Chapter 3 - Functions

Outline

Chapter 3 - Functions

Outline

3.1 Introduction

- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program

3.2 Program Components in C++

- Modules: functions and classes
- Programs use new and “prepackaged” modules
  - New: programmer-defined functions, classes
  - Prepackaged: from the standard library
- Functions invoked by function call
  - Function name and information (arguments) it needs
- Function definitions
  - Only written once
  - Hidden from other functions

3.2 Program Components in C++

- Boss to worker analogy
  - A boss (the calling function or caller) asks a worker (the called function) to perform a task and return (i.e., report back) the results when the task is done.

3.3 Math Library Functions

- Perform common mathematical calculations
  - Include the header file **<cmath>**
- Functions called by writing
  - functionName (argument);

  or
  - functionName(argument1, argument2, …);
- Example

  ```
  cout << sqrt( 900.0 );
  ```
  - sqrt (square root) function The preceding statement would print 30
  - All functions in math library return a **double**

3.3 Math Library Functions

- Function arguments can be
  - Constants
    - `sqrt( 4 );`
  - Variables
    - `sqrt( x );`
  - Expressions
    - `sqrt( sqrt( x ) ) ;`
    - `sqrt( 3 - 6x );`

Method	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 *ex*	exp(1.0) is 2.71828 exp(2.0) is 7.38906
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(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)	log10(10.0) is 1.0 log10(100.0) is 2.0
pow(x, y)	*x* raised to power *y* (*xy*)	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 Math library functions.

3.4 Functions

- Functions
  - Modularize a program
  - Software reusability
    - Call function multiple times
- Local variables
  - Known only in the function in which they are defined
  - All variables declared in function definitions are local variables
- Parameters
  - Local variables passed to function when called
  - Provide outside information

3.5 Function Definitions

- Function prototype
  - Tells compiler argument type and return type of function
  - `int square( int );`
    - Function takes an **`int`** and returns an **`int`**
  - Explained in more detail later

- Calling/invoking a function
  - `square(x);`
  - Parentheses an operator used to call function
    - Pass argument x
    - Function gets its own copy of arguments
  - After finished, passes back result

3.5 Function Definitions

- Format for function definition

  return-value-type function-name **(** *parameter-list* **)**
  {
  declarations and statements
  }

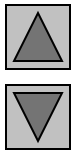
  – Parameter list
    - Comma separated list of arguments
      – Data type needed for each argument
    - If no arguments, use **`void`** or leave blank
  – Return-value-type
    - Data type of result returned (use **`void`** if nothing returned)

3.5 Function Definitions

- Example function

```
int square( int y )
{
  return y * y;
}
```

- **`return`** keyword
  - Returns data, and control goes to function's caller
    - If no data to return, use **`return;`**
  - Function ends when reaches right brace
    - Control goes to caller
- Functions cannot be defined inside other functions
- Next: program examples



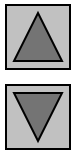
Outline

fig03_03.cpp (1 of 2)

```
1  // Fig. 3.3: fig03_03.cpp
2  // Creating and using a programmer-defined function.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int square( int );   // funct
9
10 int main()
11 {
12    // loop 10 times and calculate and o
13    // square of x each time
14    for ( int x = 1; x <= 10; x++ )
15       cout << square( x ) << "  ";  // function call
16
17    cout << endl;
18
19    return 0;  // indicates successful termination
20
21 } // end main
22
```

Function prototype: specifies data types of arguments and return values. **square** expects and **int**, and returns an **int**.

Parentheses **( )** cause function to be called. When done, it returns the result.



Outline

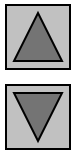
```
23 // square function definition returns square of an integer
24 int square( int y )  // y is a copy of argument to function
25 {
26    return y * y;      // returns square of y as an int
27
28 } // end function square
```

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

Definition of **square**. **y** is a copy of the argument passed. Returns **y * y**, or **y** squared.

fig03_03.cpp (2 of 2)

fig03_03.cpp output (1 of 1)

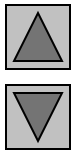


Outline

fig03_04.cpp
(1 of 2)

```
1  // Fig. 3.4: fig03_04.cpp
2  // Finding the maximum of three floating-point numbers.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  double maximum( double, double, double ); // function prototype
10
11 int main()
12 {
13    double number1;
14    double number2;
15    double number3;
16
17    cout << "Enter three floating-point numbers: ";
18    cin >> number1 >> number2 >> number3;
19
20    // number1, number2 and number3 are arguments to
21    // the maximum function call
22    cout << "Maximum is: "
23         << maximum( number1, number2, number3 ) << endl;
24
25    return 0;  // indicates successful termination
```

Function **maximum** takes 3 arguments (all **double**) and returns a **double**.



