Chapter 3 - Functions

3.1 Introduction
3.2 Program Components in C++
3.3 Math Library Functions
3.4 Functions
3.5 Function Definitions
3.6 Function Prototypes
3.7 Header Files
3.8 Random Number Generation
3.11 Scope Rules
3.12 Recursion
3.13 Example Using Recursion: The Fibonacci Series
3.17 References and Reference Parameters
3.18 Default Arguments
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
  ```cpp
  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 );
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil(x)</td>
<td>rounds $x$ to the smallest integer not less than $x$</td>
<td>ceil(9.2) is 10.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ceil(-9.8) is -9.0</td>
</tr>
<tr>
<td>cos(x)</td>
<td>trigonometric cosine of $x$ ($x$ in radians)</td>
<td>cos(0.0) is 1.0</td>
</tr>
<tr>
<td>exp(x)</td>
<td>exponential function $e^x$</td>
<td>exp(1.0) is 2.71828</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exp(2.0) is 7.38906</td>
</tr>
<tr>
<td>fabs(x)</td>
<td>absolute value of $x$</td>
<td>fabs(5.1) is 5.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fabs(0.0) is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fabs(-8.76) is 8.76</td>
</tr>
<tr>
<td>floor(x)</td>
<td>rounds $x$ to the largest integer not greater than $x$</td>
<td>floor(9.2) is 9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>floor(-9.8) is -10.0</td>
</tr>
<tr>
<td>fmod(x, y)</td>
<td>remainder of $x/y$ as a floating-point number</td>
<td>fmod(13.657, 2.333) is 1.992</td>
</tr>
<tr>
<td>log(x)</td>
<td>natural logarithm of $x$ (base $e$)</td>
<td>log(2.71828) is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log(7.389056) is 2.0</td>
</tr>
<tr>
<td>log10(x)</td>
<td>logarithm of $x$ (base 10)</td>
<td>log10(10.0) is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log10(100.0) is 2.0</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>$x$ raised to power $y$ ($x^y$)</td>
<td>pow(2, 7) is 128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pow(9, .5) is 3</td>
</tr>
<tr>
<td>sin(x)</td>
<td>trigonometric sine of $x$ ($x$ in radians)</td>
<td>sin(0.0) is 0</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root of $x$</td>
<td>sqrt(900.0) is 30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sqrt(9.0) is 3.0</td>
</tr>
<tr>
<td>tan(x)</td>
<td>trigonometric tangent of $x$ ($x$ in radians)</td>
<td>tan(0.0) is 0</td>
</tr>
</tbody>
</table>

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

   \textit{return-value-type} \ \textit{function-name} \ ( \textit{parameter-list} )

   \{
     \textit{declarations \ and \ statements}
   \}

– Parameter list
  • Comma separated list of arguments
    – Data type needed for each argument
  • If no arguments, use \texttt{void} or leave blank

– Return-value-type
  • Data type of result returned (use \texttt{void} if nothing returned)
3.5 Function Definitions

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

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

int square( int ); // function prototype

int main()
{
    // loop 10 times and calculate and output
    // square of x each time
    for ( int x = 1; x <= 10; x++ )
        cout << square( x ) << "  "; // function call

    cout << endl;

    return 0; // indicates successful termination
} // end main

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.
// square function definition returns square of an integer
int square( int y ) // y is a copy of argument to function
{
    return y * y; // returns square of y as an int
} // end function square

Definition of \textit{square}. \textit{y} is a copy of the argument passed. Returns \textit{y} * \textit{y}, or \textit{y} squared.
// Fig. 3.4: fig03_04.cpp  
// Finding the maximum of three floating-point numbers.  
#include <iostream>  

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

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

int main()  
{
    double number1;
    double number2;
    double number3;

    cout << "Enter three floating-point numbers: ";
    cin >> number1 >> number2 >> number3;

    // number1, number2 and number3 are arguments to
    // the maximum function call
    cout << "Maximum is: "
         << maximum( number1, number2, number3 ) << endl;

    return 0; // indicates successful termination
// function maximum definition;
// x, y and z are parameters
double maximum( double x, double y, double z )
{
    double max = x; // assume x is largest
    if ( y > max ) // if y is larger,
        max = y; // assign y to max
    if ( z > max ) // if z is larger,
        max = z; // assign z to max
    return max; // max is largest value
}

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

<table>
<thead>
<tr>
<th>Data types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>long double</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>unsigned long int</td>
<td>(synonymous with unsigned long)</td>
</tr>
<tr>
<td>long int</td>
<td>(synonymous with long)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>(synonymous with unsigned)</td>
</tr>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>unsigned short int</td>
<td>(synonymous with unsigned short)</td>
</tr>
<tr>
<td>short int</td>
<td>(synonymous with short)</td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>(false becomes 0, true becomes 1)</td>
</tr>
</tbody>
</table>

*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
// Fig. 3.7: fig03_07.cpp
// Shifted, scaled integers produced by 1 + rand() % 6.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib>  // contains function prototype for rand

int main()
{
    // loop 20 times
    for ( int counter = 1; counter <= 20; counter++ ) {
        // pick random number from 1 to 6 and output it
        cout << setw( 10 ) << ( 1 + rand() % 6 );

        // if counter divisible by 5, begin new line of output
        if ( counter % 5 == 0 )
            cout << endl;
    }
} // end for structure

Output of \texttt{rand()} scaled and shifted to be a number between 1 and 6.
    return 0;  // indicates successful termination

}  // end main
3.8 Random Number Generation

• Next
  – Program to show distribution of \texttt{rand()} 
  – Simulate 6000 rolls of a die 
  – Print number of 1’s, 2’s, 3’s, etc. rolled 
  – Should be roughly 1000 of each
// Fig. 3.8: fig03_08.cpp
// Roll a six-sided die 6000 times.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip> // contains function prototype for rand
using std::setw;

#include <cstdlib> // contains function prototype for rand

int main()
{
    int frequency1 = 0;
    int frequency2 = 0;
    int frequency3 = 0;
    int frequency4 = 0;
    int frequency5 = 0;
    int frequency6 = 0;
    int face; // represents one roll of the die
// loop 6000 times and summarize results
for ( int roll = 1; roll <= 6000; roll++ ) {
    face = 1 + rand() % 6;  // random number from 1 to 6

    // determine face value and increment appropriate counter
    switch ( face ) {
        case 1:  // rolled 1
            ++frequency1;
            break;

        case 2:  // rolled 2
            ++frequency2;
            break;

        case 3:  // rolled 3
            ++frequency3;
            break;

        case 4:  // rolled 4
            ++frequency4;
            break;

        case 5:  // rolled 5
            ++frequency5;
            break;
    }
case 6:     // rolled 6
    ++frequency6;
    break;

default:    // invalid value
    cout << "Program should never get here!";

} // end switch

} // end for

// display results in tabular format
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; // indicates successful termination

} // end main
<table>
<thead>
<tr>
<th>Face</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1003</td>
</tr>
<tr>
<td>2</td>
<td>1017</td>
</tr>
<tr>
<td>3</td>
<td>983</td>
</tr>
<tr>
<td>4</td>
<td>994</td>
</tr>
<tr>
<td>5</td>
<td>1004</td>
</tr>
<tr>
<td>6</td>
<td>999</td>
</tr>
</tbody>
</table>
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
// Fig. 3.9: fig03_09.cpp
// Randomizing die-rolling program.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
#include <iomanip>
using std::setw;
#include <cstdlib>

// main function begins program execution
int main()
{
    unsigned seed;
    cout << "Enter seed: ";
    cin >> seed;
    srand( seed ); // seed random number generator
    Setting the seed with srand().
// loop 10 times
for (int counter = 1; counter <= 10; counter++) {
    // pick random number from 1 to 6 and output it
    cout << setw(10) << (1 + rand() % 6);
    // if counter divisible by 5, begin new line of output
    if (counter % 5 == 0)
        cout << endl;
}
// end for
return 0; // indicates successful termination
} // 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
  – \texttt{srand( time( 0 ) );}
  – \texttt{time( 0 );}
  • \texttt{<ctime>}
    • Returns current time in seconds

• General shifting and scaling
  – \textit{Number} = \textit{shiftingValue} + \texttt{rand()} \% \textit{scalingFactor}
  – \textit{shiftingValue} = first number in desired range
  – \textit{scalingFactor} = width of desired range
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
```cpp
// Fig. 3.12: fig03_12.cpp
// A scoping example.
#include <iostream>

