Chapter 23
Other Topics

C++ How to Program, 9/e

23.4 namespaces

- A program may include many identifiers defined in different scopes.
- Sometimes a variable of one scope will "overlap" (i.e., collide) with a variable of the *same* name in a *different* scope, possibly creating a *naming conflict*.
- Such overlapping can occur at many levels.
- Identifier overlapping occurs frequently in third-party libraries that happen to use the same names for global identifiers (such as functions).
- This can cause compilation errors.

23.4 namespaces (Cont.)

- C++ solves this problem with `namespaces`.
- Each `namespace` defines a scope in which identifiers and variables are placed.
- To use a `namespace` member, either the member's name must be qualified with the `name-space` name and the *scope resolution operator* (`::`), as in
  - *`MyNameSpace::member`*
- or a `using` directive *must* appear *before* the name is used in the program.
- Typically, such `using` statements are placed at the beginning of the file in which members of the `namespace` are used.

23.4 namespaces (Cont.)

- For example, placing the following `using` directive at the beginning of a source-code file
  - `using namespace` *`MyNameSpace;`*
- specifies that members of `namespace` *MyNameSpace can be used in the file without preceding each member with MyNameSpace and the scope resolution operator (`::`).*
- A `using` declaration (e.g., `using std::cout;`) brings one name into the scope where the declaration appears.
- A `using` directive (e.g., `using namespace std;`) brings *all* the names from the specified namespace into the scope where the directive appears.

Error-Prevention Tip 24.2

Precede a member with its `namespace` name and the scope resolution operator (`::`) if the possibility exists of a naming conflict.

23.4 namespaces (Cont.)

- *Not all `namespace`s are guaranteed to be unique.*
  - Two third-party vendors might inadvertently use the same identifiers for their `namespace` names.
  - Figure 23.3 demonstrates the use of `namespace`s.


```cpp
// Fig. 24.3: fig24_03.cpp
// Demonstrating namespaces.
#include <iostream>
using namespace std;

int integer1 = 98; // global variable

// create namespace Example
namespace Example
{
   // declare two constants and one variable
   const double PI = 3.14159;
   const double E = 2.71828;
   int integer1 = 8;

   void printValues(); // prototype

   // nested namespace
   namespace Inner
   {
      // define enumeration
      enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
   } // end Inner namespace
} // end Example namespace
```

Fig. 24.3 | Demonstrating the use of namespaces. (Part 1 of 3.)


```cpp

// create unnamed namespace
namespace
{
   double doubleInUnnamed = 88.22; // declare variable
} // end unnamed namespace

int main()
{
   // output value doubleInUnnamed of unnamed namespace
   cout << "doubleInUnnamed = " << doubleInUnnamed;

   // output global variable
   cout << "\n(global) integer1 = " << integer1;

   // output values of Example namespace
   cout << "\nPI = " << Example::PI << "\nE = " << Example::E
      << "\ninteger1 = " << Example::integer1 << "\nFISCAL3 = "
      << Example::Inner::FISCAL3 << endl;

   Example::printValues(); // invoke printValues function
} // end main

```

Fig. 24.3 | Demonstrating the use of namespaces. (Part 2 of 3.)

```
48 // display variable and constant values
49 void Example::printValues()
50 {
51    cout << "\nIn printValues:\ninteger1 = " << integer1 << "\nPI = "
52       << PI << "\nE = " << E << "\ndoubleInUnnamed = "
53       << doubleInUnnamed << "\n(global) integer1 = " << ::integer1
54       << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
55 } // end printValues
```

```
doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 1992

In printValues:
integer1 = 8
PI = 3.14159
E = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
FISCAL3 = 1992
```

Fig. 24.3 | Demonstrating the use of namespaces. (Part 3 of 3.)

23.4 namespaces (Cont.)

- Lines 9–24 use the keyword `namespace` to define namespace `Example`.
- The body of a namespace is delimited by braces (`{}`).
- The `namespace Example`'s members consist of two constants (`PI` and `E` in lines 12–13), an `int` (`integer1` in line 14), a function (`printValues` in line 16) and a nested namespace (`Inner` in lines 19–23).
- Notice that member `integer1` has the same name as global variable `integer1` (line 6).
- *Variables that have the same name must have different scopes*—otherwise compilation errors occur.

