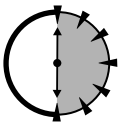
SESSION

Polymorphism

Session Checklist

- ✔ Overriding member functions in a subclass
- ✔ Applying polymorphism (alias late binding)
- ✔ Comparing polymorphism to early binding
- ✔ Taking special considerations with polymorphism



30 Min. To Go

Inheritance gives us the capability to describe one class in terms of another. Just as importantly, it highlights the relationship between classes. Once again, a microwave oven is a type of oven. However, there's still a piece of the puzzle missing.

You have probably noticed this already, but a microwave oven and a conventional oven look nothing alike. These two types of ovens don't work exactly alike either. Nevertheless, when I say "cook" I don't want to worry about the details of how each oven performs the operation. This session describes how C++ handles this problem.

Overriding Member Functions

It has always been possible to overload a member function in one class with a member function in the same class as long as the arguments are different. It is also possible to overload a member in one class with a member function in another class even if the arguments are the same.

Inheritance introduces another possibility: a member function in a subclass can overload a member function in the base class.

Overloading a member function in a subclass is called overriding. This relationship warrants a different name because of the possibilities it introduces.

Consider, for example, the simple `EarlyBinding` program shown in Listing 22-1.

Listing 22-1
EarlyBinding Demonstration Program

```
// EarlyBinding - calls to overridden member functions
//                are resolved based on the object type

#include <stdio.h>
#include <iostream.h>

class Student
{
  public:
    double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
  public:
    double calcTuition()
    {
```

```
        return 1;
    }
};

int main(int nArgc, char* pszArgs[])
{
    // the following expression calls
    // Student::calcTuition();
    Student s;
    cout << "The value of s.calcTuition is "
         << s.calcTuition()
         << "\n";

    // this one calls GraduateStudent::calcTuition();
    GraduateStudent gs;
    cout << "The value of gs.calcTuition is "
         << gs.calcTuition()
         << "\n";
    return 0;
}
```

Output

```
The value of s.calcTuition is 0
The value of gs.calcTuition is 1
```

As with any case of overriding, when the programmer refers to `calcTuition()`, C++ has to decide which `calcTuition()` is intended. Normally, the class is sufficient to resolve the call, and this example is no different. The call `s.calcTuition()` refers to `Student::calcTuition()` because `s` is declared locally as a `Student`, whereas `gs.calcTuition()` refers to `GraduateStudent::calcTuition()`.

The output from the program `EarlyBinding` shows that calls to overridden member functions are resolved according to the type of the object.

Resolving calls to overridden member functions based on the type of the object is called *compile-time* binding. This is also called *early binding.*

Enter Polymorphism

Overriding functions based on the class of the object is all very nice, but what if the class of the object making the call can't be determined unambiguously at compile time? To demonstrate how this can occur, let's change the preceding program in a seemingly trivial way. The result is the program `AmbiguousBinding` shown in Listing 22-2.

Listing 22-2
AmbiguousBinding Program

```
// AmbiguousBinding - the situation gets confusing
//                    when the compile-time type and
//                    run-time type don't match
#include <stdio.h>
#include <iostream.h>

class Student
{
  public:
    double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
  public:
    double calcTuition()
    {
        return 1;
    }
};

double fn(Student& fs)
{
    // to which calcTuition() does this call refer?
    // which value is returned?
    return fs.calcTuition();
```

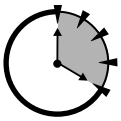
```
}

int main(int nArgc, char* pszArgs[])
{
    // the following expression calls
    // Student::calcTuition();
    Student s;
    cout << "The value of s.calcTuition when "
         << "called through fn() is "
         << fn(s)
         << "\n";

    // this one calls GraduateStudent::calcTuition();
    GraduateStudent gs;
    cout << "The value of gs.calcTuition when "
         << "called through fn() is "
         << fn(gs)
         << "\n";
    return 0;
}
```

The only difference between Listing 22-2 and Listing 22-1 is that the calls to `calcTuition()` are made through an intermediate function, `fn()`. The function `fn(Student& fs)` is declared as receiving a `Student`, but depending on how `fn()` is called, `fs` can be a `Student` or a `GraduateStudent`. (Remember? A `GraduateStudent` IS_A `Student`.) But these two types of objects calculate their tuition differently.

