (будет занята лишняя память и возникнет путаница). Чтобы это избежать, нужно при создании B и C объявить класс виртуальным.

class B : virtual public A

{**…**}

class C : virtual public A {**…**}

class D : public C, public B

В этом случае последовательность в создании объектов класса D будет следующей: сначала вызывается конструктор класса А (вызывается один раз), затем конструкторы В и С, последним вызывается конструктор класса D. Уничтожение объектов класса D производится в обратном порядке.

##### Шаблоны

Шаблоны представляют собой обобщенные объявления, из которых компилятор может создавать функции или классы с заданными параметрами. Шаблоны позволяют пользователям оперировать параметризованными типами. Имея реализацию шаблона, пользователю нет необходимости переписывать функции для конкретных типов данных.

Шаблоны функций

Объявление шаблона функций выглядит следующим образом:

template <список аргументов>

заголовок функции

{тело функции}

Здесь угловые скобки являются неотъемлемым элементом определения. Список параметров шаблона должен быть заключен именно в угловые скобки.

Список аргументов состоит из выражений типа

<class идентификатор 1,…

class идентификатор n>

идентификатор 1, идентификатор n представляют собой обозначения параметризованных типов. Эти обозначения типов можно использовать вместо типов формальных параметров или локальных переменных.

Под классом в данном случае понимается самый обширный тип данных.

Например, шаблон функции обмена для массива будет выглядеть так:

template <class T>

void Obmen(T A[], int i, int j )

{T temp;

temp=A[i];

A[i]=A[j];

A[j]=temp;}

main()

{int Z[10];

**…**

Obmen (Z,5,7);

float x[100];

Obmen (x,6,7);

**…**

}

Шаблоны функций можно специализировать, т.е. для типов данных, к которым не подходит шаблон, определить функцию с таким же именем. В этом случае алгоритм обращения компилятора к функции будет следующим:

1) Компилятор пытается найти функцию, параметры которой точно соответствуют параметрам вызова.

Если это не удалось, то

2) компилятор пытается найти шаблон, из которого можно сгенерировать функцию с указанными параметрами.

Если и это не удалось, то

3) компилятор рассматривает имеющиеся функции на предмет преобразования типов параметров.

Шаблоны классов

Шаблоны классов задают обобщенные определения семейств классов, использующих произвольные типы и константы. Из шаблона класса компилятор может генерировать объекты с определенными типами и константами.

template <список аргументов>

class NameClass

{ **…** }

Список аргументов состоит в общем случае из конструкций

< class Ид 1,…

class Ид n ,

тип Ид k… >

Идентификаторы Ид 1 - Ид n используются при определении параметризованных типов.

Идентификатор Ид k используется для задания констант.

Объекты шаблонного класса создаются следующим образом:

NameClass <int> A (список инициализации)

Будет создан объект, где формальные типы будут заменены типом int. Список аргументов модифицируется следующим образом: аргументы вида class Идентификатор заменяется конкретным типом (int, double и т.д.), в том числе здесь можно использовать и классы.

Аргумент вида тип Идентификатор заменяется соответствующей константой.

Шаблоны классов специфицировать нельзя.

Пример использования шаблонов класса:

template <class T>

class TSteck

{int n; T \*item;

public:

TSteck(int S=10)

{n=0; item=new T[S];}

void push(T t);

T pop();

}

template <class T>

void TSteck :: push(T t)

{item[n++]=t;}

T TSteck :: pop()

{return item[n--];}

TSteck <float> A(100); // создание объекта

Будет вызван конструктор, и будет моделироваться массив, имитирующий стек.

1. **Обработка исключительных ситуаций в C++**

C++ обеспечивает встроенный механизм обработки ошибок, называемый обработкой исключительных ситуаций. Он позволяет обрабатывать только синхронные исключительные ситуации, т.е. те ситуации, возникновение которых в явном виде предусмотрено в программе. Асинхронные исключения (ошибки оборудования, аппаратные прерывания) им не обрабатываются.

Для обработки исключительных ситуаций введены три ключевых слова:

try, catch, throw. Эти ключевые слова используются для предупреждения аварийного выхода из программы и корректной очистки памяти.

**try** служит дляобозначения секции кода, которая может генерировать исключения, т.е. нескольких “подозрительных” операторов, в которых возможно возникновение исключительной ситуации (например, деление на ноль, переполнение и т.д.).

Операторы, включенные в фигурные скобки за ключевым словом try, принадлежат к try-блоку. Все функции прямо или косвенно вызываемые из try-блока также принадлежат к нему.

main()

{ …

try{

f1();

f2(); }

… }

f1 и f2 могут в свою очередь вызывать другие функции. Эти функции также принадлежат к try-блоку (они также могут генерировать исключения).