23.4 namespaces (Cont.)

- A namespace can contain constants, data, classes, nested namespaces, functions, etc.
- Definitions of namespaces must occupy the *global scope* or be *nested* within other namespaces.
- Unlike classes, different namespace members can be defined in separate `namespace` blocks—each standard library header has a `namespace` block placing its contents in `namespace std`.
- Lines 27–30 create an unnamed namespace containing the member `doubleInUnnamed`.

23.4 namespaces (Cont.)

- Variables, classes and functions in an *unnamed `namespace`* are accessible only in the current translation unit (a `.cpp` file and the files it `include`s).
- However, unlike variables, classes or functions with `static` linkage, those in the *unnamed `namespace`* may be used as template arguments.
- The unnamed namespace has an implicit `using` directive, so its members appear to occupy the global namespace, are accessible directly and *do not have to be qualified with a `namespace` name.*
- Global variables are also part of the global namespace and are accessible in all scopes following the declaration in the file.

23.4 namespaces (Cont.)

- Line 35 outputs the value of variable `doubleInUnnamed`, which is directly accessible as part of the *unnamed* ***namespace***.
- Line 38 outputs the value of global variable `integer1`.
- For both of these variables, the compiler first attempts to locate a *local* declaration of the variables in `main`.
- Since there are no local declarations, the compiler assumes those variables are in the global namespace.
- Lines 41–43 output the values of `PI`, `E`, `integer1` and `FISCAL3` from namespace `Example`.

23.4 namespaces (Cont.)

- Notice that each must be *qualified* with `Example::` because the program does not provide any `using` directive or declarations indicating that it will use members of namespace `Example`.
- In addition, member `integer1` must be qualified, because a global variable has the same name.
- Otherwise, the global variable's value is output.
- `FISCAL3` is a member of nested namespace `Inner`, so it must be *qualified* with `Example::Inner::`.
- Function `printValues` (defined in lines 49–55) is a member of `Example`, so it can access other members of the `Example` namespace directly *without using a* ***namespace*** *qualifier.*

23.4 namespaces (Cont.)

- The output statement in lines 51–54 outputs `integer1`, `PI`, `E`, `doubleInUnnamed`, global variable `integer1` and `FISCAL3`.
- Notice that `PI` and `E` are *not qualified* with `Example`.
- Variable `doubleInUnnamed` is still *accessible*, because it is in the *unnamed* ***namespace*** and the variable name does *not conflict* with any other members of namespace `Example`.
- The global version of `integer1` must be *qualified* with the scope resolution operator (`::`), because its name *conflicts* with a member of namespace `Example`.
- Also, `FISCAL3` must be qualified with `Inner::`.
- When accessing members of a *nested* ***namespace***, the members must be *qualified* with the ***namespace*** name (unless the member is being used inside the nested ***namespace***).

Common Programming Error 24.1

Placing `main` in a `namespace` is a compilation error.

23.4 namespaces (Cont.)

- *namespace*s are particularly useful in large-scale applications that use many class libraries.
- In such cases, there's a higher likelihood of naming conflicts.
- When working on such projects, there should *never* be a using directive in a header.
- Having one brings the corresponding names into any file that includes the header. This could result in name collisions and subtle, hard-to-find errors.
- Instead, use only fully qualified names in headers (for example, std::cout or std::string).

23.4 namespaces (Cont.)

- *namespace*s can be *aliased*.
- For example the statement
  - `namespace CPPHTP = CPlusPlusHowToProgram;`
- creates the namespace alias `CPPHTP` for `CPlusPlusHowToProgram`.

23.5 Operator Keywords

- The C++ standard provides operator keywords (Fig. 23.4) that can be used in place of several C++ operators.
- You can use operator keywords if you have keyboards that do not support certain characters such as !, &, ^, ~, |, etc.

