Chapter 3 - Functions

Outline
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
3.9 Example: A Game of Chance and Introducing `enum`
3.10 Storage Classes
3.11 Scope Rules
3.12 Recursion
3.13 Example Using Recursion: The Fibonacci Series
3.14 Recursion vs. Iteration
3.15 Functions with Empty Parameter Lists
3.16 Inline Functions
3.17 References and Reference Parameters
3.18 Default Arguments
3.19 Unary Scope Resolution Operator
3.20 Function Overloading
3.21 Function Templates

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

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

- Function arguments can include
  - 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><code>ceil(x)</code></td>
<td>rounds x to the smallest integer not less than x</td>
<td><code>ceil(9.2)</code> is 10.0, <code>ceil(-9.8)</code> is -9.0</td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>trigonometric cosine of x (x in radians)</td>
<td><code>cos(0.0)</code> is 1.0</td>
</tr>
<tr>
<td><code>fabs(x)</code></td>
<td>absolute value of x</td>
<td><code>fabs(-5.1)</code> is 5.1</td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>rounds x to the largest integer not greater than x</td>
<td><code>floor(9.2)</code> is 9.0, <code>floor(-9.8)</code> is -10.0</td>
</tr>
<tr>
<td><code>fmod(x, y)</code></td>
<td>remainder of x/y as a floating-point number</td>
<td><code>fmod(13.657, 2.333)</code> is 1.992</td>
</tr>
<tr>
<td><code>log(x)</code></td>
<td>natural logarithm of x (base e)</td>
<td><code>log(2.71828)</code> is 1.0, <code>log(2.718282)</code> is 0.0</td>
</tr>
<tr>
<td><code>log10(x)</code></td>
<td>logarithm of x (base 10)</td>
<td><code>log10(100)</code> is 2.0, <code>log10(1000)</code> is 3.0</td>
</tr>
<tr>
<td><code>pow(x, y)</code></td>
<td>x raised to power y (xy)</td>
<td><code>pow(2, 7)</code> is 128, <code>pow(2, -3)</code> is 0.125</td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>trigonometric sine of x (x in radians)</td>
<td><code>sin(0.0)</code> is 0.0</td>
</tr>
<tr>
<td><code>sqrt(x)</code></td>
<td>square root of x</td>
<td><code>sqrt(900)</code> is 30.0, <code>sqrt(9.0)</code> is 3.0</td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>trigonometric tangent of x (x in radians)</td>
<td><code>tan(0.0)</code> is 0.0</td>
</tr>
</tbody>
</table>

- 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
    - \texttt{int square( int );}
      - Function takes an \texttt{int} and returns an \texttt{int}
      - Explained in more detail later
- Calling/invoking a function
  - \texttt{square(x);}
    - Parentheses an operator used to call function
      - Pass argument \texttt{x}
      - Function gets its own copy of arguments
      - After finished, passes back result

- Format for function definition
  \begin{verbatim}
  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 \texttt{void} or leave blank
  - Return-value-type
    - Data type of result returned (use \texttt{void} if nothing returned)
  \end{verbatim}

- Example function
  \begin{verbatim}
  int square( int y )
  {
    return y * y;
  }
  \end{verbatim}
- \texttt{return} keyword
  - Returns data, and control goes to function’s caller
    - If no data to return, use \texttt{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

// square function definition
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

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

Function prototype: specifies data types of arguments and return values.

square expects and int, and returns an int.

int square( int y ) returns square of y (y * y).

// 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;

    cout << "Maximum is: 
"    << maximum( number1, number2, number3 ) << endl;

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

// 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
} // end function maximum

Comma separated list for multiple parameters.

Enter three floating-point numbers: 99.32 37.3 27.19
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 14.3 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)`
    - 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`

