Inheritance

Introduction

Inheritance enables code reuse. It allows a *derived class* (*child class* or *subclass*) to inherit properties and behaviors from a *base class* (*parent class* or *superclass*), forming an '`is-a`' relationship. This promotes hierarchical organization and reduces code duplication.

Derived Class

Any class can inherit or be inherited by other classes. To define a derived class, use the colon (`:`) operator followed by a list of base classes, each preceded by an access specifier.

`class` *identifier* `:` *inheritance-list* `{` *body* `};`

where each base class of the *inheritance-list* is specified in the format:

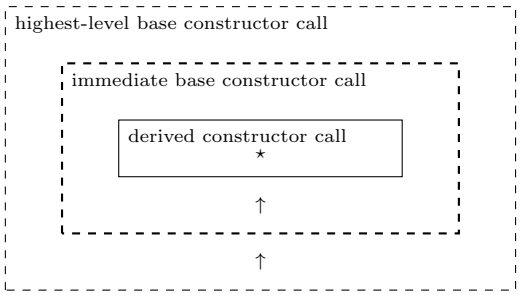
access-specifier class

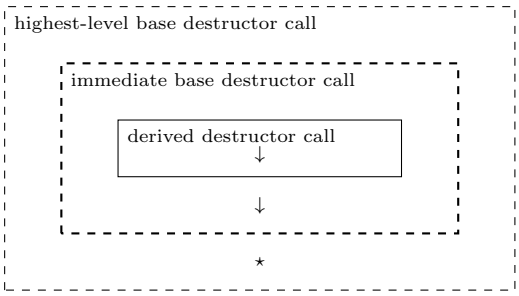
The access specifier determines how base class members are seen through the derived class.

Base Member Access In Derived Class			
Access Specifier	**Public Member**	**Protected Member**	**Private Member**
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	`private`	`private`	`private`

In addition, a derived class has direct access to the public and protected members of its base class, regardless of how it inherits the base class. It does not inherit private members, special member functions, or friends of the base class. Additionally, its objects can be used in place of its base class pointers or references.

Constructor & Destructor Hierarchy





A derived class is layered over its base class, but cannot initialize base class fields directly. When a derived class object is instantiated, the base class constructor is called first, and the derived class constructor is called afterward.

Although a derived class constructor normally invokes the base class default constructor, it can explicitly invoke another base class constructor using its member initialization list.

Furthermore, the destructors work in reverse order when a derived class object is destroyed. The derived class destructor is called first, and the base class constructor is called afterward.

Base Invocation & Upcasting

Base class members can be explicitly invoked in the derived class using the scope resolution operator (`::`) as follows

base-class`::`*member*

It is used to (1) change base class member accessibility within a derived class using '`using`' keyword:

```
using base-class::member-declaration;
```

and (2) invoke a base class method inside a derived class method; however, constructors cannot be invoked this way; they are only accessible through member initialization lists.

Last, when a base class object is required, but a derived class object is provided, *upcasting* occurs. To explicitly upcast (or *downcast*) use the function:

```
static_cast<data-type>(object)
```

Explicit upcasting is usually necessary whenever a derived class inherits multiple classes.

Example:

```
//Implicit Upcasting
class Derived : public Base
{
  public:
  Derived& operator=(const Derived& rhs)
  {
    if(this != &rhs){Base::operator=(rhs);}
    return *this;
  }
};
```

```
//Explicit Upcasting
class Derived : public Base
{
  public:
  Derived& operator=(const Derived& rhs)
  {
    if(this != &rhs){static_cast<Base&>(*this) = rhs;}
    return *this;
  }
};
```