Operator	Operator keyword	Description
Logical operator keywords		
&&	and	logical AND
||	or	logical OR
!	not	logical NOT
Inequality operator keyword		
!=	not_eq	inequality

Fig. 24.4 | Operator keyword alternatives to operator symbols. (Part 1 of 2.)

Operator	Operator keyword	Description
Bitwise operator keywords		
&	bitand	bitwise AND
|	bitor	bitwise inclusive OR
^	xor	bitwise exclusive OR
~	compl	bitwise complement
Bitwise assignment operator keywords		
&=	and_eq	bitwise AND assignment
|=	or_eq	bitwise inclusive OR assignment
^=	xor_eq	bitwise exclusive OR assignment

Fig. 24.4 | Operator keyword alternatives to operator symbols. (Part 2 of 2.)

23.5 Operator Keywords (Cont.)

- Figure 23.5 demonstrates the operator keywords.
- Microsoft Visual C++ 2010 requires the header `<ciso646>` (line 4) to use the operator keywords.
- In GNU C++ and LLVM, the operator keywords are always defined and this header is not required.


```
1  // Fig. 24.5: fig24_05.cpp
2  // Demonstrating operator keywords.
3  #include <iostream>
4  #include <ciso646> // enables operator keywords in Microsoft Visual C++
5  using namespace std;
6
7  int main()
8  {
9     bool a = true;
10    bool b = false;
11    int c = 2;
12    int d = 3;
13
14    // sticky setting that causes bool values to display as true or false
15    cout << boolalpha;
16
17    cout << "a = " << a << "; b = " << b
18       << "; c = " << c << "; d = " << d;
19
```

Fig. 24.5 | Demonstrating the operator keywords. (Part 1 of 3.)


```
20    cout << "\n\nLogical operator keywords:";
21    cout << "\n   a and a: " << ( a and a );
22    cout << "\n   a and b: " << ( a and b );
23    cout << "\n    a or a: " << ( a or a );
24    cout << "\n    a or b: " << ( a or b );
25    cout << "\n     not a: " << ( not a );
26    cout << "\n     not b: " << ( not b );
27    cout << "\na not_eq b: " << ( a not_eq b );
28
29    cout << "\n\nBitwise operator keywords:";
30    cout << "\nc bitand d: " << ( c bitand d );
31    cout << "\nc bit_or d: " << ( c bitor d );
32    cout << "\n   c xor d: " << ( c xor d );
33    cout << "\n   compl c: " << ( compl c );
34    cout << "\nc and_eq d: " << ( c and_eq d );
35    cout << "\n c or_eq d: " << ( c or_eq d );
36    cout << "\nc xor_eq d: " << ( c xor_eq d ) << endl;
37 } // end main
```

Fig. 24.5 | Demonstrating the operator keywords. (Part 2 of 3.)

```
a = true; b = false; c = 2; d = 3

Logical operator keywords:
   a and a: true
   a and b: false
    a or a: true
    a or b: true
     not a: false
     not b: true
a not_eq b: true

Bitwise operator keywords:
c bitand d: 2
c bit_or d: 3
   c xor d: 1
   compl c: -3
c and_eq d: 2
 c or_eq d: 3
c xor_eq d: 0
```

Fig. 24.5 | Demonstrating the operator keywords. (Part 3 of 3.)

23.7 Multiple Inheritance

- In Chapters 11 and 12, we discussed *single inheritance, in which each class is derived from exactly one base class.*
- In C++, a class may be derived from more than one base class—a technique known as multiple inheritance in which a derived class inherits the members of two or more base classes.

23.7 Multiple Inheritance (Cont.)

- This powerful capability encourages interesting forms of software reuse but can cause a variety of ambiguity problems.
- *Multiple inheritance is a difficult concept that should be used only by experienced programmers.*
- In fact, some of the problems associated with multiple inheritance are so subtle that newer programming languages, such as Java and C#, do not enable a class to derive from more than one base class.

Software Engineering Observation 24.3

Great care is required in the design of a system to use multiple inheritance properly; it should not be used when single inheritance and/or composition will do the job.

