C++ is fun – Part Eight

at Turbine/Warner Bros.!

Russell Hanson

Go over projects first!

For all you Black Jack-programming fiends:

http://courses.ischool.berkeley.edu/i90/f11/resources/chapter09/blackjack.py

```
# Blackjack
# From 1 to 7 players compete against a dealer

import cards, games

class BJ_Card(cards.Card):
    """ A Blackjack Card. """
    ACE_VALUE = 1

    @property
    def value(self):
        if self.is_face_up:
            v = BJ_Card.RANKS.index(self.rank) + 1
            if v > 10:
                v = 10
        else:
            v = None
        return v
…
```

Overview of Standard Library headers

Standard Library header	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 4.9 and is discussed in more detail in Chapter 15, Stream Input/Output.
`<cmath>`	Contains function prototypes for math library functions (Section 6.3).
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 6.7; Chapter 11, Operator Overloading; Class `string`; Chapter 16, Exception Handling: A Deeper Look; Chapter 21, Bits, Characters, C Strings and `structs`; and Appendix F, C Legacy Code Topics.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header is used in Section 6.7.

Standard Library header	Explanation
`<vector>`, `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these headers in Chapter 22, Standard Template Library (STL).
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 21, Bits, Characters, C Strings and `structs`.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 11, Operator Overloading; Class `string`.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 13.8.
`<exception>`, `<stdexcept>`	These headers contain classes that are used for exception handling (discussed in Chapter 16, Exception Handling: A Deeper Look).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 16, Exception Handling: A Deeper Look.
`<fstream>`	Contains function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 17, File Processing).
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 18, Class `string` and String Stream Processing).
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class `string` and String Stream Processing).
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 22.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 22.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 22.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions.
`<locale>`	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).

Function Templates

Overloaded functions are normally used to perform similar operations that involve different program logic on different data types. If the program logic and operations are *identical* for each data type, overloading may be performed more compactly and conveniently by using **function templates**. You write a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate **function template specializations** to handle each type of call appropriately. Thus, defining a single function template essentially defines a whole family of overloaded functions.

Figure 6.26 defines a `maximum` function template (lines 3–17) that determines the largest of three values. All function template definitions begin with the **`template` keyword** (line 3) followed by a **template parameter list** to the function template enclosed in angle brackets (< and >). Every parameter in the template parameter list (often referred to as a **formal type parameter**) is preceded by keyword `typename` or keyword `class` (they are synonyms in this context). The formal type parameters are placeholders for fundamental types or user-defined types. These placeholders, in this case, `T`, are used to specify the types of the function's parameters (line 4), to specify the function's return type (line 4) and to declare variables within the body of the function definition (line 6). A function template is defined like any other function, but uses the formal type parameters as placeholders for actual data types.

```
1  // Fig. 6.26: maximum.h
2  // Definition of function template maximum.
3  template < typename T >  // or template< typename T >
4  T maximum( T value1, T value2, T value3 )
5  {
6     T maximumValue = value1; // assume value1 is maximum
7
8     // determine whether value2 is greater than maximumValue
9     if ( value2 > maximumValue )
10       maximumValue = value2;
11
12    // determine whether value3 is greater than maximumValue
13    if ( value3 > maximumValue )
14       maximumValue = value3;
15
16    return maximumValue;
17 } // end function template maximum
```

Fig. 6.26 | Function template `maximum` header.

The function template declares a single formal type parameter `T` (line 3) as a placeholder for the type of the data to be tested by function `maximum`. The name of a type parameter must be unique in the template parameter list for a particular template definition. When the compiler detects a `maximum` invocation in the program source code, the *type of* the data passed to `maximum` is substituted for `T` throughout the template definition, and C++ creates a complete function for determining the maximum of three values of the specified data type—all three must have the same type, since we use only one type parameter in this example. Then the newly created function is compiled. Thus, templates are a means of code generation.

