Chapter 6 – User-defined Functions

Why are user-defined functions important?

- reduces the complexity of the main function
- improves the readability of the program
- allows a programmer to develop the solution to a problem in a top-down fashion – identify subproblems
- allows multiple programmers to efficiently work on a large project
- programmer can focus on implementing and debugging each function
- once a function has been debugged it can be used multiple times in the same program (or in different programs) without repeating the code

C++ has 2 types of functions

- A **value-returning function** is designed to compute and return EXACTLY ONE value using a return statement.
- A **void function** is a series of statements designed to perform a task. This type of function does not have a specific data type.

What is needed to add a function to a C++ program?

- **function definition** – consists of a heading and a body, placed after the main function
- **function prototype** – placed after using statement and const declarations, before heading for main function – provides information to compiler about the function
- **function call** – placed inside any function, used when you want the statements in the function to be executed

```
//preprocessor directives
using namespace std;
//const declarations
//function prototypes
int main()
{
      . . .
      //function call(s)
      return 0;
}
//function definitions (can include function calls)
```

Void functions –
A **void function** is a series of statements designed to perform a task. This type of function does not have a specific data type.
Typical situations when a void function is appropriate

- function will generate some type of output
- function performs a task that does not directly result in the calculation of a single value
- function performs a task that involves changing one or more values passed to it

void function definition

```
void function_name (formal parameter list)
//Describe what function does, what is passed in and out.
{
   //statements implementing function
   //can be any C++ statements including function calls
   return; //optional statement that returns control to
           //function that called current function
}
```

parameter – a parameter is a value that is passed (communicated) to a function
parameters should be the only way that functions share data

formal parameter list

- can be empty – no values shared between functions
- **pass by value** – a copy of the value of the actual parameter (what is supplied in the function call) is transmitted to the function for its use

syntax:
`(data_type parameter1, data_type parameter2, ...)`
example:
`(int length, int width, char symbol)`

function documentation – for each function provide a brief statement with the definition of

- input – what parameters are passed in and/or what will be read by the function
- output – what parameters are passed back and/or what will be printed by the function
- what does the function do? – often can be explained with the input/output

void function example – design a function that will draw a rectangle when given its length (int) and width (int) and a symbol to draw with

```
void draw_rectangle(int length, int width, char sym)
//Given the length and width of a rectangle, and a character to draw
//with (sym) draw a rectangle on the default output device.
{
   for (int i=0; i<length; i++)
   {
      for (int j=0; j<width; j++)
         cout << sym;
      cout << endl;
   }
}
```

function prototype – serves as a declaration to the compiler for a function

- specifies the function's type, name, parameter list
- should be placed before the main function
- implementation – copy the heading of the function, terminate with a semi-colon, names of parameters are optional

syntax:

```
   function_type function_name(parameter list);
```

example:

```
   void draw_rectangle(int, int, char);
```

function call – a statement used to invoke a function (cause it to be executed)

- statement can be placed in a function when you want its task to be performed
- if there are no parameters, parentheses must be present
- actual parameter list is a list of the values to be passed to the function for its use
  - do not include data types in an actual parameter list
  - make sure the number of actual parameters, their data types, and their order match the function heading and prototype parameter lists
- if parameters are passed by value, actual parameters can be
  - literal values
  - variables
  - expressions
- the names of actual parameters and formal parameters DO NOT have to be the same

syntax:

```
   function_name(actual parameter list);
```

examples:

```
      draw_rectangle(4,3,'$');
      draw_rectangel(lngth,width,draw);
      draw_rectangle(lngth,lngth+3,'#');
```

```
#include <iostream>  //~lee/cs135sampledir/draw.cpp
using namespace std;
void draw_rectangle(int,int,char);
int main()
{
   int lngth;
   int width;
   char draw;
   draw_rectangle(4,3,'$');
   cout << endl;
   cout << "Enter length followed by width of rectangle" << endl;
   cin >> lngth >> width;
   cout << "What character do you want to draw with?" << endl;
   cin >> draw;
   draw_rectangle(lngth,width,draw);
   cout << endl;
   lngth = 5;
   draw_rectangle(lngth,lngth+3,'#');
   return 0;
}
void draw_rectangle(int length, int width, char sym)
//Given the length and width of a rectangle, and a character to draw
//with (sym) draw a rectangle on the default output device.
{
   for (int i=0; i<length; i++)  // i is local to the for statement
   {
      for (int j=0; j<width; j++)
         cout << sym;
      cout << endl;
   }
}
```

local variable – a variable declared inside a function or program block is said to be local to the function/block

- when function/block is entered, it comes into existence at declaration
- ceases to exist when the function/block is exited

Value-returning functions –
A **value-returning function** is a series of statements designed COMPUTE AND RETURN EXACTLY ONE value.