Outline

```
26
27 } // end main
28
29 // function maximum definition;
30 // x, y and z are parameters
31 double maximum( double x, double y, double z )
32 {
33    double max = x;   // assume x is largest
34
35    if ( y > max )    // if y is larger,
36       max = y;       // assign y to max
37
38    if ( z > max )    // if z is larger,
39       max = z;       // assign z to max
40
41    return max;       // max is largest value
42
43 } // end function maximum
```

Comma separated list for multiple parameters.

fig03_04.cpp (2 of 2)

fig03_04.cpp output (1 of 1)

```
Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32

Enter three floating-point numbers: 1.1 3.333 2.22
Maximum is: 3.333

Enter three floating-point numbers: 27.9 14.31 88.99
Maximum is: 88.99
```

3.6 Function Prototypes

- Function prototype contains
  - Function name
  - Parameters (number and data type)
  - Return type (**`void`** if returns nothing)
  - Only needed if function definition after function call
- Prototype must match function definition
  - Function prototype

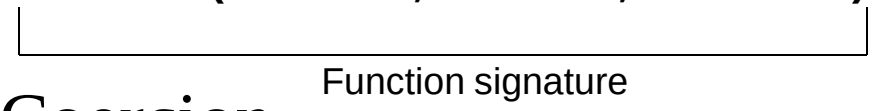
```
double maximum( double, double, double );
```

  - Definition

```
double maximum( double x, double y, double z )
{
 …
}
```


3.6 Function Prototypes

- Function signature
  - Part of prototype with name and parameters
    - `double maximum( double, double, double );`



- Argument Coercion
  - Force arguments to be of proper type
    - Converting **`int`** (4) to **`double`** (4.0)

    `cout << sqrt(4)`
  - Conversion rules
    - Arguments usually converted automatically
    - Changing from **`double`** to **`int`** can truncate data
      - 3.4 to 3
  - Mixed type goes to highest type (promotion)
    - **`Int * double`**

3.6 Function Prototypes

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`	
`char`	
`bool`	(`false` becomes 0, `true` becomes 1)

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

3.7 Header Files

- Header files contain
  - Function prototypes
  - Definitions of data types and constants
- Header files ending with .h
  - Programmer-defined header files

```
#include “myheader.h”
```

- Library header files

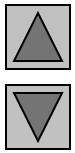
```
#include <cmath>
```


3.8 Random Number Generation

- **`rand`** function (**`<cstdlib>`**)
  - **`i = rand();`**
  - Generates unsigned integer between 0 and RAND_MAX (usually 32767)
- Scaling and shifting
  - Modulus (remainder) operator: **`%`**
    - **`10 % 3`** is **`1`**
    - **`x % y`** is between **`0`** and **`y - 1`**
  - Example

```
i = rand() % 6 + 1;
```

    - **`"Rand() % 6"`** generates a number between **`0`** and **`5`** (scaling)
    - **`"+ 1"`** makes the range 1 to 6 (shift)
  - Next: program to roll dice

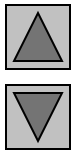


Outline

fig03_07.cpp (1 of 2)

```
1  // Fig. 3.7: fig03_07.cpp
2  // Shifted, scaled integers produced by 1 + rand() % 6.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib>   // contains function prototype for rand
13
14 int main()
15 {
16    // loop 20 times
17    for ( int counter = 1; counter <= 20; counter++ )
18
19       // pick random number from 1 to 6 and output it
20       cout << setw( 10 ) << ( 1 + rand() % 6 );
21
22       // if counter divisible by 5, begin new line of output
23       if ( counter % 5 == 0 )
24          cout << endl;
25
26    } // end for structure
```

Output of **`rand()`** scaled and shifted to be a number between 1 and 6.



Outline

```
27
28    return 0;  // indicates successful termination
29
30 } // end main
```

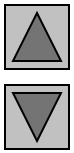
```
        6        6        5        5        6
        5        1        1        5        3
        6        6        2        4        2
        6        2        3        4        1
```

fig03_07.cpp (2 of 2)

fig03_07.cpp output (1 of 1)

3.8 Random Number Generation

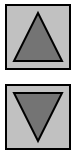
- Next
  - Program to show distribution of **`rand()`**
  - Simulate 6000 rolls of a die
  - Print number of 1's, 2's, 3's, etc. rolled
  - Should be roughly 1000 of each



Outline

fig03_08.cpp
(1 of 3)

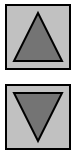
```
1  // Fig. 3.8: fig03_08.cpp
2  // Roll a six-sided die 6000 times.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib>   // contains function prototype for rand
13
14 int main()
15 {
16    int frequency1 = 0;
17    int frequency2 = 0;
18    int frequency3 = 0;
19    int frequency4 = 0;
20    int frequency5 = 0;
21    int frequency6 = 0;
22    int face;  // represents one roll of the die
23
```



Outline

fig03_08.cpp
(2 of 3)

```
24    // loop 6000 times and summarize results
25    for ( int roll = 1; roll <= 6000; roll++ ) {
26       face = 1 + rand() % 6;  // random number from 1 to 6
27
28       // determine face value and increment appropriate counter
29       switch ( face ) {
30
31          case 1:          // rolled 1
32             ++frequency1;
33             break;
34
35          case 2:          // rolled 2
36             ++frequency2;
37             break;
38
39          case 3:          // rolled 3
40             ++frequency3;
41             break;
42
43          case 4:          // rolled 4
44             ++frequency4;
45             break;
46
47          case 5:          // rolled 5
48             ++frequency5;
49             break;
```



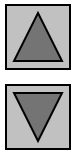
Outline

fig03_08.cpp
(3 of 3)

```
50
51          case 6:          // rolled 6
52             ++frequency6;
53             break;
54
55          default:         // invalid value
56             cout << "Program should never get here!";
57
58       } // end switch
59
60    } // end for
61
62    // display results in tabul
63    cout << "Face" << setw( 13 ) << "Frequency"
64         << "\n   1" << setw( 13 ) << frequency1
65         << "\n   2" << setw( 13 ) << frequency2
66         << "\n   3" << setw( 13 ) << frequency3
67         << "\n   4" << setw( 13 ) << frequency4
68         << "\n   5" << setw( 13 ) << frequency5
69         << "\n   6" << setw( 13 ) << frequency6 << endl;
70
71    return 0;  // indicates successful termination
72
73 } // end main
```