Figure 6.27 uses the `maximum` function template to determine the largest of three `int` values, three `double` values and three `char` values, respectively (lines 17, 27 and 37). Separate functions are created as a result of the calls in lines 17, 27 and 37—expecting three `int` values, three `double` values and three `char` values, respectively. The function template specialization created for type `int` replaces each occurrence of `T` with `int` as follows:

```
// Definition of function template maximum.
template < class T >  // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
  T maximumValue = value1; // assume value1 is maximum

  // determine whether value2 is greater than maximumValue
  if ( value2 > maximumValue )
    maximumValue = value2;

  // determine whether value3 is greater than maximumValue
  if ( value3 > maximumValue )
    maximumValue = value3;

  return maximumValue;
} // end function template maximum
```

```
#include <iostream>
#include "maximum.h" // include definition of function template
maximum
using namespace std;

int main()
{
  // demonstrate maximum with int values
  int int1, int2, int3;

  cout << "Input three integer values: ";
  cin >> int1 >> int2 >> int3;

  // invoke int version of maximum
  cout << "The maximum integer value is: "
    << maximum( int1, int2, int3 );

  // demonstrate maximum with double values
  double double1, double2, double3;

  cout << "\n\nInput three double values: ";
  cin >> double1 >> double2 >> double3;

  // invoke double version of maximum
  cout << "The maximum double value is: "
    << maximum( double1, double2, double3 );

  // demonstrate maximum with char values
  char char1, char2, char3;

  cout << "\n\nInput three characters: ";
  cin >> char1 >> char2 >> char3;

  // invoke char version of maximum
  cout << "The maximum character value is: "
    << maximum( char1, char2, char3 ) << endl;
} // end main
```

Math library fun-ctions

Function	Description	Example
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential function e^x	`exp( 1.0 )` is `2.718282` `exp( 2.0 )` is `7.389056`

`fabs( x )`	absolute value of *x*	`fabs( 5.1 )` is `5.1`
		`fabs( 0.0 )` is `0.0`
		`fabs( -8.76 )` is `8.76`

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 2.6, 1.2 )` is `0.2`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`log10( x )`	logarithm of *x* (base 10)	`log10( 10.0 )` is `1.0` `log10( 100.0 )` is `2.0`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 )` is `128` `pow( 9, .5 )` is `3`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`
`sqrt( x )`	square root of *x* (where *x* is a nonnegative value)	`sqrt( 9.0 )` is `3.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

Recursion

For some problems, it's useful to have functions *call themselves*. A **recursive function** is a function that calls itself, either directly, or indirectly (through another function). [*Note:* Although many compilers allow function `main` to call itself, Section 3.6.1, paragraph 3, and Section 5.2.2, paragraph 9, of the C++ standard document indicate that `main` should not be called within a program or recursively. Its sole purpose is to be the starting point for program execution.] Recursion is an important topic discussed at length in upper-level computer science courses. This section and the next present simple examples of recursion. Figure 6.33 (at the end of Section 6.21) summarizes the extensive recursion examples and exercises in the book.

We first consider recursion conceptually, then examine two programs containing recursive functions. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function knows how to solve only the *simplest case(s)*, or so-called **base case(s)**. If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece *must* resemble the original problem, but be a slightly simpler or smaller version. This new problem looks like the original, so the function calls a copy of itself to work on the smaller problem—this is referred to as a **recursive call** and is also called the **recursion step**. The recursion step often includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form the result passed back to the original caller, possibly `main`.

The recursion step executes while the original call to the function is still "open," i.e., it has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each new subproblem with which the function is called into two conceptual pieces. In order for the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case. At that point, the function recognizes the base case and returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original call eventually returns the final result to `main`. This sounds quite exotic compared to the kind of problem solving we've been using to this point. As an example of these concepts at work, let's write a recursive program to perform a popular mathematical calculation.

The factorial of a nonnegative integer n, written $n!$ (and pronounced "n factorial"), is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

with 1! equal to 1, and 0! defined to be 1. For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.

