CSc 10200
Introduction to Computing

Lecture 10-11
Edgardo Molina
Fall 2013 – City College of New York
Chapter 6
Modularity Using Functions
Objectives

• In this chapter, you will learn about:
  – Function and parameter declarations
  – Returning a single value
  – Returning multiple values
  – Rectangular to polar coordinate conversion
  – Variable scope
  – Variable storage categories
  – Common programming errors
Functions

- Functions allow us to organize complex programs
- Allow us to **reuse** code
- A function should have a **well defined** operation
- It takes values as inputs (**arguments**)
- It outputs some value (**return value**)

We’ve been using some functions:

```cpp
setprecision(2);
pow(3.0, 2.0);
cos(x);
...
```
Functions

- 3 parts
  - Function prototype
  - Function body
  - Function call

```cpp
double my_function(int, int);
```

return type  function name  parameter type list
Functions

• 3 parts
  – Function prototype
  – Function body
  – Function call

```c
double my_function(int x, int y) {
    double result;
    result = x*x + y*y/x;
    return result;
}
```
Functions

- 3 parts
  - Function prototype
  - Function body
  - Function call

```c
int main() {
    double a, b, z;
    ...
    z = my_function(a, b);
    ...
}
```

Another function

Argument list
Function and Parameter Declarations

• Interaction with a function includes:
  – **Passing** data to a function correctly when its called
  – **Returning** values from a function when it ceases operation

• A function is called by giving the function’s name and passing arguments in the parentheses following the function name

```
function-name (data passed to function);
```

This identifies the called function
This passes data to the function

**Figure 6.1** Calling and passing data to a function
Function and Parameter Declarations (continued)

Program 6.1

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

void findMax(int, int); // the function declaration (prototype)

int main()
{
    int firstnum, secnum;

    cout << "\nEnter a number: ";
    cin >> firstnum;
    cout << "Great! Please enter a second number: ";
    cin >> secnum;

    findMax(firstnum, secnum); // the function is called here

    return 0;
}
```

Prototype declaration

Function call

Where is the Function definition?
Function and Parameter Declarations (continued)

• Before a function is called, it must be declared to function that will do calling

• Declaration statement for a function is referred to as function prototype

• Function prototype tells calling function:
  – Type of value that will be formally returned, if any
  – Data type and order of the values the calling function should transmit to the called function

• Function prototypes can be placed with the variable declaration statements above the calling function name or in a separate header file
Calling a Function

• Requirements when calling a function include:
  – Using the name of the function
  – Enclosing any data passed to the function in the parentheses following the function name, using the same order and type declared in the function prototype
Calling a Function (continued)

- The items enclosed in the parentheses are called **arguments** of the called function

\[
\text{findMax} \ (\text{firstnum, secnum});
\]

- This identifies the \text{findMax()} function
- This causes two values to be passed to \text{findMax()}

**Figure 6.2** Calling and passing two values to \text{findMax()}
Calling a Function (continued)

Figure 6.3 The `findMax()` function receives actual values
Calling a Function (continued)

• Figure 6.3:
  – The `findMax()` function does not receive the variables named `firstnum` and `secnum` and has no knowledge of the variable names.
  – The function receives the values in these variables and must then determine where to store those values before it does anything else.
Defining a Function

• Every C++ function consists of two parts:
  • Function header
  • Function body
• Function header’s purpose:
  – Identify data type of value function returns, provide function with name, and specify number, order, and type of arguments function expects
• Function body’s purpose:
  – To operate on passed data and return, at most, one value directly back to the calling function
Defining a Function (continued)

**Figure 6.4** The general format of a function

```cpp
function header line
{
    constant and variable declarations;
    any other C++ statements;
}
```

**Figure 6.5** Storing values in parameters

```cpp
findMax(firstnum, secnum);
```

The value in `firstnum` is passed

The value in `secnum` is passed

```cpp
findMax(int x, int y)
```

The parameter named `x`

The parameter named `y`
Placement of Statements

• General rule for placing statements in a C++ program:
  – All preprocessor directives, named constants, variables, and functions must be declared or defined before they can be used
  – Although this rule permits placing both preprocessor directives and declaration statements throughout the program, doing so results in poor program structure
Functions with Empty Parameter Lists

- Although useful functions having an empty parameter list are extremely limited, they can occur.
- Function prototype for such a function requires writing the keyword `void` or nothing at all between the parenthesis following the function’s name.