Инструкции **catch** должны следовать непосредственно за try-блоком. Они представляют собой секцию кода, на которую передается управление в случае возникновения исключения. Их может быть много.

catch (int i) //перехват ошибки

{ cout<<”перехвачена ошибка номер: ”;

cout<<i<<“\n”;

}

catch-обработчик будет срабатывать при выбросе исключения целого типа.

Последним пишется обработчик с тремя точками:

catch (**…**)

{ **…**

}

Из описанных в программе обработчиков всегда срабатывает только один.

Идентификация того обработчика, который нужно вызвать, осуществляется по типу выброшенного значения. Остальные обработчики не вызываются. Можно перехватывать любые типы данных, включая и типы создаваемых пользователем классов. Фактически в качестве исключительных ситуаций часто используются именно типы классов.

**throw** выбрасывает исключения и осуществляет переход к обработчику. За этим ключевым словом в общем случае может следовать выражение, сводящееся к определенному типу. Это значение может рассматриваться как фактический параметр при вызове функции обработчика. Типом этого операнда определяем какой из обработчиков должен перехватить исключение.

Место, где написано слово throw, называется точкой выброса.

Ключевое слово throw без операнда применяется, когда заново выбрасывается то исключение, которое в данный момент обрабатывается. Этот вариант используется либо в catch-обработчике, либо в функции им вызываемом.

void f1()

{ **…**

if (<условие>)

throw “ошибка”;

}

void f2()

{ **…**

if (<условие>)

{ x=3;

throw x;}

}

Когда выполняется throw

1. создается копия выброшенного объекта;
2. “разматывается” стек, вызывая деструкторы локальных объектов, выходящих из области действия;
3. передается управление ближайшему catch-обработчику, имеющему тип, совпадающий с типом выброшенного объекта. Копия объекта передается обработчику в качестве параметра.

Рассмотрим пример программы, которая иллюстрирует работу с графическими объектами. Здесь реализованы основные принципы объектно-ориентированного программирования на языке C++.

#include <graphics.h> // директива препроцессора для работы с графикой

#include <iostream.h> //директива препроцессора для ввода-вывода

#include <conio.h> //директива препроцессора для ввода одиночного символа

enum Boolean{false, true}; //логический тип

class Location

{protected:

int X;int Y; //защищенные поля класса

public: // открытый блок

// Location (); //Такие конструкторы без параметров создаются компилятором. Его

//объявлять не надо.

Location (int nx, int ny) { X=nx;Y=ny;} // Если функцию определили сразу в классе, то

// это встроенная функция. Она вызывается не

//через стек, а подстановкой в программу.

int GetX() {return X;}

int GetY() { return Y;}

};

// класс Точка

class Point: public Location

{protected:

Boolean vis;

public:

Point (int nx,int ny); //конструктор

//виртуальные функции

virtual void Show(); //функция, реализующая отображение объекта на экране

virtual void Hide(); //функция, позволяющая скрыть объект

virtual void Drag(int by);// движение объекта по экрану в зависимости от шага by

Boolean Isvis() { return vis;}

void MoveTo (int nx ,int ny); // перенос точки из одного места на экране в другое

};

//класс Окружность

class Circle : public Point

{protected:

int R;

public:

Circle (int nx,int ny, int nr); //конструктор

void Show(); // Так как класс Circle является наследником класса Point, функции

void Hide(); // Show() и Hide() также являются виртуальными

void Expand(int by); // увеличение окружности

void Contract(int by);// уменьшение окружности

};

Boolean GetDelta (int &dx,int &dy); //функция определяет какую кнопку нажал

//пользователь

Point::Point(int nx,int ny):Location(nx,ny)

{vis=false; }

void Point:: Show()

{vis= true;putpixel(X,Y,getcolor());}// getcolor() возвращает цвет, который сейчас

// установлен

void Point :: Hide()

{vis=false;putpixel(X,Y,getbkcolor());}//getbkcolor()возвращает цвет фона

void Point :: MoveTo(int nx,int ny)

{Hide();X=nx;Y=ny;Show(); }// точка гасится на старом месте и воспроизводится

// на новом

Boolean GetDelta(int& dx,int& dy)

{char Keych;

Boolean Quit;

dx=0;dy=0;

do

{Keych=getch();

if (Keych == 13)

return (false);

if (Keych == 0)

{Quit=true;

Keych=getch();

switch (Keych){

case 72:dy=-1;break;

case 80: dy=1; break;

case 75:dx=-1; break ;

case 77: dx=1; break ;

default : Quit=false;}; };

