Programming in C/C++

Lecture 6: Functions

Kristina Shroyer

Objectives

You should be able to describe:

- Function and Parameter Declarations
- Function Prototypes
- Returning a Single Value
- Pass By Value
- Pass by Reference
- Variable Scope
- Common Programming Errors

A Note (if you have recommended prereq)

- ***If you don't have the prereq this won't make sense right now, come back and re-read this after the lecture***

- Something I've noticed that some people with prior programming experience do that they shouldn't
  - (some people in my CSIS 112 did this – I think they got old style code off the internet because I didn't teach it)
  - The point of functions is NOT to move code around on the page
    - If a function doesn't process data in some way it's useless – likewise, a function should perform ONE task
    - This means
      - void functions with no arguments are not worthwhile and merely move code around on the page in MOST cases (all cases in our class)
      - void functions that take arguments are not as useless as ones with noarguments but be careful – most of the time in our class I want functions that take arguments and return data
        - You won't get ANY CREDIT for 1970's style code (or 1980s or 1990s or 2000s style code) that "works" – code must be well designed and reusable or it is not good code for TODAY (THE PRESENT)
    - I also saw some main functions (methods in Java like this – WITH ONE LINE IN THEM A CALL TO ANOTHER METHOD!!!! – nooo!)

```
int main()
{
  aFunction();
}
```

      - DO NOT DO THIS FOR ANY REASON THIS JUST MOVES CODE AROUND ON THE PAGE AND CREATES ALL SORTS OF SCOPE ISSUES

Introduction to Functions

- What is a ***function***? (equivalent of a method in Java)
  - a collection of statements that performs a specific task or solves a specific problem
  - Better definition: a function is a small program that performs a specific task (so it is ***reusable***). The function optionally (but in almost all cases) takes some input, processes that input data in some way and optionally (but in most cases) returns some output (functions that take input and return output are better because they are more re-usable)

- All C++ programs must contain a `main()` function
  - A C++ may also contain unlimited additional functions
  - In a procedural program main drives the program – it does most of the work and decides which functions to "call" execute – so main uses the other mini programs (functions) as tools to help it accomplish the main task

- Ways we've used functions so far in our programs
  - `main()`
  - Library functions such as `pow, sqrt, rand`

- ***Why create additional functions?***
  - Break up program into small, manageable units
    - More organized, readable
    - Real world programs can easily have thousands of lines of code
      - Programs that are more readable are easier to debug and maintain
    - ***Code reuse (the focus in my classes)***
      - If a specific task is needed several times only write the function once
      - IF FUNCTIONS ARE ONLY USEFUL IN YOUR ONE PROGRAM THEY ARE NOT VERY REUSABLE – void functions tend to be this way
      - We don't reinvent the wheel and in real life this saves TIME AND MONEY – you REUSE CODE – code that is not reusable is not good business

Introduction to Functions

- **Benefits of Using Functions**
  - ***Simpler Code***
    - Code is simpler and easier to understand, more readable than one long sequence of statements
  - ***Code Reuse***
    - Functions reduce the duplication of code in the program
    - You only write the code to perform a task once and then can just call the function every time you need to perform that task again
  - ***Better Testing***
    - When each task within the program is contained within its own function, debugging and testing are easier
      - Programmers can isolate errors easier and can test each function individually
  - ***Faster Development***
    - Suppose a team of programmers is developing multiple programs
      - Each of the programs may have several common tasks
      - It doesn't make sense to re-write the code for these tasks in each program
      - Functions can be written for each common task and then incorporated into each program
  - ***Easier Facilitation of Teamwork***
    - Different programmers are assigned the task of writing different functions

Function Illustration

- Illustration: A program with and without functions in addition to the main function
  - Note: The functions do not need to be used in order, they may be called by the main function (or any other function) at any time during the program execution
- Breaking a program into small pieces makes it easier to read and understand
  - Also allows for reuse of functions – so if you use the code in the method more than once you don't have to rewrite the same code more than once
- ***NOTICE THE BLOCK SCOPE OF THE FUNCTIONS***
  - Statements and variables used inside a function are only visible inside the curly braces that define that function's definition

```
#preprosser directives

int main()
{
  statement;
  statement;
  statement;
  statement;
  statement;
  statement;
  statement;
  statement;
  statement;
  statement;
 statement;
  statement;
  statement;
  statement;
  return 0;
}
```

Question: can you "see" inside the pow function when you use it? How does your program communicate with the pow function?

```
#preprocessor directives
function prototypes

 int main()
{
    statement;
    statement;
    statement;
}

 void method2()
{
    statement;
    statement;
    statement;
}

 void method3()
{
    statement;
    statement;
    statement;
}
```

Function and Parameter Declarations

- 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

  - **Before you use a function you must know these three things:**
    1. What type of data it takes as parameters
    2. What type of data it outputs
    3. What it does (description)

- ***NOTE:*** To be reusable a function should perform ***ONE task that many programs can use, it should NOT BE PROGRAM SPECIFIC***

Defining and Calling Functions

- **There are two parts to using functions in programs**
  - Since a function is a collection of programming statements that perform a specific task, that task must be determined and the set of programming statements needed to perform that task defined (so DEFINING the function)

  - Once you have the tasks a function will perform written in code (so once you've defined and programmed the function), that code does nothing until it is told to execute
    - **`main` is the only function automatically executed by C++**
    - So in order to execute a **user defined function** the part of the program wanting to use that functions must somehow activate the function it wants to use
      - This will not be done automatically as with the main function

- **Concept:** A ***function call*** is a statement that causes a function to execute. A ***function definition*** contains the statements that make up the function.

Defining Functions

- When creating a function you must write its **function definition**. All function definitions have the following parts:
  - **Return type:** A function can send a value back to the part of the program that activated it. The return type is the data type of the value that is sent from the function to the part of the program that activated it.
    - You can think of the return value of a function as a sort of output value to the program executing that function
  - **Name:** Each function should be given a descriptive name. A name of a function is an identifier. The same rules that apply to variable names (also identifiers) also apply to function names.
    - In order to call a function, the program using that function must know its name
  - **Parameter List:** The program that activates (calls) the function can send data into the function. The parameter list is a list of variables that hold the values being passed into the function.
  - **Body:** The function body is the set of statements that perform the function's operation. The body of the function is enclosed in a set of braces
- When a program activates/uses a function causing that function to execute it is said to be **calling** that function
- **Function Header**: The first line in the function that contains the return type, name, and parameter list of the function.
  - **Example:** `int main()`

Defining Functions

Function Definition

Parameter List (this one is empty because this function takes no input data from the calling part of the program)

Name

Return type

```
int main ()
 {
  //Statements that make up the
  //function body go here

 return 0;  //the return statement
 }
```

Defining Functions

- ***Void Functions:*** It is not necessary for all functions to return a value. Some functions simply perform one or more actions and then terminate. These functions are called void functions.
  - ***In procedural programming void functions should generally only be used for functions that perform a task and then terminate – they should not be used for functions that calculate something – those functions are generally more reusable when they return a value.***
  - ***AVOID OVERUSE OF VOID FUNCTIONS AS THEY ARE NOT AS REUSABLE – just learning void functions will NOT work in this class***

- **Example:**

```
void displayMessage()
{
   cout << "This void function displays a message.\n";
}
```

