Encapsulation & Inheritance with C++

Lecture 12-1

Work hard, be kind,
and amazing things will happen.
Conan O'Brien

Encapsulation Recap: Goal

- Abstraction
  - Deal with complexity by defining public interfaces (abstractions) to interact with an object.
  - Hide all the unnecessary details under the hood of these abstractions.

- Defensive Programming
  - Protect data from misuse by the outside world.
  - Hide the sources of the changes.

Encapsulation Recap: HowTo

- Encapsulation is implemented using **classes, access specifiers,** and **setters and getters.**
- A class encapsulates essential features of an object.

Access Specifiers in C++

- Public
  - Everyone can access public members.
- Protected
  - Protected members can be accessed inside the declared class and its derived classes.
- Private
  - Private members can only be accessed inside the class in which they are defined.

Encapsulation Example Using C++

```
class SneezePill {
public:
  void take() {cout<<"no more sneeze"<<endl;}
};

class RunnyNosePill {
public:
  void take() {cout<<"no more runny nose"<<endl;}
};

class HeadAchePill {
public:
  void take() {cout<<"no more headache"<<endl;}
};
```

Encapsulation Example Using C++

```
class ColdPill {
  SneezePill sPill;
  RunnyNosePill rPill;
  HeadAchePill hPill;
public:
  void take() {sPill.take();rPill.take();hPill.take();}
};
class Patient {
public:
  void takeMedicine(ColdPill& cPill) {cPill.take();}
};
int main() {
  Patient coldPatient;
  ColdPill c;
  coldPatient.takeMedicine(c);
}
```

Output

```
no more sneeze
no more runny nose
no more headache
```

Inheritance in C++

- As in Java, a class can inherit another class to inherit its data and functions.
- Usually, parent class is called the base class, and the child class is called the derived class.

Basic Inheritance Syntax

```
class Base {
public:
  int baseInteger;
  Base() : baseInteger(0) { }
};

class Derived : public Base {
public:
  Derived() {
    cout << baseInteger << endl; // 0
  }
};
```

Derived Class Object Construction

```
class Base {
public:
  int baseInt;
  Base():baseInt(0) {cout << "Base()" << endl;}
  Base(int n):baseInt(n) {cout << "Base(n)" << endl;}
};
class Derived: public Base {
public:
  Derived() {cout << "Derived()" << endl;}
  Derived(int n): Base(n) {cout << "Derived(n)" << endl;}
};
int main() {
  Base b;
  Derived d;
  Derived d1(1);
};
```

Derived Class Object Construction

```
class Base {
public:
  int baseInt;
  Base():baseInt(0) {cout << "Base()" << endl;}
  Base(int n):baseInt(n) {cout << "Base(n)" << endl;}
};
class Derived: public Base {
public:
  Derived() {cout << "Derived()" << endl;}
  Derived(int n): Base(n) {cout << "Derived(n)" << endl;}
};
int main() {
  Base b;
  Derived d;
  Derived d1(1);
};
```

Default Base constructor is called

Output

```
Base()
Base()
Derived()
Base(n)
Derived(n)
```

Derived Class Object Construction

```
class Base {
public:
  int baseInt;
  Base():baseInt(0) {cout << "Base()" << endl;}
  Base(int n):baseInt(n) {cout << "Base(n)" << endl;}
};
class Derived: public Base {
public:
  Derived() {cout << "Derived()" << endl;}
  Derived(int n): Base(n) {cout << "Derived(n)" << endl;}
};
int main() {
  Base b;
  Derived d;
  Derived d1(1);
};
```

Default constructor of its base class is called

Output

```
Base()
Base()
Derived()
Base(n)
Derived(n)
```

Derived Class Object Construction

```
class Base {
public:
  int baseInt;
  Base():baseInt(0) {cout << "Base()" << endl;}
  Base(int n):baseInt(n) {cout << "Base(n)" << endl;}
};
class Derived: public Base {
public:
  Derived() {cout << "Derived()" << endl;}
  Derived(int n): Base(n) {cout << "Derived(n)" << endl;}
};
int main() {
  Base b;
  Derived d;
  Derived d1(1);
};
```