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

void useLocal( void );
void useStaticLocal( void );
void useGlobal( void );

int x = 1; // global variable

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

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

    {
        // start new scope
        int x = 7;

        cout << "local x in main's inner scope is " << x << endl;
    } // 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.
cout << "local x in main's outer scope is " << x << endl;

useLocal();  // useLocal has local x
useStaticLocal(); // useStaticLocal has static local x
useGlobal(); // useGlobal uses global x
useLocal();  // useLocal reinitializes its local x
useStaticLocal(); // static local x retains its prior value
useGlobal(); // global x also retains its value

cout << "\nlocal x in main is " << x << endl;

return 0;  // indicates successful termination

} // end main
// useLocal reinitializes local variable x during each call
void useLocal( void )
{
    int x = 25; // initialized each time useLocal is called
    cout << endl << "local x is " << x << " on entering useLocal" << endl;
    ++x;
    cout << "local x is " << x << " on exiting useLocal" << endl;
} // end function useLocal
// useStaticLocal initializes static local variable x only the
// first time the function is called; value of x is saved
// between calls to this function
void useStaticLocal( void )
{
    // initialized only first time useStaticLocal is called
    static int x = 50;

    cout << endl << "local static x is " << x
         << " on entering useStaticLocal" << endl;
    ++x;
    cout << "local static x is " << x
         << " on exiting useStaticLocal" << endl;
}

