Illustrations List *(Main Page)*

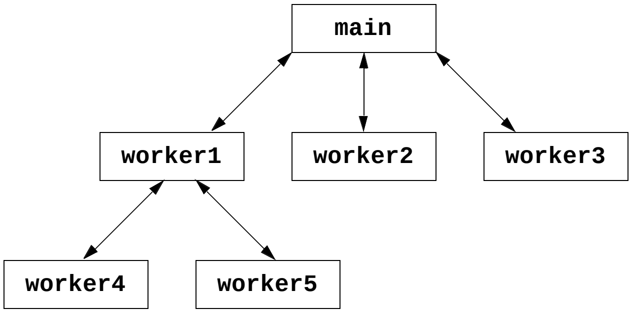


Fig. 3.1 Hierarchical boss function/worker function relationship.

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 `2.71828`
		`exp( 2.0 )` is `7.38906`

`fabs( x )`	absolute value of *x*	if `x > 0` then `abs( x )` is `x`
		if `x = 0` then `abs( x )` is `0.0`
		if `x < 0` then `abs( x )` is `x`

`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( 13.657, 2.333 )` is `1.992`

`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)	`log( 10.0 )` is `1.0`
		`log( 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*	`sqrt( 900.0 )` is `30.0`
		`sqrt( 9.0 )` is `3.0`

`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

Fig. 3.2 Commonly used math library functions.

```cpp
// Fig. 3.3: fig03_03.cpp
// Creating and using a programmer-defined function
#include <iostream.h>

int square( int );   // function prototype

int main()
{
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";

   cout << endl;
   return 0;
}

// Function definition
int square( int y )
{
   return y * y;
}
```

```
1  4  9  16  25  36  49  64  81  100
```

Fig. 3.3 Creating and using a programmer-defined function.

```cpp
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three integers
#include <iostream.h>

int maximum( int, int, int );   // function prototype

int main()
{
   int a, b, c;

   cout << "Enter three integers: ";
   cin >> a >> b >> c;
```

Fig. 3.4 Programmer-defined `maximum` function (part 1 of 2).

```cpp

   // a, b and c below are arguments to
   // the maximum function call
   cout << "Maximum is: " << maximum( a, b, c ) << endl;

   return 0;
}

// Function maximum definition
// x, y and z below are parameters to
// the maximum function definition
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 92 35 14
Maximum is: 92
```

```
Enter three integers: 45 19 98
Maximum is: 98
```

Fig. 3.4 Programmer-defined `maximum` function (part 2 of 2).

Data types	
`long double`	
`double`	
`float`	
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`short`	
`char`	

Fig. 3.5 Promotion hierarchy for built-in data types.

Standard library header file	**Explanation**
Old-style header files (used early in the book)	
`<assert.h>`	Contains macros and information for adding diagnostics that aid program debugging. The new version of this header file is `<cassert>`.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. The new version of this header file is `<cctype>`.

Fig. 3.6 Standard library header files (part 1 of 3).

Standard library header file	Explanation
`<float.h>`	Contains the floating-point size limits of the system. The new version of this header file is `<cfloat>`.
`<limits.h>`	Contains the integral size limits of the system. The new version of this header file is `<climits>`.
`<math.h>`	Contains function prototypes for math library functions. The new version of this header file is `<cmath>`.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions and information used by them. The new version of this header file is `<cstdio>`.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers, and various other utility functions. The new version of this header file is `<cstdlib>`.
`<string.h>`	Contains function prototypes for C-style string processing functions. The new version of this header file is `<cstring>`.
`<time.h>`	Contains function prototypes and types for manipulating the time and date. The new version of this header file is `<ctime>`.
`<iostream.h>`	Contains function prototypes for the standard input and standard output functions. The new version of this header file is `<iostream>`.
`<iomanip.h>`	Contains function prototypes for the stream manipulators that enable formatting of streams of data. The new version of this header file is `<iomanip>`.
`<fstream.h>`	Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 14). The new version of this header file is `<fstream>`.
New-style header files (used later in the book)	
`<utility>`	Contains classes and functions that are used by many standard library header files.
`<vector>`, `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	The header files contain classes that implement the standard library containers. Containers are use to store data during a program's execution. We discuss these header files in the chapter entitled "The Standard Template Library."
`<functional>`	Contains classes and functions used by algorithms of the standard library.
`<memory>`	Contains classes and functions used by the standard library to allocate memory to the standard library containers.
`<iterator>`	Contains classes for manipulating data in the standard library containers.
`<algorithm>`	Contains functions for manipulating data in the standard library containers.

