Functions

CS 1: Problem Solving & Program Design Using C++

Objectives

- Declare out function and parameter declarations
- Value the returning of a single value
- Refer to pass by reference
- Scope out the variable scope
- Let's get classy with a variable storage class
- Take a look at more common programming errors

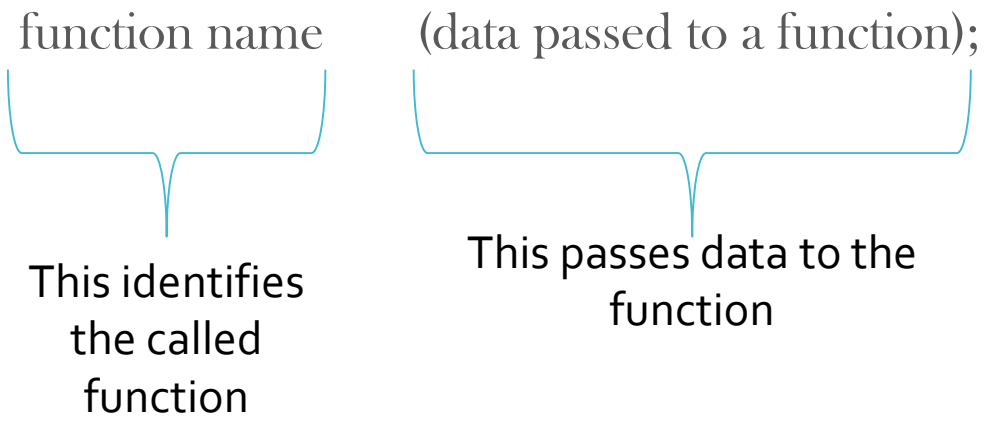
Function and Parameter Declarations

- All C++ programs must contain a main() function
  - May also contain unlimited additional functions
- Major programming concerns when creating functions
  - How does a function interact with other functions (including main)?
  - Correctly passing data to function
  - Correctly returning values from a function

Function and Parameter Declarations (2)

- Function call process
  - Give function name
  - Pass data to function as arguments in parentheses following function name
- Only after called function successfully receives data passed to it can the data be manipulated within the function

Calling and Passing Data to a Function



Function and Parameter Declarations Example

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

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

int main()
{
        int firstnum, secnum;