- Notice the format of the `displayMessage` name. The first word in the identifier starts with a lowercase letter and the second word starts with an uppercase letter. This format is not required but is used as a standard by many C++ programmers.
- The function's return type is `void`. This means the function does not return a value to the part of the program that executed it.
- Also notice the function has no return statement. It simply executes and then exits
- ***IMPORTANT NOTE (VOID FUNCTIONS ARE MAINLY A LEARNING TOOL – NOT VERY REUSABLE and very rarely the best choice in this class):*** Most functions are ***NOT*** void….a function that accepts and returns data is more useful and more reusable than a void function. Do not fall into the trap of making all of your functions void

Calling a Function

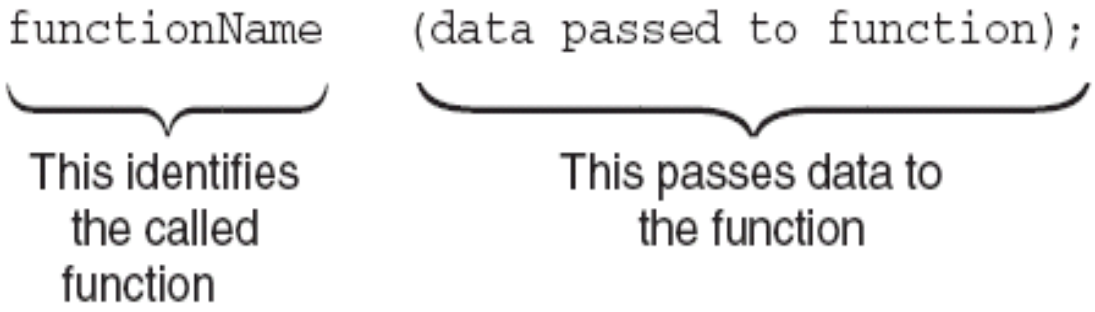
- A function is executed when it is **called**
  - Function main is automatically called when a program starts
  - All other functions must be executed by function call statements

- ***Function call process:***
  - Give function name
  - Pass data to function as arguments in parentheses following function name
  - If the function returns a value you usually want to save the output of the function into an appropriately typed variable

- Only after called function successfully receives data passed to it can the data be manipulated within the function
  - We must pass any data we want the function to see from the calling part of the program due to scope rules

Calling a Function

FIGURE 6.1 *Calling and Passing Data to a Function*



Calling a Void Function (Example)

- Example of a program with two functions: the main function and the function display message
  - The function header is part of the function definition, it displays the function's return type, name, and parameter list in that order
  - Example (***Example1*** - displayMessage.cpp)

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

/*Definition of displayMessage:  This function displays a greeting*/
void displayMessage()  //the function header
{
   cout << "Hello from the function displayMessage.\n";
}

int main()
{
   cout << "Hello from main.\n";
   displayMessage();  //the function call to the displayMessage function
   cout << "We are back in the function main again.\n";
   system("PAUSE");
   return 0;
}
```

Void Function (Example)

- **Format of the function header:** The function header is part of the function definition and is NOT terminated with a semicolon.

- **Format of the function call:** The function call is a statement that executes a function so it is terminated with a semicolon like all other C++ statements.
  - The function call must always have an argument list following the name of the function. If no data needs to be passed into the function, the parenthesis are left empty.

- Even though the function must start executing at main the function `displayMessage` is listed first before `main`.
  - This is because **the compiler must know the function's return type, number of parameters, and type of each parameter before the function is called anywhere in your program.**
    - One way to ensure this is to place the function definition before any calls to that function
    - Another way is by using a function prototype which we will go over later
      - FOR THIS CLASS YOU MUST USE THE PROTOTYPES I SHOW YOU LATER TO AVOID LOSING POINTS

- Always document your functions my writing brief comments about what they do. These comments should appear just before the function definition.

- ***A VOID FUNCTION WITH NO INPUT IS THE LEAST USEFUL, LEAST REUSABLE TYPE OF FUNCTION AND SHOULD BE AVOIDED IN THIS CLASS UNLESS SPECIFICALLY ASKED FOR***

Hierarchical Function Calls (Flow of Control)

- When a function is finished executing, control of the program passes back to the part of the program that called it at exactly the point it left off.
  - Let's Trace the flow of control in this program
- Functions may be called in a layered or hierarchical fashion
- **Example (Example 2 - flowOfControl.cpp)**

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

void C()
{
    cout << "Now we are in C.\n";
}

void B()
{
   C();
   cout << "Now we are in B.\n";
}

void A()
{
   B();
   cout << "Now we are in A.\n";
}

int main()
{
      A();
      cout << "Now we are in main.\n";
      system("PAUSE");
      return 0;
}
```

•Notice the order of the functions is not ideally what we might want for a readable program,

- •It is done this way because in C++ before a function is called the compiler must know the function's return type, number and type of parameters and name.
- •**Solution:** ***function prototypes***

•Look at what would happen if the order of the functions were changed

Function Prototypes

- **Function Prototype:** declaration statement for a function
  - Before a function can be called, it must be declared to the calling function
    - Before the C++ compiler will compile a function call it must know that functions name, return type, number of parameters and each parameter type
  - A function prototype tells contains the following information:
    - The name of the function
    - The type of returned by the function, if any
    - The number, order, and data types of the function's parameters
- A function prototype eliminates the need to place a function definition before all calls to the function
- ***IN THIS CLASS USE FUNCTION PROTOTYPES***
  - I want to see main at the top of all procedural programs

Function Prototypes

- **Prototype statement placement options:**

1. Just above the calling function name or above all functions
   - (***we'll be using this option for right now and we'll put all prototypes at the top above all the functions***)

2. In a separate header file (along with function definitions) to be included using an `#include` preprocessor statement

- At the end of this lecture we'll learn how to put our functions in separate files so they will be more reusable

Function Prototypes (Example 3 - functionPrototypes.cpp)

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

void A();
void B();
void C();

int main()
{
   A();
   cout << "Now we are in main.\n";
   system("PAUSE");
   return 0;
}

void A()
{
   B();
   cout << "Now we are in A.\n";
}

void B()
{
   C();
   cout << "Now we are in B.\n";
}

void C()
{
    cout << "Now we are in C.\n";
}
```

Calling a Function – Sending Data In

- When a function is called, the program may send values into the function (inputs)
  - Values that are passed into a function are called ***arguments***
  - In order for a function to receive arguments (inputs), the function definition must have a parameter list that requires data to be passed into it
    - So a function's parameter list specifies what types of data must be passed into a function via the arguments of the code calling that function
  - A ***parameter*** is a special variable that holds a value being passed into a function
  - Remember the ***block scope*** of functions, functions CANNOT see inside the other's scope, the only way they can communicate is through input and output

- **Note:** In this class the values being passed into the function are called arguments and the variables that receive those values are being called parameters. There are several variations of these terms in use. Some call the arguments actual parameters and the parameters formal parameters. Others use the terms formal arguments and actual arguments.
  - The terms you use in the real world are not important as long as you understand what others that use different terms are referring to.
  - Understand what these terms mean in the context of how they are used to do programming and you will easily understand what others are talking about and be able to adapt quickly.

Calling a Function – Sending Data In

- **Why do we pass values into functions?**
  - Remember a function's scope is defined by it's curly braces, a function can only see the variables and code inside of it.
  - So if the program using a function wants that function to perform some calculations on some specific values, the program must pass those values into the function
    - Otherwise the function cannot access those values due to ***scope*** limitations because they are defined in a different function (main usually)

- **What happens when values are passed (sent into) into functions?**
  - The values passed to the function are ***copied*** into it's parameter variables
    - So the parameter variable doesn't have access to the value passed into the function, instead it contains a copy of the value passed into the function so that the function can then use those values when its executing

Calling a Function – Sending Data In

- Example of a function definition that uses parameters:
  - Notice the integer variable declarations inside the parenthesis. The variables x and y are parameters
  - (x and y are the ***parameter variables***, when the function is called the values passed into the function are copied into these parameter variables).
    - They enable the function **`findMax`** to accept integer values as arguments.
    - The scope of x and y is limited to the **`findMax`** function. Their scope starts when they are declared as parameters and ends when the function finishes executing (at the closing curly brace)