Its own constructor is called

Output

```
Base()
Base()
Derived()
Base(n)
Derived(n)
```

Derived Class Object Construction

```
class Base {
public:
  int baseInt;
  Base():baseInt(0) {cout << "Base()" << endl;}
  Base(int n):baseInt(n) {cout << "Base(n)" << endl;}
};
class Derived: public Base {
public:
  Derived() {cout << "Derived()" << endl;}
  Derived(int n): Base(n) {cout << "Derived(n)" << endl;}
};
int main() {
  Base b;
  Derived d;
  Derived d1(1);
};
```

Constructor Base(n)
is called

Output

```
Base()
Base()
Derived()
Base(n)
Derived(n)
```

Derived Class Object Construction

```
class Base {
public:
  int baseInt;
  Base():baseInt(0) {cout << "Base()" << endl;}
  Base(int n):baseInt(n) {cout << "Base(n)" << endl;}
};
class Derived: public Base {
public:
  Derived() {cout << "Derived()" << endl;}
  Derived(int n): Base(n) {cout << "Derived(n)" << endl;}
};
int main() {
  Base b;
  Derived d;
  Derived d1(1);
};
```

Its corresponding constructor is called

Output

```
Base()
Base()
Derived()
Base(n)
Derived(n)
```

Derived Class Object Destruction

```
class Base {
public:
  int baseInt;
  Base(int n):baseInt(n) {cout << "Base()" << endl;}
  ~Base() {cout << "~Base()" << endl;}
};
class Derived: public Base {
public:
  Derived(int n): Base(n) {cout << "Derived()" << endl;}
  ~Derived() {cout << "~Derived()" << endl;}
};
int main() {
  Base b;
  Derived d(1);
};
```

Derived Class Object Destruction

```
class Base {
public:
  int baseInt;
  Base(int n):baseInt(n) {cout << "Base()" << endl;}
  ~Base() {cout << "~Base()" << endl;}
};
class Derived: public Base {
public:
  Derived(int n): Base(n) {cout << "Derived()" << endl;}
  ~Derived() {cout << "~Derived()" << endl;}
};
int main() {
  Base b;
  Derived d(1);
};
```

Constructors are called same as before

Output

```
Base()
Base()
Derived()
~Derived()
~Base()
~Base()
```

Derived Class Object Destruction

```
class Base {
public:
  int baseInt;
  Base(int n):baseInt(n) {cout << "Base()" << endl;}
  ~Base() {cout << "~Base()" << endl;}
};
class Derived: public Base {
public:
  Derived(int n): Base(n) {cout << "Derived()" << endl;}
  ~Derived() {cout << "~Derived()" << endl;}
};
int main() {
  Base b;
  Derived d(1);
};
```

Destructor for the Derived class is called first

Output

```
Base()
Base()
Derived()
~Derived()
~Base()
~Base()
```

Derived Class Object Destruction

```
class Base {
public:
  int baseInt;
  Base(int n):baseInt(n) {cout << "Base()" << endl;}
  ~Base() {cout << "~Base()" << endl;}
};
class Derived: public Base {
public:
  Derived(int n): Base(n) {cout << "Derived()" << endl;}
  ~Derived() {cout << "~Derived()" << endl;}
};
int main() {
  Base b;
  Derived d(1);
};
```

Destructor for the derived class's base class is then called

Output

```
Base()
Base()
Derived()
~Derived()
~Base()
~Base()
```

Derived Class Object Destruction

```
class Base {
public:
  int baseInt;
  Base(int n):baseInt(n) {cout << "Base()" << endl;}
  ~Base() {cout << "~Base()" << endl;}
};
class Derived: public Base {
public:
  Derived(int n): Base(n) {cout << "Derived()" << endl;}
  ~Derived() {cout << "~Derived()" << endl;}
};
int main() {
  Base b;
  Derived d(1);
};
```

Destructor for the Base class is called

Output

```
Base()
Base()
Derived()
~Derived()
~Base()
~Base()
```

Different Inheritance Types

- There are three different inheritance types in C++ according to the access specifier: `public`, `protected`, and `private`.
- If a class is inherited with a specifier X, all the members of the base class whose access scope is larger than X is inherited as X.
- If you do not specify the access specifier, it will be `private` inheritance by default.