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

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

        return 0;
}
```

About the Function and Parameters Declaration Example

- The program not complete
  - findMax() must be written and added
  - Done in a future slide
- Complete program components
  - main(): referred to as calling program
  - findMax(): referred to as called program
- Complete program can be compiled and executed

Function Prototypes

- FUNCTION PROTOTYPE: declaration statement for a function
  - Before a function can be called, it must be declared to the calling function
  - Tells the calling function
    - The type of value to be returned, if any
    - The data type and order of values transmitted to the called function by the calling function

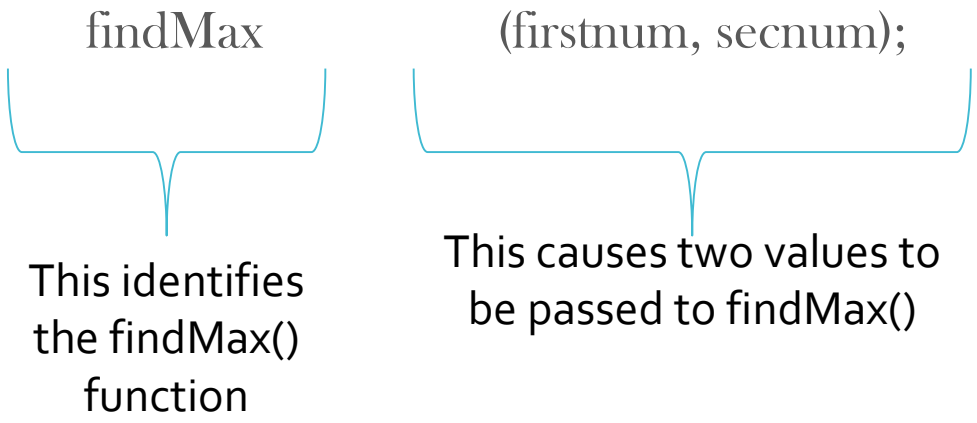
Function Prototypes (2)

- EXAMPLE: the function prototype in the example