```
void findMax (int x, int y)
{   //the curly brace start of function definition

      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 definition and end of function
```

Calling a Function – Sending Data In

- Example of a function definition that uses parameters:
  - The function ***prototype*** for **`findMax`**
    - **`void findMax(int, int);`**
      - In this function prototype it is not necessary to list the name of the parameters inside of the parenthesis just their data types
        - Note that the order of the parameter data types must be correct and the number of parameter data types must be correct
        - What if the **`findMax`** function had the parameters **`(double x, int y)`** ? How would the prototype change?
  - **Optional Alternate Prototype Format:** The prototype could be optionally written as:
    - **`void findMax(int x, int y);`**
    - However when written this way the compiler would ignore the parameter variables x and y
      - This type of format would only be used for readability

Calling a Function – Sending Data In

- Example of a function call that uses parameters:
  - How main would call **findMax**

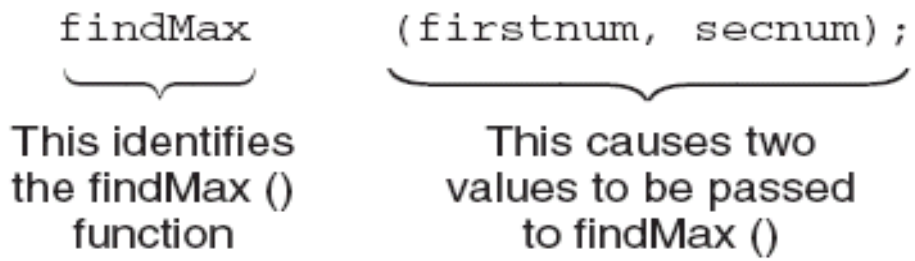
```
//first main would have to declare the variables firstnum
//and secnum as integers and assign them values with assignment
//statements
findMax(firstnum, secnum);
```

  - Alternatively main could pass integer constants into the function call:

```
findMax(5, 4);
```

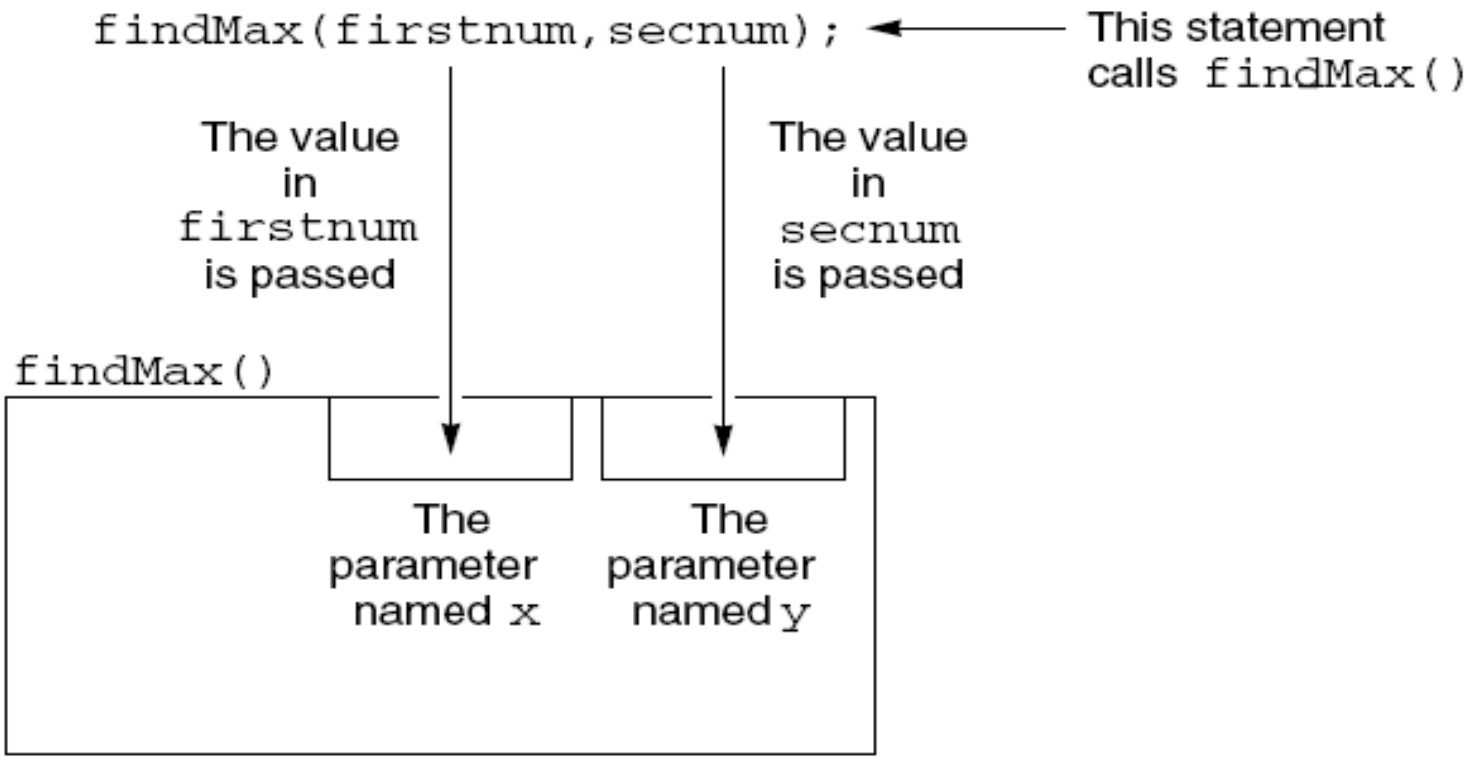
  - The arguments inside of the parenthesis of these function calls are copied into the function's parameter variables
    - In essence parameter variables are initialized to the value of their corresponding arguments (in order)
    - The order and the types of the arguments in the function call are important
      - If they types of the arguments don't match a conversion will be done if possiblel(remember coercion and no strict data typing like in Java)

FIGURE 6.2 *Calling and Passing Two Values to* `findMax()`



Calling a Function – Sending Data In

FIGURE 6.5 *Storing Values into Parameters*



Calling a Function – Sending Data In

- **Example: (Example4 - findMax.cpp)**

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

void findMax(int, int);

int main()
{
      int firstnum, secnum;

      cout << "Please enter the first number: ";
      cin >> firstnum;
      cout << "\nPlease enter the second number: ";
      cin >>secnum;

      findMax(firstnum, secnum); // variable declaration
      system("PAUSE");
      return 0;
}

/*This function finds the maximum of two numbers and
prints the result to the screen*/
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 " << x << "," << y
           << " is "<< maxnum << endl;
}   // end of function body and end of function
```