- function will have a data type
- must include a return statement – method for passing back the value of the function

value-returning function definition

```
data_type function_name (formal parameter list)
{
   //statements implementing function
   //can be any C++ statements including function calls
   return expression;     //expression represents the value of
                          //the function – data type should match
                          //the data type of the function
}
```

parameters and value-returning functions

- parameters passed to value-returning functions are almost always passed by value

value-returning function example – design a function that will compute and return the perimeter of a rectangle when given its length (int) and width (int)

```
int rectangle_perimeter(int length, int width)
//Given the length and width of a rectangle compute and return its
//perimeter.
{
   int perimeter;                    //perimeter is local to the function
   perimeter = 2*length + 2*width;
   return perimeter;
}
```

function prototype – serves as a declaration to the compiler for a function

- specifies the function's type, name, parameter list
- should be placed before the main function
- implementation – copy the heading of the function, terminate with a semi-colon, names of parameters are optional

syntax:

```
   function_type function_name(parameter list);
```

example:

```
   int rectangle_perimeter(int, int);
```

value-returning function call – used to invoke the function (cause it to be executed)

- call is not a statement – represents the value returned by the function
- can be placed used anywhere in a statement where a value of the type returned by the function would be appropriate
- parameter list rules – see void function call explanation

syntax:

```
function_name(actual parameter list)
```

examples:

```
perimeter = rectangle_perimeter(len,wid);
cout << rectangle_perimeter(7,width) << endl;
```

```
#include <iostream>  //~lee/cs135sampledir/drawmore.cpp
using namespace std;
int rectangle_perimeter(int,int);
void draw_rectangle(int,int,char);
int main()
{
   int lngth;    int width;
   char draw;    int perimeter;
   cout << "Enter length followed by width of rectangle" << endl;
   cin >> lngth >> width;
   cout << "What character do you want to draw with?" << endl;
   cin >> draw;
   draw_rectangle(lngth,width,draw);
   perimeter = rectangle_perimeter(lngth,width);
   cout << "Perimeter = " << perimeter << endl;
   cout << endl;
   lngth = 5;
   draw_rectangle(lngth,lngth+3,'#');
   cout << "Perimeter = " << rectangle_perimeter(lngth,lngth+3) << endl;
   return 0;
}
void draw_rectangle(int length, int width, char sym)
//Given the length and width of a rectangle, and a character to draw
//with (sym) draw a rectangle on the default output device.
{
   for (int i=0; i<length; i++)
   {
      for (int j=0; j<width; j++)
         cout << sym;
      cout << endl;
   }
}
int rectangle_perimeter(int length, int width)
//Given the length and width of a rectangle compute and return its
//perimeter.
{
   int perimeter;
   perimeter = 2*length + 2*width;
   return perimeter;
}
```

Passing parameters to void and value-returning functions

- most of the time the parameters passed to a value-returning function should be passed by value
- if a value-returning function is going to compute more than one value it is best to use a void function instead
- a void function that is designed to change one or more of its parameters as a result of its actions must pass those parameters by reference
- best practice, if a parameter must be permanently changed pass by reference, if it should not be permanently changed pass by value

Pass by reference parameters

- the memory location of the actual parameter is sent to the function
- pass by reference is indicated in the function heading and prototype by the addition of an ampersand (&) after the data type in the formal parameter list
- any change made inside the function will change the value of the actual parameter (in the calling function)

syntax:

```
void function_name(datatype& p1, datatype& p2, . . ., datatype& pn)
```

example:

```
void classify_number(int n, int& positive, int& negative, int& zero)
```

Given a file of integers, do the following (use functions)

- count and display the number of positive, negative, and zero values in the file
- if a value is between 0 and 12, inclusive, compute and display value! (factorial) as an int
- compute and display the number of digits and the average of the digits in each value

```
void classify_number(int n, int& positive, int& negative, int& zero)
{
  if (n > 0)
    positive++;
  else
    if (n < 0)
      negative++;
    else
      zero++;
}
```

```
int factorial(int num)
{
  int product=1;
  if (num==0 || num==1)
    return 1;
  for (int i=num; i>1; i--)
    product = product * i;
  return product;
}



void count_average(int n)
{
  int count=0;
  int digit;
  double sum=0.0;
  cout << n << " has ";
  if (n == 0)
    count = 1;
  if (n < 0)
    n = n * -1;
  while(n != 0)
    {
      digit = n % 10;
      sum = sum + digit;
      count++;
      n = n / 10;
    }
  cout << count << " digit(s), the average of its digit(s) is "
       << sum/count << endl;
}
```

```
#include <iostream>    //~lee/cs135sampledir/numbers.cpp
using namespace std;   //~lee/cs135sampledir/numdata
void classify_number(int,int&,int&,int&);
int factorial(int n);
void count_average(int);
int main()
{
  int num;      int pos=0, neg=0, zero=0;
  cin >> num;
  while (cin)
    {
      classify_number(num,pos,neg,zero);
      if (num >= 0 && num <= 12)
        cout << num << "! is " << factorial(num) << endl;
      count_average(num);
      cout << endl;
      cin >> num;
    }
  cout << "The file contained " << pos << " positive numbers, "
       << neg << " negative numbers and " << zero << " zeroes\n";
  return 0;
}
void classify_number(int n, int& positive, int& negative, int& zero)
{
  if (n > 0)
    positive++;
  else
    if (n < 0)
      negative++;
    else
      zero++;
}
int factorial(int num)
{
  int product=1;
  if (num==0 || num==1)
    return 1;
  for (int i=num; i>1; i--)
    product = product * i;
  return product;
}
void count_average(int n)
{
  int count=0;
  int digit;
  double sum=0.0;
  cout << n << " has ";
  if (n == 0)
    count = 1;
  if (n < 0)
    n = n * -1;
  while(n != 0)
    {
      digit = n % 10;
      sum = sum + digit;
      count++;
      n = n / 10;
    }
  cout << count << " digit(s), the average of its digit(s) is "
       << sum/count << endl;
}
```