Programming in Modern C++

Module M23: Inheritance: Part 3: Constructor & Destructor: Object Lifetime

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

- Discussed the effect of inheritance on Data Members and Object Layout
- Discussed the effect of inheritance on Member Functions with special reference to Overriding and Overloading

Module Objectives

- Understand `protected` access specifier
- Understand the construction and destruction process on an object hierarchy
- Revisit Object Lifetime for a hierarchy

Module Outline

1. Inheritance in C++
2. `protected` Access
   - Streaming
3. Constructor & Destructor
4. Object Lifetime
5. Module Summary

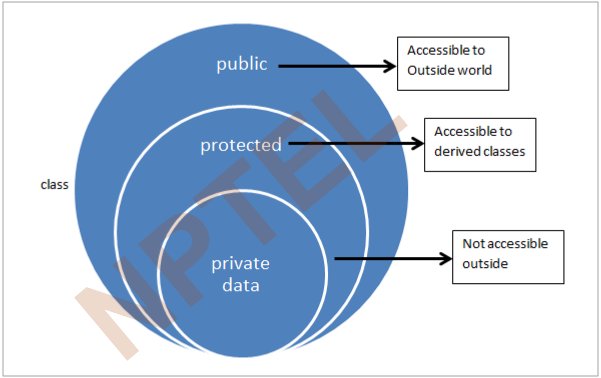
Inheritance in C++

Inheritance in C++

Inheritance in C++: Semantics

- Derived **ISA** Base
- **Data Members**
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- **Member Functions**
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
  - Derived class *may add* new member functions
- **Access Specification**
  - Derived class *cannot access* private members of Base class
  - Derived class *can access* protected members of Base class
- **Construction-Destruction**
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

protected Access



protected **Access**

Access Members of Base: `protected` Access

- Derived **ISA** Base
- **Access Specification**
  - Derived class *cannot access* private members of Base class
  - Derived class *can access* public members of Base class
- **protected Access Specification**
  - A new protected access specification is introduced for Base class
  - Derived class *can access* protected members of Base class
  - No other class or global function *can access* protected members of Base class
  - A protected member in Base class is like public in Derived class
  - A protected member in Base class is like private in other classes or global functions

Module M23

Partha Pratim Das

Objectives & Outlines

Inheritance in C++

protected Access

Streaming

Constructor & Destructor

Object Lifetime

Module Summary

protected Access

private Access

```
class B {
private:  // Inaccessible to child
          // Inaccessible to others
    int data_;
public: // ...
    void Print() { cout << "B Object: ";
        cout << data_ << endl;
    }
};
class D: public B { int info_; public: // ...
    void Print() { cout << "D Object: ";
        cout << data_ << ", "; // Inaccessible
        cout << info_ <<endl;
    }
};
B b(0);
D d(1, 2);

b.data_ = 5;  // Inaccessible to all

b.Print();
d.Print();
```

- D::Print() *cannot access* B::data_ as it is private

protected Access

```
class B {
protected: // Accessible to child
           // Inaccessible to others
    int data_;
public: // ...
    void Print() { cout << "B Object: ";
        cout<<data_<<endl;
    }
};
class D: public B { int info_; public: // ...
    void Print() { cout << "D Object: ";
        cout << data_ << ", "; // Accessible
        cout << info_ << endl;
    }
};
B b(0);
D d(1, 2);

b.data_ = 5;  // Inaccessible to others

b.Print();
d.Print();
```

- D::Print() *can access* B::data_ as it is protected

Why do we need protected access?

