



## Lecture Seven Inheritance

Ref: Herbert Schildt, Teach Yourself C++, Third Ed<sup>n</sup> (Chapter 7)

© Dr. M. Mahfuzul Islam  
Professor, Dept. of CSE, BUET



## Basic Class Access Control

- The general form of Inheritance is:  

```
class derived-class name: access base-class name{
}
```
- There are three keywords for access: **public**, **private** and **protected**.
- Access is **optional**. If the access specifier is **not present**, it is **private** for a “class” and **public** for a “struct”.
- If the access specifier is **public**, all **public** members of the base class become **public** members of the derived class and all the **protected** member of the base class become **protected** member of the derived class.
- If the access specifier is **private**, all **public** members of the base class become **private** members of the derived class.
- If the access specifier is **protected**, all **protected** and **public** members of the base class become **protected** members of the derived class.
- A **private member** of base class cannot be accessed from a **derived class**.
- **Protected members** behaves like **private members** of the base class but can be **accessed** from the derived class.



## Basic Class Access Control

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

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
    int getx() { return x; }
};

class derived: public base {
    int y;
public:
    void sety(int n) { y = n; }
    void showy() {
        cout << y << '\n';
        cout << x+y << '\n'; //Error??
        cout << y+getx() << '\n'; //ok??
    }
};
```

```
int main(){
    derived ob;

    ob.setx(10);
    ob.sety(20);
    ob.showx();
    ob.showy();

    return 0;
}
```



## Basic Class Access Control

➤ If derived **inherits** base as **private**, this cause more error within main.

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

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
    int getx() { return x; }
};

class derived: private base {
    int y;
public:
    void sety(int n) { y = n; }
    void showy() {
        cout << y << '\n';
        cout << x+y << '\n'; //Error??
        cout << y+getx() << '\n'; //ok??
    }
};
```

```
int main(){
    derived ob;

    ob.setx(10); // Error ??
    ob.sety(20);
    ob.showx(); //Error??
    ob.showy();

    return 0;
}
```



## Basic Class Access Control

➤ **Protected members** behaves like private members of the base class but can be accessed from the derived class.

➤ If the **access specifier** is **public**, all **public members** of the base class become **public members** of the derived class and all the **protected member** of the base class become **protected member** of the derived class.

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

class base {
    int a;
protected:
    int b;
public:
    int c;
    void setab(int n, int m) { a = n; b = m; }
};

class derived: public base {
public:
    void setc(int n) { c = n; }
    void showabc();
};
```

```
void derived::showabc(){
    cout << a <<" "<< b <<" "<< c<<"\n";
} // Error - a cannot be accessed
//      - b can be accessed
//      - c can be accessed
```

```
int main(){
    derived ob;

    ob.setab(10, 20);    // ok ??
    ob.setc(20);
    ob.showabc();

    return 0;
}
```



## Basic Class Access Control

➤ If the **access specifier** is **protected**, all **protected** and **public members** of the base class become **protected members** of the derived class..

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

class base {
    int a;
protected:
    int b;
public:
    int c;
    void setab(int n, int m) { a = n; b = m; }
};

class derived: protected base {
public:
    void setc(int n) { c = n; }
    void showabc();
};
```

```
void derived::showabc(){
    cout << a <<" "<< b <<" "<< c<<"\n";
} // Error - a cannot be accessed
//      - b can be accessed
//      - c can be accessed
```

```
int main(){
    derived ob;

    ob.setab(10, 20);    // Error
    ob.setc(20);
    ob.showabc();

    return 0;
}
```



## Constructors, Destructors and Inheritance

- Constructors are executed in order of derivation (i. e., **base class constructor is executed first, then derived class constructor is executed** ) and destructors are executed in reverse order.
- For **passing arguments** to the constructor of base class, a **chain** of argument passing is necessary from **derived class to base class**.
- The general form of argument passing from derived class to base class is as follows:  
**Derived-constructor(arg-list): base(arg-list){**  
**}**
- **Same arguments** can be used in both base class and derived class.
- Derived class may simply **pass the argument** to the base class.