```c
void foo();
void foo(void);
```
Default Arguments

• C++ provides **default arguments** in a function call for added flexibility

• Primary use of default arguments is to extend parameter list of existing functions without requiring any change in calling parameter lists already used in a program

• Default arguments are listed in the function prototype and transmitted automatically to the called function when the corresponding arguments are omitted from the function call
Default Arguments (continued)

• Example: Function prototype with default arguments
  void example(int, int = 5, double = 6.78)

• Sample valid calls to the example function
  example(7, 2, 9.3) // no defaults used
  example(7, 2)     // same as example(7, 2, 6.78)
  example(7)       // same as example(7, 5, 6.78)
Reusing Function Names (Overloading)

• C++ provides the capability of using the same function name for more than one function
  – Referred to as function overloading
• Only requirement for creating more than one function with same name:
  – Compiler must be able determine which function to use based on the parameters’ data types (not the data type of the return value, if any)
Reusing Function Names (Overloading) (continued)

• Which of the functions is called depends on the argument type supplied at the time of the call

```cpp
void cdabs(float x) // compute and display the absolute value of a float
{
    if ( x < 0 )
        x = -x;
    cout << "The absolute value of the float is " << x << endl;
}

void cdabs(double x) // compute and display the absolute value of a double
{
    if ( x < 0 )
        x = -x;
    cout << "The absolute value of the double is " << x << endl;
}
```
Returning a Single Value

- Function receiving an argument passed by value cannot inadvertently alter value stored in the variable used for the argument.
- Function receiving passed by value arguments can process the values sent to it in any fashion and return one, and only one, “legitimate” value directly to the calling function.

Figure 6.6 A function directly returns at most one value.
Returning a Single Value (continued)

- **findmax function**

```cpp
int findMax(int x, int y) // function header
{
    int maxnum;           // start of function body
    if (x >= y)
        maxnum = x;
    else
        maxnum = y;

    return maxnum;        // return statement
}
```
Inline Functions

• Calling a function places a certain amount of overhead on a computer
  – Placing argument values in a reserved memory region (called the stack) that the function has access to
  – Passing control to the function
  – Providing a reserved memory location for any returned value (again, using the stack for this purpose
  – Returning to the correct point in the calling program
Inline Functions (continued)

• Paying overhead associated with calling a function is justified when a function is called many times
  • Can reduce a program’s size substantially
• For small functions that are not called many times, overhead of passing and returning values might not be warranted
• **Inline functions:**
  – Group repeating lines of code together under a common function name
  – Have the compiler place this code in the program wherever the function is called
#include <iostream>
using namespace std;

inline double tempvert(double inTemp) // an inline function
{
    return (5.0/9.0) * (inTemp - 32.0);
}

int main()
{
    const CONVERTS = 4;    // number of conversions to be made
    int count;
    double fahrenheit;

    for(count = 1; count <= CONVERTS; count++)
    {
        cout << "Enter a Fahrenheit temperature: ";
        cin >> fahrenheit;
        cout << "The Celsius equivalent is "
            << tempvert(fahrenheit) << endl;
    }

    return 0;
}
Inline Functions (continued)

• Advantage: Increase in execution speed
  – Because the inline function is expanded and included in every expression or statement calling it, no execution time is lost because of the call and return overhead a non-inline function requires

• Each time an inline function is referenced the complete code is reproduced and stored as an integral part of the program

• A non-inline function is stored in memory only once

• Inline functions should be used only for small functions that aren’t called extensively in a program
Variable Scope

- A function can be thought of as a closed box, with slots at the top to receive values and a single slot at the bottom to return a value.

**Figure 6.14** A function can be considered a closed box.
Variable Scope (continued)

- **Local variables:** Variables created in a function that are conventionally available only to the function
- **Scope:** Section of the program where the identifier is valid or “known”
- A variable with **local scope** is simply one with storage locations set aside for it by a declaration statement inside the function that declared them
- A variable with **global scope** has storage created for it by a declaration statement located outside any function
Variable Scope (continued)

Figure 6.15  The three storage areas reserved by Program 6.15
Scope Resolution Operator

- When a local variable has the same name as a global variable, all references to the variable name made within the local variable’s scope refer to the local variable.

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

double number = 42.8; // a global variable named number

int main()
{
    double number = 26.4; // a local variable named number
    cout << "The value of number is " << number << endl;
    return 0;
}
```
Scope Resolution Operator (continued)