Calling a Function – Sending Data In

- What happens if you pass an argument into a function whose type is not the same as the parameter type?
  - If it's possible, the argument gets promoted or demoted automatically
  - **Example:** If you called `findMax` with the following function call: `findMax(3.1, 4.3);` what would happen?
    - The function `findMax` has two parameter variables both of which are type integer
    - **Answer:** the arguments 3.1 and 4.3 would be truncated causing the values 3 and 4 to be passed to `findMax`
    - Show in Example Program

Calling a Function – Sending Data In

- Often you will want to pass several values into a function
  - **Warning:** Each variable in the **function parameter list in the *function definition*** must have its type listed before its name. You can't leave out the data type of any variable in the parameter list

    - **Incorrect:**
```
void findMax(int x, y)
{
   code statements here;
}
```
  - **Notice this is different than the *function CALL* where you pass arguments into the function variables. The arguments in the function CALL should NOT have their type listed.**

- Like all variables, parameters have a scope. ***The scope of a parameter variable is limited to the body of the function which uses it***
  - Remember the ***scope*** **is the place where a variable can be seen and used in a program**
    - A parameter variable's scope begins when it is declared and ends at the closing curly brace of the function it was declared in

Passing Data by Value

- **Concept:** When an argument is passed into a parameter variable by value only a copy of the argument's value is passed. Changes to the parameter variable do not affect the original argument.
  - When we pass values to functions they are automatically passed by value unless we specify otherwise
  - **Parameters** are special all purpose variables defined within the parenthesis of a function *definition*
    - Their purpose is to hold the information passed to them by the arguments from the function call and usually use those arguments in some way in the body of the function
  - Normally when information is passed to a function it is ***passed by value*** This means the parameter receives a ***copy*** of the value that is passed to it.
    - Any changes made to the parameter's value inside the function has no affect on the original argument

Passing Data by Value

Example: (Example 5 - passByValue.cpp)

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

//function prototype
void changeThem(int, float);

int main()
{
   int whole = 12;
   float real = 3.5;

   cout << "In main the value of whole is: " << whole << endl;
   cout << "In main the value of real is: " << real << endl;

   changeThem(whole, real);  //function call to the changeThem function

   cout << "Now back in main the value of whole is " << whole << endl;
   cout << "Now back in main the value of real is " << real << endl;

   system("PAUSE");
   return 0;
}

/*Definition of function changeThem.  It uses i an int parameter and f a
float parameter.  The values of i and f are changed and then displayed*/
void changeThem(int i, float f)
{
   i = 100;
   f = 27.5;
   cout << "In change them the value of i is changed to " << i  << endl;
   cout << "In change them the value of f is changed to " << f  << endl;
}
```

Question: What happens to the parameters **`i`** and **`f`** when the **`changeThem`** function is finished executing?

Passing Data by Value

Example:

- Even though the parameters f and i are changed in the function *changeThem*, the arguments whole and real are not modified. The *changeThem* function does not have access to the original arguments

Original Argument (whole) (in its memory location)

Function Parameter (i) (in its memory location)

The `return` statement

DO NOT DO THIS IN THIS CLASS (points off – A LOT OF POINTS – this is no different than "break"!)

- **Concept:** The `return` statement (without anything after it) causes a function to end immediately
  - In a void function, when the last statement in a function has finished executing the function terminates (so a function that is void and has no return statement automatically terminates at the function's closing curly brace)
    - The program returns to the part of the program (usually another function) that called the function at the point immediately following the function call
  - It is possible to **force** a program to return to where it was called from **before** the last function statement has been executed (before the closing curly braces)
    - When the `return` statement is encountered the function immediately terminates and the program returns to the point immediately following the function call
    - You should only use this when you want a void function to return before the last statement in the function…before the closing curly brace
- ***Question:*** Do you think forcing a program to end with return is good design in the general sense? When should this be used? (Hint: think about the rules we have for using `break` and `continue`)

IMPORTANT: Forcing a program to end with return is NOT good design and is spaghetti code which means it's hard to read, debug and maintain. ***DO NOT DO THIS IN THIS CLASS (points off)*** - always use LOGIC to end loops, functions and selection statements. Points WILL BE taken off.

--This would only be ok in SPECIAL SITUATIONS where a rare condition forces you to have to do this – similar to break and continue (ONLY for VERY RARE situations)

The `return` statement

Example: (Example6 - quotientReturnEx.cpp)

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

//function prototype
void divide(float, float);

int main()
{
   float num1, num2;

   cout << "Enter two numbers and I will divide the 1st by the 2nd:";
   cin >> num1 >> num2;
   divide(num1, num2); //function call

   system("PAUSE");
   return 0;
}

void divide(float x, float y)
{
   //since division is not allowed by zero immediately terminate
   if(y == 0)
   {
      cout << "Sorry division by zero is not allowed.\n";
      return;
   }
   cout << "The quotient is " << x/y << endl;
}
```

Returning a Single Value

- **Concept:** A function may send (return) a value back to the part of the program that called the function
  - Think about a situation where your program passes some data to a function for it to manipulate
    - That function then processes the data in some way (say for example it calculates the total of the arguments passed into it)
    - The program that called the function is outside of the function's scope
      - What if the program that called the function wanted to use the value the function calculated in some way….it can't see the variables inside the function
        - Remember the scope rules as well as the pass by value rules
      - The solution is for the function to pass a value of meaning back to the program that called it (in this example the value of meaning would be the average)

- The idea behind using functions in a procedural program is to pass data back and forth between the functions
  - Remember main is a function
  - You almost always want to be passing data between your functions
    - ***You rarely want to use a void function unless the entire purpose of your function is to display some data***
    - Functions should be reusable and perform 1 task to reach this goal
    - ***THE FIRST PROGRAMMING MIDTERM WILL REQUIRE YOU PASS DATA IN AND OUT OF YOUR FUNCTIONS (void will not work)***

Returning a Single Value

- Example **(Example 7 - whyReturnValues.cpp)** – Example doesn't work , no return value to main (calling function)

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

//function prototype
void sum3(int, int, int);

int main()
{
    int total;
    int num1, num2, num3;

    cout << "Please enter the three numbers:";
    cin >> num1 >> num2 >> num3;

    sum3(num1, num2, num3);  //function call - DOES THE FUNCTION HELP ME GET THE AVERAGE the WAY IT'S WRITTEN?

    //NOW let's say we want to average the three numbers
    //The main function can't access the sum calculated with our function due to the function's scope.
    //We need the sum3 function to return the calculated sum to main so main
    //can use that calculated sum to then calculate the average
    system("PAUSE");
    return 0;
}

