**CS 10A – Introduction to Programming: Chapter #6 - C++ Functions (Information Sheets)**

You can write C++ programs as a collection of functions - using functions allows you to write modular programs. That is, you can break down large problems into a collection of small problems and then write a function to solve each of the smaller problems. This programming approach avoids duplication of code and lets you write programs that are easier to read, write, debug, and maintain.

You can arrange the collection of functions that make up a C++ program in any order within a file. In addition, you can store functions in multiple files. You can write a C++ function that will behave as a function or a subroutine.

As a C++ programmer, you can either write your own functions or use the functions supplied by the system. Functions and subroutines are blocks of code that have a name and perform a specified task. **Functions return a value, while subroutines (also called void functions) return nothing. In C++, you will write only functions.**

**Today's learning objectives:**

1. **Identify the key elements/syntax of C++ Program structure with multiple user-defined functions.**
2. **Understand the Argument/Parameter - Passing Mechanisms.**
3. **Show the logical order in which statements are executed in structured C++ programming.**

---

1. Identify the following items in the programs shown below:

| | |
|---|---|
| function prototype | function definition |
| function heading | parameters |
| arguments | function call |
| local variables | function body |

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

void Test( int, int, int );

int main()
{
        int a;
        int b;
        int c;
        :
        :
        Test(a, c, b);
        Test(b, a, c);
        :
        :
}

void Test (int d, int e, int f)
{
        int g;
        int h;
        :
        :
}
```

For the program fragment above, fill in the blanks with variable names to show the matching that takes place between the arguments and parameters in each of the two calls to the Test function.

| First Call to Test | | Second Call to Test | |
|---|---|---|---|
| **Parameter** | **Argument** | **Parameter** | **Argument** |
| 1._____ | _____ | 1._____ | _____ |
| 2._____ | _____ | 2._____ | _____ |
| 3._____ | _____ | 3._____ | _____ |

2. Number the marked statements in the following program to show the order in which they are executed (the logical order of execution). - **q2.cpp**

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

void DoThis( int& , int& );

int main()
{
                int number1;
                int number2;

_____           cout <<"Exercise ";

_____           DoThis(number1, number2);

_____           cout<<number1<<' '<<number2<<endl;
                return 0;
}

void DoThis (int& value1, int& value2)
{
                int value3;

_____           cin>>value3>>value1;

_____           value2 = value1 + 10;

}
```

If this program were run with the data values 10 and 15, what would be the values of the following variables just before execution of the Return statement in the main function?

number1_____number2_____value3_____

**Choosing Between Pass By Value and Pass By Reference**

Choosing between the two parameter-passing mechanisms is certainly a difficult task. This choice is not a language issue but a design decision. In designing the function interface, each parameter must be classified as an In, Out, or Inout parameter according to the data flow. Once this classification has occurred, the choice of mechanism is trivial each In parameter is passed by value, and each Out and Inout parameter is passed by reference. Requiring each parameter to be labeled with the comment /* in */, /* out */, or /* inout */ (a useful and concise terminology borrowed from the Ada language) helps in thinking carefully about the data flow.

For a given parameter, then, determining the direction of data flow is ultimately the hard part. Here is a thumbnail guide you might give to the students:

| *Use of the Parameter* | *Data Flow* |
|---|---|
| Inspected only | In |
| Modified only | Out |
| Inspected and modified | Inout |

Sample fill-in-the-blank function definitions to practice identifying the data flow of parameters:

```
void DoThis(_____int__ alpha,
        _____float  beta )
{
   beta = 3.8 * float(alpha);
}
```

For each parameter, do the following in the order shown:

1. Examine how the parameter is used within the function.
2. Fill in the first blank with /* in */, /* out */, or /* inout */.
3. Fill in the second blank with either an ampersand (&) or nothing.

For the example above, alpha is inspected only, so it's an In parameter and there should be no ampersand after int (pass by value). The parameter beta is modified only, so it's an Out parameter and there should be an ampersand after float (pass by reference).

7. For the function definition

```
void Func(int& gamma)
{
        gamma = 245;
}
```

which of the following comments describes the direction of data flow for gamma?

a. /*in*/
b. /*out*/
c. /*inout*/

8. For the function definition

```
void Func(int gamma)
{
        cout<<3 *gamma;
}
```

which of the following comments describes the direction of data flow for gamma?

a. /*in*/
b. /*out*/
c. /*inout*/

9. For the function definition

```
void Func(int& gamma)
{
        gamma = 3 * gamma;
}
```

which of the following comments describes the direction of data flow for gamma?

a. /*in*/
b. /*out*/
c. /*inout*/

10. Consider the function definition

```
void DoThis (int& alpha, int beta)
{
        int temp;

        alpha = alpha + 100;
        temp = beta;
        beta = 999;
}
```

Suppose that the caller has integer variables gamma and delta whose values are 10 and 20, respectively. What are the values of gamma and delta after return from the following function call?

```
        DoThis (gamma, delta)
```

a. gamma = 10 and delta =20
b. gamma = 110 and delta = 20
c. gamma = 10 and delta = 999
d. gamma = 110 and delta = 999
e. none of the above

**Type Mismatches Between Formal and Arguments**

With simple data types (int, float, and so on), the C++ compiler does not complain if you pass an argument of one data type to a parameter of another data type. However, the consequence can be wrong output whose cause is difficult for students to track down.