Default case included even though it should never be reached. This is a matter of good coding style

```
Face    Frequency
   1         1003
   2         1017
   3          983
   4          994
   5         1004
   6          999
```

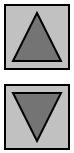


Outline

fig03_08.cpp
output (1 of 1)

3.8 Random Number Generation

- Calling rand() repeatedly
  - Gives the same sequence of numbers
- Pseudorandom numbers
  - Preset sequence of "random" numbers
  - Same sequence generated whenever program run
- To get different random sequences
  - Provide a seed value
    - Like a random starting point in the sequence
    - The same seed will give the same sequence
  - **`srand(seed);`**
    - **`<cstdlib>`**
    - Used before **`rand()`** to set the seed

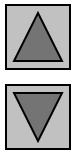


Outline

fig03_09.cpp
(1 of 2)

```
1  // Fig. 3.9: fig03_09.cpp
2  // Randomizing die-rolling program.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 // contains prototypes for functions srand and rand
14 #include <cstdlib>
15
16 // main function begins program execution
17 int main()
18 {
19    unsigned seed;
20
21    cout << "Enter seed: ";
22    cin >> seed;
23    srand( seed );  // seed random number generator
24
```

Setting the seed with **`srand()`**.



Outline

fig03_09.cpp (2 of 2)

fig03_09.cpp output (1 of 1)

```
25    // loop 10 times
26    for ( int counter = 1; counter <= 10; counter++ ) {
27
28       // pick random number from 1 to 6 and output it
29       cout << setw( 10 ) << ( 1 + rand() % 6 );
30
31       // if counter divisible by 5, begin new line of output
32       if ( counter % 5 == 0 )
33          cout << endl;
34
35    } // end for
36
37    return 0;  // indicates s
38
39 } // end main
```

`rand()` gives the same sequence if it has the same initial seed.

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

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

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

3.8 Random Number Generation

- Can use the current time to set the seed
  - No need to explicitly set seed every time
  - `srand( time( 0 ) );`
  - `time( 0 );`
    - `<ctime>`
    - Returns current time in seconds
- General shifting and scaling
  - *Number* `=` *shiftingValue* `+ rand() %` *scalingFactor*
  - shiftingValue = first number in desired range
  - scalingFactor = width of desired range

3.9 Example: Game of Chance and Introducing enum

- Enumeration
  - Set of integers with identifiers

    enum *typeName* **{***constant1*, *constant2*…**};**
  - Constants start at 0 (default), incremented by 1
  - Constants need unique names
  - Cannot assign integer to enumeration variable
    - Must use a previously defined enumeration type
- Example

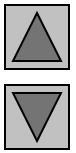
```
enum Status {CONTINUE, WON, LOST};
Status enumVar;
enumVar = WON; // cannot do enumVar = 1
```


3.9 Example: Game of Chance and Introducing enum

- Enumeration constants can have preset values

```
enum Months { JAN = 1, FEB, MAR, APR, MAY,
   JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

  - Starts at 1, increments by 1
- Next: craps simulator
  - Roll two dice
  - 7 or 11 on first throw: player wins
  - 2, 3, or 12 on first throw: player loses
  - 4, 5, 6, 8, 9, 10
    - Value becomes player's "point"
    - Player must roll his point before rolling 7 to win



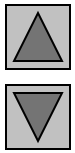
Outline

fig03_10.cpp
(1 of 5)

```
1  // Fig. 3.10: fig03_10.cpp
2  // Craps.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // contains function prototypes for functions srand and rand
9  #include <cstdlib>
10
11 #include <ctime>   // contains pro
12
13 int rollDice( void );  // function prototype
14
15 int main()
16 {
17    // enumeration constants represent game status
18    enum Status { CONTINUE, WON, LOST };
19
20    int sum;
21    int myPoint;
22
23    Status gameStatus;  // can contain CONTINUE, WON or LOST
24
```

Function to roll 2 dice and return the result as an **`int`**.

Enumeration to keep track of the current game.

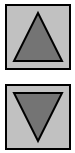


Outline

fig03_10.cpp
(2 of 5)

```
25    // randomize random number generator using current time
26    srand( time( 0 ) );
27
28    sum = rollDice();   // first
29
30    // determine game status and
31    switch ( sum ) {
32
33       // win on first roll
34       case 7:
35       case 11:
36          gameStatus = WON;
37          break;
38
39       // lose on first roll
40       case 2:
41       case 3:
42       case 12:
43          gameStatus = LOST;
44          break;
45
```

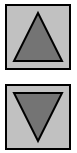
`switch` statement determines outcome based on die roll.



Outline

fig03_10.cpp
(3 of 5)

```
46       // remember point
47       default:
48          gameStatus = CONTINUE;
49          myPoint = sum;
50          cout << "Point is " << myPoint << endl;
51          break;                // optional
52
53    } // end switch
54
55    // while game not complete ...
56    while ( gameStatus == CONTINUE ) {
57       sum = rollDice();          // roll dice again
58
59       // determine game status
60       if ( sum == myPoint )      // win by making point
61          gameStatus = WON;
62       else
63          if ( sum == 7 )         // lose by rolling 7
64             gameStatus = LOST;
65
66    } // end while
67
```

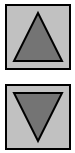


Outline

fig03_10.cpp
(4 of 5)

