APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS

a complete app for ktu students

Get it on Google Play

www.ktuassist.in

MODULE 4

FUNCTIONS

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

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

```
return_type function_name( parameter list ) {
   body of the function
}
```

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.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as 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.

DOWNLOADED FROM KTUASSIST.IN

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

Example

Given below is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;
}
```

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return_type function_name(parameter list);

For the above defined function max(), the function declaration is as follows −

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function

DOWNLOADED FROM KTUASSIST.IN

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 the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

Control of the program is transferred to the user-defined function by calling it.

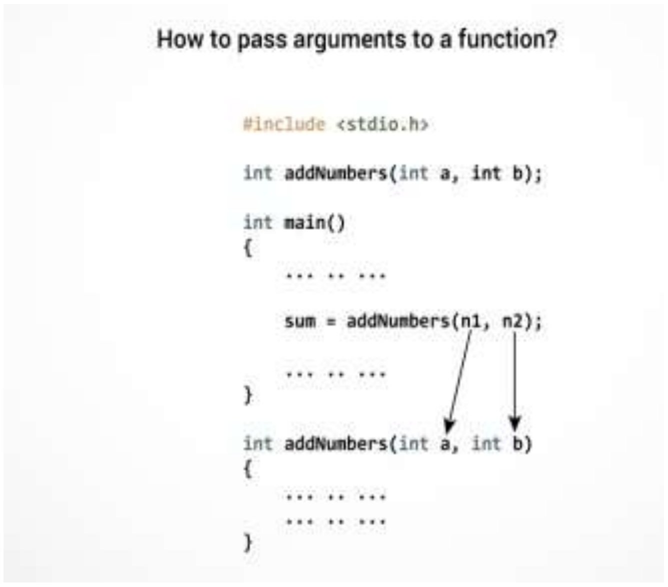
Syntax of function call

```
functionName(argument1, argument2, ...);
```

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables **n1** and **n2** are passed during function call.

The parameters **a** and **b** accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.



```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error.

If **n1** is of char type, **a** also should be of char type. If **n2** is of float type, variable **b** also should be of float type.

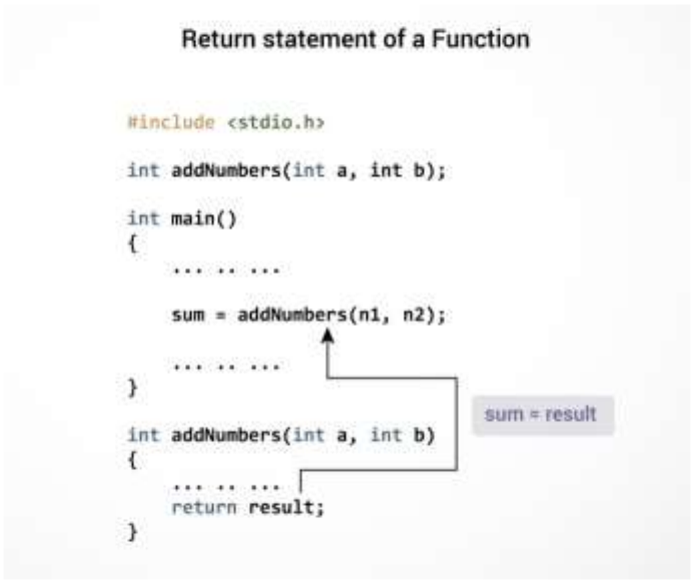
DOWNLOADED FROM KTUASSIST.IN

A function can also be called without passing an argument.

Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.

In the above example, the value of variable **result** is returned to the variable **sum** in the **main()**function.



```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

Parameter Passing

Two Ways of Passing Argument to Function in C Language :

Call by Reference

Call by Value

Let us discuss different ways one by one –

A.Call by Value :

```
#include<stdio.h>
void interchange(int number1,int number2);
int main()
{
   int num1=50,num2=70;
   interchange(num1,num2);
   printf("\nNumber 1 : %d",num1);
```

DOWNLOADED FROM KTUASSIST.IN

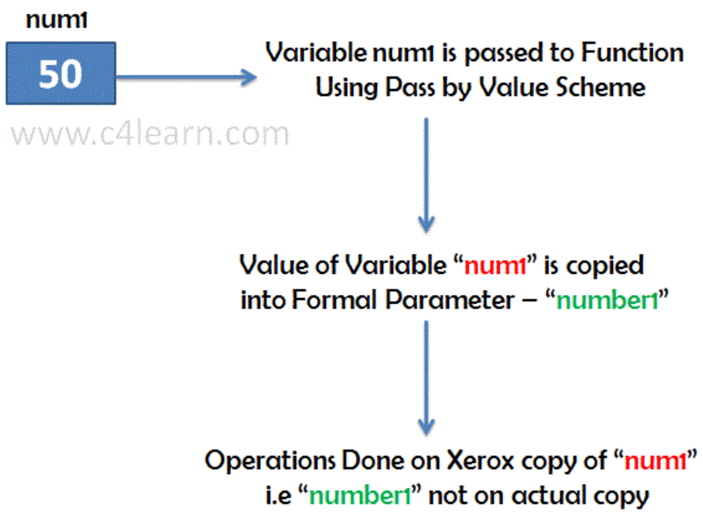
```
   printf("\nNumber 2 : %d",num2);
   return(0);
}