The factorial of an integer, `number`, greater than or equal to 0, can be calculated **iteratively** (nonrecursively) by using a `for` statement as follows:

```
factorial = 1;
for ( int counter = number; counter >= 1; --counter )
   factorial *= counter;
```

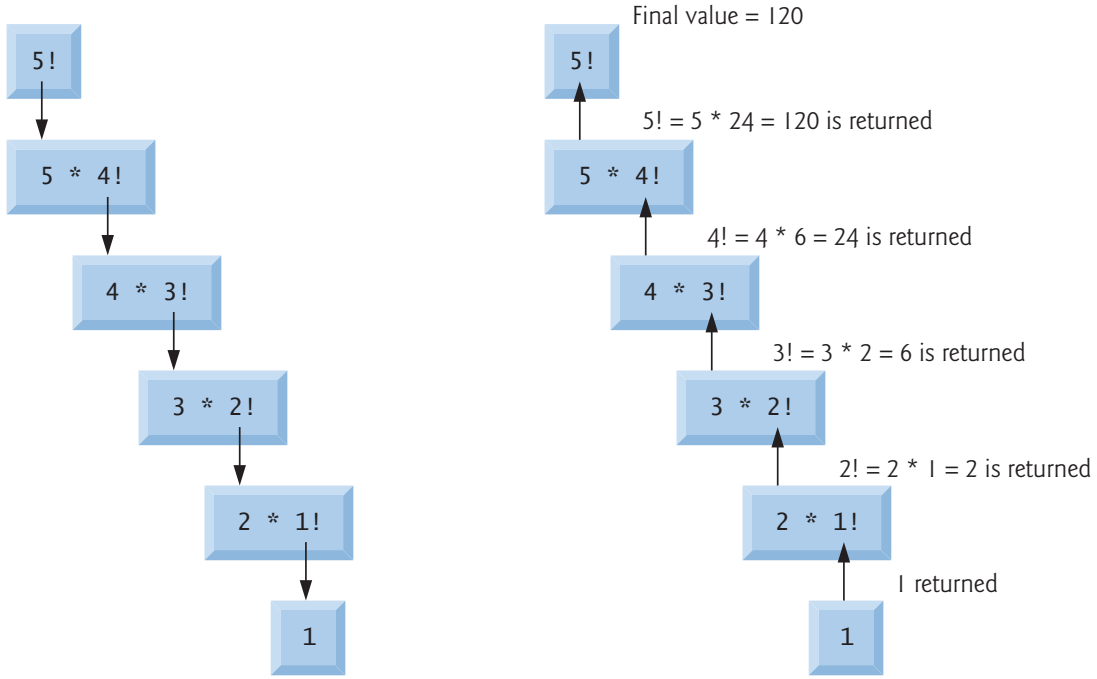
A *recursive* definition of the factorial function is arrived at by observing the following algebraic relationship:

$$n! = n \cdot (n-1)!$$

For example, 5! is clearly equal to 5 * 4! as is shown by the following:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in Fig. 6.28, which illustrates how the succession of recursive calls proceeds until 1! is evaluated to be 1, terminating the recursion. Figure 6.28(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.



(a) Procession of recursive calls

(b) Values returned from each recursive call

```
// Demonstrating the recursive function factorial.
#include <iostream>
#include <iomanip>
using namespace std;

unsigned long factorial( unsigned long ); // function prototype

int main()
{
  // calculate the factorials of 0 through 10
  for ( int counter = 0; counter <= 10; ++counter )
    cout << setw( 2 ) << counter << "! = " << factorial( counter )
      << endl;
} // end main

// recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
  if ( number <= 1 ) // test for base case
    return 1; // base cases: 0! = 1 and 1! = 1
  else // recursion step
    return number * factorial( number - 1 );
} // end function factorial
```

