UNIT 4
Inheritance

Base-Class Access Control

class *derived-class-name : access base-class-name* {

// body of class

};

Access Specifier in Base Class	Access Specifier in Derived Class		
	Public Inheritance	Protected Inheritance	Private Inheritance
public	public	protected	private
protected	protected	protected	private
Private	Not accessible	Not accessible	Not accessible

Public inheritance

```
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n";
}
};
class derived : public base {
int k;
public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob(3);
ob.set(1, 2); // access member of base
ob.show(); // access member of base
ob.showk(); // uses member of derived class
return 0;
}
```

Private inheritance

```
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n";}
};
// Public elements of base are private in
derived.
class derived : private base {
int k;
public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob(3);
ob.set(1, 2); // error, can't access set()
ob.show(); // error, can't access show()
return 0;
}
```

Inheritance & Protected members

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

class base {
protected:
int i, j; // private to base, but accessible by derived
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
// derived may access base's i and j
void setk() { k=i*j; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob;
ob.set(2, 3); // OK, known to derived
ob.show(); // OK, known to derived
ob.setk();
ob.showk();
return 0;
```

Protected Base-Class Inheritance

```
#include <iostream>
using namespace std;
class base {
protected:
int i, j; // private to base, but accessible by derived
public:
void setij(int a, int b) { i=a; j=b; }
void showij() { cout << i << " " << j << "\n"; }
};
// Inherit base as protected.
class derived : protected base{
int k;
public:
// derived may access base's i and j and setij().
void setk() { setij(10, 12); k = i*j; }
// may access showij() here
void showall() { cout << k << " "; showij(); }
};
int main()
{
derived ob;
// ob.setij(2, 3); // illegal, setij() is
// protected member of derived
ob.setk(); // OK, public member of derived
ob.showall(); // OK, public member of derived
// ob.showij(); // illegal, showij() is protected
// member of derived
return 0;
}
```

Inheriting Multiple Base Classes

```
#include <iostream>
using namespace std;
class base1 {
protected:
int x;
public:
void showx() { cout << x << "\n"; }
};
class base2 {
protected:
int y;
public:
void showy() {cout << y << "\n";}
};
// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
void set(int i, int j) { x=i; y=j; }
};
int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

Constructors, Destructors and Inheritance

- Output

Constructing base

Constructing derived

Destructing derived

Destructing base

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// do nothing but construct and destruct ob
return 0;
}
```

OUTPUT

Constructing base

Constructing derived1

Constructing derived2

Destructing derived2

Destructing derived1

Destructing base

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived1 : public base {
public:
derived1() { cout << "Constructing derived1\n"; }
~derived1() { cout << "Destructing derived1\n"; }
};
class derived2: public derived1 {
public:
derived2() { cout << "Constructing derived2\n"; }
~derived2() { cout << "Destructing derived2\n"; }
};
int main()
{
derived2 ob;
// construct and destruct ob
return 0;
}
```

Multiple base classes

- Output

Constructing base1

Constructing base2

Constructing derived

Destructing derived

Destructing base2

Destructing base1

```
#include <iostream>
using namespace std;
class base1 {
public:
base1() { cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
public:
base2() { cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// construct and destruct ob
return 0;
}
```

Passing Parameters to Base-Class Constructors

derived-constructor(arg-list) : base1(arg-list),

base2(arg-list),

// ...

baseN(arg-list)

{

// body of derived constructor

}

```
#include <iostream>
using namespace std;
class base {
protected:
int i;
public:
base(int x) { i=x; cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
int j;
public:
// derived uses x; y is passed along to base.
derived(int x, int y): base(y)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
int j;
public:
derived(int x, int y, int z): base1(y), base2(z)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4, 5);
ob.show(); // displays 4 3 5
return 0;
}
```

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
/* Derived constructor uses no parameter,
but still must be declared as taking them to
pass them along to base classes.
*/
derived(int x, int y): base1(x), base2(y)
{ cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 3 4
return 0;
}
```

Granting Access

To grant certain public members of the base class public status in the derived class even though the base class is inherited as **private**.

An access declaration takes this general form:

base-class::member;

```
#include <iostream>
using namespace std;
class base {
int i; // private to base
public:
int j, k;
void seti(int x) { i = x; }
int geti() { return i; }
};
// Inherit base as private.
class derived: private base {
public:
base::j; // make j public again - but not k
base::seti; // make seti() public
base::geti; // make geti() public
// base::i; // illegal, you cannot elevate access
int a; // public
};
int main()
{
derived ob;
//ob.i = 10; // illegal because i is private in
derived
ob.j = 20; // legal because j is made public in
derived
//ob.k = 30; // illegal because k is private in
derived
ob.a = 40; // legal because a is public in derive
ob.seti(10);
cout << ob.geti() << " " << ob.j << " " << ob.a;
return 0;
}
```

Virtual Base Classes

```
#include <iostream>
using namespace std;
class base {
public:
int i;
};

class derived1 : public base {
public:
int j;
};

class derived2 : public base {
public:
int k;
};

/* derived3 inherits both derived1 and
derived2.
This means that there are two copies of
base in derived3! */

class derived3 : public derived1, public
derived2 {
public:
int sum;
};

int main(){
derived3 ob;
ob.i = 10; // this is ambiguous, which i???
ob.j = 20;
ob.k = 30;
// i ambiguous here, too
ob.sum = ob.i + ob.j + ob.k;
// also ambiguous, which i?
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

Method 1 to resolve ambiguity
Resolving scope using :: operator

```
int main()
{
derived3 ob;
ob.derived1::i = 10; // scope resolved, use
derived1's i
ob.j = 20;
ob.k = 30;
// scope resolved
ob.sum = ob.derived1::i + ob.j + ob.k;
// also resolved here
cout << ob.derived1::i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

Method 2 to resolve ambiguity

Virtual Base classes

```
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
```

```
int main()
{
derived3 ob;
ob.i = 10; // now unambiguous
ob.j = 20;
ob.k = 30;
// unambiguous
ob.sum = ob.i + ob.j + ob.k;
// unambiguous
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

```
int main()
{
derived3 ob;
ob.i = 10; // now unambiguous
ob.j = 20;
ob.k = 30;
// unambiguous
ob.sum = ob.i + ob.j + ob.k;
// unambiguous
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```