/*This void function calculates the sum of the 3 numbers passed to it as parameters. When the function ends so
does the scope of the parameter variables a,b,c and the local variable sum.  The main function can't see these
variables.  The solution is to return, or pass data back to the function. RIGHT NOW NOTHING'S HAPPENING…main
    can't see the sum variable.  We need some way to get sum back to main*/
void sum3(int a, int b, int c)
{
    int sum;
    sum =  a + b + c;
    cout << "The sum is " << sum << endl;
}
```

Returning a Single Value

- Passing (sending) 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/part of the program**

- Returning data (sending output) from a function
  - A function may ***only return one value*** (of the specified data type) to the part of the program that called it
    - just like only copies of arguments are sent into a function only a copy of a value is sent out of a function

Returning a Single Value

- **Returning data from a function**
  - Only one value directly returned from function
    - Think of a function as having multiple communications channels for receiving data but only one channel for sending data out (the return value)
  - Called function header indicates type of data returned
    - The data type for the return value precedes the function name

- **Examples**:

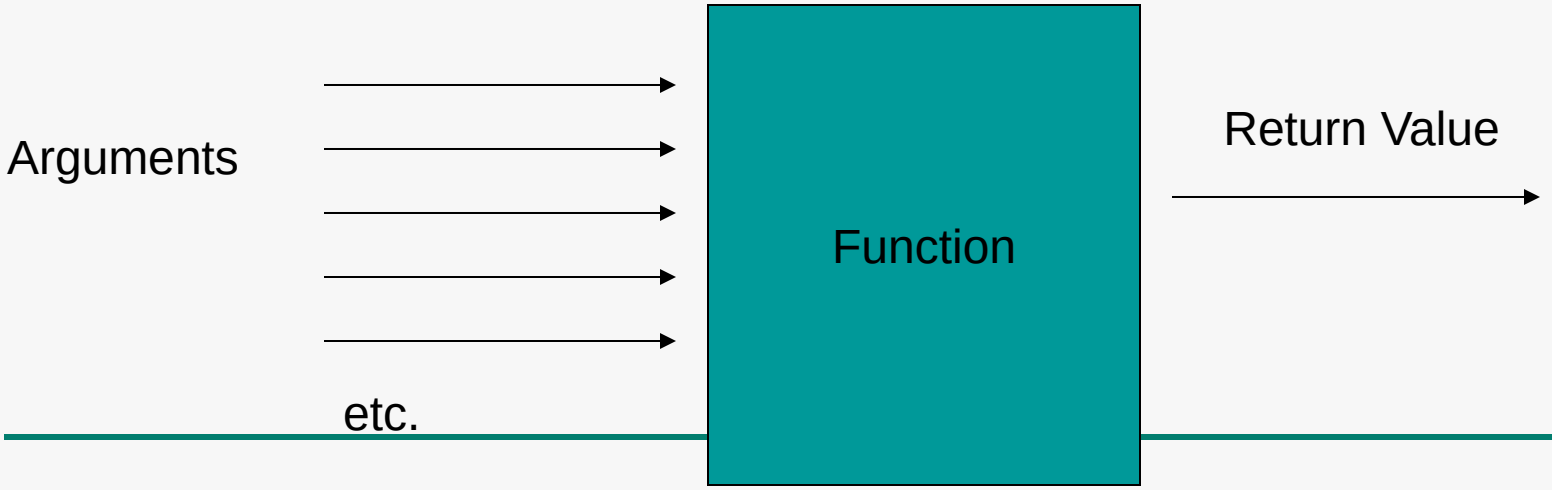
```
void findMax(int x, int y)
```

  - **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

- Note that is possible to return multiple values from a function IF they are packaged in a such a way that they are treated as a single value.
  - We'll learn about this when we learn structs later in the class.



Returning a Single Value (Example 8)

Example: (Example 8 - squareReturn.cpp)

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

//function prototype
int square(int);

int main()
{
   int number, result;

   cout << "Enter a number to be squared: ";
   cin >> number;
   result = square(number);  //notice the function call
   cout << number << " squared is " << result << endl;
   system("PAUSE");
   return 0;
}

/*This function accepts an int argument and returns the square of the argument as an int*/
int square(int num)
{
 return num * num;
}
```

Returning a Single Value (Example 8)

- This is the line that calls the square function
  - `result = square(number);`
    - An expression is something that has a value
    - If a function returns a value a call to that function is an expression
    - The statement above assigns the value returned from the function call to square to the variable result
  - Alternatively the return value of the square function could have been displayed by the `cout` statement:
    - `cout << number << " squared is " << square(number) << endl;`
  - The above examples show how a value returned by a function can be assigned to a variable or can be printed.
  - It is also possible to use the value returned by the function in a relational test or in an arithmetic expression
    - `result = square(number);`
    - ```
      if(result > 100)
      {
        cout << "big square\n";
      }
      ```
    - `sum = 1000 + result;`
      ```

Returning a Single Value (Example 9 – findMaxReturn.cpp)

```
/*Notice how the findMax function can be more useful & REUSABLE when it returns a value*/
#include <iostream>
using namespace std;

int findMax(int, int); //function prototype

int main()
{
      int firstnum, secnum;
      int max;

      cout << "Please enter the first number: ";
      cin >> firstnum;
      cout << "\nPlease enter the second number: ";
      cin >>secnum;

      max = findMax(firstnum, secnum);

      cout << "\nThe maximum number " << max << " doubled is " << (max * 2) << endl;

      if(max % 3 == 0)
      {
        cout << "The maximum number " <<  max << " is divisible by 3." << endl;
      }
      system("PAUSE");
      return 0;
}

/*This function finds the maximum of two numbers and returns the maximum number*/
int findMax (int x, int y)
{ // start of function definition
      int maxnum;       // variable declaration
      if (x >= y)       // find the maximum number
        maxnum = x;
      else
        maxnum = y;

      return maxnum;
}    // end of function definition and end of function
```

Returning a Single Value (Example 10 – average.cpp)

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

//function prototype
int sum3(int, int, int);

int main()
{
    int sum;
    double average;
    int num1, num2, num3;

    cout << "Please enter the three numbers:";
    cin >> num1 >> num2 >> num3;

    sum = sum3(num1, num2, num3);  //function call

    average = (double)sum/3; //what would happen without the cast?

    cout << "The average of the numbers is " << average << endl;
    system("PAUSE");
    return 0;
}

/*This function calculates the sum of the 3 numbers passed to it as
parameters and returns the sum to the calling function*/
int sum3(int a, int b, int c)
{
    int sum;
    sum =  a + b + c;
    return sum;
}
```

Returning a Single Value

- Note that the return type of the function should be the type of data you are wanting to use from that function in the calling program.
  - If a function returning a float value is assigned to an **`int`** variable, the value of the float returned would be truncated.
- If you give a function a return type other than void, you must have a return statement in that function
- Functions may return any type of value: **`bool`**, **`float`**, **`int`**, an object etc.

Returning a Single Value

- **Example: (Example 11 - isEven.cpp) returns a bool**

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

//function prototype
bool isEven(int);

int main()
{
   int value;

   cout << "Please enter an integer number: ";
   cin >> value;

   if(isEven(value))  //function call is used as a boolean expr
   {
     cout << value << " is an even number." << endl;
   }

   else
   {
     cout << value << " is an odd number." << endl;
   }
   system("PAUSE");
   return 0;
}

/*The isEven function returns true if the parameter is even and false otherwise*/
bool isEven(int number)
{
   if((number % 2) != 0)
      return false; //the number is odd if there's a remainder
   else
      return true;  //otherwise its even
}
```

Variable Scope

- ***Scope:*** section of program where identifier is valid (known or visible)
  - Remember an identifier is just a part of the program named by the user (examples: variables and function names)

- ***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
  - You should use local variables in this class except for when I say it is ok
    - ***it is NOT OK to use globals just to make code "work" (POINTS OFF)***
    - This is to get you to learn good programming practice, in real life globals are RARELY used except in special situations which you will learn to identify in your advanced class

- ***Global variables*** (global scope): variables created outside any function (meaning outside of all functions)
  - We haven't used any global variables yet
  - Can be used by all functions physically placed after global variable declaration

Local and Global Variables - Scope

- **Concept:** A ***local variable*** defined inside a function is not accessible outside of that function. A global variable is defined outside of all functions and is accessible to all functions in its scope

- **Example: (localVar.cpp – Example 12)** Because the variables defined within a function are hidden from the statements in other functions, other functions may have distinct separate variables with the same name. The program can only "see one of the variables at a time.

