VIRTUAL BASE CLASSES IN CPP

Virtual Base Classes

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited.

For example, consider this incorrect program:

// This program contains an error and will not compile.

```
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
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;
 }
```

As the comments in the program indicate, both **derived1** and **derived2** inherit **base**. However, **derived3** inherits both **derived1** and **derived2**. This means that there are two copies of **base** present in an object of type **derived3**. Therefore, in an expression like

ob.i = 10;

which **i** is being referred to, the one in **derived1** or the one in **derived2?** Because there are two copies of **base** present in object **ob**, there are two **ob.i**s! As you can see, the statement is inherently ambiguous.

There are two ways to remedy the preceding program. The first is to apply the scope resolution operator to **i** and manually select one **i.** For example, this version of the program does compile and run as expected:

```
// This program uses explicit scope resolution to select i.
#include  <iostream>
using namespace std;
```

```
  class base {
  public:
  int i;
  };
  // derived1 inherits base.
  class derived1 : public base {
  public:
  int j;
  };
  // derived2 inherits base.
  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.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;
}
```

As you can see, because the **::** was applied, the program has manually selected **derived1**'s version of **base**. However, this solution raises a deeper issue: What if only one copy of **base** is actually required? Is there some way to prevent two copies from being included in **derived3**? The answer, as you probably have guessed, is yes. This solution is achieved using **virtual** base classes.

When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited. You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited. For example, here is another version of the example program in which **derived3** contains only one copy of **base**:

```
// This program uses virtual base classes.
#include <iostream>
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;
}
```

As you can see, the keyword **virtual** precedes the rest of the inherited **class**' specification. Now that both **derived1** and **derived2** have inherited **base** as **virtual**, any multiple inheritance involving them will cause only one copy of **base** to be present. Therefore, in **derived3**, there i only one copy of **base** and **ob.i = 10** is perfectly valid and unambiguous. One further point to keep in mind: Even though both **derived1** and **derived2** specify **base** as **virtual**, **base** is stil present in objects of either type. For example, the following sequence is perfectly valid:

```
// define a class of type derived1
derived1 myclass;
myclass.i = 88;
```

The only difference between a normal base class and a **virtual** one is what occurs when an object inherits the base more than once. If **virtual** base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.