COE 251

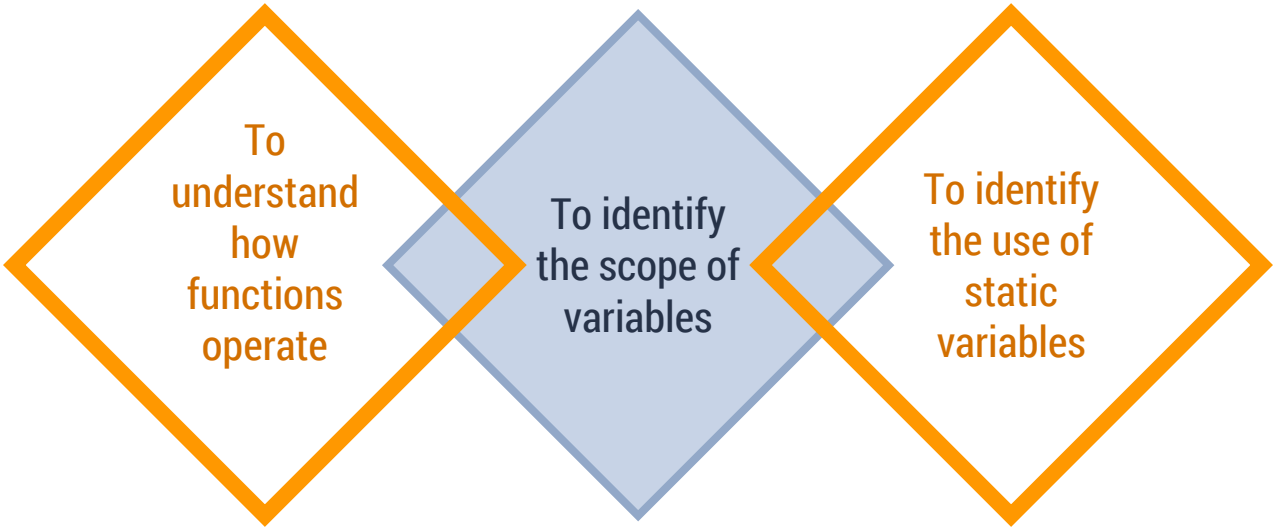
FUNCTIONS

Dr. Eliel Keelson

OUTLINE

- FUNCTIONS
- LOCAL & GLOBAL VARIABLES
- STATIC VARIABLES

INTENDED LEARNING OUTCOMES (ILOs)



1

FUNCTIONS

FUNCTIONS

- Functions are a group of statements that perform a coherent task of some kind.
- This group of statements is given a name (an identifier) so that whenever we need that particular task to be done we call/invoke the function by its name.
- This helps the programmer to avoid typing those same lines of instructions every time that task is to be carried out.

FUNCTIONS

- In our study of functions we would understand what the following are:

1. Function Definition
2. Function Declaration/Prototype
3. Function Call

FUNCTION DEFINITION

FUNCTION DEFINITION

- The function definition is the place in code where the task is actually carried out.
- It contains all the steps/instructions/statements to performing the particular task requested.
- It usually consists of five main parts.

FUNCTION DEFINITION

- The 5 main parts that a function definition may contain are:

1. Data type (which is the same as the return value's data type)
2. Function name (any valid identifier)
3. Declared Parameters/Arguments (what the function needs ahead of time before it can perform its task)
4. Statements (set of instructions the function follows in performing the task)
5. The return value (the value returned after performing the task)

FUNCTION DEFINITION

- Depending on the function, some of these parts may not always appear.
- For example we would not always return a value at the end of the function definition.
- So since the data type of the function is based on the data type of the return value in places we do not return a value the data type of the function is void.

FUNCTION DEFINITION

- Another feature of a function definition which may not appear is the parameter.
- In cases where the function does not require anything ahead of time before performing its task no parameter or argument is needed/provided.