Another Example Recursion

The Fibonacci series

```
0, 1, 1, 2, 3, 5, 8, 13, 21, …
```

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

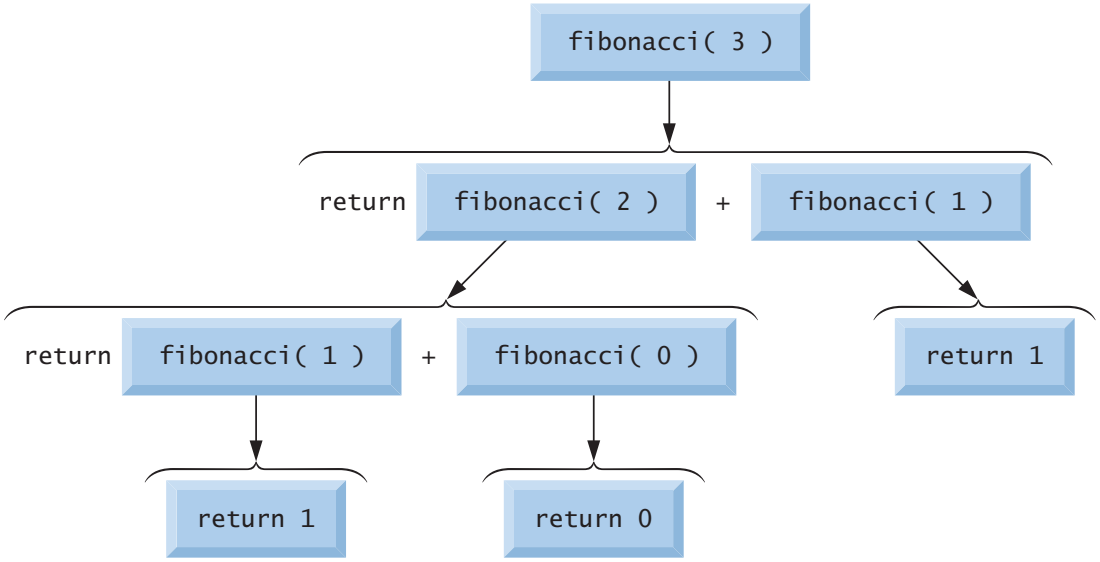
The series occurs in nature and, in particular, describes a form of spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618…. This number, too, frequently occurs in nature and has been called the **golden ratio** or the **golden mean.** Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden mean length/width ratio.

The Fibonacci series can be defined recursively as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)
```

The program of Fig. 6.30 calculates the *n*th Fibonacci number recursively by using function `fibonacci`. Fibonacci numbers tend to become large quickly, although slower than factorials do. Therefore, we chose the data type `unsigned long` for the parameter type and the return type in function `fibonacci`. Figure 6.30 shows the execution of the program, which displays the Fibonacci values for several numbers.

Figure 6.31 shows how function `fibonacci` would evaluate `fibonacci(3)`. This figure raises some interesting issues about the order in which C++ compilers evaluate the operands of operators. This is a separate issue from the order in which operators are applied to their operands, namely, the order dictated by the rules of operator precedence and associativity. Figure 6.31 shows that evaluating `fibonacci(3)` causes two recursive calls, namely, `fibonacci(2)` and `fibonacci(1)`. In what order are these calls made?



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

unsigned long fibonacci( unsigned long ); // function prototype

int main()
{
  // calculate the fibonacci values of 0 through 10
  for ( int counter = 0; counter <= 10; ++counter )
    cout << "fibonacci( " << counter << " ) = "
      << fibonacci( counter ) << endl;

  // display higher fibonacci values
  cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
  cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
  cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
} // end main

// recursive function fibonacci
unsigned long fibonacci( unsigned long number )
{
  if ( ( number == 0 ) || ( number == 1 ) ) // base cases
    return number;
  else // recursion step
    return fibonacci( number - 1 ) + fibonacci( number - 2 );
} // end function fibonacci
```