```
68    // display won or lost message
69    if ( gameStatus == WON )
70       cout << "Player wins" << endl;
71    else
72       cout << "Player loses" << endl;
73
74    return 0;  // indicates successful termination
75
76 } // end main
77
78 // roll dice, calculate sum and d
79 int rollDice( void )
80 {
81    int die1;
82    int die2;
83    int workSum;
84
85    die1 = 1 + rand() % 6;  // pick random die1 value
86    die2 = 1 + rand() % 6;  // pick random die2 value
87    workSum = die1 + die2;  // sum die1 and die2
88
```

Function **`rollDice`** takes no arguments, so has **`void`** in the parameter list.



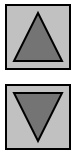
Outline

```
89    // display results of this roll
90    cout << "Player rolled " << die1 << " + " << die2
91         << " = " << workSum << endl;
92
93    return workSum;         // return sum of dice
94
95 } // end function rollDice
```

fig03_10.cpp (5 of 5)

```
Player rolled 2 + 5 = 7
Player wins

Player rolled 6 + 6 = 12
Player loses

Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

fig03_10.cpp output (1 of 2)



Outline

fig03_10.cpp
output (2 of 2)

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

3.10 Storage Classes

- Variables have attributes
  - Have seen name, type, size, value
  - Storage class
    - How long variable exists in memory
  - Scope
    - Where variable can be referenced in program
  - Linkage
    - For multiple-file program (see Ch. 6), which files can use it

3.10 Storage Classes

- Automatic storage class
  - Variable created when program enters its block
  - Variable destroyed when program leaves block
  - Only local variables of functions can be automatic
    - Automatic by default
    - keyword **`auto`** explicitly declares automatic
  - **`register`** keyword
    - Hint to place variable in high-speed register
    - Good for often-used items (loop counters)
    - Often unnecessary, compiler optimizes
  - Specify either **`register`** or **`auto`**, not both
    - **`register int counter = 1;`**

3.10 Storage Classes

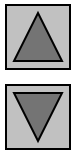
- Static storage class
  - Variables exist for entire program
    - For functions, name exists for entire program
  - May not be accessible, scope rules still apply (more later)
- **`static`** keyword
  - Local variables in function
  - Keeps value between function calls
  - Only known in own function
- **`extern`** keyword
  - Default for global variables/functions
    - Globals: defined outside of a function block
  - Known in any function that comes after it

3.11 Scope Rules

- Scope
  - Portion of program where identifier can be used
- File scope
  - Defined outside a function, known in all functions
  - Global variables, function definitions and prototypes
- Function scope
  - Can only be referenced inside defining function
  - Only labels, e.g., identifiers with a colon (`case:`)

3.11 Scope Rules

- Block scope
  - Begins at declaration, ends at right brace `}`
    - Can only be referenced in this range
  - Local variables, function parameters
  - **`static`** variables still have block scope
    - Storage class separate from scope
- Function-prototype scope
  - Parameter list of prototype
  - Names in prototype optional
    - Compiler ignores
  - In a single prototype, name can be used once



Outline

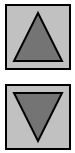
fig03_12.cpp
(1 of 5)

```
1  // Fig. 3.12: fig03_12.cpp
2  // A scoping example.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  void useLocal( void );
9  void useStaticLocal( void );
10 void useGlobal( void );
11
12 int x = 1;       // global variable
13
14 int main()
15 {
16    int x = 5;    // local variable to main
17
18    cout << "local x in main's out
19
20    { // start new scope
21
22       int x = 7;
23
24       cout << "local x in main's inner scope is " << x << endl;
25
26    } // end new scope
```

Declared outside of function; global variable with file scope.

Local variable with function scope.

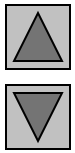
Create a new block, giving **x** block scope. When the block ends, this **x** is destroyed.



Outline

fig03_12.cpp
(2 of 5)

```
27
28    cout << "local x in main's outer scope is " << x << endl;
29
30    useLocal();       // useLocal has local x
31    useStaticLocal(); // useStaticLocal has static local x
32    useGlobal();      // useGlobal uses global x
33    useLocal();       // useLocal reinitializes its local x
34    useStaticLocal(); // static local x retains its prior value
35    useGlobal();      // global x also retains its value
36
37    cout << "\nlocal x in main is " << x << endl;
38
39    return 0;   // indicates successful termination
40
41 } // end main
42
```

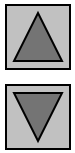


Outline

fig03_12.cpp
(3 of 5)

```
43 // useLocal reinitializes local variable x during each call
44 void useLocal( void )
45 {
46    int x = 25;  // initialized each time useLocal is called
47
48    cout << endl << "local x i
49         << " on entering useL
50    ++x;
51    cout << "local x is " << x
52         << " on exiting useLo
53
54 } // end function useLocal
55
```

Automatic variable (local variable of function). This is destroyed when the function exits, and reinitialized when the function begins.

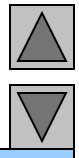


Outline

fig03_12.cpp
(4 of 5)

```
56 // useStaticLocal initializes static local variable x only the
57 // first time the function is called; value of x is saved
58 // between calls to this function
59 void useStaticLocal( void )
60 {
61    // initialized only first time useStaticLocal is called
62    static int x = 50;
63
64    cout << endl << "local static x is " << x
65         << " on entering useStaticLocal" << endl;
66    ++x;
67    cout << "local static x is " <<
68         << " on exiting useStaticL
69
70 } // end function useStaticLocal
71
```

Static local variable of function; it is initialized only once, and retains its value between function calls.



