A Short Introduction to Functions
for use in
CMIS 102, Introduction to Problem Solving and Algorithm Design

A. What is a function?

You can think of a function as a black box that accepts some data in an inbox, performs some processing using that data, and reports a result through an outbox. The items going into the inbox are *parameters*. The result placed in the outbox is called the *return value*.

Figure 1 illustrates a function with three parameters.

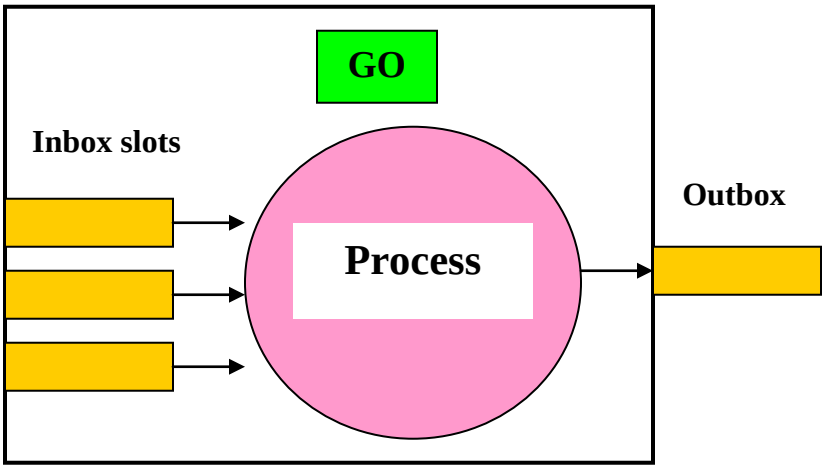


Figure 1. A function with three parameters

A function has absolutely no information about the environment in which it is expected to operate. You can think of a function as a contractor providing some service, operating out of a storefront. The contractor has no idea who his customers will be, what parameters they will bring, or what they will do with the results.

Functions are of two types: *library* or *built-in* functions, that are provided with a programming language development system; and *user-written* functions.

When a function is used in a program, it is the responsibility of the client program to provide the parameter values by placing them in the slots of the inbox, and then push the GO button to start the function's process. It is also the responsibility of the client to grab the result from the outbox and do something with it. Doing these things is called *calling* or *invoking* the function.

In order to use a function in a program, there are certain things you have to know about it:

- The function's name;
- The number of parameters (that is, the number of slots in the inbox);
- The data type required for each of the parameters;
- The data type of the result; and of course:
- What the function does.

B. Some library functions

All program development systems provide large inventories of library functions. The details vary widely from one language to another. Table 1 shows the five functions that are used in the examples in the Pseudocode Demo package. Table 2 lists a few typical functions that do useful things with Strings. We will not need to use any of the functions in Table 2 in this course. This table is just to provide you some insight into what is out there in the world of library functions.

Table 1. Information the Functions used in the Pseudocode Demos

Name	Number of Parameters	Parameter Data Types(s)	Result Data Type	What it does
`Sqrt`	1	Float	Float	Computes a square root
`isValidInteger`	1	String	Boolean	Determines if the parameter represents an integer value
`isValidFloat`	1	String	Boolean	Determines if the parameter represents a float value
`getIntegerValue`	1	String	Integer	Returns the integer value represented
`getFloatValue`	1	String	Float	Returns the float value represented.

Table 2. A Few Typical String-oriented Functions

Name	Number of Parameters	Parameter Data Types(s)	Result Data Type	What it does
`length`	1	String	Integer	Returns the number of characters in the string.
`getCharacterAt`	2	String s, Integer n	Character	Returns the character in position n of string s.

`findFirstPosition`	2	String s, Character c	Integer	Returns the position of the first occurance of character c in string s, or -1 if the character c does not occur in string s.

`printf` and `scanf` are also library functions, but they have unusual properties specifically for output and input operations, and are unique to the C language. We will not consider `printf` and `scanf` in this note.

