LECTURE 9

Virtual Functions and Polymorphism

It has always been possible to overload a function in one class with a member function from another class. With inheritance, you can overload a base class member function with a member function in a subclass as well. For example:

```
class Student
{
  public:
  float calcTuition();
  //...other stuff
};
class GraduateStudent: public Student
{
  public:
  float calcTuition();
  //...other stuff
};
main()
{
 Student s;
 GraduateStudent gs;
 s.calcTuition();          //calls Student::calcTuition()
 gs.calcTuition();         //call GraduateStudent::calcTuition()
}
```

But what if the exact class of the object can't be determined at compile time? To demonstrate how this can occur, let's change the preceding program in a seemingly trivial way:

```
class Student
{
  public:
  float calcTuition();
  //...other stuff
};
class GraduateStudent: public Student
{
  public:
  float calcTuition();
  //...other stuff
};
void func(Student &x)
```

```
{
 x.calcTuition();              //to which calcTuition does this refer?
}                              //(answer: base class Student::calcTuition)
main()
{
 Student s;
 GraduateStudent gs;
 func(s);
 func(gs);
}
```

Instead of calling `calcTuition()` directly, the call is now made through an intermediate function, `func(Student &x)`. Depending on how `func(Student &x)` is called, `x` can be a `Student` or a `GraduateStudent`. You would like `x.calcTuition()` to call `Student::calcTuition()` when `x` is a `Student` but call `GraduateStudent::calcTuition()` when `x` is a `GraduateStudent`. Normally the compiler decides which function to call at compile time. Even when a function is overloaded, the compiler uses the different argument types to decide at compile time. But here the decision cannot be made until run time, when the actual type of the object can be determined.

The capability to decide at run time which of several overloaded member functions to call based on the actual type is called *polymorphism*. *Poly* means many and *morph* means form (as in amorphous). *Late binding* is the mechanism C++ uses to implement polymorphism. I will tend to use the two terms interchangably. Deciding which overloaded member functions to call at compile time is called *early binding* because that sounds like the opposite of late binding. The default for C++ is early binding because polymorphism adds a small amount of overhead both in terms of data storage and code needed to perform the call.

Terminology: Another name for the actual type is the *run–time type*. In the previous example, the run–time type of `x` is `Student` in the call `func(s)` and the run–time type of `x` is `GraduateStudent` in the call `func(gs)`. The `declared type` of `x` is `Student` because that's what the declaration of `func` says.

To indicate polymorphism, the programmer must flag the member function with the keyword *virtual*. Virtual functions allow the programmer to declare the functions in a base class that can be redefined in each derived class.

```
class Base
  {
    public:
      virtual void func();
  };
class DerivedClass: public Base
  {
    public:
        virtual void func();
    };
  void extfunc(Base &b)
    {
      b.func();
    }
  main()
    {
      Base bc;
      DerivedClass dc;
      extfunc(bc);            //calls Base::func()
      extfunc(dc);            //calls DerivedClass::func()
    }
```

You need to declare the function virtual only in the base class. The "virtualness" is carried down to the derived class automatically. So we could have written:

```
  class Base
    {
      public:
        virtual void func();
    };
  class DerivedClass: public Base
    {
      public:
        void func();         //implicitly virtual
    };
```

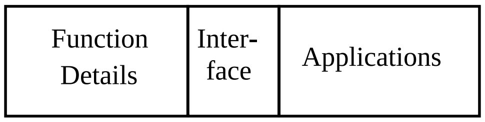
To allow a virtual function declaration to act as an interface to functions defined in derived classes, the argument types specified for a function in a derived class cannot differ from the argument types declared in the base class. If the arguments don't match, there is no late binding and the function is specified at compile time just like any ordinary overloaded function. This is true even if the keyword "virtual" is used. Only very slight changes are allowed for the return type. In particular, if the member function of the base class returns a pointer or reference to a base class object, an overloaded 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:
        virtual Base* func();
    };
   class DerivedClass: public Base
```