Outline

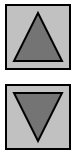
```
72 // useGlobal modifies global variable x during each call
73 void useGlobal( void )
74 {
75    cout << endl << "global x is " << x
76         << " on entering useGlobal" << endl;
77    x *= 10;
78    cout << "global x is " << x
79         << " on exiting useGlobal" << endl;
80
81 } // end function useGlobal
```

This function does not declare any variables. It uses the global **x** declared in the beginning of the program.

03_12.cpp
of 5)

03_12.cpp
output (1 of 2)

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

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

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal
```



Outline

fig03_12.cpp
output (2 of 2)

```
local x is 25 on entering useLocal
local x is 26 on exiting useLocal

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

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

3.12 Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
- If not base case
  - Break problem into smaller problem(s)
  - Launch new copy of function to work on the smaller problem (recursive call/recursive step)
    - Slowly converges towards base case
    - Function makes call to itself inside the return statement
  - Eventually base case gets solved
    - Answer works way back up, solves entire problem

3.12 Recursion

- Example: factorial

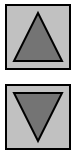
$$n! = n * (n – 1) * (n – 2) * … * 1$$

  - Recursive relationship (n! = n * (n – 1)!)

$$5! = 5 * 4!$$

$$4! = 4 * 3!…$$

  - Base case (1! = 0! = 1)

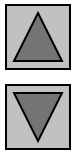


Outline

fig03_14.cpp
(1 of 2)

```
1  // Fig. 3.14: fig03_14.cpp
2  // Recursive factorial function.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 unsigned long factorial( unsigned long ); // function prototype
13
14 int main()
15 {
16    // Loop 10 times. During each iteration, calculate
17    // factorial( i ) and display result.
18    for ( int i = 0; i <= 10; i++ )
19       cout << setw( 2 ) << i << "! = "
20            << factorial( i ) << endl;
21
22    return 0;  // indicates successful termination
23
24 } // end main
```

Data type **`unsigned long`** can hold an integer from 0 to 4 billion.



Outline

```
25
26 // recursive definition of function factorial
27 unsigned long factorial( unsigned long
28 {
29    // base case
30    if ( number <= 1 )
31       return 1;
32
33    // recursive step
34    else
35       return number * factorial( number - 1 );
36
37 } // end function factorial
```

The base case occurs when we have **0!** or **1!**. All other cases must be split up (recursive step).

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

fig03_14.cpp (2 of 2)

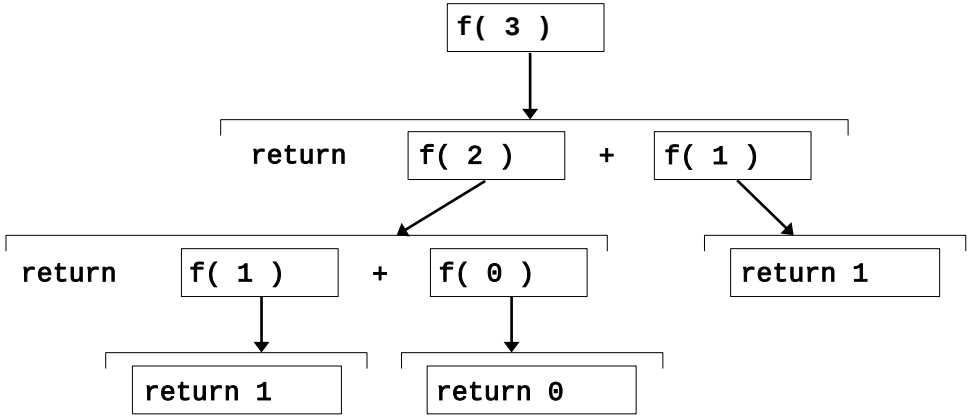
fig03_14.cpp output (1 of 1)

3.13 Example Using Recursion: Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number sum of two previous ones
  - Example of a recursive formula:
    - *fib(n) = fib(n-1) + fib(n-2)*
- C++ code for Fibonacci function

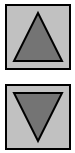
```
long fibonacci( long n )
{
  if ( n == 0 || n == 1 )  // base case
      return n;
  else
   return fibonacci( n - 1 ) +
          fibonacci( n – 2 );
}
```


3.13 Example Using Recursion: Fibonacci Series



3.13 Example Using Recursion: Fibonacci Series

- Order of operations
  - `return fibonacci( n - 1 ) + fibonacci( n - 2 );`
- Do not know which one executed first
  - C++ does not specify
  - Only `&&, ||` and `?:` guaranteed left-to-right evaluation
- Recursive function calls
  - Each level of recursion doubles the number of function calls
    - 30^{th} number = 2^30 ~ 4 billion function calls
  - Exponential complexity

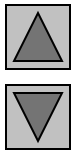


Outline

fig03_15.cpp
of 2)

```
1  // Fig. 3.15: fig03_15.cpp
2  // Recursive fibonacci function.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  unsigned long fibonacci( unsigned long ); // fu
10
11 int main()
12 {
13    unsigned long result, number;
14
15    // obtain integer from user
16    cout << "Enter an integer: ";
17    cin >> number;
18
19    // calculate fibonacci value for number input by user
20    result = fibonacci( number );
21
22    // display result
23    cout << "Fibonacci(" << number << ") = " << result << endl;
24
25    return 0;  // indicates successful termination
```

The Fibonacci numbers get large very quickly, and are all non-negative integers. Thus, we use the **unsigned long** data type.