3.7 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>
  ```

Data types

<table>
<thead>
<tr>
<th>Data types</th>
<th>long double</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>long int</td>
<td>short int</td>
<td></td>
</tr>
<tr>
<td>unsigned long int</td>
<td>short int</td>
<td></td>
</tr>
<tr>
<td>short int</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td>bool</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.5 Promotion hierarchy for built-in data types.
3.8 Random Number Generation

- **rand function (<cstdlib>)**
  - \( i = \text{rand}(); \)
  - Generates unsigned integer between 0 and RAND_MAX (usually 32767)
- Scaling and shifting
  - Modulus (remainder) operator: \( \text{mod} \)
    - \( 10 \mod 3 = 1 \)
    - \( x \mod y \) is between 0 and \( y - 1 \)
  - Example
    - \( i = \text{rand}() \mod 6 + 1; \)
    - "\( \text{Rand}() \mod 6 \)" generates a number between 0 and 5 (scaling)
    - "+ 1" makes the range 1 to 6 (shift)
  - Next: program to roll dice

Next: program to show distribution of \( \text{rand}() \)
- 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>

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!";
                break;
        }
    }

    // display results in tabular format
    cout << "Face" << setw( 13 ) << "Frequency"
        << endl;
    cout << "   1" << setw( 13 ) << frequency1
        << endl;
    cout << "   2" << setw( 13 ) << frequency2
        << endl;
    cout << "   3" << setw( 13 ) << frequency3
        << endl;
    cout << "   4" << setw( 13 ) << frequency4
        << endl;
    cout << "   5" << setw( 13 ) << frequency5
        << endl;
    cout << "   6" << setw( 13 ) << frequency6 << endl;

    return 0;  // indicates successful termination
}

// loop 4000 times and summarize results
for ( int roll = 1; roll <= 4000; roll++ ) {
    face = 1 + rand() % 4;  // random number from 1 to 4

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

// display results in tabular format
cout << "Face" << setw( 13 ) << "Frequency"
    << endl;
for ( int face = 1; face <= 4; face++ ) {
    cout << "   " << face << setw( 13 ) << frequency3
        << endl;
}

return 0;  // indicates successful termination
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

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 ) );
    • <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
  ```cpp
eenum 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
  ```cpp
eenum Status {CONTINUE, WON, LOST};
eStatus enumVar;
enumVar = WON; // cannot do enumVar = 1
```

- Enumeration constants can have preset values
  ```cpp
eenum 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

---

```cpp
// Fig. 3.10: fig03_10.cpp
// Craps.
#include <iostream>
#include <cstdlib>
#include <ctime>

int rollDice( void );

int main()
{
   enum Status { CONTINUE, WON, LOST }; // enumeration constants represent game status
    int sum; // contains function prototypes for functions srand and rand
    int myPoint; // declare function prototypes
    int main() // function prototype
    {
        // enumeration constants represent game status
        enum Status { CONTINUE, WON, LOST }; // can contain CONTINUE, WON or LOST
        int sum; // randomize random number generator using current time
        srand( time( 0 ) );
        sum = rollDice(); // first roll of the dice
        // determine game status and point based on sum of dice
        switch ( sum ) {
            // win on first roll
            case 7:
            case 11:
                gameStatus = WON;
                break;
            // lose on first roll
            case 2:
            case 3:
            case 12:
                gameStatus = LOST;
                break;
            default:
                gameStatus = CONTINUE;
        }
    }
    // contains function prototypes for functions srand and rand
    #include <cstdlib>
    #include <ctime>
    #include <iostream>
    using std::cout;
    using std::endl;
}
```

---

```
// contains function prototypes for functions srand and rand
#include <cstdlib>
#include <ctime>
#include <iostream>

int rollDice( void ) // function prototype
{
    // function to roll 2 dice and return the result as an int.
    int roll1, roll2;
    roll1 = rand() % 6 + 1; // random number between 1 and 6
    roll2 = rand() % 6 + 1; // random number between 1 and 6
    int total = roll1 + roll2; // total of the two rolls
    return total; // return the total
}
```
```cpp
// Remember point
default:
    gameStatus = CONTINUE;
    myPoint = sum;
    cout << "Point is " << myPoint << endl;
    break; // optional
} // end switch
// While game not complete ...
while ( gameStatus == CONTINUE ) {
    sum = rollDice(); // Roll dice again
    // Determine game status
    if ( sum == myPoint ) // Win by making point
        gameStatus = WON;
    else
        if ( sum == 7 ) // Lose by rolling 7
            gameStatus = LOST;
    } // end while

// Display won or lost message
if ( gameStatus == WON )
    cout << "Player wins" << endl;
else
    cout << "Player loses" << endl;

return 0; // Indicates successful termination
```

Function `rollDice` takes no arguments, so has `void` in the parameter list.
### 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
    - `auto` keyword 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:)

© 2003 Prentice Hall, Inc. All rights reserved.
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
#include <iostream>

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

void useLocal( void ) {  // function prototype
    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;
}

int main() {
    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 << "local x in main's inner scope is " << x << endl;
    useLocal();      // useLocal reinitializes local variable x during each call
    useStaticLocal(); // static local x retains its prior value
    useGlobal();      // global x also retains its value
    return 0;  // indicates successful termination
}
```

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.