- ***DO NOT USE GLOBAL VARIABLES IN YOUR PROGRAMS – ONLY USE GLOBALS I TELL YOU ARE OK IN CLASS – WE'LL SEE HOW USING GLOBALS INAPPROPRIATELY IS BAD PROGRAM DESIGN LATER IN THE CLASS***
  - ***I will take off A LOT of points on both the midterm and on the homework if you use global variables instead of passing data – do NOT use them in this class except for the few exceptions I will tell you about***

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

//function prototype - the prototype is global - ok to have global prototypes
void anotherFunction();

int main()
{
   int num = 1;  //local variable to main
   cout << "In main, num is " << num << endl;
   anotherFunction();
   cout << "Back in main, num is still " << num << endl;
   system("PAUSE");
   return 0;
}

/*Definition of anotherFunction.  This function has a local variable num whose value is
displayed*/
void anotherFunction()
{
 int num = 20;  //local variable
 cout << "In anotherFunction num is " << num << endl;
}
```

Local and Global Variables

- Since ***local variables*** are hidden from other functions they do not provide a convenient way of sharing data – however they protect data and make code reusable and maintainable as we will discuss in a minute (this is the key to good code)
- Global variables are one alternative when it is necessary to make **large** amounts of data accessible to **all functions** in a program
  - ***NONE of the programs we do in this class need to make large amounts of data accessible to all functions***
    - The general rule is ***NO GLOBAL VARIABLES for this class unless it is an exception I've told you about*** – they cause programming design problems
    - Passing data back and forth between functions is generally the better solution in procedural programs (which is what we are doing right now)
- A ***global variable*** is any variable defined outside all the functions in a program. The scope of a global variable is the portion of the program from the variable definition to the end of the program.

Global Variables

Example (globalVar.cpp – Example 13)

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

void anotherFunction(); //function prototype
int num = 2;            //global variable - BAD DESIGN FOR THIS PROGRAM

int main()
{
   cout << "In main num is " << num << endl;
   anotherFunction();
   cout << "In main num is now changed to " << num << endl;
   system("PAUSE");
   return 0;
}

/*Definition of another function.  This function changes the value
of the global variable num*/
void anotherFunction()
{
   cout << "In anotherFunction num is first " << num << endl;
   num = 50;
   cout << "Now num in anotherFunction is changed to " << num << endl;
}
```

Scope Resolution Operator

- **Local and Global variables with the same name**
  - If a function has a local variable with the same name as a global variable, only the local variable can be seen in the function
  - 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
```

Global and Local Variables

- Numeric ***Global*** variables are initialized to zero by default if no explicit initialization is made
  - This is different than local variables in which no default initializations are made

- ***WARNING regarding the use of global variables:***
  - It's tempting to make all of your variables global especially when you are first learning to program because you can access global variables from any function without passing their parameters as variables.
    - **This is NOT a good idea (and A LOT OF points WILL BE taken off for this).** It will most likely cause you problems later and will cause problems in larger programs.
      - When debugging if you find a global variable is incorrect you must track down all of the statements accessing and using it to determine where the bad data is coming from
      - When two or more functions access/modify the same global variable you have to make sure that one function does not affect the correctness of the other
  - When programming always use global variables sparingly.

Misuse of Globals

- Why avoid overuse of globals?
  1. Too many globals eliminates safeguards provided by C++ to make functions independent
     - **The idea is to have independent functions that perform one task and can be reused**
       - Always think, could I copy this function into a totally different program and reuse it? If the answer is no, it is not as good of a function.
  2. Functions that RELY on global variables are NOT reusable
     - The functions are tied to the program containing the global variable
  3. 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

- Misuse rule does not apply to function prototypes
  - Prototypes are typically global

Static Local Variables

- A function's variables (locals and parameters) are destroyed when the function terminates and recreated when the function is called again.
  - As a result if a function in called more than once in a program the values in the function's local variables are lost
  - So the lifetime of a regular local variable is the block in which that variable is declared in
- Sometimes you may want your program to "remember" what is stored in a local variable between function calls. This can be accomplished by making the variable **`static`**.
  - Static local variables are not destroyed when a function returns
  - **`static`** local variable lifetime = lifetime of program
  - Value stored in variable when function is finished is available to function next time it is called
  - ***NOTE this does NOT change the scope or visibility of the variable – it only makes it so the variable is not destroyed when the function ends***
- Initialization of **`static`** variables (local and global – but we will never use global)
  - Done ***one time only, when program first compiled***
  - Only literals or constant expressions allowed
- We'll talk more about **`static`** variables later in the class, they will be useful in object oriented programming

Static Local Variables

- ***Example 14:*** (`static.cpp`)

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

int alienCounter(int);

int main()
{
    int aliensMet;
    int totalAliens;

    cout << "During each level of a five level game you encounter aliens "
           << endl  << " enter the aliens you encounter at each level and "
           << endl << "this program will keep track of the total aliens in "
          << "the game " << endl << endl;

    for(int level = 1; level <= 5; level++)
    {
          cout << "Enter the aliens encountered in Level # " << level << ": ";
          cin >> aliensMet;

          totalAliens = alienCounter(aliensMet);
    }

    cout << "\n\nYou encountered a total of : " << totalAliens << " aliens." << endl;
    system("PAUSE");
    return 0;
}


int alienCounter(int additionalAliens)
{
    static int alienCount = 0;  //variable gets created ONCE, initialization is done ONCE
    alienCount = alienCount + additionalAliens;
    return alienCount;
}
```

Default Arguments

- ***Default arguments*** are passed to parameters automatically if no argument is provided in a function call
  - Default arguments are usually listed in the function's prototype
    - They are literal values or constants with an = operator in front of them appearing after the data types listed in the function prototype.
    - Since parameter names are optional in function prototypes they could be included for readability
  - Example Prototypes that use default arguments:
    - `float calcArea(float = 20.0, float = 10.0);`
    - `float calcArea(float length = 20.0, float width = 10.0);`
  - The **default arguments are only used then the actual arguments are omitted from the function call:**
    - `area = calcArea();  //both default values used`
    - `area = calcArea(12.0); //12.0 passed to length, default value for width`
    - `area = calcArea(12.0,5.5);  //12.0 and 5.5 are passed`
  - **Default Arguments need to be assigned in the earliest occurrence of the function name which is usually the prototype**

Default Arguments

- ***Example 15:*** **– default.cpp**

```
#include <iostream>
#include <stdlib.h>

using namespace std;

double calcArea(double = 20.0, double = 10.0);

int main()
{
    double area;

    //call the function and pass no arguments, default arguments are used
    area = calcArea();
    cout << "When no arguments are passed to the function area  = " << area << endl;

    //call the function and pass one argument
    area = calcArea(5.0);
    cout << "When one argument is passed to the function area  = " << area << endl;

    //call the function and pass two arguments
    area = calcArea(5.0, 1.0);
    cout << "When two arguments are passed to the function area  = " << area << endl;

    system("PAUSE");
    return 0;
}

double calcArea(double length, double width)
{
    return length * width;
}
```

Pass by Reference

- ***IMPORTANT:*** ***Do NOT use pass by reference on HW #4 or on the Programming Midterm***
  - That is not what those tests are about
  - **HOWEVER – you will NEED to know pass by reference for the *objective midterm***

- Called function usually receives values as ***pass by value***
  - Only ***copies*** of values in arguments are provided
  - Any changes to the parameter's value do not affect the value of the original argument

- Sometimes it is desirable to allow function to have **direct access** to variables passed into it (note this is the opposite of pass by value)
  - ***Address*** of variable must be passed to function
  - Function can directly access and change the value stored there
  - ***NOTE:*** Like globals this can be ABUSED – so on HW #4 you are not allowed pass by reference – it should only be used when the situation warrants it

- ***Pass by reference:*** passing addresses of variables received from calling function

Passing and Using Reference Parameters

- **Concept:** A ***reference variable*** is an alias for another variable. When used as a parameter, a reference variable allows the function to access the parameter's original argument. Any change to the parameter is made to the original argument.