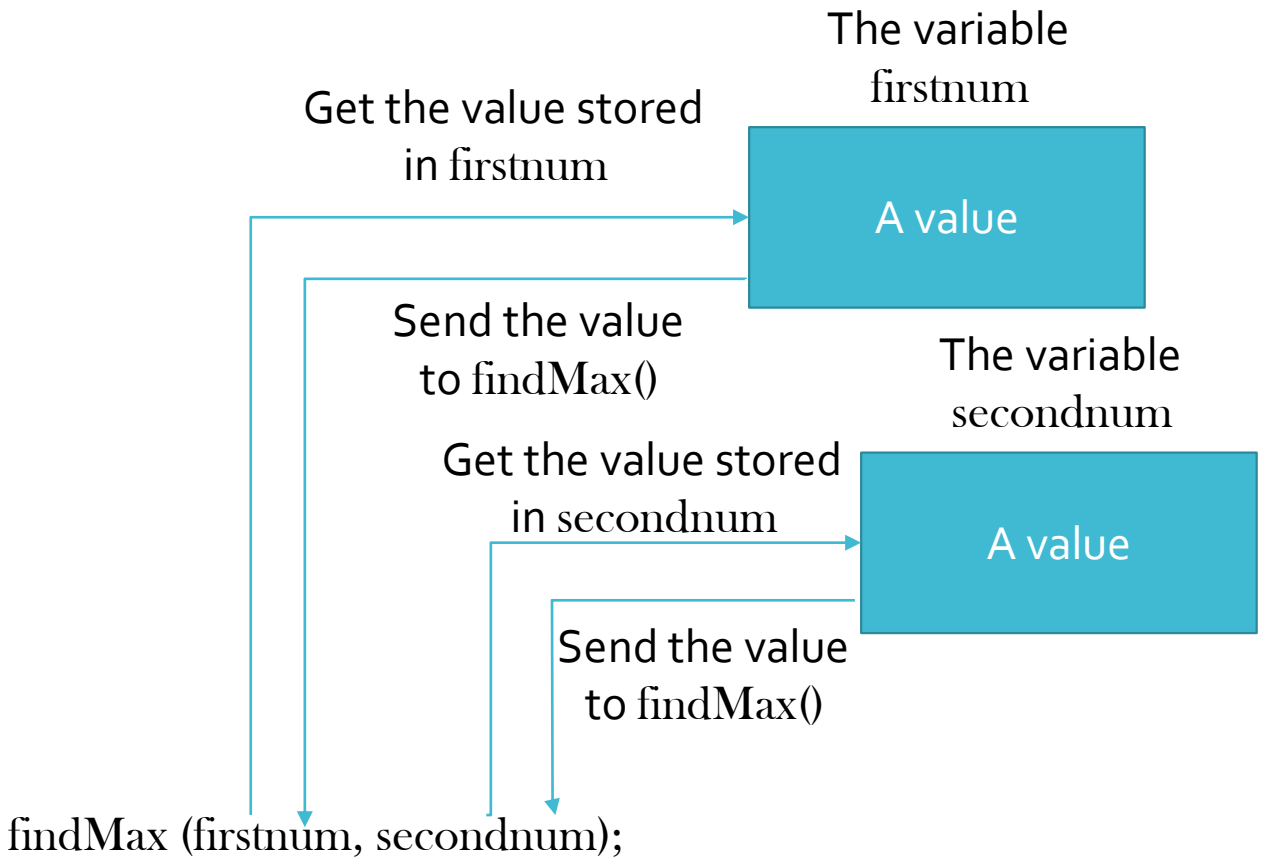
  void findMax(int, int);

  - Declares that findMax() expects two integer values sent to it
  - findMax() returns no value (void)
- Prototype statement placement options
  - Together with variable declaration statements just above calling function name (as in the example)
  - In a separate header file to be included using a #include preprocessor statement

Calling a Function



findMax() Receives Actual Values



Defining a Function

- A function is defined when it is written
  - Can then be used by any other function that suitably declares it
- FORMAT: two parts
  - Function header identifies
    - Data type returned by the function
    - Function name
    - Number, order and type of arguments expected by the function
  - Function body: statements that operate on data
    - Returns one value back to the calling function

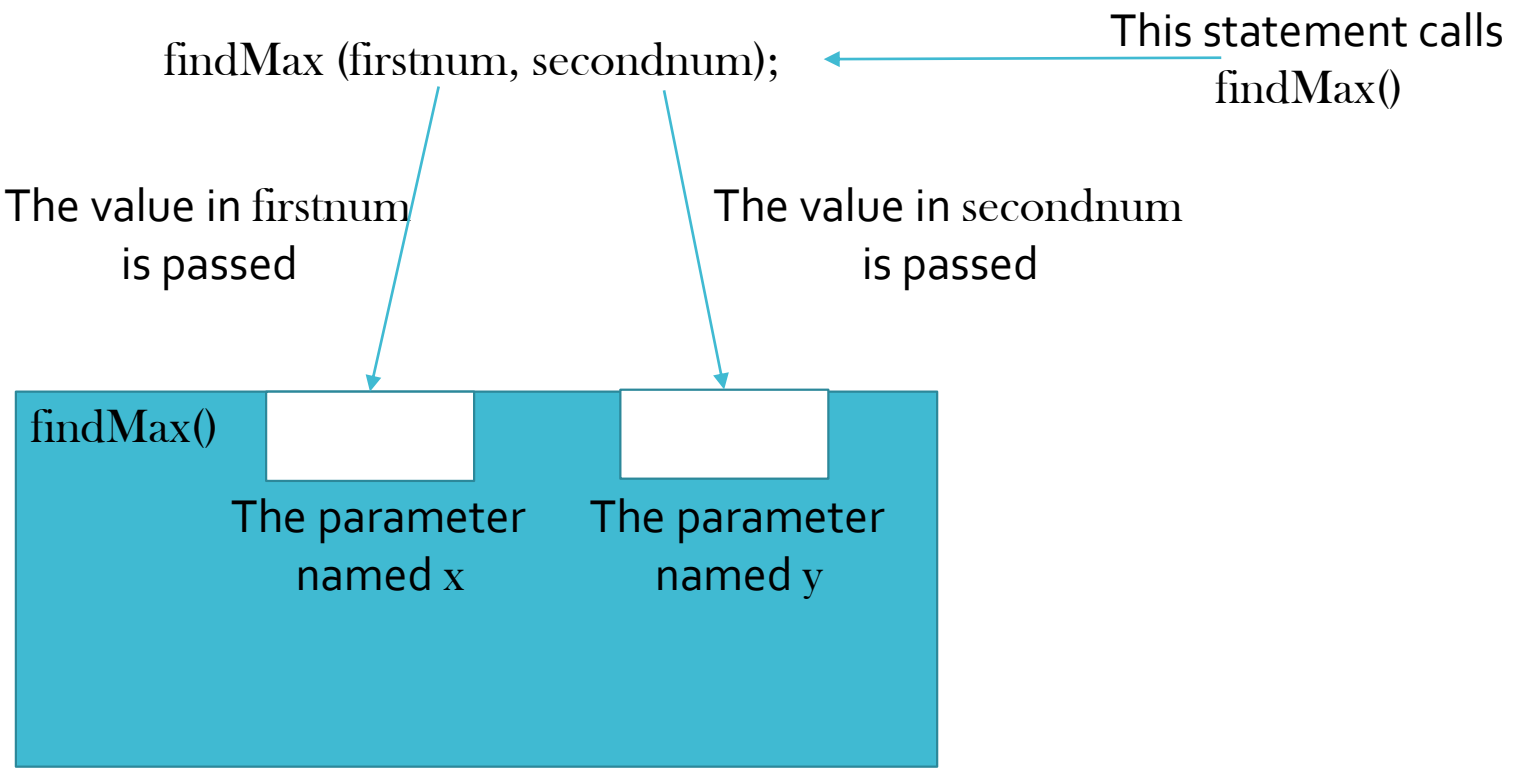
General Format of a Function