Different Inheritance Types

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Public Inheritance

- Since there are no access specifier whose scope is larger than public, public inheritance results in the members remaining as is in the derived class.
- This is the most commonly used type of inheritance.

Public Inheritance

```
class Base {
public:
  int publicInt;
protected:
  int protectedInt;
private:
  int privateInt;
};
class Derived: protected Base {
public:
  void printInt(){
    this->publicInt; // ok
    this->protectedInt; // ok
    this->privateInt; // forbidden; private
  }
};
```

```
int main() {
  Derived d;
  d.publicInt; // ok; public
  d.protectedInt; // ok; public
  d.privateInt; // forbidden; private
}
```

Protected Inheritance

- Protected inheritance results in all the public members of the base class becoming protected inside the derived class.

Protected Inheritance

```
class Base {
public:
  int publicInt;
protected:
  int protectedInt;
private:
  int privateInt;
};
class Derived: protected Base {
public:
  void printInt(){
    this->publicInt; // ok
    this->protectedInt; // ok
    this->privateInt; // forbidden; private
  }
};
```

```
int main() {
  Derived d;
  d.publicInt; // forbidden; protected
  d.protectedInt; // forbidden; protected
  d.privateInt; // forbidden; private
}
```

Private Inheritance

- Private inheritance results in all the public and protected members of the base class becoming private in the derived class.
- Of course, private members of the base class are not visible to the derived class.

Private Inheritance

```
class Base {
public:
  int publicInt;
protected:
  int protectedInt;
private:
  int privateInt;
};
class Derived: protected Base {
public:
  void printInt(){
    this->publicInt; // ok
    this->protectedInt; // ok
    this->privateInt; // forbidden; private
  }
};
```

```
int main() {
  Derived d;
  d.publicInt; // forbidden; private
  d.protectedInt; // forbidden; private
  d.privateInt; // forbidden; private
}
```

Multiple Inheritance

- In C++, a class can inherit multiple base classes at once.
- Similar to implementing several interfaces in Java, but implementation of the base classes exist.

Multiple Inheritance Syntax

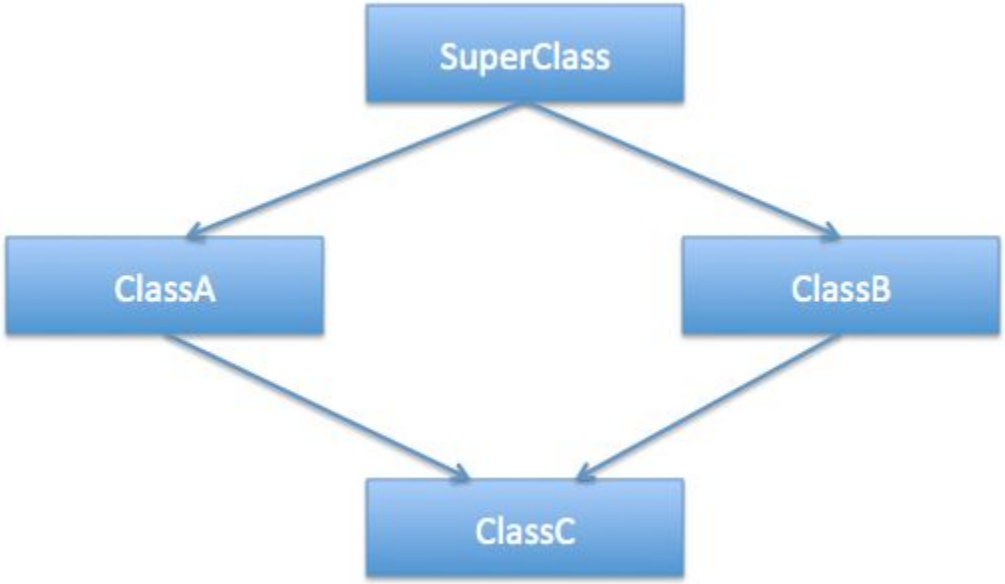
- Multiple classes can be inherited using commas.
- We can select which type of inheritance a derived class should go by for each base class.

Syntax

```
class BaseOne {...};
class BaseTwo {...};
class Derived: public BaseOne, protected BaseTwo {
};
```

The Diamond Problem

- Compiler raises an error if there is a member that has the same name over different base classes.



