Programming in Modern C++

Module M25: Inheritance: Part 5: `private` & `protected` Inheritance

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Using the Phone Hierarchy as an example analyzed the design process with inheritance

Module Objectives

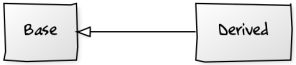
- Explore restricted forms of inheritance (`private` and `protected`) in C++ and their semantic implications

Module Outline

Inheritance in C++

Inheritance in C++: Semantics

- Derived **ISA** Base




```
class Base;                     // Base Class = Base
class Derived: public Base;     // Derived Class = Derived
```

- Use keyword `public` after class name to denote inheritance
- Name of the Base class follow the keyword

Inheritance Exercise: What is the output?


```
class B {
public:
    B()  { cout << "B "; }
    ~B() { cout << "~B "; }
};

class C {
public:
    C()  { cout << "C "; }
    ~C() { cout << "~C "; }
};

class D : public B {
    C data_; // Embedded Object
public:
    D()  { cout << "D " << endl; } // Intrinsic Base Class Object
    ~D() { cout << "~D "; }
};

int main() {
    D d;
}
```

Inheritance Exercise: What is the output?


```
class B {
public:
    B()  { cout << "B "; }
    ~B() { cout << "~B "; }
};
class C {
public:
    C()  { cout << "C "; }
    ~C() { cout << "~C "; }
};
class D : public B {
    C data_; // Embedded Object
public:
    D()  { cout << "D " << endl; } // Intrinsic Base Class Object
    ~D() { cout << "~D "; }
};
int main() {
    D d;
}
```

Output:

```
B C D       // First base class object, then embedded object, finally self
~D ~C ~B
```

private Inheritance

private Inheritance

- private Inheritance
  - Definition

    ```
    class Base;
    class Derived: private Base;
    ```

  - Use keyword private after class name
  - Name of the Base class follow the keyword
  - private inheritance does not mean generalization / specialization

private Inheritance

public Inheritance

```
class Person { ... };

class Student:
    public Person { ... };

void eat(const Person& p);     // anyone can eat
void study(const Student& s); // only students study

Person p;  // p is a Person
Student s; // s is a Student

eat(p);    // fine, p is a Person
eat(s);    // fine, s is a Student,
           // and a Student is-a Person

study(s);  // fine
study(p);  // error! p isn't a Student
```

- Compilers *converts* a derived class object (Student) into a base class object (Person) if the inheritance relationship is public

private Inheritance

```
class Person { ... };

class Student: // inheritance is now private
    private Person { ... };

void eat(const Person& p);     // anyone can eat
void study(const Student& s); // only students study

Person p;  // p is a Person
Student s; // s is a Student

eat(p);    // fine, p is a Person
eat(s);    // error! a Student isn't a Person
```

- Compilers *will not convert* a derived class object (Student) into a base class object (Person) if the inheritance relationship is private

Uncopyable Class

- Suppose we want to design a `MyClass` every object must be *unique*. That is instance objects must not be copied (the class needs to be Uncopyable)

```
MyClass m1;
MyClass m2;

MyClass m3(m1); // attempt to copy m1 - should not compile!
m1 = m2;        // attempt to copy m2 - should not compile!
```

- Naturally, we do not want to provide copy constructor or copy assignments operator. But that does not work as the compiler will provide free versions of these functions. How to stop that?

```
class MyClass {
public:
...
private:
...
    MyClass(const MyClass&); // declarations only
    MyClass& operator=(const MyClass&);
};
```

The last trick is not to provide the implementations (bodies) of these copy functions.

- With the above any global function or other class would not be able to copy - there will be *compilation error*; and any member function of `MyClass` will get *linker error* on missing implementation
- This is, of course, not an elegant solution and has to depend on the programmer to do things right for every such Uncopyable class. private inheritance helps out

Uncopyable

- `class Uncopyable` is designed as a root class that can make copy functions of any child class `private`

```
class Uncopyable { protected:  // allow construction and destruction of derived objects ...
        Uncopyable() { }
        ~Uncopyable() { }
    private:
        Uncopyable(const Uncopyable&); // ... but prevent copying
        Uncopyable& operator=(const Uncopyable&);
};
```