Neither `main()` nor `fn()` really care anything about how tuition is calculated. We would like `fs.calcTuition()` to call `Student::calcTuition()` when `fs` is a `Student`, but call `GraduateStudent::calcTuition()` when `fs` is a `GraduateStudent`. But this decision can only be made at run time when the actual type of the object passed is determinable.

In the case of the `AmbiguousBinding` program, we say that the compile-time type of `fs`, which is always `Student`, differs from the run-time type, which may be `GraduateStudent` or `Student`.

The capability to decide which of several overridden member functions to call based on the run-time type is called *polymorphism,* or *late binding. Polymorphism* comes from *poly* (meaning multiple) and *morph* (meaning form).



20 Min. To Go

Polymorphism and Object-Oriented Programming

Polymorphism is key to the power of object-oriented programming. It's so important that languages that don't support polymorphism cannot advertise themselves as object-oriented languages. Languages that support classes but not polymorphism are called *object-based languages*. Ada is an example of such a language.

Without polymorphism, inheritance has little meaning.

Remember how I made nachos in the oven? In this sense, I was acting as the late binder. The recipe read: "Heat the nachos in the oven." It didn't read: "If the type of oven is a microwave, do this; if the type of oven is conventional, do that; if the type of oven is convection, do this other thing." The recipe (the code) relied on me (the late binder) to decide what the action (member function) `heat` means when applied to the oven (the particular instance of class `Oven`) or any of its variations (subclasses), such as a microwave oven (`Microwave`). This is the way people think, and designing a language along these lines enables the software model to more accurately describe what people are thinking.

There also are the mundane issues of maintenance and reusability. Suppose that I had written this great program that used the class `Student`. After months of design, coding, and testing, I release this application.

Time passes and my boss asks me to add to this program the capability to handle graduate students who are similar but not identical to normal students. Deep within the program, `someFunction()` calls the `calcTuition()` member function as follows:

```
void someFunction(Student &s)
{
    //...whatever it might do...
    s.calcTuition();
    //...continues on...
}
```

If C++ did not support late binding, I would need to edit `someFunction()` to something similar to the following to add class `GraduateStudent`:

```
#define STUDENT 1
#define GRADUATESTUDENT 2
void someFunction(Student &s)
{
    //...whatever it might do...
    //add some member type that indicates
```

```
    //the actual type of the object
    switch (s.type)
    {
        STUDENT:
            s.Student::calcTuition();
            break;
        GRADUATESTUDENT:
            s.GraduateStudent::calcTuition();
            break;
    }
    //...continues on...
}
```

By using the full name of the function, the expression `s.GraduateStudent::calcTuition()` forces the call to the `GraduateStudent` version even though `s` is declared to be a `Student`.

I would add the member `type` to the class, which I would then set to `STUDENT` in the constructor for `Student` and to `GRADUATESTUDENT` in the constructor for `GraduateStudent`. The value of `type` would refer to the run-time type of `s`. I would then add the test in the preceding code snippet to call the proper member function depending on the value of this member.

That doesn't seem so bad, except for three things. First, this is only one function. Suppose `calcTuition()` is called from a lot of places and suppose that `calcTuition()` is not the only difference between the two classes. The chances are not good that I will find all the places that need to be changed.

Second, I must edit (read "break") code that was debugged, checked in, and working, introducing opportunities for error. Edits can be time-consuming and boring, which increases the possibility of error. Any one of my edits may be wrong or may not fit in with the existing code. Who knows?

Finally, after I've finished editing, redebugging, and retesting everything, I now have two versions to track (unless I can drop support for the original version). This means two sources to edit when bugs are found and some type of accounting system to keep them straight.

What happens when my boss wants yet another class added? (My boss is like that.) Not only do I get to repeat the process, but I'll also have three copies to track.

