Polymorphism

Before getting any deeper into this chapter, you should have a proper understanding of pointers and class inheritance. If you are not really sure of the meaning of any of the following expressions, you should review the indicated sections:

在深入本章之前，您应该对指针和类继承有一个正确的理解。如果你不确定下列任何一个表达式的意思，你应该查看指示部分:

Statement:	Explained in:
`int A::b(int c) { }`	Classes
`a->b`	Data structures
`class A: public B {};`	Friendship and inheritance

Pointers to base class

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature.

类继承的一个关键特性是，指向派生类的指针与指向基类的指针类型兼容。*多态性*是利用这个简单但强大和通用特性的艺术。

The example about the rectangle and triangle classes can be rewritten using pointers taking this feature into account:

考虑到这个特性，矩形和三角形类的例子可以使用指针重写:

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Function `main` declares two pointers to `Polygon` (named `ppoly1` and `ppoly2`). These are assigned the addresses of `rect` and `trgl`, respectively, which are objects of type `Rectangle` and `Triangle`. Such assignments are valid, since both `Rectangle` and `Triangle` are classes derived from `Polygon`.

函数main声明了两个指向Polygon的指针(命名为poly1和poly2)。它们分别被赋给rect和trgl的地址，它们是矩形和三角形类型的对象。这样的赋值是有效的，因为Rectangle和Triangle都是从Polygon派生的类。

Dereferencing `ppoly1` and `ppoly2` (with `ppoly1->` and `ppoly2->`) is valid and allows us to access the members of their pointed objects. For example, the following two statements would be equivalent in the previous example:

解引用ppoly1和ppoly2(使用ppoly1->和ppoly2->)是有效的，并允许我们访问它们的指向对象的成员。例如，下面两个语句在前面的例子中是等价的:

```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```

But because the type of both `ppoly1` and `ppoly2` is pointer to `Polygon` (and not pointer to `Rectangle` nor pointer to `Triangle`), only the members inherited from `Polygon` can be accessed, and not those of the derived classes `Rectangle` and `Triangle`. That is why the program above accesses the `area` members of both objects using `rect` and `trgl` directly, instead of the pointers; the pointers to the base class cannot access the `area` members.

但是因为poly1和poly2的类型都是指向Polygon的指针(而不是指向Rectangle或Triangle的指针)，所以只能访问从Polygon继承来的成员，而不能访问派生类Rectangle和Triangle的成员。这就是为什么上面的程序直接使用rect和trgl来访问两个对象的area成员，而不是指针;指向基类的指针不能访问区域成员。

Member `area` could have been accessed with the pointers to `Polygon` if `area` were a member of `Polygon` instead of a member of its derived classes, but the problem is that `Rectangle` and `Triangle` implement different versions of `area`, therefore there is not a single common version that could be implemented in the base class.

区域成员可以访问的成员指针指向多边形面积要是多边形代替它的派生类的成员,但问题是,矩形和三角形实现不同版本的区域,因此没有一个共同的版本,可以在基类中实现。

Virtual members

A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the `virtual` keyword:

虚成员是可以在派生类中重新定义的成员函数，同时通过引用保留其调用属性。要变成虚函数的语法是在它的声明之前加上virtual关键字:

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
```

```
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

In this example, all three classes (`Polygon`, `Rectangle` and `Triangle`) have the same members: `width`, `height`, and functions `set_values` and `area`.

在这个例子中，所有三个类(Polygon、Rectangle和Triangle)都有相同的成员:width、height和函数set_values和area。

The member function `area` has been declared as `virtual` in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if `virtual` is removed from the declaration of `area` in the example above, all three calls to `area` would return zero, because in all cases, the version of the base class would have been called instead.

成员函数区域已在基类中声明为虚函数区域，因为稍后将在每个派生类中重新定义它。也可以在派生类中重新定义非虚成员，但派生类的非虚成员不能通过基类的引用访问:例如，如果在上面的例子中从area的声明中删除了virtual，那么对area的所有三个调用都将返回0，因为在所有情况下，都将调用基类的版本。

Therefore, essentially, what the `virtual` keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class that is pointing to an object of the derived class, as in the above example.

因此,从本质上讲,什么虚拟关键字是允许派生类的成员具有相同的名称作为一个基类的适当从指针,更准确地说,当指针的类型是一个指向基类的指针指向派生类的对象,在上面的例子中。

A class that declares or inherits a virtual function is called a *polymorphic class*.

声明或继承虚函数的类称为多态类。

Note that despite of the virtuality of one of its members, `Polygon` was a regular class, of which even an object was instantiated (`poly`), with its own definition of member `area` that always returns 0.

注意，尽管它的一个成员是虚的，但Polygon是一个常规类，其中甚至实例化了一个对象(poly)，它自己定义的成员区域总是返回0。

Abstract base classes

Abstract base classes are something very similar to the `Polygon` class in the previous example. They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions). The syntax is to replace their definition by `=0` (an equal sign and a zero):

抽象基类与前面例子中的Polygon类非常相似。它们是只能用作基类的类，因此允许具有没有定义的虚成员函数(称为纯虚函数)。语法是将它们的定义替换为=0(一个等号和一个0):

An abstract base `Polygon` class could look like this:

一个抽象的基本多边形类看起来是这样的:

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Notice that `area` has no definition; this has been replaced by `=0`, which makes it a *pure virtual function*. Classes that contain at least one *pure virtual function* are known as *abstract base classes*.

注意，区域没有定义;这个已经被替换为=0，这使它成为一个纯虚函数。包含至少一个纯虚函数的类称为抽象基类。

Abstract base classes cannot be used to instantiate objects. Therefore, this last abstract base class version of `Polygon` could not be used to declare objects like:

抽象基类不能用于实例化对象。因此，Polygon的最后一个抽象基类版本不能用来声明如下对象:

```
Polygon mypolygon;   // not working if Polygon is abstract base class
```

But an *abstract base class* is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities. For example, the following pointer declarations would be valid:

但抽象基类并非完全无用。它可以用来创建指向它的指针，并利用它的所有多态能力。例如，下面的指针声明是有效的:

```
Polygon * ppoly1;
Polygon * ppoly2;
```

And can actually be dereferenced when pointing to objects of derived (non-abstract) classes. Here is the entire example:

当指向派生(非抽象)类的对象时，实际上可以解除引用。下面是整个例子:

```
// abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

In this example, objects of different but related types are referred to using a unique type of pointer (`Polygon*`) and the proper member function is called every time, just because they are virtual. This can be really useful in some circumstances. For example, it is even possible for a member of the abstract base class `Polygon` to use the special pointer `this` to access the proper virtual members, even though `Polygon` itself has no implementation for this function:

在这个例子中，不同但相关类型的对象使用唯一类型的指针(Polygon*)来引用，并且每次都会调用适当的成员函数，因为它们是虚的。这在某些情况下非常有用。例如，抽象基类Polygon的成员甚至可以使用特殊的this指针来访问适当的虚成员，尽管Polygon本身没有实现这个函数:

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area() =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

Virtual members and abstract classes grant C++ polymorphic characteristics, most useful for object-oriented projects. Of course, the examples above are very simple use cases, but these features can be applied to arrays of objects or dynamically allocated objects.

虚成员和抽象类赋予c++多态特征，这对面向对象的项目最有用。当然，上面的示例是非常简单的用例，但是这些特性可以应用于对象数组或动态分配的对象。

Here is an example that combines some of the features in the latest chapters, such as dynamic memory, constructor initializers and polymorphism:

下面是一个结合了最新章节中的一些特性的例子，比如动态内存、构造函数初始化和多态性:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};

int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

补充说明：

如果基类里面有虚成员函数，则最好把析构函数也设置为虚函数
否则，派生类在用基类指针调用时，发生内存泄露，析构时不会调用派生类的析构函数

`virtual ~Polygon(){}`

Notice that the `ppoly` pointers:

注意ppoly指针:

```
Polygon * ppoly1 = new Rectangle (4,5);
Polygon * ppoly2 = new Triangle (4,5);
```

are declared being of type "pointer to `Polygon`", but the objects allocated have been declared having the derived class type directly (`Rectangle` and `Triangle`).

被声明为“pointer to Polygon”类型，但是被分配的对象被直接声明为具有派生类类型(Rectangle和Triangle)。