// end function useStaticLocal

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

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
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

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \]

- Recursive relationship \((n! = n \times (n-1)!)\)

\[ 5! = 5 \times 4! \]

\[ 4! = 4 \times 3! \ldots \]

- Base case \((1! = 0! = 1)\)
// Fig. 3.14: fig03_14.cpp
// Recursive factorial function.
#include <iostream>

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

#include <iomanip>
using std::setw;

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

int main()
{
    // Loop 10 times. During each iteration, calculate factorial( i ) and display result.
    for ( int i = 0; i <= 10; i++ )
        cout << setw( 2 ) << i << "! = "
        << factorial( i ) << endl;

    return 0; // indicates successful termination
}

// end main
// recursive definition of function factorial

unsigned long factorial(unsigned long number) {
    // base case
    if (number <= 1)
        return 1;
    // recursive step
    else
        return number * factorial(number - 1);
} // end function factorial

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

The base case occurs when we have 0! or 1!. All other cases must be split up (recursive step).
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:
    • \( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \)

• C++ code for Fibonacci function
  
  ```cpp
  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

\[ f(3) = f(2) + f(1) \]
\[ f(2) = f(1) + f(0) \]
\[ f(1) = 1 \]
\[ f(0) = 0 \]
3.13 Example Using Recursion: Fibonacci Series

• Order of operations
  - \( \text{return fibonacci}(n - 1) + \text{fibonacci}(n - 2) \);

• 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 = \(2^{30} \sim 4\) billion function calls
  - Exponential complexity
The Fibonacci numbers get large very quickly, and are all non-negative integers. Thus, we use the `unsigned long` data type.
```cpp
unsigned long fibonacci(unsigned long n)
{
    // base case
    if ( n == 0 || n == 1 )
        return n;
    // recursive step
    else
        return fibonacci(n - 1) + fibonacci(n - 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
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
// Fig. 3.20: fig03_20.cpp
// Comparing pass-by-value and pass-by-reference
// with references.
#include <iostream>
using std::cout;
using std::endl;

int squareByValue( int ); // function prototype
void squareByReference( int & ); // function prototype

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

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

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

return 0; // indicates successful termination
}

// squareByValue multiplies number by itself, stores the result in number and returns the new value
int squareByValue( int number )
{
    return number *= number; // caller's argument not modified
}

// squareByReference multiplies numberRef by itself and stores the result in the variable to which numberRef refers
void squareByReference( int &numberRef )
{
    numberRef *= numberRef; // caller's argument modified
}

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
3.17 References and Reference Parameters

• Pointers (chapter 5)
  – Another way to pass-by-reference

• References as aliases to other variables
  – Refer to same variable
  – Can be used within a function
    
    ```c
    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
// Fig. 3.21: fig03_21.cpp
// References must be initialized.
#include <iostream>

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

int main()
{
    int x = 3;

    // y refers to (is an alias for) x
    int &y = x;

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

    return 0; // indicates successful termination
} // end main
// Fig. 3.22: fig03_22.cpp
// References must be initialized.
#include <iostream>

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

int main()
{
    int x = 3;
    int &y;  // Error: y must be initialized
    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7;
    cout << "x = " << x << endl << "y = " << y << endl;
    return 0;  // indicates successful termination
} // end main

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
  
  \[
  \text{int myFunction( int x = 1, int y = 2, int z = 3 );}
  \]
  
  – \text{myFunction(3)}
    • \text{x = 3, y and z get defaults (rightmost)}
  
  – \text{myFunction(3, 5)}
    • \text{x = 3, y = 5 and z gets default}
// Fig. 3.23: fig03_23.cpp
// Using default arguments.
#include <iostream>

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

// function prototype that specifies default arguments
int boxVolume(int length = 1, int width = 1, int height = 1);

int main()
{
    // no arguments--use default values for all dimensions
    cout << "The default box volume is: " << boxVolume();

    // specify length; default width and height
    cout << "\n\nThe volume of a box with length 10,\n" << "width 1 and height 1 is: " << boxVolume(10);

    // specify length and width; default height
    cout << "\n\nThe volume of a box with length 10,\n" << "width 5 and height 1 is: " << boxVolume(10, 5);
}
// specify all arguments
cout << "\n\nThe volume of a box with length 10, \n" << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 ) << endl;

return 0;  // indicates successful termination

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

// 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