Some library functions will fail (and the client program will be terminated) if the input to the function does not make sense:

- `Sqrt` will fail if the input is a negative number. For example, `Sqrt(-37.2)` will fail. You can't take a square root of a negative number!
- `getIntegerValue` will fail if the input is not a valid representation of an integer value. For example, `getIntegerValue("3.1567 ")` and `getIntegerValue("Hello there! ")` will fail. Similarly with `getFloatValue`.
- `getCharacterAt` will fail if there is no character at the specified position. For example, `getCharacterAt("qwert", -3)` and `getCharacterAt("ABCDE", 37)` will fail.

When working with a function that can fail, the prudent programmer would first do a test on the intended input values to make sure that a failure won't happen.

C. Examples of User-Written Functions

In the Pseudocode Demo package, you will find two examples that contain user-written functions: Function Example A and Function Example B. We will discuss the function in Function Example A in some detail, then briefly discuss the function in Function Example B.

1. The `Larger` *function*

Suppose you have two float variables `m` and `n`, and want to set `t` to the larger of the values of `m` and `n`. You would have to write something like this:

```
If (m >= n)
     t = m
Else
     t = n
Endif
```

That is not all that awkward, but if you were writing code to do some complex financial computation (some sort of taxes, maybe) and had to write variations on this many times,

it would get rather tiresome. Here is a better way: write a function `Larger` that will simplify your task.

```
Define Function Larger(Float A, Float B) Returns Float
     If (A >= B)
          return A
     Else
          return B
     Endif
End Function
```

The first line of this definition provides all of the information a client needs in order to use the function. The name of the function is `Larger`. It takes two parameters (there are two slots in its inbox). Both parameters are of type Float. The result returned to the client is of type Float. This first line is the *header* or *prototype* of the function. The following lines, which are the *body* of the function, define what the function does and how it does it.

Now suppose that you have to find the largest of three numbers X, Y, and Z. You would do this in two steps. First, find the larger of X and Y; store it in a variable you might call Temp. Then, find the larger of Temp and Y.

At this point, you should go to the Pseudocode Demonstration web site. Then start the demonstration program by clicking on the yellow button. Select Function Example A from the dropdown list in the upper left corner, then select Initialize. Figure 2 shows what you will see.

Read through the pseudocode in the left panel. (It will be much clearer on the web site than in this document.) Now look at the variables in the right panel. There are two groups. The first group consists of the five variables declared in the main program. The second group consists of the two variables A and B of the `Larger` function. These are the *formal parameters*; they are the two slots in `Larger`'s inbox. This function does not have any more variables, but most functions will have additional variables.

Now single-step through the program, entering values of your choice for X, Y and Z. Watch for the following key events associated to the two invocations of the function `Larger`:

- The values of X and Y are copied to A and B.
- The function does its work.
- The result from the function is picked up from the function's outbox by the main program, and placed in Temp. (The outbox itself is not visible.)
- The values of Temp and Z are copied to A and B.
- The function does its work.
- The result from the function is picked up from the function's outbox by the main program, and placed in Biggest.