- **Handling Encapsulation**: Encapsulation, the first principle of OOAD, can be enforced in a single class by `private` and `public` access specifiers
  - `private` hides the state (data) of the object and `public` allows the service (method / interface) to be exposed
  - We fine-grain this by `get/set` paradigm to achieve effective information hiding
  - Further `friend` provides a way to sneak through encapsulation for *easy yet safe coding*
- **Encapsulation-Inheritance Conflict**: The above approach to Encapsulation conflicts with Inheritance, the second principle of OOAD
  What should be the access specification for data members of a Base class?
  - If they are `public`, the encapsulation is lost for the base class objects
  - If they are `private`, even the derived class methods cannot access them
  - So the derived class object contains the base class data members but cannot access them
    Notably, the state of the derived class object depends on the state of its base class part
  - The `get/set` paradigm does not work as it is clumsy and creates an encapsulation hole like `public` if used for all data members
- **Solution**: The `protected` access specifier provides a neat solution by making `protected` base class members available to the derived class while being hidden from the rest of the world
- **Caveat**: `protected` specifier still does not solve all situations and we need to use `friend` to provide a way to sneak through encapsulation as the next example illustrates

Streaming

Streaming in B

```
class B { protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) { os << "B Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    //friend ostream& operator<<(ostream& os,
    //    const D& d) { os << "D Object: ";
    //    os << d.data_ << ' ' << d.info_ << endl;
    //    return os;
    //}
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a B object
```

```
B Object: 0
B Object: 1
```

- d printed as a B object; info_ missing

Streaming in B & D

```
class B { protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) { os << "B Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    friend ostream& operator<<(ostream& os,
        const D& d) { os << "D Object: ";
        os << d.data_ << ' ' << d.info_ << endl;
        return os;
    }
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a D object
```

```
B Object: 0
D Object: 1 2
```

- d printed as a D object as expected

Constructor and Destructor

Constructor and Destructor

Constructor and Destructor

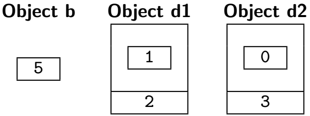
- Derived **ISA** Base
- **Constructor-Destructor**
  - Derived class *does not inherit* the Constructors and Destructor of Base class but *must have access to them*
  - Derived class *must provide* its own Constructors and Destructor
  - Derived class *cannot override* or *overload* a Constructor or the Destructor of Base class
- **Construction-Destruction**
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

Constructor and Destructor

```
class B { protected: int data_; public:
    B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
    ~B() { cout << "B::~B(): " << data_ << endl; }
    // ...
};
class D: public B { int info_; public:
    D(int d, int i) : B(d), info_(i) // ctor-1: Explicit construction of Base
    { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
    D(int i) : info_(i)              // ctor-2: Default construction of Base
    { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
    ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
    // ...
};

B b(5);
D d1(1, 2);    // ctor-1: Explicit construction of Base
D d2(3);       // ctor-2: Default construction of Base
```

Object Layout



Object Lifetime

Object Lifetime

Object Lifetime

```
class B { protected: int data_; public:
    B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
    ~B() { cout << "B::~B(): " << data_ << endl; }
    // ...
};
class D: public B { int info_; public:
    D(int d, int i) : B(d), info_(i) // ctor-1: Explicit construction of Base
    { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
    D(int i) : info_(i)              // ctor-2: Default construction of Base
    { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
    ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
    // ...
};
B b;
D d1(1, 2);    // ctor-1: Explicit construction of Base
D d2(3);       // ctor-2: Default construction of Base
```

Construction O/P

```
B::B(int): 0         // Object b
B::B(int): 1         // Object d1
D::D(int, int): 1, 2 // Object d1
B::B(int): 0         // Object d2
D::D(int): 0, 3      // Object d2
```

Destruction O/P

```
D::~D(): 0, 3      // Object d2
B::~B(): 0         // Object d2
D::~D(): 1, 2      // Object d1
B::~B(): 1         // Object d1
B::~B(): 0         // Object b
```

- **First construct base class object, then derived class object**
- **First destruct derived class object, then base class object**

Module Summary

- Understood the need and use of `protected` Access specifier
- Discussed the Construction and Destruction process of class hierarchy and related Object Lifetime