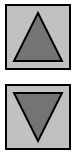
Outline

fig03_15.cpp (2 of 2)

fig03_15.cpp output (1 of 2)

```
26
27 } // end main
28
29 // recursive definition of function fibonacci
30 unsigned long fibonacci( unsigned long n )
31 {
32    // base case
33    if ( n == 0 || n == 1 )
34       return n;
35
36    // recursive step
37    else
38       return fibonacci( n - 1 ) + fibonacci( n - 2 );
39
40 } // end function fibonacci
```

```
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
```



Outline

fig03_15.cpp
output (2 of 2)

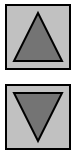
```
Enter an integer: 4
Fibonacci(4) = 3

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
```

3.14 Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance (iteration) and good software engineering (recursion)

3.15 Functions with Empty Parameter Lists

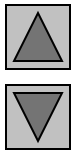
- Empty parameter lists
  - **`void`** or leave parameter list empty
  - Indicates function takes no arguments
  - Function **`print`** takes no arguments and returns no value
    - **`void print();`**
    - **`void print( void );`**



Outline

fig03_18.cpp
(1 of 2)

```
1  // Fig. 3.18: fig03_18.cpp
2  // Functions that take no arguments.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  void function1();        // function prototype
9  void function2( void );  // function prototype
10
11 int main()
12 {
13    function1();  // call function1 with no arguments
14    function2();  // call function2 with no arguments
15
16    return 0;     // indicates successful termination
17
18 } // end main
19
```



Outline

```
20 // function1 uses an empty parameter list to specify that
21 // the function receives no arguments
22 void function1()
23 {
24    cout << "function1 takes no arguments" << endl;
25
26 } // end function1
27
28 // function2 uses a void parameter list to specify that
29 // the function receives no arguments
30 void function2( void )
31 {
32    cout << "function2 also takes no arguments" << endl;
33
34 } // end function2
```

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

fig03_18.cpp (2 of 2)

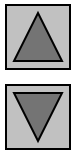
fig03_18.cpp output (1 of 1)

3.16 Inline Functions

- Inline functions
  - Keyword **`inline`** before function
  - Asks the compiler to copy code into program instead of making function call
    - Reduce function-call overhead
    - Compiler can ignore **`inline`**
  - Good for small, often-used functions
- Example

```
inline double cube( const double s )
   { return s * s * s; }
```

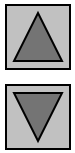
  - **`const`** tells compiler that function does not modify **`s`**
    - Discussed in chapters 6-7



Outline

fig03_19.cpp
(1 of 2)

```
1  // Fig. 3.19: fig03_19.cpp
2  // Using an inline function to calculate.
3  // the volume of a cube.
4  #include <iostream>
5
6  using std::cout;
7  using std::cin;
8  using std::endl;
9
10 // Definition of inline function cube. Definition of function
11 // appears before function is called, so a function prototype
12 // is not required. First line of function definition acts as
13 // the prototype.
14 inline double cube( const double side )
15 {
16    return side * side * side;  // calculate cube
17
18 } // end function cube
19
```



Outline

```
20 int main()
21 {
22    cout << "Enter the side length of your cube: ";
23
24    double sideValue;
25
26    cin >> sideValue;
27
28    // calculate cube of sideValue and display result
29    cout << "Volume of cube with side "
30         << sideValue << " is " << cube( sideValue ) << endl;
31
32    return 0;  // indicates successful termination
33
34 } // end main
```

fig03_19.cpp (2 of 2)

fig03_19.cpp output (1 of 1)

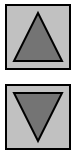
```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

3.17 References and Reference Parameters

- Call by value
  - Copy of data passed to function
  - Changes to copy do not change original
  - Prevent unwanted side effects
- Call by reference
  - Function can directly access data
  - Changes affect original

3.17 References and Reference Parameters

- Reference parameter
  - Alias for argument in function call
    - Passes parameter by reference
  - Use **&** after data type in prototype
    - **`void myFunction( int &data )`**
    - Read "**`data`** is a reference to an **`int`**"
  - Function call format the same
    - However, original can now be changed

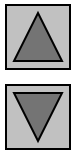


Outline

fig03_20.cpp (1 of 2)

```
1  // Fig. 3.20: fig03_20.cpp
2  // Comparing pass-by-value and pass-by-reference
3  // with references.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  int squareByValue( int );        // function prototype
10 void squareByReference( int & );  // function prototype
11
12 int main()
13 {
14    int x = 2;
15    int z = 4;
16
17    // demonstrate squareByValue
18    cout << "x = " << x << " before squareByValue\n";
19    cout << "Value returned by squareByValue: "
20         << squareByValue( x ) << endl;
21    cout << "x = " << x << " after squareByValue\n" << endl;
22
```

Notice the **&** operator, indicating pass-by-reference.



Outline

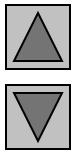
fig03_20.cpp (2 of 2)

```
23    // demonstrate squareByReference
24    cout << "z = " << z << " before squareByReference" << endl;
25    squareByReference( z );
26    cout << "z = " << z << " after squareByReference" << endl;
27
28    return 0;  // indicates successful termination
29 } // end main
30
31 // squareByValue multiplies number by itsel
32 // result in number and returns the new val
33 int squareByValue( int number )
34 {
35    return number *= number;  // caller's argument not modified
36
37 } // end function squareByValue
38
39 // squareByReference multiplies numberRef by its
40 // stores the result in the variable to which nu
41 // refers in function main
42 void squareByReference( int &numberRef )
43 {
44    numberRef *= numberRef;   // caller's argument modified
45
46 } // end function squareByReference
```

Changes **number**, but original parameter (**x**) is not modified.

Changes **numberRef**, an alias for the original parameter. Thus, **z** is changed.

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

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



Outline

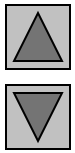
fig03_20.cpp
output (1 of 1)

3.17 References and Reference Parameters

- Pointers (chapter 5)
  - Another way to pass-by-refernce
- References as aliases to other variables
  - Refer to same variable
  - Can be used within a function

