IS 0020
Program Design and Software Tools
Introduction to C++ Programming

Multiple Inheritance

March 1-3, 2005

22.9 Multiple Inheritance

- Multiple inheritance
  - Derived class has several base classes
  - Powerful, but can cause ambiguity problems
    - If both base classes have functions of the same name
    - Solution: specify exact function using **::**
      - **myObject.BaseClass1::function()**
  - Format
    - Use comma-separated list

```
class Derived : public Base1, public Base2{
   contents
}
```

Outline

base1.h (1 of 1)

```
1  // Fig. 22.14: base1.h
2  // Definition of class Base1
3  #ifndef BASE1_H
4  #define BASE1_H
5
6  // class Base1 definition
7  class Base1 {
8  public:
9     Base1( int parameterValue ) { value = parameterValue; }
10    int getData() const { return value; }
11
12 protected:      // accessible to derived classes
13    int value;   // inherited by derived class
14
15 };  // end class Base1
16
17 #endif  // BASE1_H
```

There are two base classes in this example, each has its own **`getData`** function.

This base class contains an **`int`**.

Outline

base2.h (1 of 1)

```
1  // Fig. 22.15: base2.h
2  // Definition of class Base2
3  #ifndef BASE2_H
4  #define BASE2_H
5
6  // class Base2 definition
7  class Base2 {
8  public:
9     Base2( char characterData ) { letter = characterData; }
10    char getData() const { return letter; }
11
12 protected:        // accessible to derived classes
13    char letter;   // inherited by derived class
14
15 };  // end class Base2
16
17 #endif  // BASE2_H
```

Outline

derived.h (1 of 1)

```
1  // Fig. 22.16: derived.h
2  // Definition of class Derived which inherits
3  // multiple base classes (Base1 and Base2).
4  #ifndef DERIVED_H
5  #define DERIVED_H
6
7  #include <iostream>
8
9  using std::ostream;
10
11 #include "base1.h"
12 #include "base2.h"
13
14 // class Derived definition
15 class Derived : public Base1, public Base2 {
16    friend ostream &operator<<( ostream &, const Derived & );
17
18 public:
19    Derived( int, char, double );
20    double getReal() const;
21
22 private:
23    double real;   // derived class's private data
24
25 };  // end class Derived
26
27 #endif  // DERIVED_H
```

Use comma-separated list.

Outline

(1 of 1)

```
1  // Fig. 22.17: derived.cpp
2  // Member function definitions for class Derived
3  #include "derived.h"
4
5  // constructor for Derived calls constructors for
6  // class Base1 and class Base2.
7  // use member initializers to call base-class constructors
8  Derived::Derived( int integer, char character, double double1 )
9     : Base1( integer ), Base2( character ), real( double1 ) { }
10
11 // return real
12 double Derived::getReal() const { return real; }
13
14 // display all data members of Derived
15 ostream &operator<<( ostream &output, const Derived &derived )
16 {
17    output << "    Integer: " << derived.value
18           << "\n  Character: " << derived.letter
19           << "\nReal number: " << derived.real;
20
21    return output;   // enables cascaded calls
22
23 }  // end operator<<
```

Note use of base-class constructors in derived class constructor.

Outline

fig22_18.cpp
(1 of 2)

```
1  // Fig. 22.18: fig22_18.cpp
2  // Driver for multiple inheritance example.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include "base1.h"
9  #include "base2.h"
10 #include "derived.h"
11
12 int main()
13 {
14    Base1 base1( 10 ), *base1Ptr = 0;  // create Base1 object
15    Base2 base2( 'Z' ), *base2Ptr = 0; // create Base2 object
16    Derived derived( 7, 'A', 3.5 );    // create Derived object
17
18    // print data members of base-class objects
19    cout << "Object base1 contains integer "
20         << base1.getData()
21         << "\nObject base2 contains character "
22         << base2.getData()
23         << "\nObject derived contains:\n" << derived << "\n\n";
24
```


Outline

fig22_18.cpp
(2 of 2)

```
25    // print data members of derived-class object
26    // scope resolution operator resolves getData ambiguity
27    cout << "Data members of Derived can be"
28         << " accessed individually:"
29         << "\n    Integer: " << derived.Base1::getData()
30         << "\n  Character: " << derived.Base2::getData()
31         << "\nReal number: " << derived.getReal() << "\n\n";
32
33    cout << "Derived can be treated as an "
34         << "object of either base class:\n";
35
36    // treat Derived as a Base1 object
37    base1Ptr = &derived;
38    cout << "base1Ptr->getData() yields "
39         << base1Ptr->getData() << '\n';
40
41    // treat Derived as a Base2 object
42    base2Ptr = &derived;
43    cout << "base2Ptr->getData() yields "
44         << base2Ptr->getData() << endl;
45
46    return 0;
47
48 }  // end main
```

Note calls to specific base class functions.

Can treat derived-class pointer as either base-class pointer.

