MIRZO ULUG’BEK NOMIDAGI O’ZBEKISTON MILLIY UNIVERSITETI

MATEMATIKA FAKULTETI

1 – KURS MAGISTRANTI

Kurs ishi

MAVZU: Function.

Bajardi: G’aniyev F.

Qa’bul qildi: Ibragimov F.

Toshkent 2017

MUNDARIJA

1. About Function.
2. Terms.
3. Recursion function.
4. Defining a Function
5. Nesting fuctions.

Functions

Functions are "self contained" modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. Once a function is written, it can be used over and over and over again. Functions can be "called" from the inside of other functions. Let's translate the above program in C programming language −

# i n c l u d e < s t d i o . h >

m a i n ( ) {

i n t s e t 1 [ 5 ] = { 1 0 , 2 0 , 3 0 , 4 0 , 5 0 } ;

i n t s e t 2 [ 5 ] = { 1 0 1 , 2 0 1 , 3 0 1 , 4 0 1 , 5 0 1 } ;

i n t i , m a x ;

/ \* P r o c e s s f i r s t s e t o f n u m b e r s a v a i l a b l e i n s e t 1 [ ] \* /

m a x = s e t 1 [ 0 ] ;

i = 1 ;

w h i l e ( i < 5 ) {

i f ( m a x < s e t 1 [ i ] ) {

m a x = s e t 1 [ i ] ;

}

i = i + 1 ;

}

p r i n t f ( " M a x i n f i r s t s e t = % d \ n " , m a x ) ;

/ \* N o w p r o c e s s s e c o n d s e t o f n u m b e r s a v a i l a b l e i n s e t 2 [ ] \* /

m a x = s e t 2 [ 0 ] ;

i = 1 ;

w h i l e ( i < 5 ) {

i f ( m a x < s e t 2 [ i ] ) {

m a x = s e t 2 [ i ] ;

}

i = i + 1 ;

}

p r i n t f ( " M a x i n s e c o n d s e t = % d \ n " , m a x ) ;

}

When the above code is compiled and executed, it produces the following result –

M a x i n f i r s t s e t = 5 0

M a x i n s e c o n d s e t = 5 0 1

If you are clear about the above example, then it will become easy to understand why we need a function. In the above example, there are only two sets of numbers, set1 and set2, but consider a situation where we have 10 or more similar sets of numbers to find out the maximum numbers from each set. In such a situation, we will have to repeat, processing 10 or more times and ultimately, the program will become too large with repeated code. To handle such situation, we write our functions where we try to keep the source code which will be used again and again in our programming.

Now, let's see how to define a function in C programming language and then in the subsequent sections, we will explain how to use them.

**Defining a Function**

The general form of a function definition in C programming language is as follows –

r e t u r n \_ t y p e f u n c t i o n \_ n a m e ( p a r a m e t e r l i s t ) {

b o d y o f t h e f u n c t i o n

r e t u r n [ e x p r e s s i o n ] ;

}

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function –

**Return Type** − A function may return a value. The return\_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword void.

**Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

**Parameter List** − A parameter is like a placeholder. When a function is invoked, you pass a value as a parameter. This value is referred to as the actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

**Function Body** − The function body contains a collection of statements that defines what the function does.

**Functions**

Functions "Encapsulate" a task (they combine many instructions into a single line of code). Most programming languages provide many built in functions that would otherwise require many steps to accomplish, for example computing the square root of a number. In general, we don't care how a function does what it does, only that it "does it"!

**Calling a Function**

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform a defined task.

Now, let's write the above example with the help of a function −

# i n c l u d e < s t d i o . h >

i n t g e t M a x ( i n t s e t [ ] ) {

i n t i , m a x ;

m a x = s e t [ 0 ] ;

i = 1 ;

w h i l e ( i < 5 ) {

i f ( m a x < s e t [ i ] ) {

m a x = s e t [ i ] ;

}

i = i + 1 ;

}

r e t u r n m a x ;

}

m a i n ( ) {

i n t s e t 1 [ 5 ] = { 1 0 , 2 0 , 3 0 , 4 0 , 5 0 } ;

i n t s e t 2 [ 5 ] = { 1 0 1 , 2 0 1 , 3 0 1 , 4 0 1 , 5 0 1 } ;

i n t m a x ;

/ \* P r o c e s s f i r s t s e t o f n u m b e r s a v a i l a b l e i n s e t 1 [ ] \* /

m a x = g e t M a x ( s e t 1 ) ;

p r i n t f ( " M a x i n f i r s t s e t = % d \ n " , m a x ) ;