```
int count = 1; // declare integer variable count
Int &cRef = count; // create cRef as an alias for count
++cRef; // increment count (using its alias)
```

- References must be initialized when declared
  - Otherwise, compiler error
  - Dangling reference
    - Reference to undefined variable



Outline

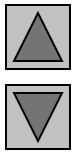
fig03_21.cpp (1 of 1)

fig03_21.cpp output (1 of 1)

```
1  // Fig. 3.21: fig03_21.cpp
2  // References must be initialized.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int x = 3;
11
12    // y refers to (is an alias for) x
13    int &y = x;
14
15    cout << "x = " << x << endl << "y = " << y << endl;
16    y = 7;
17    cout << "x = " << x << endl << "y = " << y << endl;
18
19    return 0;  // indicates successful termination
20
21 } // end main
```

y declared as a reference to **x**.

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



Outline

fig03_22.cpp (1 of 1)

fig03_22.cpp output (1 of 1)

```
1  // Fig. 3.22: fig03_22.cpp
2  // References must be initialized.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int x = 3;
11    int &y;      // Error: y must be initialized
12
13    cout << "x = " << x << endl << "y = " << y << endl;
14    y = 7;
15    cout << "x = " << x << endl << "y = " << y << endl;
16
17    return 0;  // indicates successful termination
18
19 } // end main
```

Uninitialized reference – compiler error.

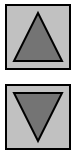
```
Borland C++ command-line compiler error message:
 Error E2304 Fig03_22.cpp 11: Reference variable 'y' must be
   initialized- in function main()

Microsoft Visual C++ compiler error message:
 D:\cpphtp4_examples\ch03\Fig03_22.cpp(11) : error C2530: 'y' :
   references must be initialized
```

3.18 Default Arguments

- Function call with omitted parameters
  - If not enough parameters, rightmost go to their defaults
  - Default values
    - Can be constants, global variables, or function calls
- Set defaults in function prototype

```
int myFunction( int x = 1, int y = 2, int z = 3 );
```

  - **`myFunction(3)`**
    - **`x = 3`**, **`y`** and **`z`** get defaults (rightmost)
  - **`myFunction(3, 5)`**
    - **`x = 3`**, **`y = 5`** and **`z`** gets default



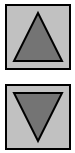
Outline

```
1  // Fig. 3.23: fig03_23.cpp
2  // Using default arguments.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function prototype that specifies default arguments
9  int boxVolume( int length = 1, int width = 1, int height = 1 );
10
11 int main()
12 {
13    // no arguments--use default values for all dimensions
14    cout << "The default box volume is: " << boxVolume();
15
16    // specify length; default width and height
17    cout << "\n\nThe volume of a box with length 10,\n"
18         << "width 1 and height 1 is: " << boxVolume( 10 );
19
20    // specify length and width; default height
21    cout << "\n\nThe volume of a box with length 10,\n"
22         << "width 5 and height 1 is: " << boxVolume( 10, 5 );
23
```

Set defaults in function prototype.

Function calls with some parameters missing – the rightmost parameters get their defaults.

fig03_23.cpp
(1 of 2)



Outline

```
24    // specify all arguments
25    cout << "\n\nThe volume of a box with length 10,\n"
26         << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
27         << endl;
28
29    return 0;  // indicates successful termination
30
31 } // end main
32
33 // function boxVolume calculates the volume of a box
34 int boxVolume( int length, int width, int height )
35 {
36    return length * width * height;
37
38 } // end function boxVolume
```

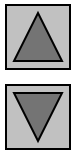
```
The default box volume is: 1

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
```

fig03_23.cpp
(2 of 2)

fig03_23.cpp
output (1 of 1)

3.19 Unitary Scope Resolution Operator

- Unary scope resolution operator (`::`)
  - Access global variable if local variable has same name
  - Not needed if names are different
  - Use **`::variable`**
    - **`y = ::x + 3;`**
  - Good to avoid using same names for locals and globals



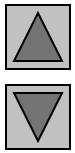
Outline

fig03_24.cpp
(1 of 2)

```
1  // Fig. 3.24: fig03_24.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setprecision;
11
12 // define global constant PI
13 const double PI = 3.14159265358979;
14
15 int main()
16 {
17    // define local constant PI
18    const float PI = static_cast< float >( ::PI );
19
20    // display values of local and global PI constants
21    cout << setprecision( 20 )
22         << "  Local float value of PI = " << PI
23         << "\nGlobal double value of PI = " << ::PI << endl;
24
25    return 0;  // indicates successful termination
```

Access the global **PI** with **::PI**.

Cast the global **PI** to a **float** for the local **PI**. This example will show the difference between **float** and **double**.



Outline

```
26
27  } // end main
```

```
Borland C++ command-line compiler output:
  Local float value of PI = 3.141592741012573242
Global double value of PI = 3.141592653589790007