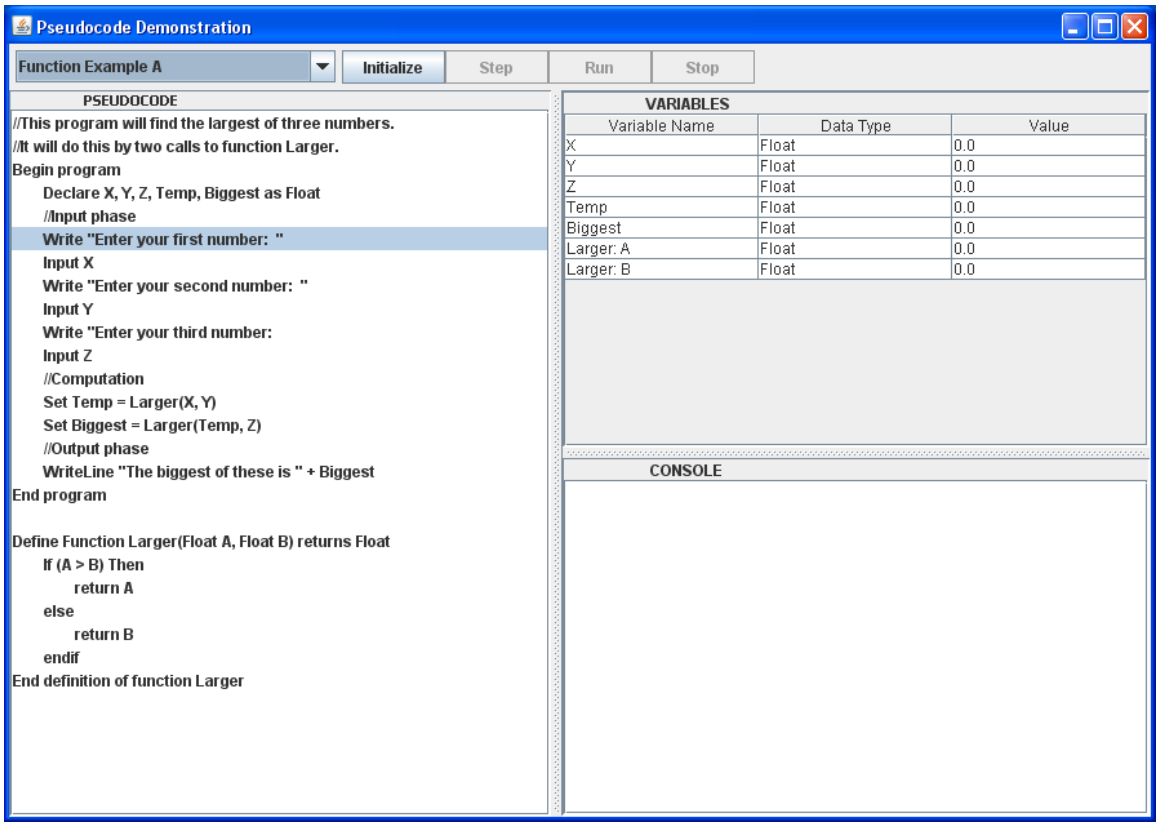


Figure 2. Function Example A in the Pseudocode Demonstration

In the first use of `Larger`, the values of the variables X and Y are the *actual parameters* sent to the function. In the second use of `Larger`, the actual parameters are the values of Temp and Y.

Things to remember:

- *Actual parameters* are values. They are sent to the function and placed in the inbox.
- *Formal parameters* are variables in the function. They are the slots of the inbox.
- The client module has no access to any of the variables of the function, and the function has no access to any of the variables of the client. All communication takes place through the function's inbox and outbox.

If you writing a program that consists of a main module and one or more functions, it is highly recommended that you **never use the same variable name in two different program units**. Doing so is not illegal (because the main module and each function have their own namespaces), but it significantly increases your chances of making mistakes.

2. The `Diag` *function*

Function Example B in the Pseudocode Demonstration program is similar to Function Example A but is a bit more involved. Before working through the example, read the general information about this example. There is one function, `Diag`, that finds the length of the hypotenuse of a right triangle, given the lengths of the sides. The `Diag` function in turn uses the library function `Sqrt`. Then start Function Example B. As before, watch how control flows from the main module to the `Diag` function and back each time the function is called.

D. Writing and Using Your Own Function: A Cookbook Approach

To design a function, you first have to answer a few questions:

> How many slots does the function's Inbox have?
> For each slot in the Inbox, what is the data type of the data you expect to find there?
> What is the data type of the data to be returned to the client?

Then, you have to choose some names:

> Choose a name for the function
> Choose a name for each slot in the Inbox. (These names are called *formal parameters*.)

Your next step is to build a framework for the function. You do this by filling in a standard template. Here is the template for a function of two variables. If there are only one, or more than two, Inbox slots, modify the template accordingly.

```
Define Function FunctionName ( type1 parameterName1, type2
                parameterName2) Returns returnType
        Pseudocode for accomplishing the function's task
        Return ( Value to be placed in Outbox )
End Function Definition
```

You should copy the blue text (including the parentheses and comma) exactly, then replace each red item with appropriate names or code. Note that each of type1, type2, … returnType must be a recognized data type, such as Integer or Float or String.