void interchange(int number1,int number2)
{
   int temp;
   temp = number1;
   number1 = number2;
   number2 = temp;
}
```

Output :

```
Number 1 : 50

Number 2 : 70
```

Explanation : Call by Value

1. While Passing Parameters using call by value , xerox copy of original parameter is created and passed to the called function.
2. Any update made inside method will not affect the original value of variable in calling function.
3. In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1,number2 variable of sum function respectively.
4. As their scope is limited to only function so they cannot alter the values inside main function.


num1
50
Variable num1 is passed to Function Using Pass by Value Scheme
www.c4learn.com
Value of Variable "num1" is copied into Formal Parameter – "number1"
Operations Done on Xerox copy of "num1" i.e "number1" not on actual copy

<u>B.Call by Reference/Pointer/Address :</u>

```
#include<stdio.h>
void interchange(int *num1,int *num2);
int main()
```

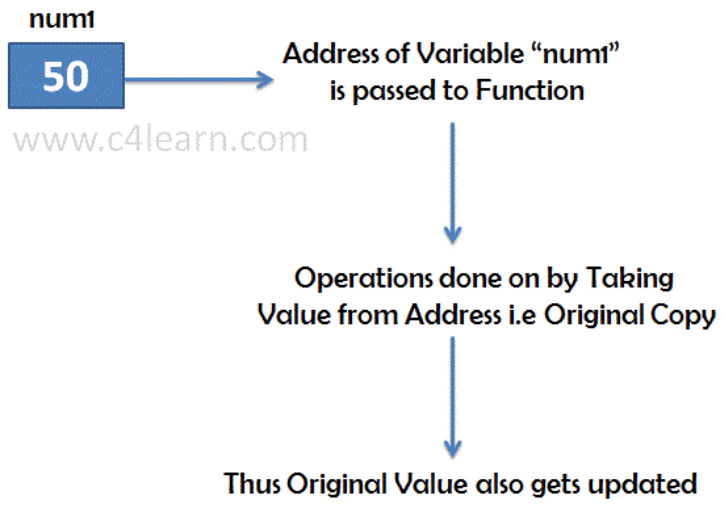
```
 {
   int num1=50,num2=70;
   interchange(&num1,&num2);
   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);
   return(0);
}

void interchange(int *num1,int *num2)
{
   int temp;
   temp  = *num1;
   *num1 = *num2;
   *num2 = temp;
}
```

Output :

```
Number 1 : 70
Number 2 : 50
```

Explanation : Call by Reference

1. While passing parameter using call by address scheme , we are passing the actual address of the variable to the called function.
2. Any updates made inside the called function will modify the original copy since we are directly modifying the content of the exact memory location.


num1
50
Address of Variable "num1" is passed to Function
www.c4learn.com
Operations done on by Taking Value from Address i.e Original Copy
Thus Original Value also gets updated

Point	Call by Value	Call by Reference
Copy	Duplicate Copy of Original Parameter is Passed	Actual Copy of Original Parameter is Passed

DOWNLOADED FROM KTUASSIST.IN

Point	Call by Value	Call by Reference
Modification	No effect on Original Parameter after modifying parameter in function	Original Parameter gets affected if value of parameter changed inside function

Passing Array to Function

Passing One-dimensional Array In Function

Single element of an array can be passed in similar manner as passing variable to a function.

C program to pass a single element of an array to function

```
#include <stdio.h>
void display(int age)
{
   printf("%d", age);
}
int main()
{
   int ageArray[] = { 2, 3, 4 };
   display(ageArray[2]); //Passing array element ageArray[2] only.
   return 0;
}
```

Output

```
4
```

Passing an entire one-dimensional array to a function

While passing arrays as arguments to the function, only the name of the array is passed (,i.e, starting address of memory area is passed as argument).

C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.