`<exception>`
`<stdexcept>`
	These header files contain classes that are used for exception handling (discussed in Chapter 13).

`<string>`	Contains the definition of class `string` from the standard library (discussed in Chapter 19, "Strings").
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 14).

Fig. 3.6 Standard library header files (part 2 of 3).

Standard library header file	Explanation
`<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.).
`<limits>`	Contains a class for defining the numerical data type limits on each computer platform.
`<typeinfo>`	Contains classes for run-time type identification (determining data types at execution time).

Fig. 3.6 Standard library header files (part 3 of 3).

```
// Fig. 3.7: fig03_07.cpp
// Shifted, scaled integers produced by 1 + rand() % 6
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

int main()
{
   for ( int i = 1; i <= 20; i++ ) {
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      if ( i % 5 == 0 )
         cout << endl;
   }

   return 0;
}
```

```
         5         5         3         5         5
         2         4         2         5         5
         5         3         2         2         1
         5         1         4         6         4
```

Fig. 3.7 Shifted, scaled integers produced by `1 + rand() % 6`.

```
// Fig. 3.8: fig03_08.cpp
// Roll a six-sided die 6000 times
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

int main()
{
   int frequency1 = 0, frequency2 = 0,
       frequency3 = 0, frequency4 = 0,
       frequency5 = 0, frequency6 = 0,
       face;

   for ( int roll = 1; roll <= 6000; roll++ ) {
      face = 1 + rand() % 6;

      switch ( face ) {
         case 1:
            ++frequency1;
            break;
         case 2:
            ++frequency2;
            break;
         case 3:
            ++frequency3;
            break;
         case 4:
            ++frequency4;
            break;
         case 5:
            ++frequency5;
            break;
         case 6:
            ++frequency6;
            break;
         default:
            cout << "should never get here!";
      }
   }

   cout << "Face" << setw( 13 ) << "Frequency"
        << "\n   1" << setw( 13 ) << frequency1
        << "\n   2" << setw( 13 ) << frequency2
        << "\n   3" << setw( 13 ) << frequency3
        << "\n   4" << setw( 13 ) << frequency4
        << "\n   5" << setw( 13 ) << frequency5
        << "\n   6" << setw( 13 ) << frequency6 << endl;

   return 0;
}
```

Fig. 3.8 Rolling a six-sided die 6000 times (part 1 of 2).

```
Face    Frequency
1             987
2             984
3            1029
4             974
5            1004
6          1022
```

Fig. 3.8 Rolling a six-sided die 6000 times (part 2 of 2).

```
// Fig. 3.9: fig03_09.cpp
// Randomizing die-rolling program
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

int main()
{
   unsigned seed;

   cout << "Enter seed: ";
   cin >> seed;
   srand( seed );

   for ( int i = 1; i <= 10; i++ ) {
      cout << setw( 10 ) << 1 + rand() % 6;

      if ( i % 5 == 0 )
         cout << endl;
   }

   return 0;
}
```

```
Enter seed: 67
1         6         5         1         4
5         6         3         1         2
```

```
Enter seed: 432
4         2         6         4         3
2         5         1         4         4
```

```
Enter seed: 67
1         6         5         1         4
5         6         3         1         2
```

Fig. 3.9 Randomizing the die-rolling program.

```
// Fig. 3.10: fig03_10.cpp
// Craps
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

int rollDice( void );   // function prototype

int main()
{
   enum Status { CONTINUE, WON, LOST };
   int sum, myPoint;
   Status gameStatus;

   srand( time( NULL ) );
   sum = rollDice();            // first roll of the dice

   switch ( sum ) {
      case 7:
      case 11:                  // win on first roll
         gameStatus = WON;
         break;
      case 2:
```

```
      case 3:
      case 12:                  // lose on first roll
         gameStatus = LOST;
         break;
      default:                  // remember point
         gameStatus = CONTINUE;
         myPoint = sum;
         cout << "Point is " << myPoint << endl;
         break;                 // optional
   }

   while ( gameStatus == CONTINUE ) {    // keep rolling
      sum = rollDice();

      if ( sum == myPoint )       // win by making point
         gameStatus = WON;
      else
         if ( sum == 7 )          // lose by rolling 7
            gameStatus = LOST;
   }

   if ( gameStatus == WON )
      cout << "Player wins" << endl;
   else
      cout << "Player loses" << endl;

   return 0;
}
```

