Chapter 3 - Functions

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++

- 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);
  - 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`

3.5 Function Definitions

- Only written once
- Hidden from other functions
### 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

- **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

### 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
3.6 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 )
    {
      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
    }
    ```

- Comma separated list for multiple parameters.

- 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
    - Mixed type goes to highest type (promotion)
    - int * double

### 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)
### 3.7 Header Files

- Header files contain
  - Function prototypes
  - Definitions of data types and constants
- Header files ending with `.h`
  - Programmer-defined header files
    ```c
    #include "myheader.h"
    ```
- Library header files
  ```c
  #include <cmath>
  ```

### 3.8 Random Number Generation

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

#### fig03_07.cpp

```c
// Fig. 3.7: fig03_07.cpp
// Shifted, scaled integers produced by 1 + rand() % 6.
#include <iostream>
#include <iomanip>
#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
  return 0;
// indicates successful termination
}
```

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

- 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

#### fig03_08.cpp

```c
// Fig. 3.8: fig03_08.cpp
// Roll a six-sided die 6000 times.
#include <iostream>
#include <iomanip>
#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 die
  for (int roll = 1; roll <= 6000; roll++) {
    face = rand() % 6 + 1;
    if (face == 1)
      frequency1++;
    else if (face == 2)
      frequency2++;
    else if (face == 3)
      frequency3++;
    else if (face == 4)
      frequency4++;
    else if (face == 5)
      frequency5++;
    else
      frequency6++;
  }
  // output results
  cout << "Die Rolls:
       1: ", frequency1, " times.
       2: ", frequency2, " times.
       3: ", frequency3, " times.
       4: ", frequency4, " times.
       5: ", frequency5, " times.
       6: ", frequency6, " times."
  return 0;
// indicates successful termination
}
```

**Output of rand() scaled and shifted to be a number between 1 and 6.**
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
  - \texttt{rand(time(0));}
  - \texttt{time(0);}
    - <ctime>
  - Returns current time in seconds
- General shifting and scaling
  - \texttt{Number = shiftingValue + rand() \% scalingFactor}
  - shiftingValue = first number in desired range
  - 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 (\texttt{case:})
- Block scope
  - Begins at declaration, ends at right brace \texttt{)}
  - Can only be referenced in this range
  - Local variables, function parameters
  - \texttt{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

fig03_12.cpp

```cpp
// Fig. 3.12: fig03_12.cpp
// A scoping example.

#include <iostream>

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

void useLocal(void);
// function prototype

void useStaticLocal(void);
// function prototype

void useGlobal(void);
// function prototype

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; // local variable within new scope
    cout << "local x in main's inner scope is " << x << endl;
  } // end new scope

  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 is " << x << endl;
  return 0; // indicates successful termination
}
```

// Fig. 3.12: fig03_12.cpp (1 of 5)
// A scoping example.

#include <iostream>

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

void useLocal(void);
// function prototype

void useStaticLocal(void);
// function prototype

void useGlobal(void);
// function prototype

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; // local variable within new scope
    cout << "local x in main's inner scope is " << x << endl;
  } // end new scope

  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 is " << x << endl;
  return 0; // indicates successful termination
}
```
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

// Fig. 3.14: fig03_14.cpp
// Recursive factorial function.

#include <iostream>

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

unsigned long factorial(unsigned long n)
{
  // Base case (1! = 0! = 1)
  if ( n == 1 ) return 1;
  // Recursive relation ( n! = n * ( n - 1 ) * ... * 1)
  else return n * factorial( n - 1 );
}

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

3.12 Recursion

- Example: factorial
  - Recursive relationship ( n! = n * ( n - 1 ) * ... * 1 )
  - Base case (1! = 0! = 1)
The base case occurs when we have 0 or 1. All other cases must be split up (recursive step).

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

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:
    \( f(n) = f(n-1) + f(n-2) \)
- C++ code for Fibonacci function
  ```c++
  long fibonacci( long n )
  {
      if ( n == 0 || n == 1 ) // base case
          return n;
      else
          return fibonacci( n - 1 ) + fibonacci( n - 2 );
  }
  ```

Order of operations
- 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

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

```c++
unsigned long factorial( unsigned long number )
{
    // base case
    if ( number <= 1 )
        return 1;
    // recursive step
    else
        return number * factorial( number - 1 );
}
```
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 )
  - Function call format the same
    - However, original can now be changed

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

Changes numberRef, an alias for the original parameter. Thus, width is changed.
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
    - int count = 1; // declare integer variable count
    - int &ref = count; // create ref as an alias for count
    - ++ref; // increment ref
- References must be initialized when declared
  - Otherwise, compiler error
  - Dangling reference
  - Reference to undefined variable

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)
    - myFunction(3, 5)
    - myFunction(3, 5, 7)

Function calls with same parameters taking the rightmost parameters get their defaults.