23.7 Multiple Inheritance (Cont.)

- A common problem with multiple inheritance is that each of the base classes might contain data members or member functions that have the same name.
- This can lead to ambiguity problems when you attempt to compile.
- Consider the multiple-inheritance example (Figs. 23.7-23.11).
- Class `Base1` (Fig. 23.7) contains one `protected` `int` data member—`value` (line 20), a constructor (lines 10–13) that sets `value` and `public` member function `getData` (lines 15–18) that returns `value`.


```
1  // Fig. 24.7: Base1.h
2  // Definition of class Base1
3  #ifndef BASE1_H
4  #define BASE1_H
5
6  // class Base1 definition
7  class Base1
8  {
9  public:
10    Base1( int parameterValue )
11    {
12       value = parameterValue;
13    } // end Base1 constructor
14
15    int getData() const
16    {
17       return value;
18    } // end function getData
19 protected: // accessible to derived classes
20    int value; // inherited by derived class
21 }; // end class Base1
22
23 #endif // BASE1_H
```

Fig. 24.7 | Demonstrating multiple inheritance—`Base1.h`.

23.7 Multiple Inheritance (Cont.) (Cont.)

- Class `Base2` (Fig. 23.8) is similar to class `Base1`, except that its `protected` data is a `char` named `letter` (line 20).
- Like class `Base1`, `Base2` has a `public` member function `getData`, but this function returns the value of `char` data member `letter`.


```
1  // Fig. 24.8: Base2.h
2  // Definition of class Base2
3  #ifndef BASE2_H
4  #define BASE2_H
5
6  // class Base2 definition
7  class Base2
8  {
9  public:
10    Base2( char characterData )
11    {
12       letter = characterData;
13    } // end Base2 constructor
14
15    char getData() const
16    {
17       return letter;
18    } // end function getData
19 protected: // accessible to derived classes
20    char letter; // inherited by derived class
21 }; // end class Base2
22
23 #endif // BASE2_H
```

Fig. 24.8 | Demonstrating multiple inheritance—`Base2.h`.

23.7 Multiple Inheritance (Cont.)

- Class `Derived` (Figs. 23.9–23.10) inherits from both class `Base1` and class `Base2` through *multiple inheritance*.
- Class `Derived` has a `private` data member of type `double` named `real` (Fig. 23.9, line 20), a constructor to initialize all the data of class `Derived` and a `public` member function `getReal` that returns the value of `double` variable `real`.
- To indicate *multiple inheritance* we follow the colon (`:`) after `class Derived` with a comma-separated list of base classes (line 13).
- In Fig. 23.10, notice that constructor `Derived` explicitly calls base-class constructors for each of its base classes—`Base1` and `Base2`—using the member-initializer syntax (line 9).


```
1  // Fig. 24.9: 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  #include "Base1.h"
9  #include "Base2.h"
10 using namespace std;
11
12 // class Derived definition
13 class Derived : public Base1, public Base2
14 {
15    friend ostream &operator<<( ostream &, const Derived & );
16 public:
17    Derived( int, char, double );
18    double getReal() const;
19 private:
20    double real; // derived class's private data
21 }; // end class Derived
22
23 #endif // DERIVED_H
```

Fig. 24.9 | Demonstrating multiple inheritance—`Derived.h`.


```
1  // Fig. 24.10: 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
13 {
14    return real;
15 } // end function getReal
16
17 // display all data members of Derived
18 ostream &operator<<( ostream &output, const Derived &derived )
19 {
20    output << "    Integer: " << derived.value << "\n  Character: "
21         << derived.letter << "\nReal number: " << derived.real;
22    return output; // enables cascaded calls
23 } // end operator<<
```

Fig. 24.10 | Demonstrating multiple inheritance—`Derived.cpp`.

23.7 Multiple Inheritance (Cont.)

- The *base-class constructors are called in the order that the inheritance is specified, not in the order in which their constructors are mentioned*; also, *if the base-class constructors are not explicitly called in the member-initializer list, their default constructors will be called implicitly.*
- The overloaded stream insertion operator (Fig. 23.10, lines 18–23) uses its second parameter—a reference to a `Derived` object—to display a `Derived` object's data.