Fig. 3.10 Program to simulate the game of craps (part 1 of 2).

```
int rollDice( void )
{
   int die1, die2, workSum;

   die1 = 1 + rand() % 6;
   die2 = 1 + rand() % 6;
   workSum = die1 + die2;
   cout << "Player rolled " << die1 << " + " << die2
        << " = " << workSum << endl;

   return workSum;
}
```

Fig. 3.10 Program to simulate the game of craps (part 2 of 2).

```
Player rolled 6 + 5 = 11
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

Fig. 3.11 Sample runs for the game of craps.

```
// Fig. 3.12: fig03_12.cpp
// A scoping example
#include <iostream.h>

void a( void );   // function prototype
void b( void );   // function prototype
void c( void );   // function prototype
```

Fig. 3.12 A scoping example (part 1 of 3).

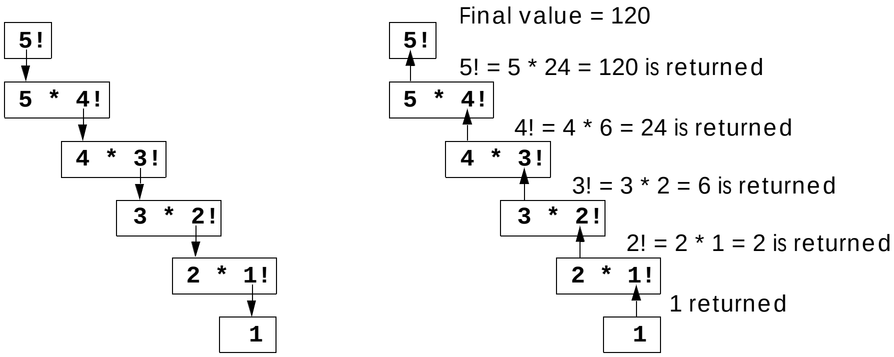
```

int x = 1;      // global variable

int main()
{
   int x = 5;   // local variable to main

   cout << "local x in outer scope of main is " << x << endl;

   {            // start new scope
      int x = 7;

      cout << "local x in inner scope of main is " << x << endl;
   }            // end new scope

   cout << "local x in outer scope of main is " << x << endl;

   a();         // a has automatic local x
   b();         // b has static local x
   c();         // c uses global x
   a();         // a reinitializes automatic local x
   b();         // static local x retains its previous value
   c();         // global x also retains its value

   cout << "local x in main is " << x << endl;

   return 0;
}

void a( void )
{
   int x = 25;  // initialized each time a is called

   cout << endl << "local x in a is " << x
        << " after entering a" << endl;
   ++x;
   cout << "local x in a is " << x
        << " before exiting a" << endl;
}

void b( void )
{
    static int x = 50;  // Static initialization only
                        // first time b is called.
    cout << endl << "local static x is " << x
         << " on entering b" << endl;
    ++x;
    cout << "local static x is " << x
         << " on exiting b" << endl;
}

```

Fig. 3.12 A scoping example (part 2 of 3).

```
void c( void )
{
   cout << endl << "global x is " << x
        << " on entering c" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting c" << endl;
}
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Fig. 3.12 A scoping example (part 3 of 3).


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1

a) Procession of recursive calls. b) Values returned from each recursive call.

Fig. 3.13 Recursive evaluation of 5!.

```cpp
// Fig. 3.14: fig03_14.cpp
// Recursive factorial function
#include <iostream.h>
#include <iomanip.h>

unsigned long factorial( unsigned long );

int main()
{
   for ( int i = 0; i <= 10; i++ )
      cout << setw( 2 ) << i << "! = " << factorial( i ) << endl;

   return 0;
}

// Recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
   if ( number <= 1 )  // base case
      return 1;
   else                // recursive case
      return number * factorial( number - 1 );
}
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 3.14 Calculating factorials with a recursive function.

