

Inheritance, Polymorphism, & Dynamic Allocation

In order for a programming language to be considered **Object-Oriented**, it must include the following features:

- | | |
|-----------------------|---------------------------------|
| 1) Class Construction | (covered in preceding lectures) |
| 2) Inheritance | } (covered in this lecture) |
| 3) Polymorphism | |

Class Inheritance

Definition: **Inheritance** occurs when a newly constructed class receives the properties of an existing class.

Example:

Suppose we have a **circle** class that has member functions for returning the following:

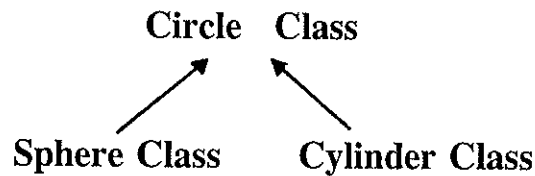
radius
diameter
circumference
area

We could create another class for **sphere** which would inherit all the data and function members of the circle class with the following modifications:

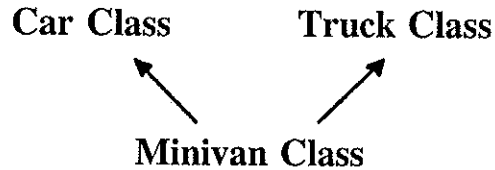
- 1) Since all the functions except **area** are equivalent, we would redefine the sphere's area function to be $4\pi r^2$.
- 2) Since there is an additional property of volume, we would add that function to the sphere class.

The initial class used as the basis for the derived class is called the **base class**, or **parent class**, or **super class**. The derived class is referred to as the **derived class**, or **child class**, or **sub class**.

Simple inheritance is where each derived class has only one immediate base class:



Multiple inheritance is when a derived class has two or more base classes:



These class derivation illustrations are called **class hierarchies**. The lines of derivation always point back to the parent class.

The general syntax for a derived class is:

`class derived_class_name : class_access base_class_name`

Example: If **circle** is an existing class, a new class named **cylinder** can be derived as follows:

```
class cylinder : public circle
{
    // add any additional data members
    // and function members
};
```

The only thing new here is the **class access specifier**.

Access Control

C++ defines 3 different **access control specifiers**.

- 1) Private
- 2) Protected
- 3) Public

These specifiers are used in 2 places:

- 1) To specify how data members are accessed.
- 2) To specify how a sub class is going to transmit the access to those base members in its own class.

1) Access Specifier for Data Members

Example:

```
class Base_name;
{
    private:
        int a;
    protected:
        int b;
    public:
        int c;
};
```

These access specifiers may occur in any order.

This data member is accessible only to this class. (points to `private: int a;`)

This is accessible to this class and to all sub classes that may be derived from it. (points to `protected: int b;`)

This is accessible to all. (points to `public: int c;`)

2) Access Specifiers in Derived Classes

The **class access specifier** in a derived class simply defines the type of inheritance it is willing to take on for the 3 accessible ways data members may be specified in the base class. Since there are 3 ways that data members may be accessed, and 3 ways for specifying a sub class's access, there are $3 \times 3 = 9$ possibilities which are summarized in the table below:

For class inheritance, these are the most commonly used set of access specifiers for data members.

... most common for function members.

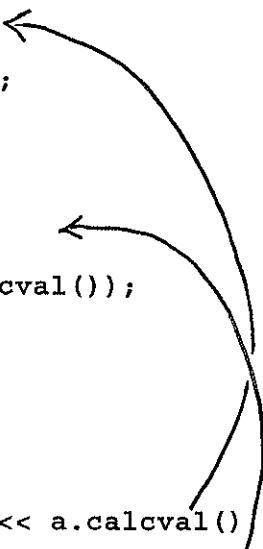
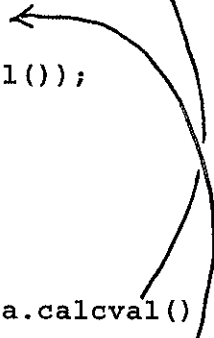
Access Specifier for a Data Member in the Base Class	Class Access Specifier in the Derived Class	How the Access to that Base Data Member is transmitted to the Derived Class
private	private	inaccessible
protected	private	private
public	private	private
private	public	inaccessible
protected	public	protected
public	public	public
private	protected	inaccessible
protected	protected	protected
public	protected	protected

When you develop a class, if you ever expect it to become a parent class, it is better to define the access to its data members as **protected** rather than **private**.

Protected access behaves identically to **private** access in that it only permits member or friend function access, but it permits this restriction to be inherited by any derived class — whereas a private base member is inaccessible to all derived classes.

Polymorphism

Polymorphism allows a base class function and a derived class function to have the same name. When that function name is encountered, the one that is executed is determined by what type the object argument is.

```
...  
double Circle::calcval(void)   
{  
    return(PI * radius * radius);  
}  
...  
double Cylinder::calcval(void)   
{  
    return (length * Circle::calcval());  
}  
...  
  
    Circle    a(1);  
    Cylinder  b(3,4);  
  
    cout << "The area of a is " << a.calcval() << endl;  
    cout << "The volume of b is " << b.calcval() << endl;  
...  