Automatic variable (local variable of function). This is destroyed when the function exits, and reinitialized when the function begins.
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 \( n! = n \times (n-1)! \)
  - Base case \( 1! = 0! = 1 \)

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

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

### Data type unsigned long
Can hold an integer from 0 to 4 billion.

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

• 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
    - 30th number = $2^{30} \approx 4$ 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
#include <iostream>

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

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

int main()
{
    unsigned long result, number;

    // obtain integer from user
    cout << "Enter an integer: ";
    cin >> number;

    // calculate fibonacci value for number input by user
    result = fibonacci(number);

    // display result
    cout << "Fibonacci(" << number << ") = " << result << endl;

    return 0;  // indicates successful termination
}
```

```cpp
// recursive definition of function fibonacci
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);
}
```
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 );
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
  ```cpp
  inline double cube( const double s )
  { return s * s * s; }
  ```
  - **const** tells compiler that function does not modify `s`
  - Discussed in chapters 6-7

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

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

// Definition of inline function cube. Definition of function appears before function is called, so a function prototype is not required. First line of function definition acts as the prototype.
inline double cube( const double side )
{
  return side * side * side;  // calculate cube
}

int main()
{
  cout << "Enter the side length of your cube: ";
  double sideValue;
  cin >> sideValue;
  // calculate cube of sideValue and display result
  cout << "Volume of cube with side " << sideValue << " is " << cube( sideValue ) << endl;
  return 0;  // indicates successful termination
}
```

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

### Outline

```
// Fig. 3.20: fig03_20.cpp
// Comparing pass-by-value and pass-by-reference
#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
    " << squareByValue( x ) << endl;
    cout << "x = " << x << " after squareByValue
    " << endl;

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

    return 0;  // indicates successful termination
}

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

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

**Changes**

- `numberRef` is an alias for the original parameter `z`. Thus, `z` is changed.
- `number` is a reference to an `int`, so changes made to `numberRef` affect the original parameter.

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

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

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
  
  \[
  \text{int } \text{myFunction}( \text{int } \text{x} = 1, \text{int } \text{y} = 2, \text{int } \text{z} = 3 );
  \]

- \text{myFunction}(3)
  - \text{x} = 3, \text{y} and \text{z} get defaults (rightmost)
- \text{myFunction}(3, 5)
  - \text{x} = 3, \text{y} = 5 and \text{z} gets default

### Example Code

```cpp
#include <iostream>

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

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 << "The volume of a box with length 10, width 1 and height 1 is: " << boxVolume( 10 );
    
    // specify length and width; default height
    cout << "The volume of a box with length 10, width 5 and height 1 is: " << boxVolume( 10, 5 );
    
    // specify all arguments
    cout << "The volume of a box with length 10, width 5 and height 2 is: " << boxVolume( 10, 5, 2 ) << endl;
    
    return 0;  // indicates successful termination
}
```

### Box Volume Function

```cpp
int boxVolume( int length, int width, int height )
{
    return length * width * height;
}
```

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
    - \text{y} = ::x + 3;
  - Good to avoid using same names for locals and globals
3.20 Function Overloading

- Function overloading
  - Functions with same name and different parameters
  - Should perform similar tasks
    - i.e., function to square int and function to square floats
      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

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.
# 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**
  
  ```cpp
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
  
  ```cpp
  int x;
  int y = square(x);
  ```

---

**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**
- (1 of 3)

```cpp
// Fig. 3.27: fig03_27.cpp
// Using a function template.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// definition of function template
template < class T >  // or template < typename 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()
{
    // demonstrate maximum with int values
    int int1, int2, int3;
    cout << "Input three integer values: ";
    cin >> int1 >> int2 >> int3;
    cout << "The maximum integer value is: " << maximum( int1, int2, int3 );

    // demonstrate maximum with double values
    double double1, double2, double3;
    cout << "Input three double values: ";
    cin >> double1 >> double2 >> double3;
    cout << "The maximum double value is: " << maximum( double1, double2, double3 );

    // demonstrate maximum with char values
    char char1, char2, char3;
    cout << "Input three characters: ";
    cin >> char1 >> char2 >> char3;
    cout << "The maximum character value is: " << maximum( char1, char2, char3 );
    return 0;  // indicates successful termination
}
```

**Output (1 of 1)**

```
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
```