- To reference a global variable when a local variable of the same name is in scope, use C++’s scope resolution operator, which is `::`

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

double number = 42.5;     // a global variable named number

int main()
{
    double number = 26.4;  // a local variable named number

    cout << "The value of number is " << ::number << endl;

    return 0;
}
```
Misuse of Globals

• Global variables allow programmers to “jump around” the normal safeguards provided by functions

• Instead of passing variables to a function, it is possible to make all variables global: *do not do this*
  – Indiscriminate use of global variables destroys the safeguards C++ provides to make functions independent and insulated from each other
  – Using only global variables can be especially disastrous in large programs with many user-created functions
Stop here
Variable Storage Categories

• A variable’s scope can be thought of as the space in the program where the variable is valid.
• In addition to space dimension represented by scope, variables have a time dimension that refers to the length of time storage locations are reserved for a variable.
• This time, dimension is referred to as the variable’s lifetime.
• When and how long a variable’s storage locations are kept before they are released can be determined by the variable’s storage category.
Variable Storage Categories (continued)

• The four available storage categories are:
  – auto
  – static
  – extern
  – register

```c++
auto int num; // auto storage category and int data type
static int miles; // static storage category and int data type
register int dist; // register storage category and int data type
extern int volts; // extern storage category and int data type
auto float coupon; // auto storage category and float data type
static double yrs; // static storage category and double data type
extern float yld; // extern storage category and float data type
auto char inKey; // auto storage category and char variable type
```
Local Variable Storage Categories

• Local variables can be members only of the auto, static, or register storage categories
  – auto is short for automatic
• Storage for automatic local variables is reserved or created automatically each time a function
  – As long as the function hasn’t returned control to its calling function, all automatic variables local to the function are “alive”
• A local static variable isn’t created and destroyed each time the function declaring it is called
  – Local static variables remain in existence for the program’s lifetime
Local Variable Storage Categories (continued)

Program 6.17

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

void testauto();  // function prototype

int main()
{
  int count;     // count is a local auto variable
  for(count = 1; count <= 3; count++)
    testauto();

  return 0;
}

void testauto()
{
  int num = 0;  // num is a local auto variable
    // initialized to 0
  cout << "The value of the automatic variable num is ",
    << num << endl;
  num++;

  return;
}
```
Local Variable Storage Categories (continued)

Program 6.18

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

void teststat();  // function prototype
int main()
{
    int count;   // count is a local auto variable

    for(count = 1; count <= 3; count++)
        teststat();

    return 0;
}

void teststat()
{
    static int num = 0;   // num is a local static variable
    cout << "The value of the static variable num is now "
         << num << endl;
    num++;

    return;
}
```
Local Variable Storage Categories (continued)

• Most computers have a few high-speed storage areas, called **registers**, located in the CPU that can also be used for variable storage
  – Because registers are located in the CPU, they can be accessed faster than normal memory storage areas located in the computer’s memory unit
Global Variable Storage Categories

• Global variables are created by definition statements external to a function
• By their nature, global variables do not come and go with the calling of a function
• After a global variable is created, it exists until the program in which it’s declared has finished executing
• Global variables can be declared with the `static` or `extern` storage category, but not both
Global Variable Storage Categories (continued)

Figure 6.17 A program can extend beyond one file
Figure 6.18 Extending the scope of global variables
Common Programming Errors

• Passing incorrect data types
• Errors that occur when the same variable is declared locally in both the calling and the called functions
• Omitting the called function’s prototype before or within the calling function
• Terminating a function header with a semicolon
• Forgetting to include the data type of a function’s parameters in the function header
Chapter Summary

• A function is called by giving its name and passing any data to it in the parentheses following the name.
• A function’s return type is the data type of the value the function returns.
• Arguments passed to a function when it is called must conform to the parameters specified by the function header in terms of order, number of arguments, and specified data type.
Chapter Summary (continued)

• Functions can be declared to all calling functions by means of a function prototype
• Every variable has a storage category, which determines how long the value in the variable is retained