/ \* N o w p r o c e s s s e c o n d s e t o f n u m b e r s a v a i l a b l e i n s e t 2 [ ] \* /

m a x = g e t M a x ( s e t 2 ) ;

p r i n t f ( " M a x i n s e c o n d s e t = % d \ n " , m a x ) ;

}

When the above code is compiled and executed, it produces the following result –

M a x i n f i r s t s e t = 5 0

M a x i n s e c o n d s e t = 5 0 1

When a function is "called" the program "leaves" the current section of code and begins to execute the first line inside the function. Thus the function "flow of control" is:

1 . The program comes to a line of code containing a "function call".

2 . The program enters the function (starts at the first line in the function code).

3 . All instructions inside of the function are executed from top to bottom.

4 . The program leaves the function and goes back to where it started from.

5 . Any data computed and RETURNED by the function is used in place of the function in the

original line of code.

**Why do we Write Functions?**

1 . They allow us to conceive of our program as a bunch of sub-steps. (Each sub-step can be its own function. When any program seems too hard, just break the overall program into substeps!)

2 . They allow us to reuse code instead of rewriting it.

3 . Functions allow us to keep our variable namespace clean (local variables only "live" as long as the function does). In other words, function\_1 can use a variable called i, and function\_2 can also use a variable called i and there is no confusion. Each variable i only exists when the computer is executing the given function.

4 . Functions allow us to test small parts of our program in isolation from the rest. This is especially true in interpreted langaues, such as Matlab, but can be useful in C, Java, ActionScript, etc.

**Steps to Writing a Function**

1 . Understand the purpose of the function.

2 . Define the data that comes into the function from the caller (in the form of parameters)!

3 . Define what data variables are needed inside the function to accomplish its goal.

4 . Decide on the set of steps that the program will use to accomplish this goal. (The Algorithm)

Parts of a "black box" (i.e., a function)

Functions can be called "black boxes" because we don't need to know how they work. Just what is supposed to go into them, and what is supposed to come out of them.

When defining a program as a black box, we must describe the following attributes of the function.

**Note: most documentation systems are just this, the attributes of a function with no code associated with it.**

1 . **The Name** - describes the purpose of the function. Usually a verb or phrase, such as "compute\_Average", or just "average".

2 . **The Inputs** - called parameters. Describe what data is necessary for the function to work and gives each piece of data a Symbolic Name for use in the function.

3 . **The Calculation** - varies for each function

4 . **The Output** - Usually one (but sometimes zero or sometimes many) values that are calculated inside the function and "returned" via the output variables.

**Function Workspace**

Every function has its own Workspace. This means that every variable inside the function is only usable during the execution of the function (and then the variables go away).

Having a separate "workspace" for each function is critical to proper software engineering. If every function shared every variable in an entire program, it would be easy to inadvertently change the values of variables that you shouldn't. Further, it would be hard to remember what "names" have been used elsewhere, and coming up with new names to represent similar ideas would be challenging.

A side-effect of function variables not existing after the end of the function is that the only way to get information "out" of a function is by "returning" that information via the output of the function.

Additionally, the function can only "see" the information that is "passed" to it via parameters. Thus the only way information can get "in" to the function is by using parameters.

**Note: In certain object oriented languages (e.g., C++, Java, ActionScript), a function can also see all of the variables associated with its containing object.**

**Formal vs. Actual Parameters**

When we create a function, it should represent a "generic" action that can be applied in many circumstances. For example, if we want to find the average grade, it doesn't matter if it is on a test, or on a quiz, or an assignment, or a midterm, etc... given any list of grades we can compute an

average!

...but if it can be any list of grades, how do we know what the list of grades will be called? The answer: we don't care. You, the programmer of the function, provide your own name for the data. This is much the same as when a sales person calls you and reads a script trying to sell something to you, they say: Dear \_insert customer name here\_, let me sell you our wonderful product.

