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
      ```cpp
class Derived : public Base1, public Base2{
    contents
}
```
There are two base classes in this example, each has its own `getData` function.
This base class contains an `int`.

// Fig. 22.14: base1.h
// Definition of class Base1
#ifndef BASE1_H
#define BASE1_H

// class Base1 definition
class Base1 {
public:
    Base1( int parameterValue ) { value = parameterValue; }
    int getData() const { return value; }

protected:      // accessible to derived classes
    int value;   // inherited by derived class
};  // end class Base1
#endif // BASE1_H

// Fig. 22.15: base2.h
// Definition of class Base2
#ifndef BASE2_H
#define BASE2_H

// class Base2 definition
class Base2 {
public:
    Base2( char characterData ) { letter = characterData; }
    char getData() const { return letter; }

protected:      // accessible to derived classes
    char letter;   // inherited by derived class
};  // end class Base2
#endif // BASE2_H
// Fig. 22.16: derived.h
// Definition of class Derived which inherits
// multiple base classes (Base1 and Base2).
#ifndef DERIVED_H
#define DERIVED_H
#include <iostream>
using std::ostream;
#include "base1.h"
#include "base2.h"
// class Derived definition
class Derived : public Base1, public Base2 {
    friend ostream &operator<<( ostream &, const Derived & );
public:
    Derived(int integer, char character, double double1);
    double getReal() const;
private:
    double real;   // derived class’s private data
};  // end class Derived
#endif  // DERIVED_H

// Fig. 22.17: derived.cpp
// Member function definitions for class Derived
#include "derived.h"
// constructor for Derived calls constructors for
// class Base1 and class Base2.
Derived::Derived(int integer, char character, double double1):
    Base1(integer), Base2(character), real(double1) {
}
// return real
double Derived::getReal() const { return real; }
// display all data members of Derived
ostream operator<<(ostream &output, const Derived &derived)
{    output << " Integer: " << derived.value
        << " Character: " << derived.letter
        << " Real number: " << derived.real;
    return output;  // enables cascaded calls
}  // end operator<<
// Fig. 22.18: fig22_18.cpp
// Driver for multiple inheritance example.
#include <iostream>

using std::cout;
using std::endl;

#include "base1.h"
#include "base2.h"
#include "derived.h"

int main()
{
    Base1 base1( 10 ), *base1Ptr = 0;  // create Base1 object
    Base2 base2( 'Z' ), *base2Ptr = 0; // create Base2 object
    Derived derived( 7, 'A', 3.5 );    // create Derived object

    // print data members of base-class objects
    cout << "Object base1 contains integer \\
    \"<< base1.getData() \n\n    Object base2 contains character \\
    \"<< base2.getData() \n\n    Object derived contains:
    \"<< derived << \n\n    // print data members of derived-class object
    // scope resolution operator resolves getData ambiguity
    cout << "Data members of Derived can be" \\
    \" accessed individually:"
    \" Integer: \"<< derived.Base1::getData() \\
    \" Character: \"<< derived.Base2::getData() \\
    \" Real number: \"<< derived.getReal() << \n\n    cout << "Derived can be treated as an " \\
    \" object of either base class:"

    // treat Derived as a Base1 object
    base1Ptr = &derived;
    cout << "base1Ptr->getData() yields " \\
    \"<< base1Ptr->getData() \n\n    // treat Derived as a Base2 object
    base2Ptr = &derived;
    cout << "base2Ptr->getData() yields " \\
    \"<< base2Ptr->getData() \n
    return 0;
}

© 2003 Prentice Hall, Inc.
All rights reserved.
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

---

Multiple Inheritance and virtual Base Classes

- Ambiguities from multiple inheritance

[Diagram showing multiple inheritance relationships]

- `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

```
// Fig. 22.20: fig22_20.cpp
// Attempting to polymorphically call a function that is
// multiply inherited from two base classes.
#include <iostream>

using std::cout;
using std::endl;

// class Base definition
class Base {
  virtual void print() const = 0;  // pure virtual
};  // end class Base

// class DerivedOne definition
class DerivedOne : public Base {
  public:
    virtual void print() const { cout << "DerivedOne\n"; }  
};  // end class DerivedOne

This example will demonstrate the ambiguity of multiple inheritance.
```

© 2003 Prentice Hall, Inc. All rights reserved.
// class DerivedTwo definition
class DerivedTwo : public Base {
public:
    void print() const { cout << "DerivedTwo\n"; }
}; // end class DerivedTwo

// class Multiple definition
class Multiple : public DerivedOne, public DerivedTwo {
public:
    void print() const { DerivedTwo::print(); }
}; // end class Multiple

int main()
{
    Multiple both;    // instantiate Multiple object
    DerivedOne one;   // instantiate DerivedOne object
    DerivedTwo two;   // instantiate DerivedTwo object

    // create array of base-class pointers
    Base *array[ 3 ];
    array[ 0 ] = &both;   // ERROR -- ambiguous
    array[1 ] = &one;
    array[2 ] = &two;

    // polymorphically invoke print
    for ( int i = 0; i < 3; i++ )
        array[i] -> print();

    return 0;
} // end main

Which base subobject will be used?
// Fig. 22.21: fig22_21.cpp
// Using virtual base classes.
#include <iostream>

using std::cout;
using std::endl;

// class Base definition
class Base {
public:
  // implicit default constructor
  virtual void print() const = 0; // pure virtual
}; // end Base class

// class DerivedOne definition
class DerivedOne : virtual public Base {
public:
  // implicit default constructor calls
  // Base default constructor
  // override print function
  void print() const { cout << "DerivedOne\n"; }
}; // end DerivedOne class

// class DerivedTwo definition
class DerivedTwo : virtual public Base {
public:
  // implicit default constructor calls
  // Base default constructor
  // override print function
  void print() const { cout << "DerivedTwo\n"; }
}; // end DerivedTwo class

// class Multiple definition
class Multiple : public DerivedOne, public DerivedTwo {
public:
  // implicit default constructor calls
  // DerivedOne and DerivedTwo default constructors
  // qualify which version of function print
  void print() const { DerivedTwo::print(); }
}; // end Multiple class
int main()
{
    Multiple both;  // instantiate Multiple object
    DerivedOne one;  // instantiate DerivedOne object
    DerivedTwo two;  // instantiate DerivedTwo object
    // declare array of base-class pointers and initialize
    // each element to a derived-class type
    Base *array[ 3 ];
    array[ 0 ] = &both;
    array[ 1 ] = &one;
    array[ 2 ] = &two;
    // polymorphically invoke function print
    for ( int i = 0; i < 3; i++ )
        array[ i ]->print();
    return 0;
}  // end main