```
#include <stdio.h>
float average(float age[]);
int main()
{
   float avg, age[] = { 23.4, 55, 22.6, 3, 40.5, 18 };
   avg = average(age); /* Only name of array is passed as argument. */
   printf("Average age=%.2f", avg);
   return 0;
}
```

DOWNLOADED FROM KTUASSIST.IN

```
float average(float age[])
{
   int i;
   float avg, sum = 0.0;
   for (i = 0; i < 6; ++i) {
      sum += age[i];
   }
   avg = (sum / 6);
   return avg;
}
```

Output

```
Average age=27.08
```

Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
int main()
{
  recursion();
}
void recursion()
{
  recursion(); /* function calls itself */
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Eg: Factorial of Number

```
#include <stdio.h>
int factorial(int n);
int main()
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d", &n);
   f= factorial(n);
   printf("Factorial of %d = %d", n, f);
   return 0;
```

DOWNLOADED FROM KTUASSIST.IN

```
}

int factorial (int n)
{
   if (n >= 1)
       return n* factorial(n-1);
   else
       return 1;
}
```

Eg: Fibonacci Series

```
#include<stdio.h>
void fibonacciSeries(int n);
int main()
{
   int a,n;
   long int i=0,j=1,f;
   printf("How many number you want to print in the fibonnaci series :\n");
   scanf("%d",&n);
   printf("\nFibonacci Series: ");
   printf("%d  %d",i,j);
   fibonacciSeries(n);
   return 0;
}

void fibonacciSeries(int n)
{
   static int d=0,e=1;
   int c;
   if(n>1)
   {
       c = d + e;
       d = e;
       e = c;
       printf("%d ",c);
       fibonacciSeries(n-1);
   }
}
```

Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program –

DOWNLOADED FROM KTUASSIST.IN

1. Auto
2. Register
3. Static
4. Extern

The auto Storage Class

A variable declared inside a function without any storage class specification, is by default an automatic variable. They are created when a function is called and are destroyed automatically when the function exits. Automatic variables can also be called local variables because they are local to a function. By default they are assigned garbage value by the compiler.
The auto storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when static is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.

DOWNLOADED FROM KTUASSIST.IN

```
#include <stdio.h>
/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

   while(count--) {
      func();
   }

   return 0;
}

/* function definition */
void func( void ) {

   static int i = 5; /* local static variable */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

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

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

DOWNLOADED FROM KTUASSIST.IN

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main() {

   count = 5;
   write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
   printf("count is %d\n", count);
}
```

Here, extern is being used to declare count in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows −

```
$gcc main.c support.c
```

It will produce the executable program a.out. When this program is executed, it produces the following result −

```
count is 5
```

DOWNLOADED FROM KTUASSIST.IN

try it now

A KTU
STUDENTS
PLATFORM

SYLLABUS

KTU NOTIFICATION

NOTES

TEXT BOOKS

QUESTION PAPERS

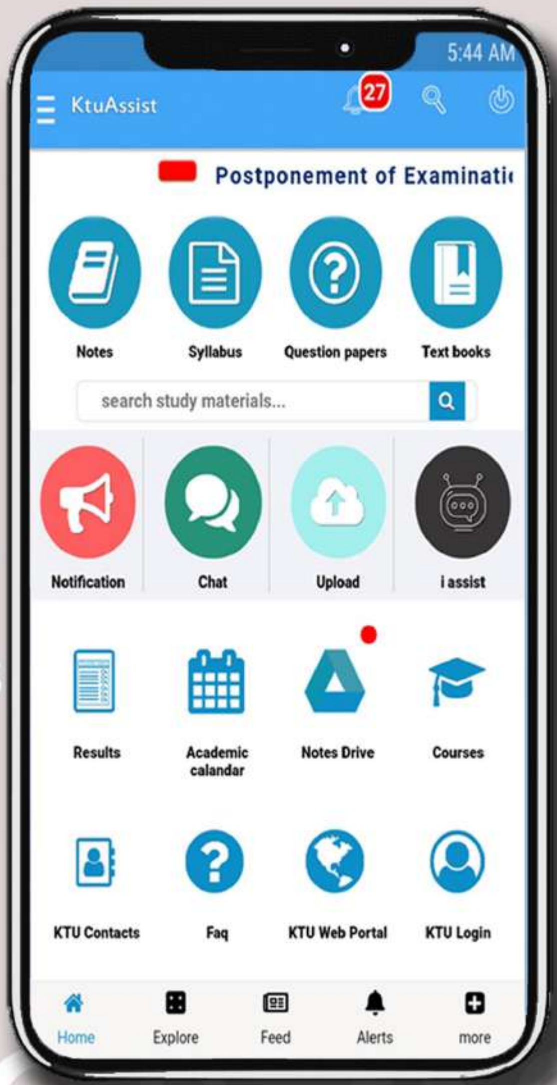
DOWNLOAD
IT
FROM
GOOGLE PLAY

CHAT

CALENDAR

FAQ

LOGIN



MUCH MORE

DOWNLOAD APP

ktuassist.in

instagram.com/ktu_assist

facebook.com/ktuassist