When writing a function, the programmer must provide a blank to plug in what ever data is of current interest; the blank should have a good symbolic name saying what it will represent. Here is a pseudocode function example:

f u n c t i o n a v e r a g e \_ g r a d e ( l i s t \_ o f \_ g r a d e s )

. . . .

e n d f u n c t i o n

Inside the average\_grade function, the name list\_of\_grades will be used in place of whatever variable some other user has stored his or her grades in. Thus to call the function, I might write:

/ / T h i s s o m e o t h e r c o d e ( n o t t h e f u n c t i o n c o d e )

m i d t e r m \_ g r a d e s = . . . / / c r e a t e a r r a y o f g r a d e s

p r i n t " t h e a v e r a g e o f t h e m i d t e r m w a s "

p r i n t a v e r a g e \_ g r a d e ( m i d t e r m \_ g r a d e s )

In "My" code, the grades are stored in the variable, "midterm\_grades". Inside the function, the grades are stored in the variable "list\_of\_grades". Thus, during the execution of the program, both names will refer to the same thing but at different times.

The parameter "list\_of\_grades" is called a Formal paramater; again, this just means a place holder name for any possible set of grades.

The variable midterm\_grades is the Actual paramater. This means "what is actually used" for this call to the function, such as [90, 100, 70];

**Nesting Functions?**

Often the question comes up: Can we Nest functions (can we place the code for one function inside another function)? The answer is a resounding: NO!

It should be noted, that we can utilize (call) other functions inside a function, but we cannot create the recipe for a new function there.

**Note: Sadly this question often comes up when we are careless with our indentation and syntax, and accidentally forget to end our first function, thus, to the computer, placing our second function inside our first function:**

/ / H e r e i s p s e u d o c o d e o f t h e c o r r e c t l a y o u t o f t w o f u n c t i o n s

f u n c t i o n 1 ( )

{

c o d e ;

c o d e ;

c o d e ;

}

f u n c t i o n 2 ( )

{

c o d e ;

c o d e ;

c o d e ;

}

/ / H e r e i s p s e u d o c o d e o f t h e I N C O R R E C T l a y o u t o f t w o f u n c t i o n s

f u n c t i o n 1 ( )

{

c o d e ;

c o d e ;

c o d e ;

/ / f o r g o t t o e n d t h e f u n c t i o n p r o p e r l y w i t h

f u n c t i o n 2 ( )

{

c o d e ;

c o d e ;

c o d e ;

}

}/ / a c c i d e n t a l l y e n d e d t h e f i r s t f u n c t i o n A F T E R t h e s e c o n d !

It will often be convenient to use the same function name twice. Suppose, for instance, that you have a function that returns the volume of a box, given the three sides:

i n t v o l u m e ( i n t x , i n t y , i n t z ) {

r e t u r n x \* y \* z ;

}

and that you also want to have a similar function compute the same thing, except for floating point values:

f l o a t v o l u m e ( f l o a t x , f l o a t y , f l o a t z) {

r e t u r n x \* y \* z ;

}

**Recursive function**

**What's a Recursive Function?**

Technically, a recursive function is a function that makes a call to itself. To prevent infinite recursion, you need an if-else statement (of some sort) where one branch makes a recursive call, and the other branch does not. The branch without a recursive call is usually the base case (base cases do not make recursive calls to the function).

Functions can also be mutually recursive. For example, function f() can call function g() and function g() can call function f(). This is still considered recursion because a function can eventually call itself. In this case, f() indirectly calls itself.

Functions can be tail-recursive. In a tail-recursive function, none of the recursive call do additional work after the recursive call is complete (additional work includes printing, etc), except to return the value of the recursive call.

The following is typical of a tail-recursive function return.

r e t u r n r e c \_ f u n c ( x , y ) ; / / n o w o r k a f t e r r e c u r s i v e c a l l , j u s t r e t u r n t h e v a l u e o f c a l l

The following is NOT tail-recursive.

r e t u r n r e c \_ f u n c ( x , y ) + 3 ; / / w o r k a f t e r r e c u r s i v e c a l l , a d d 3 t o r e s u l t

because once rec\_func is done, it must add 3, and then return that value. Hence, additional work. Hence, not tail-recursive.

It's common, in tail-recursive functions, to have one of the parameters be pass-by-reference (though this is not necessary), which is used to accumulate the answer. Once the base case is reached, you simply return the parameter value, which has the answer. Thus, tailrecursive functions often require an additional parameter, where non tail-recursive functions do not.

The main drawback with recursion is that it can use O(n) space (stack space) when a simple loop may only use O(1). For example, printing an array should only require O(1) space in addition to the array. You just need a looping variable, which requires O(1) space. The recursive solution uses O(n) space.