```
// Fig. 3.15: fig03_15.cpp
// Recursive fibonacci function
#include <iostream.h>

long fibonacci( long );

int main()
{
    long result, number;

    cout << "Enter an integer: ";
    cin >> number;
    result = fibonacci( number );
    cout << "Fibonacci(" << number << ") = " << result << endl;
    return 0;
}

// Recursive definition of function fibonacci
long fibonacci( long n )
{
   if ( n == 0 || n == 1 )  // base case
      return n;
   else                     // recursive case
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

Fig. 3.15 Recursively generating Fibonacci numbers (part 1 of 2).

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Fig. 3.15 Recursively generating Fibonacci numbers (part 2 of 2).

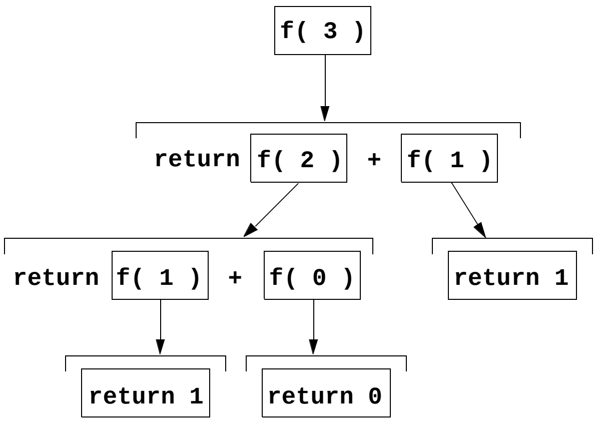


Fig. 3.16 Set of recursive calls to method `fibonacci`.

Chapter	**Recursion Examples and Exercises**
Chapter 3	Factorial function
	Fibonacci function
	Greatest common divisor
	Sum of two integers
	Multiply two integers
	Raising an integer to an integer power
	Towers of Hanoi
	Printing keyboard inputs in reverse
	Visualizing recursion
Chapter 4	Sum the elements of an array
	Print an array
	Print an array backwards
	Print a string backwards
	Check if a string is a palindrome
	Minimum value in an array
	Selection sort
	Eight Queens
	Linear search
	Binary search
Chapter 5	Quicksort
	Maze traversal
	Printing a string input at the keyboard backwards
Chapter 15	Linked list insert
	Linked list delete
	Search a linked list
	Print a linked list backwards

Fig. 3.17 Summary of recursion examples and exercises in the text (part 1 of 2).

Chapter	Recursion Examples and Exercises
	Binary tree insert
	Preorder traversal of a binary tree
	Inorder traversal of a binary tree
	Postorder traversal of a binary tree

Fig. 3.17 Summary of recursion examples and exercises in the text (part 2 of 2).

```
// Fig. 3.18: fig03_18.cpp
// Functions that take no arguments
#include <iostream.h>

void function1();
void function2( void );

int main()
{
   function1();
   function2();

   return 0;
}

void function1()
{
   cout << "function1 takes no arguments" << endl;
}

void function2( void )
{
   cout << "function2 also takes no arguments" << endl;
}
```

```
function1 takes no arguments
function2 also takes no arguments
```

Fig. 3.18 Two ways to declare and use functions that take no arguments.

```
// Fig. 3.19: fig03_19.cpp
// Using an inline function to calculate
// the volume of a cube.
#include <iostream.h>

inline float cube( const float s ) { return s * s * s; }

int main()
{
   cout << "Enter the side length of your cube:  ";

   float side;

   cin >> side;
   cout << "Volume of cube with side "
        << side << " is " << cube( side ) << endl;

   return 0;
}
```

```
Enter the side length of your cube:  3.5
Volume of cube with side 3.5 is 42.875
```

Fig. 3.19 Using an `inline` function to calculate the volume of a cube.

```
// Fig. 3.20: fig03_20.cpp
// Comparing call-by-value and call-by-reference
// with references.
#include <iostream.h>

int squareByValue( int );
void squareByReference( int & );

int main()
{
   int x = 2, z = 4;

   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue( x ) << endl
        << "x = " << x << " after squareByValue\n" << endl;

   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;

   return 0;
}
```

Fig. 3.20 An example of call-by-reference (part 1 of 2).

```
int squareByValue( int a )
{
   return a *= a;   // caller's argument not modified
}

void squareByReference( int &cRef )
{
   cRef *= cRef;    // caller's argument modified
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 3.20 An example of call-by-reference (part 2 of 2).