- Reference variables are defined like regular variables except there is an ampersand (**&**) in front of the name

```
void doubleNum(int &refVar)
{
   refVar *= 2;
}
```

  - This function doubles **refVar** by multiplying it by 2. Since **refVar** is a reference variable, this action is actually performed on the variable that was passed to the function as an argument

  - The variable **refVar** is called "a reference to an **int**"

Passing and Using Reference Parameters

- When prototyping a reference variable, be sure to include the ampersand (&) after the data type:

```
void doubleNum(int &);
```

- Some programmers do not but a space between the ampersand and the data type. Both styles are equivalent.

```
void doubleNum(int&);
```

- **The ampersand must appear both in the prototype and in the header of the function that uses the reference variable as a parameter. It does not appear in the function call**

Passing and Using Reference Parameters - Summary

- ***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&);
```

- **Function Call**
  - When you call a function with reference parameters you pass the variables names you want the parameters to refer to with no **`&`**
  - So if you were in **`main`** and wanted to call the **`newval`** function above with variables **`a`** and **`b`** (previously defined in main) the call would look like this:
    - `newval(a, b);`

Passing and Using Reference Parameters (Example 16 - refVar1.cpp)

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

//function prototype - the parameter is a reference variable
void doubleNum(int&);

int main()
{
   int value = 4;

   cout << "In main the value is " << value << endl;
   cout << "Now I'm calling the doubleNum function..." << endl;
   doubleNum(value);  //function call, no ampersand
   cout << "Now back in main, value is " << value << endl;
   system("PAUSE");
   return 0;
}

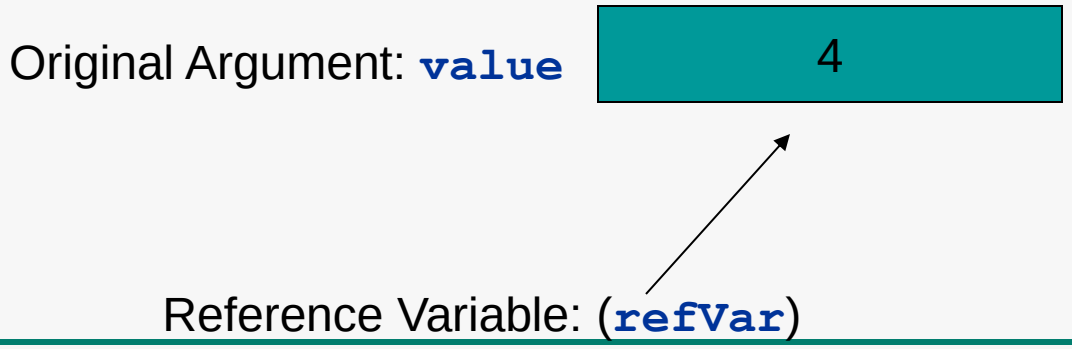
/*Function Definition for doubleNum.  The parameter refVar
is a reference variable.  The value in refVar is doubled.*/
void doubleNum(int &refVar)
{
   refVar *= 2;
}
```

Passing and Using Reference Parameters – Example 14

- Program Output:

```
In main the value is 4
Now I'm calling the doubleNum function...
Now back in main, value is 8
Press any key to continue . . .
```

- The parameter variable **`refVar`** "points" to the **`value`** variable in the function main. It references **`value`**. When a program works with a reference variable it is actually working with the variable it references or points to.



Passing and Using Reference Parameters – Example #15

- **Example (Example 17 - refVar2.cpp)**

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

//function prototypes
//both functions use reference variables as parameters
void doubleNum(int&);
void getNum(int&);

int main()
{
   int value;
   getNum(value);  //function call
   doubleNum(value);  //function call
   cout << "That value doubled is " << value << endl;
   system("PAUSE");
   return 0;
}

/*Function Definition for getNum. The parameter userNum is a reference variable.  The user is asked
to enter a number which is stored in userNum*/
void getNum(int &userNum)
{
 cout << "Enter a number: ";
 cin >> userNum;
}

/*Function Definition for doubleNum.  The parameter refVar
is a reference variable.  The value in refVar is doubled.*/
void doubleNum(int &refVar)
{
   refVar *= 2;
}
```

Important Point: A function's parameters do not have to be **all** pass by value or all pass by reference, ***some parameters can be pass by value and some pass by reference***

Pay close attention to details (HINT for your exam)

Passing and Using Reference Parameters

- Only Variables may be passed by reference (literals and expressions may NOT be passed by reference)
  - If you attempt to pass a non-variable argument such as a literal or an expression an error will result

```
doubleNum(5);  //error
doubleNum(value + 10); //error
```

- If a function uses more than one reference variable as a parameter be sure to place the ampersand before ***each*** reference variable name.

```
//prototype 4 reference variables as parameters
void addFourThree(int&, int&, int&, int&);