However, a good compiler can determine if a function is tail-recursive, and internally produce code that runs in O(1) space, thus giving you the benefits of recursion (i.e., recursion is "neat" and compact to write) but with the space efficiency of a loop. Languages other than C++ (say, ML) often require tail-recursion be converted to loops by the compiler. Typical C++ compilers generally (I believe) do not make this optimization.

**When to Use Recursion?**

Some people think recursion is so neat, they would like to use it all the time. In general, recursion should be used when you know the number of recursive calls isn't excessive. That number depends somewhat on how much memory you have. Usually, a 1000 recursive calls should be fine. Stack sizes can now be several megabytes of memory, which allows recursion to go fairly deep without causing a core dump.

Sometimes recursion is used because it produces a cleaner answer compared to the iterative version. For example, nearly all code written for tree-like structures is recursive. Many sorting algorithms are more naturally written recursively as well.

However, recursive solutions can be VERY inefficient, if you aren't careful. For example, the

obvious recursive solution to compute the Nth Fibonacci numbers has exponential running time, even though the loop version runs in O(n). This is one good reason to study algorithms (in CMSC 351)---so you know what the running time of a recursive algorithm is, and decide if that solution is worthwhile.

If it sounds like recursion is always bad, realize that recursion often produces solutions that

are very compact (requires few lines of typed code). Code with fewer lines are easier to debug than code with many lines. Writing recursive functions can give you greater confidence that you are coding correctly. Often, writing code that ought to be recursive without recursion (i.e., as loops) produces messy code. You may even have to simulate a stack (recursion uses the program stack behind the scenes) to get the behavior you want.

**How to Think Recursively**

When you want to write a recursive function, begin by writing its prototype. Define what the recursive function does in English (or whatever language you prefer).

For example, consider the following stand-alone (non-member) function:

/ / S u m s f i r s t n e l e m e n t s o f a r r

i n t s u m ( i n t a r r [ ] , i n t n ) ;

When you make the call:

s u m ( a r r , 1 0 ) ;

You know this is summing the first 10 elements of arr.

Writing down the purpose of the function may seem like a trivial step, but it's important. Students often fail to write recursive functions correctly because they forget what the function is trying to do. Then, repeat the following to yourself several times: Recursion solves a big problem (of size n, say) by solving one or more smaller problems, and using the solutions of the smaller problems, to solve the bigger problem. For example, suppose you are running for some fundraising marathon. You have collected a large stack of pledges, but want to know what the total amount that people have pledged to you. This stack is rather large, and you'd rather not do the work yourself.

However, you have many willing friends. You divide the stack of pledges and ask your friend "Could you please add up the dollar amount in pledges in this stack? I've only given you half, so there's half the work to do.". As a sneaky person, you give the other half toanother friend, and say the same thing. Once both are done, they will give their answer to you, and you add their results.

Thus, you have broken down the problem into two smaller parts, and asked your friends to do the work.

Now those friends are clever, so they divide the stack into two parts (now each has two stacks with size 1/4) and ask two of their friends. When their friends are done, they return their answer, and the result is summed.

Eventually, there is only a stack of two pledges, and these are given to two friends, and those friends, seeing how silly the problem is now, just tell the first friend the only value on the pledge. There's no need to ask any more friends, because you're down to one pledge (this is the base case).

Thus, recursion is all about breaking a problem down, and solving that, and that smaller problem is solved by breaking it down some more, and trying to solve that. Eventually, you reach an easy solution (the base case), and return the solution.

It's interesting to think about what happens to the stack when this happens. As you make each recursive call, the stack grows larger and larger. When you reach the base case, the recursion is done, and the stack becomes smaller and smaller, as it passes the solution back.

Once you have the prototype written, think about solving the next smaller sized problem.

Thus, if the call is:

s u m ( a r r , n ) ;

what's the next smallest size? What about n - 1?

s u m ( a r r , n - 1 ) ;

Suppose someone gave you the answer to this sum. What would you have? This is where it's important to remember how you defined the function. This would give you the sum of the first n - 1 elements.

Now that you have this solution (which you can assume), what's needed to solve the entire problem? Well, you haven't added arr[ n - 1 ]. So, do that.

Finally, you should deal with the base case, which is the smallest problem (in terms of input size, n) you can solve without any recursive calls. It turns out that

s u m ( a r r , 0 ) ;

is the smallest input size. While having an array of size 0 may not make sense, it's fine. Just let this value be 0 (since 0, summed to any value, just gives you that value). 0 is the additive

identity.