- Any class that inherits from `class Uncopyable`, will not have copy functionality:

```
class MyClass: private Uncopyable { // class no longer declares copy ctor or copy assign. operator
    // ...
    void ProhibitiveCopy() { MyClass test1, test2; // Member functions cannot perform copy
        MyClass test3(test1); // Error 1: 'Uncopyable::Uncopyable' : cannot access private member
        test2 = test1;        // Error 2: 'Uncopyable::operator =' : cannot access private member
    }
};
int main() { MyClass test1, test2; // Global functions cannot perform copy
    MyClass test3(test1); // Error 1: 'Uncopyable::Uncopyable' : cannot access private member
    test2 = test1;        // Error 2: 'Uncopyable::operator =' : cannot access private member
}
```

- **The inheritance from `Uncopyable` need not be `public` (though it will work), hence using `private` we express that it is purely for implementation - not for modeling ISA that actually does not exist**
- You may also use the `boost::noncopyable`
- **C++11 provides an explicit support by `delete` to stop compilers from providing free functions**

Car HAS–A Engine: Composition OR `private` Inheritance?

Simple Composition

```
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    void start() { } // Starts this Engine
};
class Car {
public:
     // Initializes this Car with 8 cylinders
    Car() : e_(8) { }

    // Start this Car by starting its Engine
    void start() { e_.start(); }
private:
    Engine e_;  // Car has-a Engine
};
int main() {
    Car c;

    c.start();
}
```

`private` **Inheritance**

```
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    void start() { } // Starts this Engine
};
class Car : private Engine {  // Car has-a Engine
public:
     // Initializes this Car with 8 cylinders
    Car() : Engine(8) { }

    // Start this Car by starting its Engine
    using Engine::start;


};
int main() {
    Car c;

    c.start();
}
```

private Inheritance

- For **HAS-A**: Use composition when you can, `private` inheritance when you have to

> **• Private inheritance means nothing during software design, only during software implementation**
>
> **• Private inheritance means is-implemented-in-terms of. It is usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions**
>
> – Scott Meyers in Item 32, Effective C++ (3rd. Edition)

protected Inheritance

protected Inheritance

- protected Inheritance
  - Definition

    ```
    class Base;
    class Derived: protected Base;
    ```

  - Use keyword protected after class name
  - Name of the Base class follow the keyword
  - protected inheritance does not mean generalization / specialization

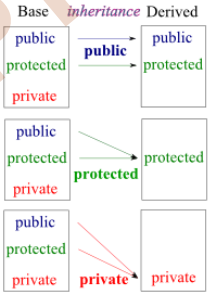
> • **Private inheritance means something entirely different (from public inheritance), and protected inheritance is something whose meaning eludes me to this day**
>
> – *Scott Meyers in Item 32, Effective C++ (3rd. Edition)*

Visibility

Visibility across Access and Inheritance

- Visibility Matrix

Visibility \ Inheritance	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private



Use and Examples

Inheritance Exercise: What is the output?


```
class B {
protected:
    B()  { cout << "B "; }
    ~B() { cout << "~B "; }
};
class C : public B {
protected:
    C()  { cout << "C "; }
    ~C() { cout << "~C "; }
};
class D : private C {
    C data_;
public:
    D()  { cout << "D " << endl; }
    ~D() { cout << "~D "; }
};

int main() {
    D d;
}
```

Inheritance Exercise: What is the output?

```
class B {
protected:
    B()  { cout << "B "; }
    ~B() { cout << "~B "; }
};
class C : public B {
protected:
    C()  { cout << "C "; }
    ~C() { cout << "~C "; }
};
class D : private C {
    C data_;
public:
    D()  { cout << "D " << endl; }
    ~D() { cout << "~D "; }
};

int main() {
    D d;
}
```

Output:

```
B C B C D
~D ~C ~B ~C ~B
```

Inheritance Exercise: Access Rights

Inaccessible Members

```
class A { private: int x;
    protected: int y;
    public: int z;
};
class B : public A { private: int u;
    protected: int v;
    public: int w; void f() { x; }
};
class C: protected A { private: int u;
    protected: int v;
    public: int w; void f() { x; }
};
class D: private A { private: int u;
    protected: int v;
    public: int w; void f() { x; }
};
class E : public B { public: void f() { x; u; }
};
class F : public C { public: void f() { x; u; }
};
class G : public D { public: void f() { x; y; z; u; }
};
```

Accessible Members

```
void f(A& a,
       B& b, C& c, D& d,
       E& e, F& f, G& g) {
    a.z;

    b.z;
    b.w;

    c.w;

    d.w;

    e.z;
    e.w;

    f.w;

    g.w;
}
```

Module Summary

- Introduced restricted forms of inheritance and `protected` specifier
- Discussed how `private` inheritance is used for *Implemented-As* Semantics