} while (!Quit);

return (true); }

// движение объекта по экрану в зависимости от шага by

// Можно нажимать две кнопки одновременно, точка

// будет перемещаться сразу по x и по y под углом 45 градусов

void Point::Drag(int by)

{ int dlx,dly;

int fx,fy;

Show();

fx= GetX();fy= GetY();

while (GetDelta(dlx,dly))

{ fx+=dlx\*by;

fy+=dly\*by;

MoveTo(fx,fy);};}//перемещение точки, пока стрелка нажата

Circle::Circle(int nx,int ny,int nr):Point(nx,ny)

{ R=nr;}

void Circle:: Show()

{vis=true;circle(X,Y,R); }

void Circle :: Hide()

{unsigned tcol; // tcol - текущее значение цвета

tcol=getcolor();

setcolor(getbkcolor());// установка цвета фона

vis=false;

circle(X,Y,R); //стандартная функция рисования окружности

setcolor(tcol); }//гашение окружности, восстановление исходных значений цвета

void Circle :: Expand(int by) // увеличение окружности

{Hide();

R+=by;

if (R<0)

R=0;

Show(); }

void Circle :: Contract(int by) // уменьшение окружности

{Expand(-by);}

// класс Кольцо

class Ring: public Circle

{protected:

int R; int R1;

public:

Ring(int nx,int ny, int nr, int nr1); //конструктор

void Show();

void Hide();

void Expand(int by);

void Contract(int by);

};

Ring::Ring(int nx,int ny, int nr, int nr1):Circle(nx,ny,nr)

{R=nr;R1=nr1;}

void Ring::Show()

{vis=true;circle(X,Y,R);circle(X,Y,R1);}

void Ring::Hide()

{unsigned tcol;

tcol=getcolor();

setcolor(getbkcolor());

vis=false;

circle(X,Y,R);

circle(X,Y,R1);

setcolor(tcol);

}

void Ring:: Expand(int by)

{Hide();

R+=by;

R1+=by;

if (R<0)

R=0;

if (R1<0)

R1=0;

Show();

}

void Ring:: Contract(int by)

{Expand(-by);}

//класс Ауди

class Audi : public Ring

{protected:

int R;int R1;

public:

Audi(int nx,int ny, int nr, int nr1);

void Show();

void Hide();

void Expand(int by);

void Contract(int by);

};

Audi::Audi(int nx,int ny, int nr, int nr1):Ring(nx,ny,nr,nr1)

{R=nr;R1=nr1;}

void Audi::Show()

{vis=true;

circle(X,Y,R);circle(X,Y,R1); //рисование взаимосвязанных колец

circle(X+120,Y,R);circle(X+120,Y,R1);

circle(X+240,Y,R);circle(X+240,Y,R1);

circle(X+360,Y,R);circle(X+360,Y,R1);

}

void Audi::Hide()

{unsigned tcol;

tcol=getcolor();

setcolor(getbkcolor());

vis=false;

circle(X,Y,R);circle(X,Y,R1);

circle(X+120,Y,R);circle(X+120,Y,R1);

circle(X+240,Y,R);circle(X+240,Y,R1);

circle(X+360,Y,R);circle(X+360,Y,R1);

setcolor(tcol);

}

void Audi:: Expand(int by)

{Hide();

R+=by;

R1+=by;

if (R<0)

R=0;

if (R1<0)

R1=0;

Show();

}

void Audi:: Contract(int by)

{Expand(-by);}

//основная часть программы

void main()

//инициализация графического режима

{ int graphdriver=DETECT,graphmode;

initgraph(&graphdriver,&graphmode,"c:\\borlandc\\bgi");

// создание объекта класса Точка и действия с ним

Point P(100,50);

P.Show(); getch();

P.MoveTo(300,100); getch();

P.Hide();

// создание объекта класса Окружность и действия с ним

Circle C(200,200,100);

C.Show(); getch();

C.MoveTo(300,200); getch();

C.Contract(50);getch();

C.Expand(50);getch();

C.Drag(5);

C.Hide();

// создание объекта класса Кольцо и действия с ним

Ring R(300,300,100,80);

R.Show();getch();

R.MoveTo(200,200); getch();

R.Contract(50);getch();

R.Expand(50);getch();

R.Drag(10);

R.Hide();

// создание объекта класса Ауди и действия с ним

Audi A(100,300,80,70);

A.Show();getch();

A.MoveTo(150,200); getch();

A.Contract(50);getch();

A.Expand(50);getch();

A.Drag(10);

A.Hide();

//закрытие графического режима

closegraph();

}