Here's the code:

i n t s u m ( i n t a r r [ ] , i n t n )

{

i f ( n = = 0 ) / / b a s e c a s e

r e t u r n 0 ; / / n o r e c u r s i v e c a l l

e l s e

{

i n t s m a l l = s u m ( a r r , n - 1 ) ; / / s o l v e s m a l l e r p r o b l e m

/ / u s e s o l u t i o n o f s m a l l e r t o s o l v e l a r g e r

r e t u r n s m a l l + a r r [ n - 1 ] ;

}

}

So, here are the steps to writing a recursive function.

1 . Write a prototype for the recursive function.

2 . Write a comment that describes what the function does.

3 . Determine the base case (there may be more than one), and its solution(s).

4 . Determine what smaller problem (or problems) to solve. If it makes it easier for you to follow, save the solutions to the smaller problems to local variables (e.g., small in the sum() example).

ASSUME the recursive call works (similar to inductive hypothesis in CMSC 250), i.e., that it will correctly compute the answer.

5 . Use the solutions of the smaller problem to solve the larger problem. (If this is done INCORRECTLY, the solutions of the smaller problems will also be computed incorrectly, thus, the assumption in the previous step will fail).

**What Makes Recursion Work**

Recursion works only when a problem has a recursive structure. This means that the problem's solution is similar for a large input size as well as a small input size. If reducing the input size causes the solution to look different, then it will be hard to use recursion. However, for the same reason, it can be hard to use loops too. Loops also depend on doing the same thing over and over, but on, say, a different index of an array, or a different node in a linked list.

Also, you must be able to use the solutions of the small problem to help solve the larger one. Again, if the solutions aren't useful to you, then recursion isn't going to be useful.

Fortunately, many problems exhibit this kind of behavior. If you can write it in a loop, there are ways to convert it to recursion (and vice versa)

**Misconceptions about Recursion**

Suppose function f() calls g() and then function g() calls h() which then calls i(). What happens when i() is done?

As you know, the flow of execution goes back to h() and when that's done, it goes back to g(), when when g() is done, it goes back to f().

**Avoiding Static/Global Variables**

Generally, I put static variables in the same category as global variables, and prefer to avoid them when I can. In particular, using static variables to do recursion can lead to problems. For example, static variables are initialized when the function is initially called for the first time. However, it's hard to re-initialize them afterwards.

Consider

i n t s u m ( i n t a r r [ ] , i n t n )

{

i n t r e s u l t = 0 ;

i f ( n = = 0 )

r e t u r n r e s u l t ;

e l s e {

r e s u l t + = a r r [ n - 1 ] ;

s u m ( a r r , n - 1 ) ;

}

}

This should work fine the first time you call it. However, the second time, result is not 0, and your answer will be wrong.

Instead, you should always update arguments to the recursive function.

**"Real" Recursion**

We can try to write this function more "recursively". That is, to think about how to solve the smaller problem. Initially, you'd think about what kind of function you want. The goal is to print the numbers from 0 to n - 1. So you can imagine having a function to do this:

v o i d p r i n t I t ( i n t n ) ;

The recursive call passes in n.

Now, what would be a smaller recurisve call?

v o i d p r i n t I t ( n - 1 ) ;

This function would print out 0 up to n - 2. If this occurred, what would we do to print the rest? You would print n - 1. The base case is when n == 0.

So, here's how you would write the code in a more traditional recursive way.

v o i d p r i n t I t ( i n t n ) {

i f ( n = = 0 ) { / / b a s e c a s e

c o u t < < 0 < < e n d l ;

}

e l s e { / / r e c u r s i v e c a s e

p r i n t I t ( n - 1 ) ; / / r e c u r s i v e c a l l

c o u t < < n - 1 < < e n d l ;

}

}

Notice that you pass one fewer parameter this way, although it doesn't match up with the loop nearly as well. In general, you would "prefer" to do it this way, because it's more "natural" when writing recursive functions.

Here's an analogy. There are two ways to learn a foreign language. Either learn a foreign language by learning it like native speakers speak it, or learn to translate every word in your native language to the foreign language. The second method works, somewhat, but isn't as "fluent" as the first.

For coding, of course, it doesn't matter which method, as long as it works, but nevertheless, the second method we looked at it more "fluent" since the problem is solved from the usual methodology of solving smaller problems and using it to solve bigger problems.

Occasionally, you must use the loop method to come up with a recursive call, because not all problems are nearly as easy to break down into smaller cases.