**Module-III**

-------------------------------------------------------------------------------------------------------------------

**FUNCTIONS**

A function is a group of statements that together perform a task.

**Definition**: A function is a self contained block of statements to perform a task.

C program is a collection of functions. Every C program has at least one function, which is **main()**.

We can divide up your code into separate functions. The logical the division of each function is such that each function performs a specific task.

Functions are of two types

* Pre defined or Library functions.

Pre defined functions: These are the **functions** which already have a definition in header files (.h files like stdio.h ), so we just call them whenever there is a need to use them.Eg:scanf(),printf(),gets(),puts(),pow(),sqrt(), etc.

* User defined functions: These are functions for which the user will define the code according to the need.

Ex:main(),add(),print(),etc.

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

A function comes under derived data type.

A function can be either a calling function or called function.

**Calling function**:A function which is calling another function is known as calling function.

**Called function**: A function which has been called by another function is known as called function.

Syntax for Function defintion

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

Terminology associated with functions

1.function declaration/prototype

2.function definition

3.function call

4.function header/declaratory

**Function declaration/prototype:**A function **declaration** tells the compiler about a function's name, return type, and parameters.

Syntax for function declaration is

ret\_type function\_name(parameter\_list);

For example 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.

**Function definition:** A function **definition** provides the actual body of the function.

**Function call**: 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.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

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;

}

For example −

#include <stdio.h>

/\* function declaration \*/

int max(int num1, int num2);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* 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;

}

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

Max value is : 200

**Categories of function**: Functions are categorized into four types based on return values and passing arguments. They are

**i)No return value and No arguments**

**ii)No return value and arguments**

**iii)Return value and no arguments**

**iv)Return value and arguments**

**No return value and no arguments:**In this techniquecalling function doesn’t pass arguments to called function and called function doesn’t return value to calling function.

**Example:**

void add()

{

int x,y,z;

printf(“\n enter two values”);

scanf(“%d%d”,&x,&y);

z=x+y;

printf(“\n sum of %d & %d is %d”,x,y,z);

}

void main()

{

void add();

add();

}

**No return value and arguments:** In this techniquecalling function passes arguments to called function and called function doesn’t return value to calling function.

**Example:**

void add(int x,int y)

{

int z;

z=x+y;

printf(“\n sum of %d & %d is %d”,x,y,z);

}

void main()

{

void add(int,int);

int a,b,c;

printf(“\n enter two values”);

scanf(“%d%d”,&a,&b);

add(a,b);

}

**Return value and no arguments:**In this techniquecalling function doesn’t pass arguments to called function and called function returns value to calling function.

**Example:**

int add()

{

int x,y,z;

printf(“\n enter two values”);

scanf(“%d%d”,&x,&y);

z=x+y;

return z;

}

void main()

{

int add();

int a;

a=add();

printf(“sum=%d”,a);

}

**Return value and arguments:**In this techniquecalling function does pass arguments to called function and called function returns value to calling function.

**Example:**

int add(int x, int y)

{

return (x+y);

}

void main()

{

int add(int,int);

int a,b,c;

printf(“\n enter two values”);

scanf(“%d%d”,&a,&b);

c=add(a,b);

printf(“sum of %d & %d=%d”,a,b,c);

}

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function −

|  |  |
| --- | --- |
| **S.N.** | **Call Type & Description** |
| 1 | [**Call by value**](https://www.tutorialspoint.com/cprogramming/c_function_call_by_value.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| 2 | [**Call by reference**](https://www.tutorialspoint.com/cprogramming/c_function_call_by_reference.htm)  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Sample program for call by value

void swap(int x, int y)

{

int t;

t=x;

x=y;

y=t;

printf(“values after swapping: %d%d”, x, y);

}

void main()

{

void swap(int, int);

int x, y;

printf(“\n enter two values: “);

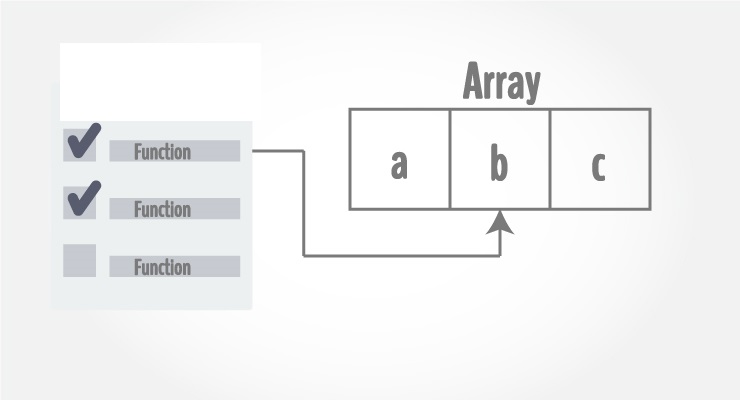
scanf(“%d%d”, &x, &y);

swap(x,y);

}

***PASSING ARRAYS TO FUNCTIONS:***

Passing entire array to function :



In C programming, a single array element or an entire array can be passed to a function.