23.7 Multiple Inheritance (Cont.)

- This operator function is a `friend` of `Derived`, so `operator<<` can directly access all of class `Derived`'s `protected` and `private` members, including the `protected` data member `value` (inherited from class `Base1`), `protected` data member `letter` (inherited from class `Base2`) and `private` data member `real` (declared in class `Derived`).
- Now let's examine the `main` function (Fig. 23.11) that tests the classes in Figs. 23.7–23.10.

23.7 Multiple Inheritance (Cont.)

- Line 11 creates `Base1` object `base1` and initializes it to the `int` value `10`.
- Line 12 creates `Base2` object `base2` and initializes it to the `char` value `'Z'`.
- Line 13 creates `Derived` object `derived` and initializes it to contain the `int` value `7`, the `char` value `'A'` and the `double` value `3.5`.


```
1  // Fig. 24.11: fig24_11.cpp
2  // Driver for multiple-inheritance example.
3  #include <iostream>
4  #include "Base1.h"
5  #include "Base2.h"
6  #include "Derived.h"
7  using namespace std;
8
9  int main()
10 {
11    Base1 base1( 10 ), *base1Ptr = 0; // create Base1 object
12    Base2 base2( 'Z' ), *base2Ptr = 0; // create Base2 object
13    Derived derived( 7, 'A', 3.5 ); // create Derived object
14
15    // print data members of base-class objects
16    cout << "Object base1 contains integer " << base1.getData()
17       << "\nObject base2 contains character " << base2.getData()
18       << "\nObject derived contains:\n" << derived << "\n\n";
19
```

Fig. 24.11 | Demonstrating multiple inheritance. (Part 1 of 3.)


```
20    // print data members of derived-class object
21    // scope resolution operator resolves getData ambiguity
22    cout << "Data members of Derived can be accessed individually:"
23       << "\n    Integer: " << derived.Base1::getData()
24       << "\n  Character: " << derived.Base2::getData()
25       << "\nReal number: " << derived.getReal() << "\n\n";
26    cout << "Derived can be treated as an object of either base class:\n";
27
28    // treat Derived as a Base1 object
29    base1Ptr = &derived;
30    cout << "base1Ptr->getData() yields " << base1Ptr->getData() << '\n';
31
32    // treat Derived as a Base2 object
33    base2Ptr = &derived;
34    cout << "base2Ptr->getData() yields " << base2Ptr->getData() << endl;
35 } // end main
```

Fig. 24.11 | Demonstrating multiple inheritance. (Part 2 of 3.)

```
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
```

Fig. 24.11 | Demonstrating multiple inheritance. (Part 3 of 3.)

23.7 Multiple Inheritance (Cont.)

- Lines 16-18 display each object's data values.
- For objects `base1` and `base2`, we invoke each object's `getData` member function.
- Even though there are *two* `getData` functions in this example, the calls are *not ambiguous*.
- In line 16, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`'s `getData` is called.
- In line 17, the compiler knows that `base2` is an object of class `Base2`, so class `Base2`'s `getData` is called.
- Line 18 displays the contents of object `derived` using the overloaded stream insertion operator.

23.7 Multiple Inheritance (Cont.)

- Lines 22–25 output the contents of object `derived` again by using the *get* member functions of class `Derived`.
- However, there is an *ambiguity* problem, because this object contains two `getData` functions, one inherited from class `Base1` and one inherited from class `Base2`.
- This problem is easy to solve by using the scope resolution operator.

23.7 Multiple Inheritance (Cont.)

- The expression `derived.Base1::getData()` gets the value of the variable inherited from class `Base1` (i.e., the `int` variable named `value`) and `derived.Base2::getData()` gets the value of the variable inherited from class `Base2` (i.e., the `char` variable named `letter`).
- The `double` value in `real` is printed without ambiguity with the call `derived.getReal()`—there are no other member functions with that name in the hierarchy.

23.7 Multiple Inheritance (Cont.)

- The *is-a relationships* of *single inheritance* also apply in *multiple-inheritance* relationships.
- To demonstrate this, line 29 assigns the address of object `derived` to the `Base1` pointer `base1Ptr`.
- This is allowed because an object of class `Derived` *is an* object of class `Base1`.
- Line 30 invokes `Base1` member function `getData` via `base1Ptr` to obtain the value of only the `Base1` part of the object `derived`.