## Constructors, Destructors and Inheritance

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

class base {
    int x, y;
public:
    base(int n, int p){
        x = n;
        y = p;
        cout << "Constructing base class\n";
    }
    ~base(){
        cout << "destructing base class\n";
    }
    void showxy(){
        cout << x << " " << y << '\n';
    }
};

int main(){
    derived ob(10, 20, 30);

    ob.showxy();
    ob.showij();
    return 0;
}
```

```
class derived: public base {
    int i, j;
public:
    derived(int n, int m, int p): base (n, p){
        i = n;
        j = m;
        cout << "Constructing derived class\n";
    }
    ~derived(){
        cout << "destructing derived class\n";
    }
    void showij(){
        cout << i << " " << j << '\n';
    }
};
```

**OUTPUT:**  
Constructing base class  
Constructing derived class  
10 30  
10 20  
Destructing derived class  
Destructing base class



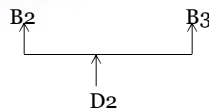
## Multiple Inheritances

➤ There are **two ways** that a derived class can inherit more than one base class: the **hierarchical inheritance** and the **base inheritance**.

➤ **The hierarchical inheritance:** a derived class can be used as a base class for another derived class, creating a multilevel class hierarchy.

B1 ← D1 ← D2

➤ **The multiple base inheritances:** a derived class can directly inherit more than one base class.



➤ The general for **multiple base inheritance** is as follows:

```

derived-constructor(arg-list): base1(arg-list), base2(arg-list), ..., baseN(arg-list){
}
  
```

➤ When **multiple base** classes are inherited, **constructors** are executed from **left to right**, and destructors are executed in **opposite order**.

➤ When a derived class inherits a **hierarchy of classes**, each derived class in the chain must pass back to its preceding base any arguments it needs.



## Multiple Inheritances

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

```
class B1 {
    int x;
public:
    B1(int i) {
        x = i;
        cout << "Constructing B1\n";
    }
    ~B1() { cout << "Destructing B1\n"; }
    int getx() { return x; }
};
```

```
class B2 {
    int y;
public:
    B2(int j) {
        y = j;
        cout << "Constructing B2\n";
    }
    ~B2() { cout << "Destructing B2\n"; }
    int gety() { return y; }
};
```

```
class B3 {
    int z;
public:
    B3(int k) {
        z = k;
        cout << "Constructing B3\n";
    }
    ~B3() { cout << "Destructing B3\n"; }
    int getz() { return k; }
};
```

```
class D1: public B1 {
public:
    D1(int j):B1(j) {
        cout << "Constructing D1\n";
    }
    ~D1() { cout << "Destructing D1\n"; }
};
```



## Multiple Inheritances

```
class D2: public D1, public B2, public B3 {
public:
    D2(int i, int j, int k): D1(i), B2(j), B3(k) {
        cout << "Constructing D2\n";
    }
    ~D2(){ cout << "Destructing D2\n";}
    void show(){
        cout << getx() << " " << gety() << " ";
        cout << getz() << "\n";
    }
};
```

```
int main(){
    D2 ob(10, 20, 30);

    ob.show();
    return 0;
}
```

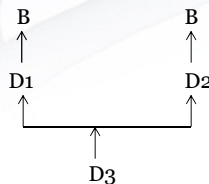
### OUTPUT:

```
Constructing B1
Constructing D1
Constructing B2
Constructing B3
Constructing D2
10 20 30
Destructing D2
Destructing B3
Destructing B2
Destructing D1
Destructing B1
```



## Virtual Base Classes

➤ In the following scenario, **B** is inherited by **D1** and **D2** and **D3** directly inherits **D1** and **D2**. This causes **ambiguity** (whether calling through **D1** or **D2**) when a member of **B** is inherited in **D3**, because two copies of **B** is included in **D3**.



➤ The keyword “**virtual**” precedes the base class in both cases inherits only one copy in **D3**.



## Virtual Base Classes

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

```
class B{
public:
    int i;
};
```

```
class D1: virtual public B{
public:
    int j;
};
```

```
class D2: virtual public B{
public:
    int k;
};
```

```
class D3: public D1, public D2{
public:
    int product(){ return i *j; }
};
```

```
int main(){
    D1 ob1;
    D3 ob3;

    ob1.i = 100;
    ob2.i = 10;
    return 0;
}
```