Inheritance

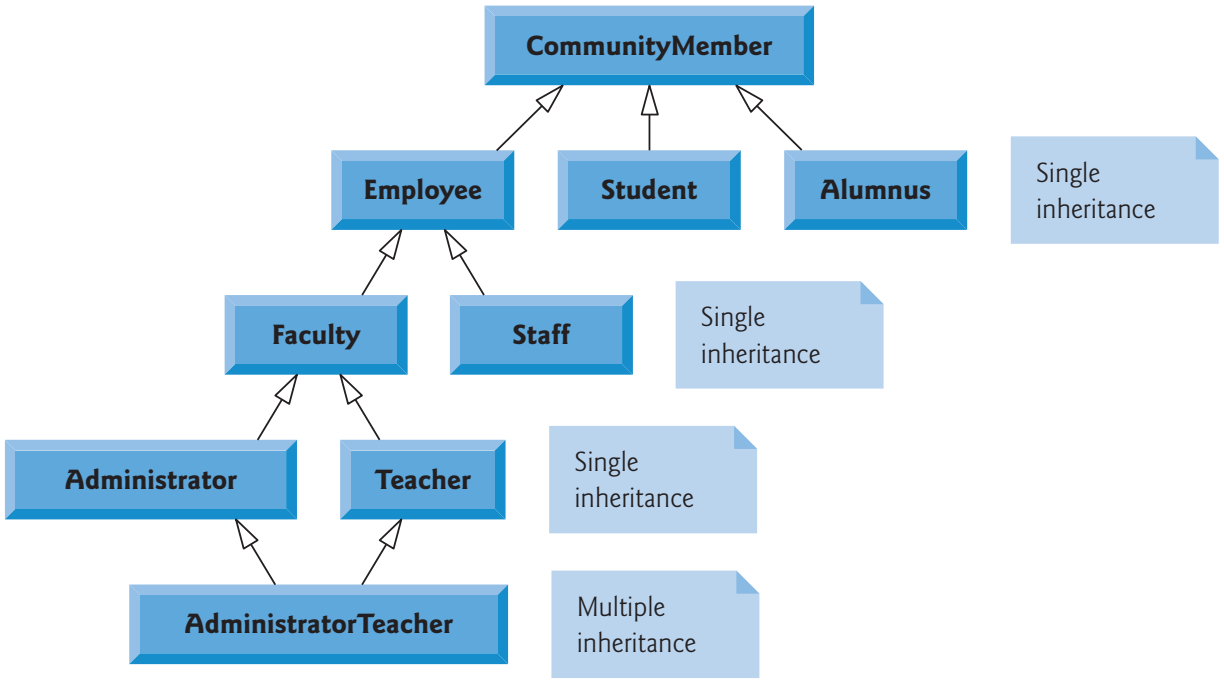
This chapter continues our discussion of object-oriented programming (OOP) by introducing **inheritance**—a form of software reuse in which you create a class that absorbs an existing class's capabilities, then *customizes* or enhances them. Software reuse saves time during program development by taking advantage of proven, high-quality software.

When creating a class, instead of writing completely new data members and member functions, you can specify that the new class should **inherit** the members of an existing class. This existing class is called the **base class**, and the new class is called the **derived class**. Other programming languages, such as Java and C#, refer to the base class as the **superclass** and the derived class as the **subclass**. A derived class represents a *more specialized* group of objects.

C++ offers `public`, `protected` and `private` inheritance. In this chapter, we concentrate on `public` inheritance and briefly explain the other two. *With `public` inheritance, every object of a derived class is also an object of that derived class's base class.* However, base-class objects are *not* objects of their derived classes. For example, if we have `Vehicle` as a base class and `Car` as a derived class, then all `Car`s are `Vehicle`s, but not all `Vehicle`s are `Car`s—for example, a `Vehicle` could also be a `Truck` or a `Boat`.