With polymorphism, there's a good chance that all I need to do is add the new subclass and recompile. I may need to modify the base class itself, but at least it's all in one place. Modifications to the application should be minimal to none.

This is yet another reason to leave data members protected and access them through public member functions. Data members cannot be polymorphically overridden by a subclass, whereas a member function can.

How Does Polymorphism Work?

Given all that I've said so far, it may be surprising that the default for C++ is early binding. The output from the `AmbiguousBinding` program is shown below.

```
The value of s.calcTuition when called through fn() is 0
The value of gs.calcTuition when called through fn() is 0
```

The reason is simple. Polymorphism adds a small amount of overhead both in terms of data storage and code needed to perform the call. The founders of C++ were concerned that any additional overhead they introduced would be used as a reason not to adopt C++ as the systems language of choice, so they made the more efficient early binding the default.

To indicate polymorphism, the programmer must flag the member function with the C++ keyword `virtual`, as shown in program LateBinding contained in Listing 22-3.

Listing 22-3
LateBinding Program

```
// LateBinding - in late binding the decision as to
//               which of two overridden functions
//               to call is made at run-time
#include <stdio.h>
#include <iostream.h>

class Student
{
  public:
    virtual double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
```

```
  public:
    virtual double calcTuition()
    {
        return 1;
    }
};

double fn(Student& fs)
{
    // because calcTuition() is declared virtual this
    // call uses the run-time type of fs to resolve
    // the call
    return fs.calcTuition();
}


int main(int nArgc, char* pszArgs[])
{
    // the following expression calls
    // fn() with a Student object
    Student s;
    cout << "The value of s.calcTuition when\n"
         << "called virtually through fn() is "
         << fn(s)
         << "\n\n";

    // the following expression calls
    // fn() with a GraduateStudent object
    GraduateStudent gs;
    cout << "The value of gs.calcTuition when\n"
         << "called virtually through fn() is "
         << fn(gs)
         << "\n\n";
    return 0;
}
```

The keyword `virtual` added to the declaration of `calcTuition()` is a virtual member function. That is to say, calls to `calcTuition()` will be bound late if the run-time type of the object being used cannot be determined.

The LateBinding program contains the same call to fn() as shown in the two earlier versions. In this version, however, the call to calcTuition() goes to Student::calcTuition() when fs is a Student and to GraduateStudent::calcTuition() when fs is a GraduateStudent.

The output from LateBinding is shown below. Declaring calcTuition() virtual tells fn() to resolve calls based on the run-time type.

```
The value of s.calcTuition when
called virtually through fn() is 0

The value of gs.calcTuition when
called virtually through fn() is 1
```

When defining the virtual member function, the virtual tag goes only with the declarations and not with the definition, as the following example illustrates:

```
class Student
{
  public:
    // declare function to be virtual here
    virtual double calcTuition()
    {
        return 0;
    }
};

// don't include the 'virtual' in the definition
double Student::calcTuition()
 {
     return 0;
 }
```

10 Min. To Go

When Is a Virtual Function Not?

Just because you think a particular function call is bound late doesn't mean it is. C++ generates no indication at compile time of which calls it thinks are bound early and late.

The most critical thing to watch for is that all the member functions in question are declared identically, including the return type. If not declared with the same

arguments in the subclasses, the member functions are not overridden polymorphically, whether or not they are declared virtual. Consider the following code snippet:

```
#include <iostream.h>
class Base
{
  public:
    virtual void fn(int x)
    {
        cout << "In Base class, int x = " << x << "\n";
    }
};
class SubClass : public Base
{
  public:
    virtual void fn(float x)
    {
        cout << "In SubClass, float x = " << x << "\n";
    }
};

void test(Base &b)
{
    int i = 1;
    b.fn(i);          //this call not bound late
    float f = 2.0;
    b.fn(f);          //neither is this one
}
```