Microsoft Visual C++ compiler output:
  Local float value of PI = 3.1415927410125732
Global double value of PI = 3.14159265358979
```

fig03_24.cpp (2 of 2)

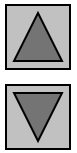
fig03_24.cpp output (1 of 1)

3.20 Function Overloading

- Function overloading
  - Functions with same name and different parameters
  - Should perform similar tasks
    - I.e., function to square **`int`**s and function to square **`float`**s

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

- Overloaded functions distinguished by signature
  - Based on name and parameter types (order matters)
  - Name mangling
    - Encodes function identifier with parameters
  - Type-safe linkage
    - Ensures proper overloaded function called

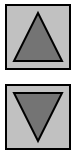


Outline

fig03_25.cpp
(1 of 2)

```
1  // Fig. 3.25: fig03_25.cpp
2  // Using overloaded functions.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function square for int values
9  int square( int x )
10 {
11    cout << "Called square with int argument: " << x << endl;
12    return x * x;
13
14 } // end int version of function square
15
16 // function square for double values
17 double square( double y )
18 {
19    cout << "Called square with double argument: " << y << endl;
20    return y * y;
21
22 } // end double version of function square
23
```

Overloaded functions have the same name, but the different parameters distinguish them.



Outline

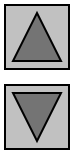
fig03_25.cpp (2 of 2)

fig03_25.cpp output (1 of 1)

```
24 int main()
25 {
26    int intResult = square( 7 );          // calls int version
27    double doubleResult = square( 7.5 ); // calls double version
28
29    cout << "\nThe square of intege
30         << "\nThe square of doubl
31         << endl;
32
33    return 0;  // indicates successful termination
34
35 } // end main
```

The proper function is called based upon the argument (**int** or **double**).

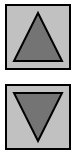
```
Called square with int argument: 7
Called square with double argument: 7.5

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



Outline

fig03_26.cpp
(1 of 2)

```
1  // Fig. 3.26: fig03_26.cpp
2  // Name mangling.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  }
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 }
15
16 // function that receives arguments of types
17 // int, float, char and int *
18 void nothing1( int a, float b, char c, int *d )
19 {
20    // empty function body
21 }
22
```



Outline

fig03_26.cpp (2 of 2)

fig03_26.cpp output (1 of 1)

```
23 // function that receives arguments of types
24 // char, int, float * and double *
25 char *nothing2( char a, int b, float *c, double *d )
26 {
27    return 0;
28 }
29
30 int main()
31 {
32    return 0;  // indicates successful termination
33
34 } // end main
```

```
_main
@nothing2$qcipfpd
@nothing1$qifcpi
@square$qd
@square$qi
```

Mangled names produced in assembly language.

$q separates the function name from its parameters. **c** is **char**, **d** is **double**, **i** is **int**, **pf** is a pointer to a **float**, etc.

3.21 Function Templates

- Compact way to make overloaded functions
  - Generate separate function for different data types
- Format
  - Begin with keyword **`template`**
  - Formal type parameters in brackets **`<>`**
    - Every type parameter preceded by **`typename`** or **`class`** (synonyms)
    - Placeholders for built-in types (i.e., **`int`**) or user-defined types
    - Specify arguments types, return types, declare variables
  - Function definition like normal, except formal types used

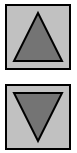
3.21 Function Templates

- Example

```
template < class T > // or template< typename T >
T square( T value1 )
{
   return value1 * value1;
}
```

  - **T** is a formal type, used as parameter type
    - Above function returns variable of same type as parameter
  - In function call, T replaced by real type
    - If **int**, all **T**'s become **int**s

```
int x;
int y = square(x);
```

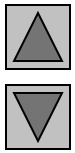
Outline

fig03_27.cpp
(1 of 3)

```
1  // Fig. 3.27: fig03_27.cpp
2  // Using a function template.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  // definition of function templat
10 template < class T >  // or template < typename T >
11 T maximum( T value1, T value2, T value3 )
12 {
13    T max = value1;
14
15    if ( value2 > max )
16       max = value2;
17
18    if ( value3 > max )
19       max = value3;
20
21    return max;
22
23 } // end function template maximum
24
```

Formal type parameter **T** placeholder for type of data to be tested by **maximum**.

maximum expects all parameters to be of the same type.

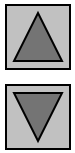


Outline

fig03_27.cpp
(2 of 3)

```
25 int main()
26 {
27    // demonstrate maximum with int values
28    int int1, int2, int3;
29
30    cout << "Input three integer values: ";
31    cin >> int1 >> int2 >> int3;
32
33    // invoke int version of maximum
34    cout << "The maximum integer value is: "
35         << maximum( int1, int2, int3 );
36
37    // demonstrate maximum with double values
38    double double1, double2, double3;
39
40    cout << "\n\nInput three double values: ";
41    cin >> double1 >> double2 >> double3;
42
43    // invoke double version of maximum
44    cout << "The maximum double value is: "
45         << maximum( double1, double2, double3 );
46
```

maximum called with various data types.



Outline

```
47    // demonstrate maximum with char values
48    char char1, char2, char3;
49
50    cout << "\n\nInput three characters: ";
51    cin >> char1 >> char2 >> char3;
52
53    // invoke char version of maximum
54    cout << "The maximum character value is: "
55         << maximum( char1, char2, char3 )
56         << endl;
57
58    return 0;  // indicates successful termination
59
60 } // end main
```

```
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
```

fig03_27.cpp (3 of 3)

fig03_27.cpp output (1 of 1)