```

However there are some cases in which this method of overriding will not work.

```
#include <iostream.h>
#include <math.h>
```

```
class One ← Base class
{
```

```
    protected:
        float x;
    public:
        One(float = 2.0);
        float f1(float);
        float f2(float);
};
```

```
One::One(float val)
{
    x = val;
}
```

```
float One::f1(float num)
{
    return(num/2);
}
```

```
float One::f2(float num)
{
    return( pow(f1(num),2) );
}
```

f2 calls f1 - but which one?

```
class Two : public One ← Derived class
{
```

```
    public:
        float f1(float);
};
```

```
float Two::f1(float num) ← This overrides the f1 function
{
    return(num/3);
    in the base class.
}
```

```
int main()
{
    One a; // a is an object of the base class
    Two b; // b is an object of the derived class

    cout << "The computed value using a base class object call is "
          << a.f2(12) << endl;

    cout << "The computed value using a derived class object call is "
          << b.f2(12) << endl;

    return 0;
}
```

These answers should differ.

Output

The computed value using a base class object call is 36
The computed value using a derived class object call is 36



We would have liked the derived class object to have referenced the **f1** function in the derived class. The reason why the **f1** function of the base class was called, is that its location was determined at compile time. This is the usual **static binding**.

In place of static binding, we would like to determine at run time which function to call based on the object argument. This is called **dynamic binding**, and is accomplished in C++ by declaring the function to be **virtual**.

A **virtual function** specification tells the compiler to create a pointer to a function, but not fill in the value of the pointer until the function is actually called. Then, at run time, and based on the object making the call, the appropriate function address is used.

Example of Virtual Functions:

```
#include <iostream.h>
#include <math.h>
```

```
class One
```

← Base class

```
{
    protected:
        float x;
    public:
        One(float = 2.0);
        virtual float f1(float); ← f1 is declared virtual
        float f2(float);
};
```

```
One::One(float val)
```

```
{
    x = val;
}
```

```
float One::f1(float num)
```

```
{
    return(num/2);
}
```

```
float One::f2(float num)
```

```
{
    return( pow(f1(num),2) );
}
```

```
class Two : public One
```

← Derived class

```
{
    public:
        virtual float f1(float); ← f1 is declared virtual
};
```

```
float Two::f1(float num)
```

```
{
    return(num/3);
}
```

```
int main()
```

```
{
    One a; // a is an object of the base class
    Two b; // b is an object of the derived class
```

```
    cout << "The computed value using a base class object call is "
          << a.f2(12) << endl;
```

```
    cout << "The computed value using a derived class object call is "
          << b.f2(12) << endl;
```

```
    return 0;
```

```
}
```

Now the answers are correct.

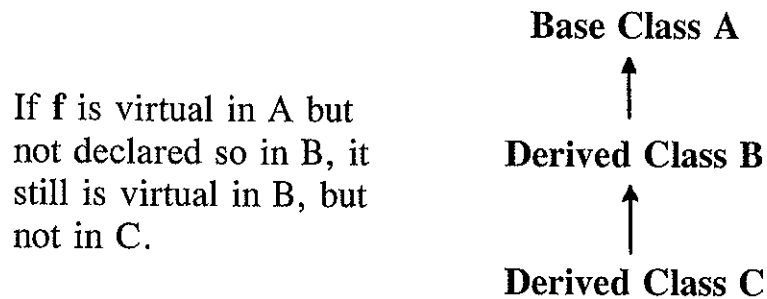
Output

The computed value using a base class object call is 36
The computed value using a derived class object call is 16



Note:

Once a function is declared as virtual, it remains virtual for the next derived class, with or without a virtual declaration in the derived class. However one should always use the word **virtual** for clarity and to ensure that any further derived classes also correctly inherit the function.



The other requirement is that once a function is declared virtual, the return type and parameter list of all subsequent derived class override versions must be the same.

Example of compile time error message if no. of parameters differ:

```
...
virtual float f1(float);
...
float One::f1(float num)
{
    return(num/2);
}
...
virtual float f1(float, float);
...
float Two::f1(float num, float extra)
{
    return(num/3);
}
...
```

Handwritten annotations:

- Base class: *1 parameter in f1* (points to `float num`)
- Derived class: *2 parameters in the override of f1.* (points to `float, float`)

Error: Too many types in declaration

Error: 'Two::f1(float, float)' is not a member of 'Two'

Compile time
Error Messages

Example of compile time error message if return types differ:

...

```
virtual float f1(float);
```

...

```
float One::f1(float num)
{
    return(num/2);
}
```

...

```
virtual int f1(float);
```

...

```
int Two::f1(float num)
{
    return(num/3);
}
```

...

← Return type
is float

Base class

← Return type
is int

Derived class

Error: Virtual function 'Two::f1(float)' conflicts with base class 'One'

Compile Time Error Message.