Lecture 11: Inheritance

Introduction

Inheritance is the third fundamental concept in object-oriented programming that 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 inherited in 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 only to the public and protected members of its base class; it does not inherit private members, special member functions, or friends of the base class, and its objects can be used in place of base class pointers or references.

Example:

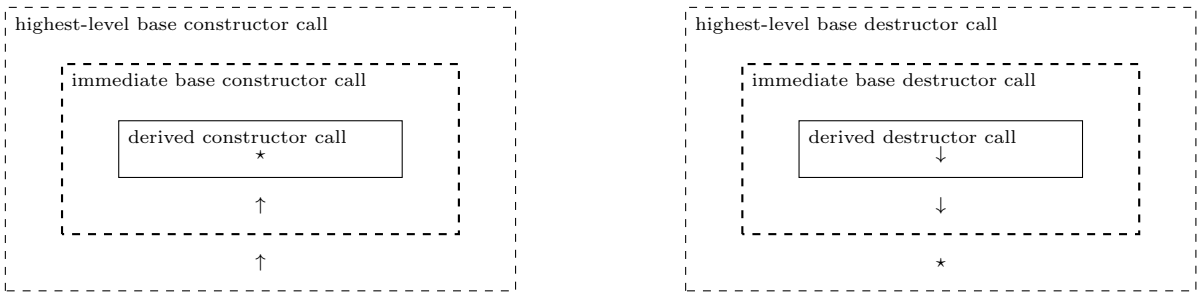
```
class Parent
{
  public:
  void msgA() {std::cout << "CA";}
  protected:
  void msgB() {std::cout << "CB";}
  private:
  void msgC() {std::cout << "CC";}
};

//Derived Classes
class A : public Parent
{
  //msgA() is public (accessible)
  //msgB() is protected (accessible)
  //msgC() is private (inaccessible)
};

class B : protected Parent
{
  //msgA() and msgB() are protected (accessible)
  //msgC() is private (inaccessible)
};

class C : private Parent
{
  //msgA() and msgB() are private (accessible)
  //msgC() is private (inaccessible)
};
```

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.

Example:

```
class Base
{
  public:
  Base() {std::cout << "base default";}
  Base(std::string a) {std::cout << "base other";}
  ~Base() {std::cout << "base destroyed";}
};

class Derived : public Base
{
  public:
  Derived() : Base("other") {std::cout << "derived default";}
  Derived(int a) {std::cout << "derived other";}
  ~Derived() {std::cout << "derived destroyed";}
};

int main()
{
  Derived a, b(3);
  return 0;
}
```

Output: `"base other derived default base default derived other derived destroyed base destroyed derived destroyed base destroyed"`.

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:

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

Method Overriding

Inheritance allows a process called *function overriding* in which a derived class can redefine its base class methods. A derived class method overrides a base class method if their function signatures (function headers) are the same [or differ only by *covariant return types* (class pointer or reference)]. Afterward, derived class objects will use their overridden version of the method.

Example:

```
class Animal
{
  public:
  std::string speak() const {return "silence";}
};

class Cat : public Animal
{
  public:
  //overridden method
  std::string speak() const {return "meow";}
};

int main()
{
  Cat a;
  std::cout << a.speak() << "\n"; //meow
  return 0;
}
```