```
function header line
{
        C++ statements;
}
```

← Function header

Function body

Storing Values Into Parameters



Defining a Function

```
void findMax (int x, int y)
{ // Start of function body
      int maxnum;   // Variable declaration
      if (x >= y)        // Find the maximum number
      {
            maxnum = x;
      }
      else
      {
            maxnum = y;
      }
      cout << "\nThe maximum of the two numbers is " <<maxnum <<
            endl;
} // End of function body and end of function
```

Defining a Function (2)

- Order of functions in a program:
  - Any order is allowed
  - main() usually first
    - main() is the driver function
    - Gives reader overall program concept before details of each function encountered
- Each function defined outside any other function
  - Each function separate and independent
  - No nesting of function definitions allowed

Placement of Statements

- REQUIREMENT: items that must be either declared or defined before they are used
  - Preprocessor directives
  - Named constants
  - Variables
  - Functions
- Otherwise, C++ is flexible in requirements for ordering of statements

Recommended Ordering of Statements

```
preprocessor directives
function prototypes
int main()
{
        symbolic constants
        variable declarations
        other executable statements
        return value
}
function definitions
```

Function Stubs

- Possible programming approach
  - Write main() first and add functions as developed
  - Program cannot be run until all functions are included
- STUB: beginning of a final function
  - Can be used as a placeholder for a function until the function is completed
  - A “fake” function that accepts parameters and returns values in proper form
  - Allows main to be compiled and tested before all functions are completed

Functions with Empty Parameter Lists

- Extremely limited use
- Prototype format:

```
int display ();

int display (void);
```

- Information provided in above prototypes:
  - Display takes no parameters
  - Display returns an integer

Default Arguments

- Values listed in function prototype
  - Automatically transmitted to the called function when the arguments omitted from function call
- Example

```
void example (int, int = 5, double = 6.78);
```

  - Provides default values for last two arguments
  - Following function calls are valid:

```
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

- FUNCTION OVERLOADING: using same function name for more than one function
  - Compiler must be able to determine which function to use based on data types of parameters (not data type of return value)
- Each function must be written separately
  - Each acts as a separate entity
- Use of same name does not require code to be similar
  - GOOD PROGRAMMING PRACTICE: functions with the same name perform similar operations

Reusing Function Names – Overloading Example

```
void cdabs(int x)   // Compute and display the absolute value of an
                    //    integer
{
   if ( x < 0 )
   {
            x = -x;
   }
   cout << "The absolute value of the integer is " << x << endl;
}

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;
}
```

About the Reusing Function Names – Overloading Example

- Function call: `cdabs(10);`
  - Causes compiler to use the function named `cdabs()` that expects and integer argument
- Function call: `cdabs(6.28f);`
  - Causes compiler to use the function named `cdabs()` that expects a double-precision argument
- Major use of overloaded functions
  - Constructor functions

Function Templates

- Most high-level languages require each function to have its own name
  - Can lead to a profusion of names
- EXAMPLE: functions to find the absolute value
  - Three separate functions and prototypes required

```
void abs (int);

void fabs (float);

void dabs (double);
```

- Each function performs the same operation
  - Only difference is data type handled

Function Templates Example

```
template <class T>
void showabs(T number)
{
   if (number < 0)
   {
      number = -number;
   }
   cout << "The absolute value of the number is " << number << endl;
   return;
}
```

- Template allows for one function instead of three
  - T represents a general data type
  - T replaced by an actual data type when compiler encounters a function call

Use and Output Run of Function Templates Example

```
int main()
{
        int num1 = -4;
        float num2 = -4.23F;
        double num3 = -4.23456;
        showabs(num1);
        showabs(num2);
        showabs(num3);
        return 0;
}
```

- Output from above program:

The absolute value of the number is 4

The absolute value of the number is 4.23

The absolute value of the number is 4.23456

Returning a Single Value

- Passing data to a function
  - Called function receives only a copy of data sent to it
  - Protects against unintended change
  - Passed arguments called pass by value arguments
  - A function can receive many values (arguments) from the calling function

Returning a Single Value (2)

- Returning data from a function
  - Only one value directly returned from function
  - Called function header indicates type of data returned
- Examples

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

  - findMax accepts two integer parameters and returns no value

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

  - findMax accepts two float values and returns an integer value

Inline Functions

- Calling functions has an associated overhead
  - Placing arguments in reserved memory (stack)
  - Passing control to the function
  - Providing stack space for any returned value
  - Returning to proper point in calling program
- Overhead justified when function is called many times
  - Better than repeating code

Inline Functions (2)

- Overhead not justified for small functions that are not called frequently
  - Still convenient to group repeating lines of code into a common function name
- INLINE FUNCTION: avoids overhead problems
  - C++ compiler instructed to place a copy of in-line function code into the program wherever the function is called

Inline Function Example

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

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

int main()
{
        const int CONVERTS = 4; // Number of conversions to be
        //    made
        int count;                    // Start of variable declarations
        double fahren;
```

Inline Function Example (2)

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

        return 0;
}
```

Pass By Reference

- Called function usually receives values as pass by value
  - Only copies of values in arguments are provided
- Sometimes desirable to allow function to have direct access to variables
  - Address of variable must be passed to function
  - Function can directly access and change the value stored there
- PASS BY REFERENCE: passing addresses of variables received from calling function