```
 {
   DerivedClass* func();
 };
```

A virtual function must be defined for the class in which it is first declared (unless it is declared to be a pure virtual function). A virtual function can be used even if no class is derived from its class, and a derived class that does not need its own version of a virtual function need not provide one. When deriving a class, simply provide an appropriate function, if it is needed.

The great benefit of polymorphism is that one can add new derived classes without changing the base class or the older derived classes. In some cases you don't even have to recompile the existing classes. Polymorphism greatly facilitates encapsulation of the code. The details of a function can be dealt with in the base class and its derived classes, while the application can call the generic function which is bound at run time. The generic function acts as a buffer or an interface between the details of the function and the applications which will use various versions of the function.



For example,

```
class StuffYouCook{};
class Nachos: public StuffYouCook {};
class Oven
 {
   public:
    virtual void cook(Nachos &nachos);
    //other stuff...
 };
class Microwave: public Oven
 {
   public:
    virtual void cook(Nachos &nachos);
    //other stuff...
 };
 Nachos makeNachos(Oven &oven)        //external function
  {
   Nachos n;
   oven.cook(n);                      //bound at run time
   return n;
  }
```

The function `makeNachos` is passed an `Oven` of some type. Given that oven, it assembles all the stuff into an object `n` and then cooks them by calling `oven.cook`. Exactly which function is used, function `Oven::cook` or `Microwave::cook`, depends on the real–time type of `oven`. The function `makeNachos` has no idea–and doesn't want to know–what the run–time type of `oven` is. Notice that `makeNachos` doesn't need to know the details of `oven`. That's what we mean by encapsulation and hiding the details of one part of the program from another part of the program. Polymorphism has allowed us to greatly simplify the code since `makeNachos` doesn't have any of the `oven` or `Nachos` details. The result is extensible. If a new class of `oven` comes along, e.g. `ToasterOven`, with a new cook function (`ToasterOven::cook(Nachos&)`), we do not need to change `makeNachos` to incorporate the new function. Polymorphism automatically includes the new function and calls it when necessary.

Polymorphism is the 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.

Comments on Virtual Functions

1. Static member functions cannot be declared virtual. Because static member functions are not called with an object, there is no run–time object to have a type.

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

```
      class Base
    {
      public:
        virtual void func();
    };
  class DerivedClass: public Base
    {
      public:
        void func();          //implicitly virtual
    };
  void test(Base &b)
    {
      b.Base::func();         //This call is not bound late
    }
```

3. A virtual function cannot be inlined. To expand a function inline, the compiler must know which function is intended at compile time.

4. 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. But you can define a virtual function that has the same effect as a virtual constructor; the function calls a constructor and returns a constructed object (see Stroustrup (3rd ed.), pages 424-425, for details).

Virtual Destructors

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

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

If the pointer passed to `finishWithObject` really points to `DerivedClass`, the `DerivedClass` destructor is not invoked properly. Declaring the destructor virtual solves the problem:

```
class Base
{
  public:
  virtual ~Base(){};             //even an empty destructor will do
};
class DerivedClass: public Base
{
  public:
  ~DerivedClass();              //implicitly virtual
};
void finishWithObject(Base *pHeapObject)
```