We distinguish between the ***is-a* relationship** and the *has-a* relationship. The *is-a* relationship represents inheritance. In an *is-a* relationship, an object of a derived class also can be treated as an object of its base class—for example, a `Car` *is a* `Vehicle`, so any attributes and behaviors of a `Vehicle` are also attributes and behaviors of a `Car`. By contrast, the *has-a* relationship represents *composition*, which was discussed in Chapter 10. In a *has-a* relationship, an object *contains* one or more objects of other classes as members. For example, a `Car` has many components—it *has a* steering wheel, *has a* brake pedal, *has a* transmission, etc.

Base class	Derived classes
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `Triangle`, `Rectangle`, `Sphere`, `Cube`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Employee`	`Faculty`, `Staff`
`Account`	`CheckingAccount`, `SavingsAccount`



`Teachers`. Some `Administrators`, however, are also `Teachers`. We've used *multiple inheritance* to form class `AdministratorTeacher`. With **single inheritance**, a class is derived from *one* base class. With **multiple inheritance**, a derived class inherits from *two or more* (possibly unrelated) base classes. We discuss multiple inheritance in Chapter 24, Other Topics.

Each arrow in the hierarchy (Fig. 12.2) represents an *is-a* relationship. For example, as we follow the arrows in this class hierarchy, we can state "an `Employee` *is a* `CommunityMember`" and "a `Teacher` *is a* `Faculty` member." `CommunityMember` is the **direct base class** of `Employee`, `Student` and `Alumnus`. In addition, `CommunityMember` is an **indirect base class** of all the other classes in the diagram. An indirect base class is inherited from two or more levels up the class hierarchy.

Starting from the bottom of the diagram, you can follow the arrows upwards and apply the *is-a* relationship to the topmost base class. For example, an `AdministratorTeacher` *is an* `Administrator`, *is a* `Faculty` member, *is an* `Employee` and *is a* `CommunityMember`.

Shape Class Hierarchy

Now consider the `Shape` inheritance hierarchy in Fig. 12.3. This hierarchy begins with base class `Shape`. Classes `TwoDimensionalShape` and `ThreeDimensionalShape` derive from base class `Shape`—a `Shape` *is a* `TwoDimensionalShape` or *is a* `ThreeDimensionalShape`. The third level of this hierarchy contains *more specific* types of `TwoDimensionalShapes` and `ThreeDimensionalShapes`. As in Fig. 12.2, we can follow the arrows from the bottom of the diagram upwards to the topmost base class in this hierarchy to identify several *is-a* relationships. For instance, a `Triangle` *is a* `TwoDimensionalShape` and *is a* `Shape`, while a `Sphere` *is a* `ThreeDimensionalShape` and *is a* `Shape`.

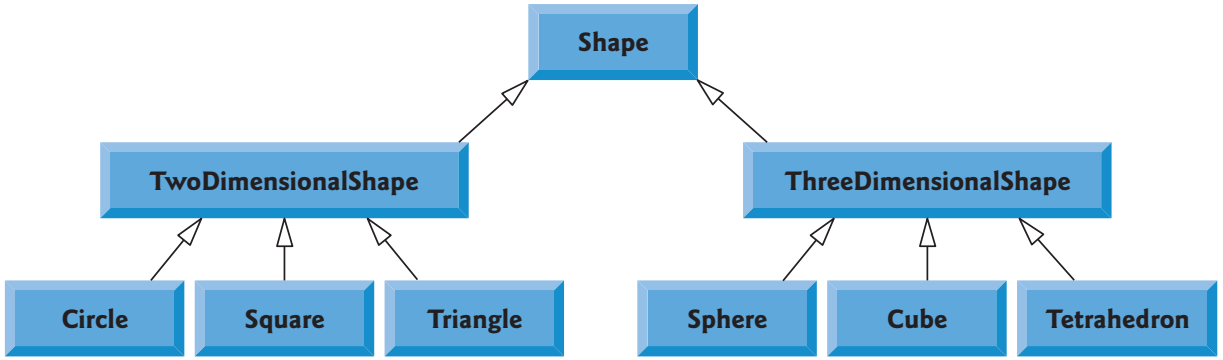


Fig. 12.3 | Inheritance hierarchy for Shapes.