With passing by value, the problem is no different from type coercion across the assignment operator. For example, passing a float value to an int parameter results in truncation of the fractional part. Do not become careless with the type coercion rules. Use explicit type casts to ensure that type conversions are intentional rather than accidental.

With pass by reference, the problem is more insidious. If the argument is not of the same type as the parameter, the system creates a temporary variable of the correct type and passes its address to the parameter. The function then works with this temporary variable, which is destroyed when control returns from the function. The net effect is that the argument mentioned in the function call is completely unaffected by execution of the function. Even an explicit type cast of the argument won't help with this situation the compiler still creates a temporary. Remember that with pass by reference, the actual and parameters must be of identical type.

**Mismatches Between Function Prototypes and Function Definitions**

In most C++ implementations, the following code produces an error message:

```
#include <iostream>

using namespace std;

void Func( float );

int main()
{
        .
        .
}

void Func( int n )
{
        .
        .
}
```

The error message says something similar to "Undefined symbol Func(float)." This is a baffling error message; after all, the program clearly provides a function definition for Func.

It turns out that the error message is from the linker, not the compiler. The compiler thinks there are two different functions named Func one taking a float parameter (seen in the function prototype) and another taking an int parameter (seen in the heading of the function definition). The program compiles successfully under the assumption that the object code will later be linked with another file containing the object code for the Func(float) function. At link time, the linker cannot find any object code for the Func(float) function and issues an error message.

**Reminder:** The function prototype and the function definition must agree *exactly* with respect to the function return type (above, void) and the number and data types of the parameters.

**Local Variables**

Local variables along with global are only briefly mentioned in this chapter 6.10; because they are so closely tied to the scope and lifetime rules. You may have some trouble with the idea that local variables exist only during execution of a function. This is your first exposure to dynamic memory allocation. When a function is called, its local variables are stored temporarily in a region of memory called the run-time stack. When the function returns, the space for the local variables is released from the stack. In this you will also learn the distinction between automatic local variables (those described here) and static local variables (those whose memory remains allocated for the lifetime of the program).

## CS 10A – Declarations and Definitions …C++ concepts to think about … (Information Sheet)

**Why does C++ offer both declarations and definitions?** The declaration/definition distinction reflects the fundamental distinction between what we need to use something (an interface) and what we need for that something to do what it is supposed to (an implementation).

For a variable (introduced in Gaddis Chapter 2), a declaration supplies the type but only the definition supplies the object (the memory).

For a function (introduced later in Gaddis 6), a declaration again provides the type (argument types plus return type) but only the definition supplies the function body (the executable statements). Note that function bodies are stored in memory as part of the program, so it is fair to say that function and variable definitions consume memory, whereas declarations don't.

The declaration/definition distinction allows us to separate a program into many parts that can be compiled separately. The declarations allow each part of a program to maintain a view of the rest of the program without bothering with the definitions in other parts. As all declarations (including the one definition) must be consistent, the use of names in the whole program will be consistent.

| **Declaration** | **Definition** |
|---|---|
| double sqrt(double d) // a function prototype | //a function defined<br>double sqrt(double d)<br>{<br>// pow() is a predefined function in math.h header file<br>return pow(d, 0.5);<br>} |
| int value; | value=10;<br>or int value=10; |

```
//see function_declaration_definition.cpp
#include<iostream>
using namespace std;

int function(/*in*/int); //a value returning function prototype (aka. function declaration)with formal parameters
void anotherfunction(/*inout*/int&); //a non value returning function prototype (aka. function declaration)with formal parameters

int main()
{
   int number=10; //a local variable

   cout<<"The local variable number in main contains "<<number<<endl;

   cout<<"The value returned in main after a call to int function "<<function(number)<<'\n'; //a value returning function call with actual argument
passed into the function

   anotherfunction(number); //a call to a non value returning function (aka. void function)with actual argument passed into the function
   //*note the statement by itself above..what was processed

   cout<<"The local variable number in main contains "<<number<<endl;//what caused number to change here
}

//value return function defined with formal parameters...note dataflow comments
int function(/*in*/int someValue)
{
someValue=someValue+100;
return someValue; //computer value returned to calling function

}
//non value returning function defined (aka. void function)with formal...note dataflow comments
void anotherfunction(/*inout*/int& someValue)
{
int newValue=someValue+200;
someValue=newValue*2;
cout<<"The local variable newValue in anotherFunction contains "<<newValue<<endl;
//return someValue;

}
```