//definition of addThree
void addThree(int &sum, int &num1, int &num2, int &num3)
{
   cout << "Enter three integer values: ";
   cin >> num1 >> num2 >> num3;
   sum = num1 + num2 + num3;
}
```

Important Point: A function's parameters do not have to be all pass by value or all pass by reference, some parameters can be pass by value and some pass by reference

Pay close attention to details (HINT for your exam)

- ***IMPORTANT:*** **Note that your function could have some variables that are passed by value and some that are passed by reference…look at the prototype below**
  - `void calcSomeThings(int &a, int &b, int num);`

Don't Get Carried Away Using Reference Variables as Function Parameters

- Anytime you allow a function to alter a variable that's outside of the function, you are creating potential debugging problems.
  - You are intertwining groups of functions…so finding an error will be harder
  - So if you use pass by reference improperly you have similar issues to when we use globals
    - In the advanced C++ class we learn a lot of additional code to make pass by reference safer and really focus on when to use it and when not to – in our class I want you to at least undestand how it works – when we get to object oriented I'll show some good uses too
- You are also giving functions access to variables you may not want them to have
  - You should never give code free access to other code for no reason – you should protect code if possible from change by "outsiders"
- Reference variables should only be used as parameters when the situation requires them.

When to Pass Arguments by Reference and When to Pass Arguments by Value

- General Guidelines
  - When an argument is a constant (literal or named constant) it must be passed by value. Only regular variables can be passed by reference.
  - When a variable passed as an argument should NOT have its value changed it should be passed by value. This protects it from being altered.
  - When exactly **one** value needs to be sent back from a function to the calling routine, it should generally be returned with a `return` statement rather than through a reference parameter. So in this case **pass by value**.
  - **When two or more variables passed as arguments to functions need to have their values changed by that function, they should be passed by reference**
  - **When a copy of an argument cannot correctly or reasonably be made, such as when the argument is a file stream object, it must be passed by reference**
  - **When you have a large argument like an object you are passing, you may want to pass by reference to save memory**

- Common Instances when Reference Parameters are Used
  - When new data values being input in a function need to be known by the calling function (or part of the program that activated the function)
  - When a function must change the existing values (more than one) in the calling function (part of the program that activated the function)
  - When a file stream object is passed to a function, when other complex objects are passed to functions
  - To save memory in an object oriented program – when you are passing an instance of a data type with a lot of member variables

- **Example (refVar3.cpp – Example 18)**

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

//function prototypes
void getNums(int&, int&);
void swap(int&, int&);

int main()
{
   int firstNum,secNum;

   getNums(firstNum, secNum); //call getNums to get the two numbers
   cout << "\nBEFORE the swap firstNum = " << firstNum << " and secNum = " << secNum;

   swap(firstNum, secNum); //call orderNums to put the two numbers in order
   cout << "\n\nAFTER the swap firstNum = " << firstNum << " and secNum = " << secNum << endl;

   system("PAUSE");
   return 0;
}

/*getNums Definition.  The arguments passed into input1 and input2 are passed by reference so the values
entered in thesetwo parameters will be stored in the memory space of main's small and big vairables*/
void getNums(int &input1, int &input2)
{
   cout << "Enter the first integer: ";
   cin >> input1;
   cout << "\nEnter the second integer: ";
   cin >> input2;
}

/*orderNums Definition.  The arguments are passed by reference into num1 and num2 so that if they are out of
order main's small and big variables are actually swapped. */
void swap(int &num1, int &num2)
{
      int temp;

      temp = num1;
      num1 = num2;
      num2 = temp;

}
```

Function Overloading

- **Concept:** Two or more functions may have the same name as long as their parameter lists (signatures) are different

- ***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
    - Overloading is VERY COMMON in object oriented programming and in programming in general
    - Think of the **+** sign, it is overloaded

Function Overloading

- **The complier determines which overloaded function to call based on the function's *signatures***
  - Every function with the same name must have a unique function signature.
  - **A function's signature is the *function's name* and *the function's parameter type list* (order of parameters, number of parameters and types of parameters) BUT *NOT* the return type or the parameter names**
  - Example: Function signatures for two sum functions one that takes two ints and one that takes two floats

```
int sum(int x, int y)  //signature sum(int, int)
{
}

int sum(float x, float y)  //signature sum(float, float)
{
}
```

  - The above two functions have different signatures. If the second function were changed to:

```
float sum(int l, int k) //signature sum(int, int)
{
}
```

    - This function and the first one no longer have different signatures and could not be used in the same program.
    - The return type is NOT part of the function's signature

Function Overloading

Example: two functions named **`cdabs()`** that return the absolute value of the argument passed to them

```
int cdabs(int x) // compute and return the absolute value of an int
{
}

float cdabs(float x) //compute and return the absolute value of a float
{
}
```

Function Overloading

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

Reusing Functions – header files (.h)

- One of the points of functions is to be able to reuse them in multiple programs
  - One way would be to copy and paste your functions from one program to another
    - This could be cumbersome

- To facilitate reuse of similar functions you can save your functions in a ***header file*** and then include that header file in any program you're writing that uses one or more of those functions
  - It's similar to writing your own `cmath` file
  - When you do this you should only put functions that perform similar tasks in your header file

Reusing Functions – header files (.h)

- Example 19 (Rectangle)
  - For this program I'm going to define some functions that can manipulate rectangles
    - `calcArea`, `calcPerimeter`
  - I'm going to define these functions first and save them in a `.h` file
  - So let's start our project like normal but when we add a file to it we'll add a `.h` file instead of a .cpp file
  - So after you create your project, right click on the "Header" folder and select Add->New Item
    - The type of item you add should be a header file (.h)
    - Name it "rectangle"
  - Now we'll add our functions to this file

Reusing Functions – header files (.h)

- Example 19 (Rectangle) - continued
  - My **`rectangle.h`** file is below – we now have a file that can be included in any program that needs to use these functions

```
//prototypes
double calcArea(double, double);
double calcPerimeter(double, double);

double calcArea(double length, double width)
{
   return length * width;
}

double calcPerimeter(double length, double width)
{
   return (2 * length) + (2 * width);
}
```

Reusing Functions – header files (.h)

- Example 19 (Rectangle) - continued
  - Now let's add a .cpp program to our project that uses our functions, to do this we create the program like usual (add a .cpp file to the project)
  - The only new thing we need to do is include the .h file in our source code (.cpp file) so our source code can see and use the functions in the .h file, we do that with the statement below:
    - `#include "rectangle.h"`
    - The only thing different about this include is we used " " instead of < > around the file name, this is because the file is one we defined in the current project instead of one in the C++ standard library

Reusing Functions – header files (.h)

- Example 19 (Rectangle) – continued
  - My .cpp file for this project is below, note I could use the functions – no need to define them in the .cpp file now

```
#include <iostream>
#include "rectangle.h"
using namespace std;

int main()
{
    double len;
    double w;
    double area;
    double perim;

    cout << "Enter the length of the rectangle: ";
    cin >> len;
    cout << "Enter the width of the rectangle: ";
    cin >> w;

    area = calcArea(len, w);
    perim = calcPerimeter(len, w);

    cout << "\n\nRECTANGLE CALCULATIONS" << endl;
    cout << "----------------------" << endl;
    cout << "Area:\t\t" << area << endl;
    cout << "Perimeter\t" << perim << endl << endl;

   system("PAUSE");
   return 0;
}
```

Defining Functions - Format

- Order of functions in a program:
  - Any order is allowed
    - ***It's better and more readable to have main first and use function prototypes***
      - ***Do that for this class***
      - ***For this class you MUST USE PROTOTYPES OR YOU MAY USE A HEADER FILE FOR YOUR FUNCTIONS***
  - `main()` usually first and in this class you must put your main function first and use prototypes or use a header file for your functions
    - `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

Placement of Statements

- Recommended ordering of statements
  - Good programming practice

```
preprocessor directives
function prototypes

int main()
{
   symbolic constants
   variable declarations
   other executable statements
   return value
}

otherfunction definitions
```

Function Design

- Remember your program should not just "run" it should be designed well
- ***Do NOT use all void functions, do NOT use global variables to avoid passing data with your functions (if you like points don't do it!)***
  - Generally only use a void function when your function needs to display something
    - DO NOT HAVE IMPROPER VOID FUNCTIONS IN YOUR PROGRAMS (I will take off points)
  - Functions that return values are generally best because they pass data back and forth between functions
    - Allows for functions to perform one task only and thus be reusable
    - Allows local variables to be used and avoids debugging and maintenance problems associated with global variables
    - When a calculation is involved you should almost ALWAYS use a value returning function
  - Pass by reference functions can be used but should only be used in the situations described earlier in the lecture
    - When new data values being input in a function need to be known by the calling function
    - When a function must change the existing values in the calling function
    - When a file stream object is passed to a function
  - There are some cases where a pass by value or pass by reference function are appropriate depending on how you designed your function
- ***If you design your function well you should be able to copy and paste it to another program that needs the function without modifying*** it
  - If the function works for just the one program you designed it is not as useful

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
- 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

- 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

- **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