To specify that class `TwoDimensionalShape` (Fig. 12.3) is derived from (or inherits from) class `Shape`, class `TwoDimensionalShape`'s definition could begin as follows:

```
class TwoDimensionalShape : public Shape
```

This is an example of **`public` inheritance**, the most commonly used form. We'll also discuss **`private` inheritance** and **`protected` inheritance** (Section 12.6). With all forms of inheritance, `private` members of a base class are *not* accessible directly from that class's derived classes, but these `private` base-class members are still inherited (i.e., they're still considered parts of the derived classes). With `public` inheritance, all other base-class mem-

class. In this section, we introduce the access specifier **`protected`**.

Using `protected` access offers an intermediate level of protection between `public` and `private` access. A base class's `protected` members can be accessed within the body of that base class, by members and `friend`s of that base class, and by members and `friend`s of any classes derived from that base class.

Derived-class member functions can refer to `public` and `protected` members of the base class simply by using the member names. When a derived-class member function *redefines* a base-class member function, the base-class member can still be accessed from the derived class by preceding the base-class member name with the base-class name and the scope resolution operator (`::`). We discuss accessing redefined members of the base

Why use inheritance?

```
1  // Fig. 12.10: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  #include "CommissionEmployee.h" // CommissionEmployee class declaration
9  using namespace std;
10
11 class BasePlusCommissionEmployee : public CommissionEmployee
12 {
13 public:
14    BasePlusCommissionEmployee( const string &, const string &,
15       const string &, double = 0.0, double = 0.0, double = 0.0 );
16
17    void setBaseSalary( double ); // set base salary
18    double getBaseSalary() const; // return base salary
19
20    double earnings() const; // calculate earnings
21    void print() const; // print BasePlusCommissionEmployee object
22 private:
23    double baseSalary; // base salary
24 }; // end class BasePlusCommissionEmployee
25
26 #endif
```

Why?

We literally *copied* code from class `CommissionEmployee` and *pasted* it into class `BasePlusCommissionEmployee`, then modified class `BasePlusCommissionEmployee` to include a base salary and member functions that manipulate the base salary. This *copy-and-paste approach* is error prone and time consuming.

Software Engineering Observation 12.1

Copying and pasting code *from one class to another can spread many physical* copies of the same code *and can spread* errors *throughout a system, creating a code-maintenance nightmare. To avoid duplicating code (and possibly errors), use* inheritance*, rather than the "copy-and-paste" approach, in situations where you want one class to "absorb" the data members and member functions of another class.*

Software Engineering Observation 12.2

With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, you need to make the changes only *in the base class—derived classes then inherit the changes. Without inheritance, changes would need to be made to* all *the source code files that contain a copy of the code in question.*

