МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Язык С++ и основы технологии объектно-ориентированного программирования

Часть II

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

По специальности ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА (01.03.02)

ВОРОНЕЖ

2017

5. Наследование в языке C++

5.1. Модификаторы наследования

Мы, наконец, готовы поговорить о наследовании. Наследование представляет собой одно из наиболее интересных качеств языка C++. Наследование в C++ – это механизм, который позволяет строить иерархию классов, переходя от более общих характеристик к специфическим, характерным только для классов-наследников. Когда один класс наследуется другим, первый из них называется *базовым* классом, а класс-наследник называется *производным* классом. Новый класс строится на базе уже существующего с помощью конструкции следующего вида:

```
class Parent {....};
class Child : [модификатор наследования] Parent {....};
```

При определении класса потомка за его именем следует разделитель-двоеточие («:»), затем – необязательный модификатор наследования и имя родительского класса. Модификатор наследования определяет видимость наследуемых переменных и методов для пользователей и возможных потомков самого класса-потомка. Другими словами, он определяет, какие права доступа к переменным и методам класса-родителя будут «делегированы» классу-потомку. При реализации наследования, область «видимости» принадлежащих классу данных и методов можно определять выбором ключевого слова `private` (собственный), `public` (общедоступный) или `protected` (защищенный), которые могут произвольно чередоваться внутри описания класса. С двумя первыми модификаторами доступа мы уже знакомы, `private` описывает закрытые члены класса, доступ к которым имеют только методы-члены этого класса, `public` предназначен для описания общедоступных элементов, доступ к которым возможен из любого места в программе. Особый интерес представляют элементы, обладающие модификатором доступа `protected`. Модификатор `protected` используется тогда, когда необходимо, чтобы некоторые члены базового класса оставались закрытыми, но были бы доступны для производного класса. Модификатор `protected` эквивалентен `private` с единственным исключением: защищенные члены базового класса доступны для членов всех производных классов этого базового класса.

То, каким образом изменяется уровень доступа к элементам базового класса в методах производного класса в зависимости от значения модификатора наследования, видно из следующей таблицы:

Модификатор доступа (базовый класс)	**Модификатор наследования**		
	public	**protected**	**private**
public	public	protected	private
protected	protected	protected	private

```
 ~DerivedClass()
  {cout << "Работа деструктора производного класса \n";}
 };

main()
 {
 DerivedClass obj(2,3);
 }
```

Допускается также, что конструктор базового класса может иметь больше параметров, чем конструктор производного класса.

```
class BaseClass
 {
 int j, i;

 public:
 BaseClass(int jj, int ii)
  {j=jj; i=ii;}
 ~BaseClass()
  {cout << "Работа деструктора базового класса \n";}
 };

class DerivedClass: public BaseClass
 {
 int n;

 public:
 DerivedClass(int nn);
 ~DerivedClass()
  {cout << "Работа деструктора производного класса \n";}
 };

DerivedClass :: DerivedClass(int nn): BaseClass(nn/2, nn%2)
 {n=nn;}

main()
 {
 DerivedClass obj(15);
 }
```

Еще раз обратим внимание на то, что в расширенной форме объявления конструктора производного класса описывается **вызов** конструктора базового класса.

5.3. Пример построения классов при наследовании

В качестве примера выберем графические объекты, использование которых может оказаться полезным в самых различных сферах. Разумно начать с класса, который моделирует построение физических пикселей на экране.

```
struct Point
 {
 int X;
 int Y;
 };
```

Но пиксель на экране монитора кроме координат своего положения обладает еще и возможностью «светиться». Расширим структуру:

```
enum Boolean {false, true};          // false = 0, true = 1
struct Point
 {
 int X;
 int Y;
 Boolean Visible;
 };
```

Тип `Boolean` хорошо знаком программистам на Паскале. Этот код использует перечисляемый тип `enum` для проверки `true` (истина) или `false` (ложь). Так как значения перечисляемого типа начинаются с 0, то `Boolean` может иметь одно из двух значений: 0 или 1 (ложь или истина).