```
// References must be initialized
#include <iostream.h>

int main()
{
   int x = 3, &y = x;  // y is now an alias for x

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;

   return 0;
}
```

```
x = 3
y = 3
x = 7
y = 7
```

Fig. 3.21 Using an initialized reference.

```
// References must be initialized
#include <iostream.h>

int main()
{
   int x = 3, &y;   // Error: y must be initialized

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;

   return 0;
}
```

```
Compiling FIG03_21.CPP:
Error FIG03_21.CPP 6: Reference variable 'y' must be
   initialized
```

Fig. 3.22 Attempting to use an uninitialized reference.

```
// Fig. 3.23: fig03_23.cpp
// Using default arguments
#include <iostream.h>

int boxVolume( int length = 1, int width = 1, int height = 1 );

int main()
{
   cout << "The default box volume is: " << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume( 10 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume( 10, 5 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
        << endl;

   return 0;
}

// Calculate the volume of a box
int boxVolume( int length, int width, int height )
{
   return length * width * height;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 3.23 Using default arguments.

```
// Fig. 3.24: fig03_24.cpp
// Using the unary scope resolution operator
#include <iostream.h>
#include <iomanip.h>

const double PI = 3.14159265358979;

int main()
{
   const float PI = static_cast< float >( ::PI );

   cout << setprecision( 20 )
        << "  Local float value of PI = " << PI
        << "\nGlobal double value of PI = " << ::PI << endl;

   return 0;
}
```

```
  Local float value of PI = 3.14159
Global double value of PI = 3.14159265358979
```

Fig. 3.24 Using the unary scope resolution operator.

```
// Fig. 3.25: fig03_25.cpp
// Using overloaded functions
#include <iostream.h>

int square( int x ) { return x * x; }

double square( double y ) { return y * y; }

int main()
{
   cout << "The square of integer 7 is " << square( 7 )
        << "\nThe square of double 7.5 is " << square( 7.5 )
        << endl;

   return 0;
}
```

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Fig. 3.25 Using overloaded functions.

```
// Name mangling
int square(int x) { return x * x; }

double square(double y) { return y * y; }

void nothing1(int a, float b, char c, int *d)
   { }  // empty function body

char *nothing2(char a, int b, float *c, double *d)
   { return 0; }

int main()
{
   return 0;
}
```

```
public   _main
public   @nothing2$qzcipfpd
public   @nothing1$qifzcpi
public   @square$qd
public   @square$qi
```

Fig. 3.26 Name mangling to enable type-safe linkage.

```
// Fig. 3.27: fig03_27.cpp
// Using a function template
#include <iostream.h>

template < class T >
T maximum( T value1, T value2, T value3 )
{
   T max = value1;

   if ( value2 > max )
      max = value2;

   if ( value3 > max )
      max = value3;

   return max;
}

int main()
{
   int int1, int2, int3;

   cout << "Input three integer values: ";
   cin >> int1 >> int2 >> int3;
   cout << "The maximum integer value is: "
        << maximum( int1, int2, int3 );        // int version
```

Fig. 3.27 Using a function template (part 1 of 2).

```

   double double1, double2, double3;

   cout << "\nInput three double values: ";
   cin >> double1 >> double2 >> double3;
   cout << "The maximum double value is: "
      << maximum( double1, double2, double3 ); // double version

   char char1, char2, char3;

   cout << "\nInput three characters: ";
   cin >> char1 >> char2 >> char3;
   cout << "The maximum character value is: "
        << maximum( char1, char2, char3 )      // char version
        << endl;

   return 0;
}
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3
Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3
Input three characters: A C B
The maximum character value is: C
```

Fig. 3.27 Using a function template (part 2 of 2).