`fn()` in `Base` is declared as `fn(int)`, whereas the `SubClass` version is declared `fn(float)`. Because the functions have different arguments, there is no polymorphism. The first call is to `Base::fn(int)` — not surprising considering that `b` is of class `Base` and `i` is an `int`. However, the next call also goes to `Base::fn(int)` after converting the `float` to an `int`. No error is generated because this program is legal (other than a possible warning concerning the demotion of `f`. The output from calling `test()` shows no sign of polymorphism:

```
Calling test(bc)
In Base class, int x = 1
In Base class, int x = 2
Calling test(sc)
```

```
In Base class, int x = 1
In Base class, int x = 2
```

If the arguments don't match exactly, there is no late binding — with one exception: If the member function in the base class returns a pointer or reference to a base class object, an overridden member function in a subclass may return a pointer or reference to an object of the subclass. In other words, the following is allowed:

```
class Base
{
  public:
    Base* fn();
};

class Subclass : public Base
{
  public:
    Subclass* fn();
};
```

In practice, this is quite natural. If a function is dealing with `Subclass` objects, it seems natural that it should continue to deal with `Subclass` objects.

Virtual Considerations

Specifying the class name in the call forces the call to bind early. For example, the following call is to `Base::fn()` because that's what the programmer indicated, even if `fn()` is declared virtual:

```
void test(Base &b)
{
    b.Base::fn();            //this call is not bound late
}
```

A virtual function cannot be inlined. To expand a function inline, the compiler must know which function is intended at compile time. Thus, although the example member functions so far were declared in the class, all were outline functions.

Constructors cannot be virtual because there is no (completed) object to use to determine the type. At the time the constructor is called, the memory that the

object occupies is just an amorphous mass. It's only after the constructor has finished that the object is a member of the class in good standing.

By comparison, the destructor normally should be declared virtual. If not, you run the risk of improperly destructing the object, as in the following circumstance:

```
class Base
{
  public:
    ~Base();
};
class SubClass : public Base
{
  public:
    ~SubClass();
};
void finishWithObject(Base *pHeapObject)
{
    //...work with object...
    //now return it to the heap
    delete pHeapObject;  // this calls ~Base() no matter
}                        // what the run-time type
                         // of pHeapObject is
```

If the pointer passed to `finishWithObject()` really points to a `SubClass`, the `SubClass` destructor is not invoked properly. Declaring the destructor virtual solves the problem.

So, when would you not want to declare the destructor virtual? There's only one instance. Earlier I said that virtual functions introduce a "little" overhead. Let me be more specific. When the programmer defines the first virtual function in a class, C++ adds an additional, hidden pointer — not one pointer per virtual function, just one pointer if the class has any virtual functions. A class that has no virtual functions (and does not inherit any virtual functions from base classes) does not have this pointer.

Now, one pointer doesn't sound like much, and it isn't unless the following two conditions are true:

- The class doesn't have many data members (so that one pointer represents a lot compared to what's there already).
- You intend to create a lot of objects of this class (otherwise, the overhead doesn't make any difference).

Done!

If both these conditions are met and your class doesn't already have any virtual member functions, you might not want to declare the destructor virtual.

Normally, you should declare the destructor virtual. If you don't declare the destructor virtual, document it!

REVIEW

By itself, inheritance is a nice but limited capability. Combined with polymorphism, inheritance is a powerful programming aid.

- Member functions in a class may be overridden by member functions defined in the base class. Calls to these functions are resolved at compile time based on the declared (compile-time) class. This is called early binding.
- A member function may be declared virtual, in which case calls are resolved based on the run-time class. This is called polymorphic or late binding.
- Calls in which the run-time and compile-time classes are known to be identical are bound early, regardless of whether the member function is declared virtual or not.

QUIZ YOURSELF

1. What is polymorphism? (See "Enter Polymorphism.")
2. What's another word for polymorphism? (See "Enter Polymorphism.")
3. What's the alternative and what is it called? (See "Overriding Member Functions.")
4. Name three reasons that C++ includes polymorphism? (See "Polymorphism and Object-Oriented Programming.")
5. What keyword is used to declare a member function polymorphic? (See "How Does Polymorphism Work?")