Учитывая наш опыт работы со структурой `_3d`, мы должны позаботиться об интерфейсе класса `Point`. Нам потребуются методы для инициализации координат пикселя и указания, «включен» он или нет. Кроме того, если мы захотим сделать внутренние переменные недоступными, следует предоставить какой-либо способ узнать, что в них находится, прочитать их значения регламентированным образом. Следующая версия может выглядеть следующим образом:

```
enum Boolean {false, true};          // false = 0, true = 1
class Point
 {
 protected:
 int X;
 int Y;
 Boolean Visible;

 public:
 int GetX(void) {return X;}
 int GetY(void) {return Y;}
 Boolean isVisible() {return Visible;}
 Point(const Point& cp)          // конструктор копирования
  {X = cp.X; Y = cp.Y; Visible = cp.Visible;}
 Point(int newX=0, int new Y=0);
                                 // прототип конструктора

};

Point :: Point (int NewX, int NewY)          // конструктор
 {
```

```
 X = newX; Y = newY; Visible = false;
 }
```

Теперь у нас есть возможность объявлять объекты типа `Point`:

```
Point Center(320, 120);     // объект Center типа Point
Point *point_ptr;           // указатель на тип Point
point_ptr = &Center;        // указатель показывает на Center
```

Задание аргументов по умолчанию при описании прототипа конструктора дает нам возможность вызывать конструктор без аргументов или с неполным списком аргументов:

```
Point aPoint();
Point Row[80];               // массив из объектов типа Point
Point bPoint(100);
```

Пока мы может создавать объекты класса `Point` и определять их координаты, но не можем пока их показывать. Так что необходимо дополнить класс `Point` соответствующими методами:

```
class Point
 {
 ...
 public:
 ...
 void Show();
 void Hide();
 void MoveTo(int newX, int newY);
 };

void Point::Show()
 {
 Visible = true;
 putpixel(X,Y,getcolor());
 }

void Point::Hide()
 {
 Visible = false;
 putpixel(X,Y,getbkcolor());
 }

void Point::MoveTo(int newX, int newY)
 {
 Hide();
 X = newX;
 Y = newY;
 Show();
 }
```

Теперь, когда у нас есть полноценный класс Point, можно создавать объекты-точки, «включать» и «выключать» их на экране, а также перемещать их по нему.

```
Point pointA(50,50);
pointA.Show();
pointA.MoveTo(100,130);
pointA.Hide();
```

Если потребуется создать класс для другого графического объекта, то можно выбрать один из двух способов: либо начать его реализацию «с нуля», либо воспользоваться уже готовым классом Point, сделав его базовым. Второй способ кажется более предпочтительным, поскольку он предполагает использование уже готовых модулей, все, что при этом нужно – это создать новый производный от Point класс, дополнив его новыми состояниями и методами и/или переопределив некоторые методы базового класса.

Попробуем создать класс Circle для описания окружности. Окружность, в известном смысле, является жирной точкой. Она имеет все, что имеет точка (позицию X и Y и видимое/невидимое состояние), а также радиус. Может показаться, что класс Circle появляется только затем, чтобы иметь единственный дополнительный элемент Radius, однако не следует забывать обо всех элементах, которые наследует Circle, являясь классом, порожденным из Point: Circle имеет X, Y, а также Visible, даже если их не видно в определении класса для Circle.

```
class Circle: public Point
 {
 int Radius;           // private по умолчанию

 public:
 Circle(int initX, int initY, int initR);
 void Show();
 void Hide();
 void Expand(int deltaR);
 void Contract(int deltaR);
 void MoveTo(int newX, int newY);
 };
Circle::Circle(int initX, int initY, int initR)
                        // конструктор
  :Point(initX, initY)  // вызов конструктора базового
                        // класса
 {
 Radius = initR;
 }

void Circle::Show()
 {
 Visible = true;
 Circle(X,Y, Radius);
```

```
 }

void Circle::Hide()    // скрыть = зарисовать цветом фона
 {
 Visible = false;
 unsigned int tempColor = getcolor();
 setcolor (getbkcolor());
 Circle(X,Y, Radius);
 setcolor(tempColor);
}

void Circle::Expand(int deltaR)
 {
 Hide();
 Radius += deltaR;
 Show();
 }

void Circle::Contract(int deltaR)
 {
 Expand (-deltaR);
 }

void Circle::MoveTo(int newX, int newY)
 {
 Hide();
 X = newX;
 Y = newY;
 Show();
 }

main()
 {
 int graphDr = DETECT, graphMode;
 initgraph ( &graphDr, &graphMode, "");

 Circle C(150,200,50);    /* создать объект окружность
                             с центром в т.(150, 200) и
                             радиусом 50 */
 C.Show();                // отобразить окружность
 getch();
 C.MoveTo(300,100);       // переместить
 getch();
 C.Expand(50);            // растянуть
 getch();
 C.Contract(70);          // сжать
 getch();
 closegraph();
 }
```

Поскольку класс `Circle` – производный от класса `Point`, то, соответственно, класс `Circle` наследует из класса `Point` состояния `X`, `Y`, `Visible`, а