The Diamond Problem Example

```
class SuperClass {
public:
  void func() {}
};

class ClassA:public SuperClass {

};

class ClassB:public SuperClass {

};
```

```
class ClassC: public
  ClassA,public ClassB {
public:
  void doubleFunc() {
    func(); // ? error
    func(); // ? error
  }
};
```

ClassC is indirectly inheriting the SuperClass twice through ClassA and ClassB

The Diamond Problem Example

```
class SuperClass {
public:
  void func() {}
};

class ClassA:public SuperClass {

};

class ClassB:public SuperClass {

};
```

```
class ClassC: public
  ClassA,public ClassB {
public:
  void doubleFunc() {
    ClassA::func();
    ClassB::func();
  }
};
```

The problem can be addressed by specifying from which class to call the function from.

Virtual Inheritance

- We can also avoid the above error by using what's called the *virtual inheritance.*
- Using the `virtual` keyword results in only one instance of overlapping `func()` function.
- The `func()` function belonging to the SuperClass is used.
  - If func() does not exist in SuperClass, there will be a compilation error since which func() to call is still ambiguous.

Virtual Inheritance

```
class SuperClass {
public:
  void func() {cout<<"SuperClass"<<endl;}
};

class ClassA:virtual public SuperClass {

};

class ClassB:virtual public SuperClass {

};
```

Virtual Inheritance

```
class ClassC: public ClassA,public ClassB {
public:
  void doubleFunc() {
    func();
    func();
  }
};
int main() {
  ClassC c;
  c.doubleFunc();
}
```

Output

```
SuperClass
SuperClass
```

Virtual Functions

- Virtual function is a member function in the base class that you redefine in the derived class.
- It tells the compiler to dynamically bind the function.
- This could be understood as C++’s way of implementing function overriding as in Java.
- It is declared using the `virtual` keyword.

Virtual Function Rules

- Cannot be static members.
- Are accessed through object pointers.
- Must be defined in the base class.
- Need to have identical prototypes in both the base and the derived class.
  - same name and different prototype → compiler considers them as overloaded functions

Example without Virtual Function

```
class A {
public:
    void test() { cout << "A" << endl; }
};
class B: public A {
public:
    void test() {cout << "B" << endl; }
};
int main() {
    A a;
    a.test(); // A
    B b;
    b.test(); // B
    A* aptr = &b;
    aptr->test(); // A
}
```

Example with Virtual Function

```
class A {
public:
    virtual void test() { cout << "A" << endl; }
};
class B: public A {
public:
    void test() {cout << "B" << endl; }
};
int main() {
    A a;
    a.test(); // A
    B b;
    b.test(); // B
    A* aptr = &b;
    aptr->test(); // B
}
```

Pure Virtual Functions

- Pure virtual function is a virtual function whose implementation is not given.
- We can declare a pure virtual function by assigning 0 in the declaration.

```
class A {
public:
    virtual void pureVirtualFunc() = 0;
};
```

Abstract Class

- An abstract class is a class with **at least one pure virtual function** as a member.
- It can have normal functions along with pure virtual functions.
- **It cannot be instantiated**, but pointers and references of it can be created.
- The derived class of an abstract class **must implement all the pure virtual functions**, or else it will become abstract too.

Virtual Destructors

- Deleting a derived class object with a pointer to the base class whose destructor is not virtual results in an undefined behavior.
- All the destructors in the inheritance chain should be called in the process of destruction of an object.

Virtual Destructor Example

```
class First {
public:
    First() {}
    ~First() {cout<<"~First()"<<endl;}
};
class Second : public First {
public:
    Second() {}
    ~Second() {cout<<"~Second()"<<endl;}
};
int main() {
    First* fp = new Second();
    delete fp;
}
```

Destructor of Second not called. Memory leak!

Output

```
~First()
```

Virtual Destructor Example

```
class First {
public:
    First() {}
    virtual ~First() {cout<<"~First()"<<endl;}
};
class Second : public First {
public:
    Second() {}
    ~Second() {cout<<"~Second()"<<endl;}
};
int main() {
    First* fp = new Second();
    delete fp;
}
```

Output

```
~Second()
~First()
```