23.7 Multiple Inheritance (Cont.)

- Line 33 assigns the address of object `derived` to the `Base2` pointer `base2Ptr`.
- This is allowed because an object of class `Derived` *is an* object of class `Base2`.
- Line 34 invokes `Base2` member function `getData` via `base2Ptr` to obtain the value of only the `Base2` part of the object `derived`.

23.8 Multiple Inheritance and `virtual` Base Classes

- In Section 23.7, we discussed *multiple inheritance*, the process by which one class inherits from *two or more* classes.
- Multiple inheritance is used, for example, in the C++ standard library to form class `basic_iostream` (Fig. 23.12).

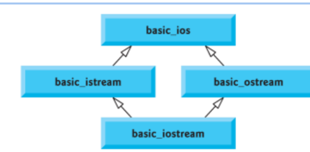

Fig. 24.12 | Multiple inheritance to form class `basic_iostream`.

23.8 Multiple Inheritance and `virtual` Base Classes (Cont.)

- Class `basic_ios` is the base class for both `basic_istream` and `basic_ostream`, each of which is formed with *single inheritance*.
- Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`.
- This enables class `basic_iostream` objects to provide the functionality of `basic_istream`s and `basic_ostream`s.
- In multiple-inheritance hierarchies, the inheritance described in Fig. 23.12 is referred to as diamond inheritance
- Because classes `basic_istream` and `basic_ostream` each inherit from `basic_ios`, a potential problem exists for `basic_iostream`.

23.8 Multiple Inheritance and `virtual` Base Classes (Cont.)

- Class `basic_iostream` could contain two copies of the members of class `basic_ios`—one inherited via class `basic_istream` and one inherited via class `basic_ostream`).
- Such a situation would be ambiguous and would result in a compilation error, because the compiler would not know which version of the members from class `basic_ios` to use.
- In this section, you'll see how using `virtual` base classes solves the problem of inheriting duplicate copies of an indirect base class.

23.8 Multiple Inheritance and `virtual` Base Classes (Cont.)

- Figure 23.13 demonstrates the *ambiguity* that can occur in *diamond inheritance*.
- Class `Base` (lines 8–12) contains pure `virtual` function `print` (line 11).
- Classes `DerivedOne` (lines 15–23) and `DerivedTwo` (lines 26–34) each publicly inherit from `Base` and override function `print`.
- Class `DerivedOne` and class `DerivedTwo-` each contain a base-class subobject—i.e., the members of class `Base` in this example.


```
1  // Fig. 24.13: fig24_13.cpp
2  // Attempting to polymorphically call a function that is
3  // multiply inherited from two base classes.
4  #include <iostream>
5  using namespace std;
6
7  // class Base definition
8  class Base
9  {
10 public:
11    virtual void print() const = 0; // pure virtual
12 }; // end class Base
13
14 // class DerivedOne definition
15 class DerivedOne : public Base
16 {
17 public:
18    // override print function
19    void print() const
20    {
21       cout << "DerivedOne\n";
22    } // end function print
23 }; // end class DerivedOne
```

Fig. 24.13 | Attempting to call a multiply inherited function polymorphically. (Part 1 of 4.)

```
24
25 // class DerivedTwo definition
26 class DerivedTwo : public Base
27 {
28 public:
29    // override print function
30    void print() const
31    {
32       cout << "DerivedTwo\n";
33    } // end function print
34 }; // end class DerivedTwo
35
36 // class Multiple definition
37 class Multiple : public DerivedOne, public DerivedTwo
38 {
39 public:
40    // qualify which version of function print
41    void print() const
42    {
43       DerivedTwo::print();
44    } // end function print
45 }; // end class Multiple
46
```

Fig. 24.13 | Attempting to call a multiply inherited function polymorphically. (Part 2 of 4.)


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

Fig. 24.13 | Attempting to call a multiply inherited function polymorphically. (Part 3 of 4.)