Passing and Using Reference Parameters

- REFERENCE PARAMETER: receives the address of an argument passed to called function
- Example: accept two addresses in function newval()
- Function header:

```
void newval (double& num1, double& num2)
```

  - Ampersand, &, means "the address of"

- Function Prototype:

```
void newval (double&, double&);
```

Variable Scope

- SCOPE: section of program where identifier is valid (known or visible)
- LOCAL VARIABLES (LOCAL SCOPE): variables created inside a function or program component
  - Meaningful only when used in expressions inside the function in which it was declared
- GLOBAL VARIABLES (GLOBAL SCOPE): variables created outside any function
  - Can be used by all functions physically placed after global variable declaration

Scope Resolution Operator

- Local variable with the same name as a global variable
  - All references to variable name within scope of local variable refer to the local variable
  - Local variable name takes precedence over global variable name
- Scope resolution operator (::)
  - When used before a variable name the compiler is instructed to use the global variable

```
::number   // scope resolution operator causes global
           // variable to be used
```

Misuse of Globals

- Avoid overuse of globals
  - Too many globals eliminates safeguards provided by C++ to make functions independent
  - Misuse does not apply to function prototypes
    - Prototypes are typically global
- Difficult to track down errors in a large program using globals
  - Global variable can be accessed and changed by any function following the global declaration

Variable Storage Class

- Scope has a space and a time dimension
- TIME DIMENSION (LIFETIME): length of time that storage locations are reserved for a variable
  - All variable storage locations released back to operating system when program finishes its run
  - During program execution interim storage locations are reserved
    - Storage class: determines length of time that interim locations are reserved
    - Four classes: auto, static, extern, register

Local Variable Storage Classes: auto Class

- Local variable can only be members of auto, static, or register class
- auto Class: default, if no class description included in variable's declaration statement
- Storage for auto local variables automatically reserved (created)
  - Each time a function declaring auto variables is called
  - Local auto variables are "alive" until function returns control to calling function

Local Variable Storage Classes: static Class

- static Storage Class: allows a function to remember local variable values between calls
  - static local variable lifetime = lifetime of program
  - Value stored in variable when function is finished is available to function next time it is called
- Initialization of static variables (local and global)
  - Done one time only, when program first compiled
  - Only constants or constant expressions allowed

Local Variable Storage Classes: static Class Example

```
#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;
}
```

Local Variable Storage Classes: static Class Example (2)

```
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 Classes: register Class

- register Storage Class: same as auto class except for location of storage for class variables
  - Uses high-speed registers
  - Can be accessed faster than normal memory areas
    - Improves program execution time
- Some computers do not support register class
  - Variables automatically switched to auto class

Global Variable Storage Classes

- GLOBAL VARIABLES: created by definition statements external to a function
  - Do not come and go with the calling of a function
  - Once created, a global variable is alive until the program in which it is declared finishes executing
  - May be declared as members of static or extern classes
- PURPOSE: to extend the scope of a global variable beyond its normal boundaries

Common Programming Errors

- Passing incorrect data types between functions
  - Values passed must correspond to data types declared for function parameters
- Declaring same variable name in calling and called functions
  - A change to one local variable does not change value in the other
- Assigning same name to a local and a global variable
  - Use of a variable's name only affects local variable's contents unless the :: operator is used

Common Programming Errors (2)

- Omitting a called function's prototype
  - The calling function must be alerted to the type of value that will be returned
- Terminating a function's header line with a semicolon
- Forgetting to include the data type of a function's parameters within the function header line

Summary

- A function is called by giving its name and passing data to it
  - If a variable is an argument in a call, the called function receives a copy of the variable's value
- Common form of a user-written function:

```
returnDataType functionName(parameter list)
{
        declarations and other C++ statements;
        return expression;
}
```

Summary (2)

- A function's return type is the data type of the value returned by the function
  - If no type is declared, the function is assumed to return an integer value
  - If the function does not return a value, it should be declared as a void type
- Functions can directly return at most a single data type value to their calling functions
  - This value is the value of the expression in the return statement

Summary (3)

- REFERENCE PARAMETER: passes the address of a variable to a function
- FUNCTION PROTOTYPE: function declaration
- SCOPE: determines where in a program the variable can be used
- VARIABLE CLASS: determines how long the value in a variable will be retained