FUNCTION DEFINITION

- The general syntax of Function definition is shown below

```
data_type functName (declared parameter list)
{
        statements;
        return value;
}
```

FUNCTION DECLARATION

FUNCTION DECLARATION/PROTOTYPE

- The function declaration is a statement that only informs the compiler that such a function exists.
- Just like we declare a variable before using it, so do we declare a function before calling/invoking it.

FUNCTION DECLARATION/PROTOTYPE

- The general syntax of Function declaration is shown below

data_type FunctName (declared parameter list);

FUNCTION CALL

FUNCTION CALL

- Whenever we need the function to perform its task we call it by its name.
- In calling the function we pass the necessary parameters (arguments) that it requires before it can perform its task.
- In places where they are not needed we pass no arguments.

FUNCTION CALL

- The general syntax of Function call is shown below

FunctName (arguments);

FUNCTIONS

- So in summary the function declaration is like an advertisement to tell the compiler that such a function exists; the function is called whenever the task is to be performed, and we go to the function definition whenever the function is called.

FUNCTIONS

- **NB:** Note that when the function is called we stop whatever we are doing and go to where the function has been defined in our program and return to the point we left off after the function has been executed.

FUNCTIONS

- **NB:** C does not support nested functions i.e. defining a function within another function. They must **always** be separated.

FUNCTIONS

- The key to understanding functions is learning how to identify a function declaration, call and definition.
- Let’s take an example and try and identify these three as well as explain how it works.

FUNCTIONS - EXAMPLE

Line	Code
1	#include <stdio.h>
2	float calcArea(float rad); /*function declaration */
3	int main()
4	{ float x,y;
5	x = 10.25;
6	y = calcArea(x); /* function call */
7	printf("the area of a circle of radius %g is %g\n", x, y)
8	return 0;
9	}
10	float calcArea(float rad)
11	{
12	float a;
13	a = 3.142 * rad * rad;
14	
15	return(a);
16	}

FUNCTIONS - EXAMPLE

- We would try to step through the code one line at a time.
- The first step is to identify the function declaration, call and definition.
- And these can be easily identified if you remember the syntax.

FUNCTIONS - EXAMPLE

- From the syntax we can then reckon that line 2 contains the function declaration, line 6 contains the function call and lines 10 to 16 contain the function definition.

FUNCTIONS - EXAMPLE

- So walking through the code line 1 is the basic preprocessor directive telling the compiler to include the standard input and output header file.
- Line 2 tells the compiler that there is a function that exists by name calcArea and the function requires one floating point (rad) as a parameter before it can carry out its task and it would return a value of data type float.

FUNCTIONS - EXAMPLE

- Line 3 to 9 is the main function.
- In the main function line 4 declares two floating point variables by name x and y.
- In line 5 we assign 10.25 to the value x.

FUNCTIONS - EXAMPLE

- In line 6 y is assigned to a function call (i.e. calcArea(x);). And we can see that it has the variable x as an argument.
- And we can see that x is a floating point just like the function requested in its declaration.
- Since the function is being called at this point on line 6 we stop whatever we are doing in the main and jump to where the function has been defined(line 10-16)

FUNCTIONS - EXAMPLE

- So now that we are in the definition (line 10). We copy the value in x from the function call (i.e. 10.25) into the variable rad. So now rad has a value 10.25.
- So on line 12 we declare a variable a.
- On line 13, we assign a to the result of 3.142*rad*rad (formula for calculating the area of a circle) which would result in 330.106
- Then on line 15 we return the value in a to where the function was called (i.e. line 6)

FUNCTIONS - EXAMPLE

- So we jump back to line 6 and put the results of the function definition exactly where the call was. Therefore y would now be y=330.106;(i.e. the results of the function definition)
- So on line 7 we print out:

The area of a circle of radius 10.25 is 330.106

- That brings us to the end of the program.