Now let's use this procedure write a function to find the length of the hypotenuse of a triangle. This is, of course, exactly the `Diag` function that you saw in Function Example B, so you have already seen the end product of this activity. We begin by answering the basic questions, then choose names.

> How many slots does the function's Inbox have? **Two – the lengths of the two sides of the triangle.**

For each slot in the Inbox, what is the data type of the data you expect to find there? **Float – because the lengths are measurements**
What is the data type of the data to be returned to the client? **Float**

Then, you have to choose some names:

Choose a name for the function: **Hypoteneuse**
Choose appropriate, valid variable name for each slot in the Inbox: **Width, Length**

Now we can copy the template, and replace most of the red items using the answers we chose above; replacements are in green:

```
Define Function Hypoteneuse ( Float Width, Float Length) Returns Float
        Pseudocode for accomplishing the function's task]
        Return ( Value to be placed in Outbox] )
End Function Definition
```

Now we have to consider what additional variables we will need. We will only need one, to hold the calculated hypoteneuse length. It will be of type Float. Lets call it Hypo. We declare this variable, then add the formula for the length of the hypoteneuse of a triangle:

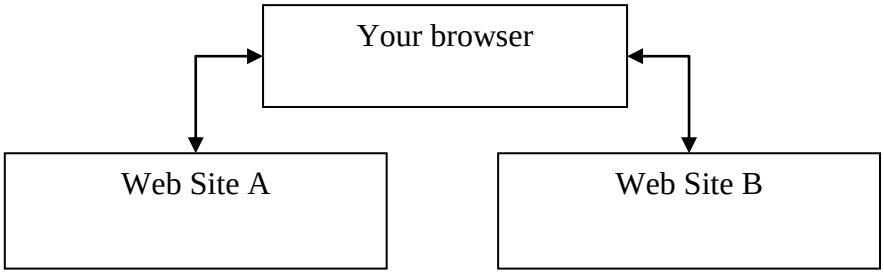
```
Define Function Hypoteneuse ( Float Width, Float Length) Returns Float
        Declare Hypo as Float
        Set Hypo =  Sqrt(Length*Length + Width*Width)
        Return (Hypo )
End Function Definition
```

And there you have it! Now in a program, you can write things like

```
Declare L, W as Float
Write "Enter the length and width of your triangle: "
Input L
Input W
Write "The length of your hypotenuse is " + Hypoteneuse(L, R)
```

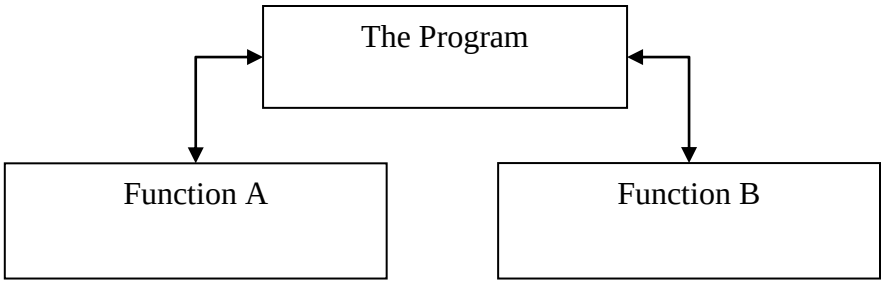
E. A Comparison to Computer Networks

You are sitting at your computer and decide to visit two web sites. At each site, you fill in a form indicating what you want done or what information you want, and each site responds with some information. Information is sent to a web site, and requested information is returned. So we have the following data flow:



The three actors here – your browser, Web Site A, and Web Site B – run on three different computers. They communicate with one another only through the network, as shown in the arrows. None of the computers has any components in common with either of the others. In particular, they do not share any memory!

Now consider a program that needs to use two functions, Function A and Function B. We have this diagram:



This is the same diagram! This time, the three actors run on the same computer, but they still communicate with one another *only through the function call mechanism*. And they do not share memory! Each actor has its own set of variables, which it does not share with either of the others.

So when writing a program and some functions, you should imagine that you are writing programs to be run on different computers. Nothing is shared!