Why use inheritance? -- Just change one function

```
84
85  // calculate earnings
86  double CommissionEmployee::earnings() const
87  {
88     return commissionRate * grossSales;
89  } // end function earnings
90
91  // print CommissionEmployee object
92  void CommissionEmployee::print() const
93  {
94     cout << "commission employee: " << firstName << ' ' << lastName
95        << "\nsocial security number: " << socialSecurityNumber
96        << "\ngross sales: " << grossSales
97        << "\ncommission rate: " << commissionRate;
98  } // end function print
```

```
32  // calculate earnings
33  double BasePlusCommissionEmployee::earnings() const
34  {
35     // derived class cannot access the base class's private data
36     return baseSalary + ( commissionRate * grossSales );
37  } // end function earnings
38
```

```
32  // calculate earnings
33  double BasePlusCommissionEmployee::earnings() const
34  {
35     // derived class cannot access the base class's private data
36     return baseSalary + ( commissionRate * grossSales );
37  } // end function earnings
38
39  // print BasePlusCommissionEmployee object
40  void BasePlusCommissionEmployee::print() const
41  {
42     // derived class cannot access the base class's private data
43     cout << "base-salaried commission employee: " << firstName << ' '
44        << lastName << "\nsocial security number: " << socialSecurityNumber
45        << "\ngross sales: " << grossSales
46        << "\ncommission rate: " << commissionRate
47        << "\nbase salary: " << baseSalary;
48  } // end function print
```

```
C:\chhttp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(36) :
   error C2248: 'CommissionEmployee::commissionRate' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhttp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(36) :
   error C2248: 'CommissionEmployee::grossSales' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhttp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(43) :
   error C2248: 'CommissionEmployee::firstName' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhttp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(44) :
   error C2248: 'CommissionEmployee::lastName' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhttp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(44) :
   error C2248: 'CommissionEmployee::socialSecurityNumber' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhttp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(45) :
   error C2248: 'CommissionEmployee::grossSales' :
   cannot access private member declared in class 'CommissionEmployee'
```

Defining Base Class `CommissionEmployee` *with* `protected` *Data*

Class `CommissionEmployee` (Fig. 12.12) now declares data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as `protected` (lines 32–37) rather than `private`. The member-function implementations are identical to those in Fig. 12.5, so `CommissionEmployee.cpp` is not shown here.

```
1  // Fig. 12.12: CommissionEmployee.h
2  // CommissionEmployee class definition with protected data.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using namespace std;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 protected:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

Class Exercise – Check the folder “CommissionEmployee” on Google Drive

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Homework Exercises (Pick 1)

1) Download and install either
http://www.appgamekit.com/ (2D only, but simpler)
OR
http://www.ogre3d.org/ . Compile, link, and run one of the sample programs.

2) Implement a simple Tic Tac Toe "AI" strategy. Some sample implementations of the game are at the following two links:
http://courses.ischool.berkeley.edu/i90/f11/resources/chapter06/tic-tac-toe.py
http://en.literateprograms.org/Tic_Tac_Toe_(Python)

3) Show the value of x after each of the following statements is performed:

```
a)  x = fabs( 7.5 )
b)  x = floor( 7.5 )
c)  x = fabs( 0.0 )
d)  x = ceil( 0.0 )
e)  x = fabs( -6.4 )
f)  x = ceil( -6.4 )
g)  x = ceil( -fabs( -8 + floor( -5.5 ) ) )
```

4)

Self-Review Exercises

12.1 Fill in the blanks in each of the following statements:

a) _____ is a form of software reuse in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.

b) A base class's _____ members can be accessed in the base-class definition, in derived-class definitions and in `friend`s of the base class its derived classes.

c) In a(n) _____ relationship, an object of a derived class also can be treated as an object of its base class.

d) In a(n) _____ relationship, a class object has one or more objects of other classes as members.

e) In single inheritance, a class exists in a(n) _____ relationship with its derived classes.

f) A base class's _____ members are accessible within that base class and anywhere that the program has a handle to an object of that class or one of its derived classes.

g) A base class's `protected` access members have a level of protection between those of `public` and _____ access.

h) C++ provides for _____, which allows a derived class to inherit from many base classes, even if the base classes are unrelated.

i) When an object of a derived class is instantiated, the base class's _____ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.

j) When deriving a class from a base class with `public` inheritance, `public` members of the base class become _____ members of the derived class, and `protected` members of the base class become _____ members of the derived class.

k) When deriving a class from a base class with `protected` inheritance, `public` members of the base class become _____ members of the derived class, and `protected` members of the base class become _____ members of the derived class.

12.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Base-class constructors are not inherited by derived classes.

b) A *has-a* relationship is implemented via inheritance.

c) A `Car` class has an *is-a* relationship with the `SteeringWheel` and `Brakes` classes.

d) Inheritance encourages the reuse of proven high-quality software.

e) When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.