FUNCTIONS - EXAMPLE

- Let's try another example involving void.

```
#include <stdio.h>
void avg (int a, int b, int c);
int main()
{
        int x=3, y = 4, z = 5;
        avg(x,y,z);
        printf("Done!\n");
        return 0;
}
```

```
void avg(int a, int b, int c)
{
    int w = (a +b + c)/3;
    printf("The average of %d, %d and %d is %d\n", a, b, c, w);
}
```

FUNCTIONS - EXAMPLE

- The output of the code above is

```
The average of 3, 4 and 5 is 4

Done!
```

- Note the order of the statements printed in the output. Also note that in this example the function does not return a value hence the void.

2

LOCAL AND GLOBAL VARIABLES

LOCAL AND GLOBAL VARIABLES

- A **local variable** is a variable that is declared within a function. It is only recognized within that function. Outside the function it is **not** known.
- A **global variable** on the other hand is variable that is declared outside all functions. It is recognized everywhere within that program.

LOCAL AND GLOBAL VARIABLES - EXAMPLES

```
#include <stdio.h>
void funct1();
int x = 50;
int main()
{
        int y = 30;
        printf("%d\n", y);
        printf("%d\n", x);
        funct1();
        printf("End\n");
        return 0;
}
```

```
void funct1()
{
        float z = 20;
        printf("%d\n", z);
        printf("%d\n", x);
        return();
}
```

LOCAL AND GLOBAL VARIABLES

- The code above would result in the following output:

```
30

50

20

50

End
```

LOCAL AND GLOBAL VARIABLES

- From the code x was declared outside all functions thus making x a **global variable**.
- y and z were declared within the main function and funct1 function respectively thus making them **local variables**.

LOCAL AND GLOBAL VARIABLES

- So unlike x which is recognized everywhere y and z are only recognized within their functions.
- So the value of x can be accessed and altered anywhere in the program whiles y can only be accessed/altered in main and z can be only accessed/altered in funct1.

3

STATIC LOCAL VARIABLES

STATIC LOCAL VARIABLES

- The value of local variable is often wiped out of memory as soon as its function comes to an end.
- Global variables however have their values stored in memory until the program comes to an end.
- So with local variables (**non-static local variables**) they are always initiated anew when their function is called and destroyed/deleted when their function comes to an end.

STATIC LOCAL VARIABLES

- **Static local variables** however behave differently. They remain in memory throughout the program after the function they belong to (are declared in) is called.
- They are only initialized **once**. Their values however may change during the course of the program.

NON-STATIC LOCAL VARIABLE EXAMPLE

```
#include <stdio.h>
int funct1(int k);
int main()
{
        int a, b = 0;
        for(a=0;a<5;a++)
        {
                b = funct1(a);
                printf("%d\t", b);
        }
return 0;
}
```

```
int funct1(int k)
{
        int j = 0;
        j += k;
        return (j);
}
```

NON-STATIC LOCAL VARIABLE EXAMPLE

- The result of the above non-static local variable code example is

```
0  1  2  3  4
```

- Now we would compare this result with a static local variable example

STATIC LOCAL VARIABLE EXAMPLE

```
#include <stdio.h>
int funct1(int k);
int main()
{
        int a, b = 0;
        for(a=0;a<5;a++)
        {
                b = funct1(a);
                printf("%d\t", b);
        }
return 0;
}
```

```
int funct1(int k)
{
        static int j = 0;
        j += k;
        return (j);
}
```

STATIC LOCAL VARIABLE EXAMPLE

- The result of the above static local variable code example is

```
0    1    3    6    10
```

- Compare the code above with the previous code and strike the difference based on the explanation of static local variables.

STATIC LOCAL VARIABLE EXAMPLE

- In your comparison you would realize that with the introduction of static int j, j becomes a **static local variable**.
- So when funct1 is called for the first time j is initialized to 0. Then the remaining statements in the function are executed.
- Upon the next function call to funct1 we **do not reinitialize** j to 0 but continue from the last value it obtained before exiting the function.

THANKS!

Any questions?

You can find me at

elielkeelson@gmail.com & ekeelson@knust.edu.gh