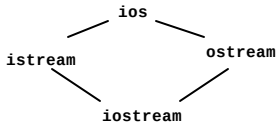
```
Object base1 contains integer 10
Object base2 contains character Z
Object derived contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Outline

fig22_18.cpp
output (1 of 1)

Multiple Inheritance and virtual Base Classes

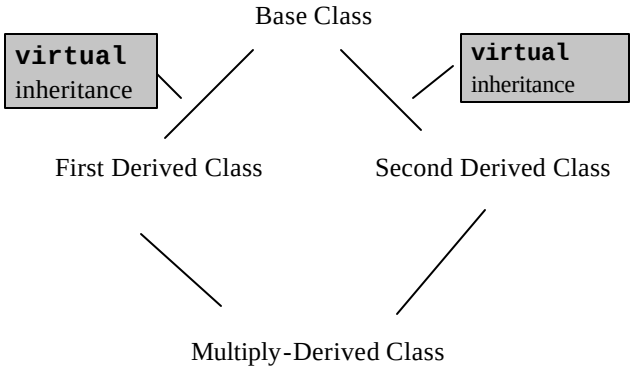
- Ambiguities from multiple inheritance



- **iostream** could have duplicate subobjects
  - Data from **ios** inherited into **ostream** and **istream**
  - Upcasting **iostream** pointer to **ios** object is a problem
    - Two **ios** subobjects could exist, which is used?
  - Ambiguous, results in syntax error
    - **iostream** does not actually have this problem

Multiple Inheritance and virtual Base Classes

- Solution: use virtual base class inheritance
  - Only one subobject inherited into multiply derived class



Outline

fig22_20.cpp
(1 of 3)

```
1  // Fig. 22.20: fig22_20.cpp
2  // Attempting to polymorphically call a function that is
3  // multiply inherited from two base classes.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  // class Base definition
10 class Base {
11 public:
12    virtual void print() const = 0;  // pure virtual
13
14 };  // end class Base
15
16 // class DerivedOne definition
17 class DerivedOne : public Base {
18 public:
19
20    // override print function
21    void print() const { cout << "DerivedOne\n"; }
22
23 };  // end class DerivedOne
24
```

This example will demonstrate the ambiguity of multiple inheritance.

Outline

fig22_20.cpp
(2 of 3)

```
25 // class DerivedTwo definition
26 class DerivedTwo : public Base {
27 public:
28
29    // override print function
30    void print() const { cout << "DerivedTwo\n"; }
31
32 };  // end class DerivedTwo
33
34 // class Multiple definition
35 class Multiple : public DerivedOne, public DerivedTwo {
36 public:
37
38    // qualify which version of function print
39    void print() const { DerivedTwo::print(); }
40
41 };    // end class Multiple
42
```


Outline

fig22_20.cpp
(3 of 3)

```
43 int main()
44 {
45    Multiple both;   // instantiate Multiple object
46    DerivedOne one;  // instantiate DerivedOne object
47    DerivedTwo two;  // instantiate DerivedTwo object
48
49    // create array of base-class point
50    Base *array[ 3 ];
51
52    array[ 0 ] = &both;   // ERROR--ambiguous
53    array[ 1 ] = &one;
54    array[ 2 ] = &two;
55
56    // polymorphically invoke print
57    for ( int i = 0; i < 3; i++ )
58       array[ i ] -> print();
59
60    return 0;
61
62 }  // end main
```

Which base subobject will be used?

Outline

fig22_21.cpp
(1 of 3)

```
1  // Fig. 22.21: fig22_21.cpp
2  // Using virtual base classes.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // class Base definition
9  class Base {
10 public:
11
12    // implicit default constructor
13
14    virtual void print() const = 0; // pure virtual
15
16 };  // end Base class
17
18 // class DerivedOne definition
19 class DerivedOne : virtual public Base {
20 public:
21
22    // implicit default constructor calls
23    // Base default constructor
24
25    // override print function
26    void print() const { cout << "DerivedOne\n"; }
27
28 };  // end DerivedOne class
```

Use virtual inheritance to solve the ambiguity problem.

The compiler generates default constructors, which greatly simplifies the hierarchy.

Outline

fig22_21.cpp
(2 of 3)

```
29
30 // class DerivedTwo definition
31 class DerivedTwo : virtual public Base {
32 public:
33
34    // implicit default constructor calls
35    // Base default constructor
36
37    // override print function
38    void print() const { cout << "DerivedTwo\n"; }
39
40 }; // end DerivedTwo class
41
42 // class Multiple definition
43 class Multiple : public DerivedOne, public DerivedTwo {
44 public:
45
46    // implicit default constructor calls
47    // DerivedOne and DerivedTwo default constructors
48
49    // qualify which version of function print
50    void print() const { DerivedTwo::print(); }
51
52 }; // end Multiple class
```

Use **`virtual`** inheritance, as before.

Outline

fig22_21.cpp
(3 of 3)

```
53
54 int main()
55 {
56    Multiple both;   // instantiate Multiple object
57    DerivedOne one;  // instantiate DerivedOne object
58    DerivedTwo two;  // instantiate DerivedTwo object
59
60    // declare array of base-class pointers and initialize
61    // each element to a derived-class type
62    Base *array[ 3 ];
63
64    array[ 0 ] = &both;
65    array[ 1 ] = &one;
66    array[ 2 ] = &two;
67
68    // polymorphically invoke function print
69    for ( int i = 0; i < 3; i++ )
70       array[ i ]->print();
71
72    return 0;
73
74 }  // end main
```

```
DerivedTwo
DerivedOne
DerivedTwo
```