Microsoft Visual C++ compiler error message:

```
c:\cpphtp8_examples\ch25\Fig24_13\fig24_13.cpp(54) : error C2594: '=' :
   ambiguous conversions from 'Multiple *' to 'Base *'
```

GNU C++ compiler error message:

```
fig24_13.cpp: In function ‘int main()’:
fig24_13.cpp:54: error: ‘Base’ is an ambiguous base of ‘Multiple’
```

Fig. 24.13 | Attempting to call a multiply inherited function polymorphically. (Part 4 of 4.)

23.8 Multiple Inheritance and `virtual` Base Classes (Cont.)

- Class `Multiple` (lines 37–45) inherits from both class `DerivedOne` and class `DerivedTwo`.
- In class `Multiple`, function `print` is overridden to call `DerivedTwo`'s `print` (line 43).
- Notice that we must *qualify* the `print` call with the class name `DerivedTwo` to specify which version of `print` to call.
- Function `main` (lines 47–61) declares objects of classes `Multiple` (line 49), `Derived-One` (line 50) and `DerivedTwo` (line 51).

23.8 Multiple Inheritance and `virtual` Base Classes (Cont.)

- Line 52 declares an array of `Base *` pointers.
- Each array element is initialized with the address of an object (lines 54–56).
- An error occurs when the address of `both`—an object of class `Multiple`—is assigned to `array[ 0 ]`.
- The object `both` actually contains two subobjects of type `Base`, so the compiler does not know which subobject the pointer `array[ 0 ]` should point to, and it generates a compilation error indicating an *ambiguous conversion*.

23.8 Multiple Inheritance and `virtual` Base Classes (Cont.)

- The problem of *duplicate subobjects* is resolved with `virtual` inheritance.
- When a base class is inherited as `virtual`, only *one* subobject will appear in the derived class—a process called `virtual` base-class inheritance.
- Figure 23.14 revises the program of Fig. 23.13 to use a `virtual` base class.


```cpp
// Fig. 24.14: fig24_14.cpp
// Using virtual base classes.
#include <iostream>
using namespace std;

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

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

```

Fig. 24.14 | Using `virtual` base classes. (Part 1 of 3.)


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

// class Multiple definition
class Multiple : public DerivedOne, public DerivedTwo
{
public:
   // qualify which version of function print
   void print() const
   {
      DerivedTwo::print();
   } // end function print
}; // end Multiple class

```

Fig. 24.14 | Using `virtual` base classes. (Part 2 of 3.)

```
46 int main()
47 {
48    Multiple both; // instantiate Multiple object
49    DerivedOne one; // instantiate DerivedOne object
50    DerivedTwo two; // instantiate DerivedTwo object
51
52    // declare array of base-class pointers and initialize
53    // each element to a derived-class type
54    Base *array[ 3 ];
55    array[ 0 ] = &both;
56    array[ 1 ] = &one;
57    array[ 2 ] = &two;
58
59    // polymorphically invoke function print
60    for ( int i = 0; i < 3; ++i )
61       array[ i ]->print();
62 } // end main
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 24.14 | Using `virtual` base classes. (Part 3 of 3.)

23.8 Multiple Inheritance and `virtual` Base Classes (Cont.)

- The key change is that classes `DerivedOne` (line 14) and `DerivedTwo` (line 25) each inherit from `Base` by specifying `virtual public Base`.
- Since both classes inherit from `Base`, they each contain a `Base` *subobject*.
- The benefit of *`virtual` inheritance* is not clear until class `Multiple` inherits from `DerivedOne`- and `DerivedTwo` (line 36).

23.8 Multiple Inheritance and `virtual` Base Classes (Cont.)

- Since each of the base classes used *`virtual` inheritance* to inherit class `Base`'s members, the compiler ensures that only one `Base` subobject is inherited into class `Multiple`.
- This eliminates the ambiguity error generated by the compiler in Fig. 23.13.
- The compiler now allows the implicit conversion of the derived-class pointer (`&both`) to the base-class pointer `array[ 0 ]` in line 55 in `main`.
- The `for` statement in lines 60–61 polymorphically calls `print` for each object.