```
  {
    //work with object...
    //now return it to heap memory
    delete pHeapObject;           //this calls the correct destructor
```

When an object is deleted, C++ runs the object's destructor. Given that we only have a pointer to the `Base` object, which destructor should C++ run, `~Base()` or `~DerivedClass()`? The answer depends on which kind of object the pointer points to. Such behavior is exactly what declaring a member to be `virtual` arranges. Since one can't predict when writing a base class whether any class eventually derived from the base class will have a destructor member, it is safest to anticipate the possibility by providing a virtual destructor in the public interface of the base class. You should definitely use a virtual destructor if you use virtual functions and pointers to the base class. In short, the presence of a virtual destructor in `Base` ensures that every class derived from it will be supplied with a destructor (thus getting the size of the object right), even if the derived class doesn't have a user–defined destructor. Even an empty destructor will do.

How Polymorphism Is Implemented

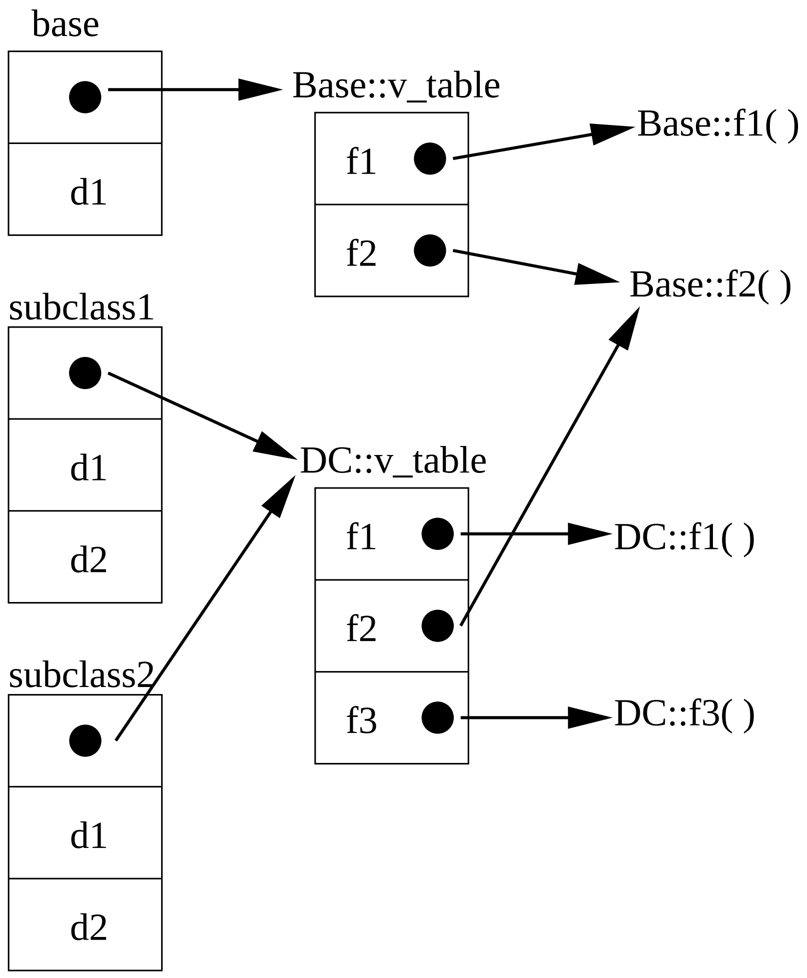
When a class has at least one virtual function, C++ adds an additional, hidden pointer–not one pointer per virtual function, just one pointer if the class has any virtual functions. This pointer points to a structure known the `v_table`. The `v_table` contains a list of pointers to all the virtual functions defined in the class. Suppose we have the following class hierarchy:

```
class Base
{
  public:
    virtual void f1();
    virtual void f2();
            void nonVirtualFn();
    int d1;
}
class DC: public Base
{
  public:
    virtual void f3();
    virtual void f1();
    int d2;
};
void func(Base &b)
{
  b.f1();
  b.f2();
};

main()
{
  Base base;
  DC subclass1;             //declare 2 subclass elements
  DC subclass2;

  func(base);
  func(subclass1);
}
```

The `v_table` configuration is shown in the figure. Notice that the two `DC` objects have their own `v_table` pointers, but they share the same `v_table`.



When `func(base)` is called in `main`, it is passed the object `base`. The call to `b.f1` leads to `Base::fn1` if you just follow the arrows in the figure. The second time `func` is called in main, it is passed the object subclass1. If you follow the arrows, the call to b.f1 now leads to DC::fn1. The call to b.f2 leads to Base::fn2 for both the object base and the object subclass1. So virtual functions introduce a little "overhead", but not much.

Abstract Classes and Pure Virtual Functions

Many classes resemble class Employee in that they are useful as themselves and also as bases for derived classes. For such classes, the techniques described in the previous section suffice. However, not all classes follow that pattern. Some classes represent abstract concepts for which objects cannot exist. For example, one can observe the different species of warm–blooded, baby–bearing animals and conclude that there is a concept mammal. You can derive classes from mammal, such as canine, feline, and hominid. However, it is impossible to find anywhere on earth a pure mammal, that is, a mammal that isn't a member of some species. Mammal is an abstract concept. Another example is the class Shape. A Shape makes sense only as the base of some class derived from it. It doesn't make sense to define functions that rotate or draw a Shape object, just like there are no photos of pure mammals. So we declare the virtual functions of class Shape to be pure virtual functions. A virtual function is "made pure" by setting it equal to zero:

```
class Shape{                 //abstract class
 public:
    virtual void rotate(int) = 0;      //pure virtual function
    virtual void draw() = 0;           //pure virtual function
    virtual bool is_closed() = 0;      //pure virtual function
 //...
};
```

A class with one or more pure virtual functions is an *abstract class*, and no objects of that abstract class can be created:

```
 Shape s;             //error: can't create object of abstract class Shape
```

But it's ok to create a pointer or a reference to an abstract class

```
 func1(Shape *s);      //legal
 func2(Shape &s);      //legal
```

An abstract class can be used only as an interface and as a base for other classes. For example:

```
class Point{ ... };

class Circle: public Shape {
  public:
     void rotate(int){  }            //overrides Shape::rotate
```

```
        void draw();                     //overrides Shape::draw
        bool is_closed(){return true;}   //overrides Shape::is_closed

        Circle(Point p, float r);
     private:
        float radius;
        Point center;
    };
```

A pure virtual function is a placeholder in the base class for the derived class to overload with its own implementation. Without that placeholder in the base class, there is no overloading. A pure virtual function that is not defined in a derived class remains a pure virtual function, so the derived class is also an abstract class. This allows us to build implementations in stages:

```
    class Polygon: public Shape {        //abstract class
      public:
        bool is_closed(){return true;}   //overrides Shape::is_closed
        //... draw and rotate not overridden...
    };

    Polygon b;   //error: declaration of an object of abstract class Polygon

    class Square: public Polygon {
      public:
        void draw();                     //overrides Shape::draw
        void rotate(int);                //overrides Shape::rotate
        //....
    };

    Square s;   //fine (assume some suitable constructor)
```

An important use of abstract classes is to provide an interface without exposing any implementation details. For example, an operating system might hide the details of its device drivers behind an abstract class:

```
   class Device {                        //abstract class
     public:
        virtual int open(int opt) = 0;
        virtual int close(int opt) = 0;
        virtual int read(char* p, int n) = 0;
        virtual int write(const char* p, int n) = 0;
        virtual ~Device() {};          //virtual destructor
   };
```

We can then specify drivers as classes derived from `Device`, and manipulate a variety of drivers through that interface.

Employee Example of Virtual Functions

One can describe a collection of objects using an array of base class pointers. One wants to use pointers if the objects have different sizes. Each pointer is constructed with `new`, which sets aside the appropriate amount of memory for that particular object. In other words, we can make an array of different types of objects if they are all derived from the same base class by making an array of pointers to the base class. For example, suppose we want to make a data base of all employees. There are different kinds of employees and each type takes up a different amount of memory.

```
class EmployeeList : public Array<Employee *>
    {
    ...
    };

EmployeeList myDept(30);

WageEmployee *wagePtr;
SalesPerson *salePtr;
Manager *mgrPtr;

wagePtr = new WageEmployee("Bill Shapiro");
myDept[0] = wagePtr;

salePtr = new SalesPerson("John Smith");
myDept[1] = salePtr;

myDept[2] = new Manager("Mary Brown");

for(int i=0; i < myDept.numElts(); i++)
    cout << myDept[i]->getName() << endl;

//Now try:

for(int i=0; i < myDept.numElts(); i++)
    cout << myDept[i]->computePay() << endl;
// error, computePay not a member of Employee
```

If Employee did have a computePay, it would not be the right one: we need a different computePay for each employee type.

Solution: Virtual functions

```
class Employee                              //abstract class
```

```
    {
public:
    Employee(const char* nm);
    char *getName() const;
    virtual float computePay() const=0;   //placeholder fcn that is never called
    virtual ~Employee() {}
private:
    char name[30];
    };
```

Now make sure WageEmployee, SalesPerson, Manager, each have the line:

```
    float computePay() const; // implicitly virtual
```

Now to use it:

```
Employee *empPtr;
float salary;

empPtr = &aWorker;
salary = empPtr->computePay(); // call WageEmployee::computePay

empPtr = &aSeller;
salary = empPtr->computePay(); // call SalesPerson::computePay

empPtr = &aBoss;
salary = empPtr->computePay(); // call Manager::computePay

//Or use our array of pointers:

for(int i=0; i < myDept.numElts(); i++)
    cout << myDept[i]->computePay() << endl;

// No problem; this works fine.
```