This can be done for both one-dimensional array or a multi-dimensional array.

**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 rgument. \*/

printf("Average age=%.2f", avg);

return 0;

}

float average(float age[])

{

int i;

floatavg, sum = 0.0;

for (i = 0; i< 6; ++i) {

sum += age[i];

}

avg = (sum / 6);

return avg;

}

**Output**

Average age=27.08

**Passing Multi-dimensional Arrays to Function:**

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

Example: Pass two-dimensional arrays to a function

#include <stdio.h>

void displayNumbers(int num[2][2]);

int main()

{

int num[2][2], i, j;

printf("Enter 4 numbers:\n");

for (i = 0; i< 2; ++i)

for (j = 0; j < 2; ++j)

scanf("%d", &num[i][j]);

// passing multi-dimensional array to displayNumbers function

displayNumbers(num);

return 0;

}

void displayNumbers(int num[2][2])

{

// Instead of the above line,

// void displayNumbers(intnum[][2]) is also valid

int i, j;

printf("Displaying:\n");

for (i = 0; i< 2; ++i)

for (j = 0; j < 2; ++j)

printf("%d\n", num[i][j]);

}

**Output**

Enter 4 numbers:

2

3

4

5

Displaying:

2

3

4

5

Recursion : A program calling itself is known as recursion. A function that calls itself is known as a recursive function.

void recursion() {

recursion(); /\* function calls itself \*/

}

int main() {

recursion();

}

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.

***STORAGE CLASSES:***

Commonly we are declaring variables with their data types. To fully define a variable it needs to mention its storage class along with its data type. Variables have a data type and storage class.

A variable in C can have any one of the four storage classes:

1. Automatic storage class
2. Register storage class
3. Static storage class
4. External storage class

The general form of a variable declaration that uses storage class is:

*Storage\_specifies data\_type variable\_name;*

**1. Automatic Storage Class:**

Features of automatic storage class are as follows:

Storage : Memory

Default Initial Value : Garbage value

Scope : Local to the block in which the variable is defined.

Life : Till the control remains within the block.

* The keyword for automatic storage class is *auto.*
* Automatic variables are declared inside a function in which they are to be used. Memory locations are allocated to these variables when a function is called.
* When the control leaves the function the memory locations allocated to these variables are released. Their contents are lost and not available to any other functions.
* If function is called again new memory locations are allocated again to these variables. The old values are lost. So, automatic variables are also known as local variables.
* Automatic storage class is default one to variables.

*Example:*

#include<stdio.h>

main()

{

void function1();

void function2();

int m=1000;

function2();

printf(“%d\n”,m);

}

void function1()

{

int m=10;

printf(“%d\n”,m);

}

void function2()

{

int m=100;

function1();

printf(“%d\n”,m);

}

**OUTPUT:**

10

100

100

**2. Register Storage Class:**

Features of register storage class are as follows:

Storage : CPU Registers

Default Initial Value : Garbage value

Scope : Local to the block in which the variable is defined.

Life : Till the control remains within the block.

* The keyword for register storage class is *register.*
* A value stored in CPU register can always be accessed faster than the one which is stored in memory. So, if a variable is used at many places in a program it is better to declare its storage class as register.
* Good examples of frequently used variables are loop counters.
* Even though we have declared storage class as register, it is not sure that the value would be stored in a CPU registers. Because the number of CPU registers are limited. At that time storage class is *auto.*
* Register storage class is not suitable for float, double and long. Because the CPU registers are usually 16-bit registers. So, it can hold int, char and pointer data types.
* If we use float as register, compiler won’t give any error. Compiler will treat that one as *auto.*

*Example:*

main()

{

register int k;

for(k=1;k<=100;k++)

printf(“%d”,k);

}

**3. Static Storage Class:**

Features of static storage class are as follows:

Storage : Memory

Default Initial Value : Zero

Scope : Local to the block in which the variable is defined.

Life : Value changes between different function calls.

* The keyword for static storage class is *static.*
* The auto and static variables are local to the block in which they are defined.
* But a static variable does not disappear when the function is no longer active. Their values persist.
* This storage is suitable for recursion.
* Avoid using static variables, because their values are kept in memory when the variables are not in active. So, this is wastage of memory.

*Example:*

main()

{

void stat();

int i;

for(i=1;i<=3;i++)

stat();

}

void stat()

{

static int x=0;

x=x+1;

printf(“x=%d\n”,x);

}

**OUTPUT:**

x=1

x=2

x=3

**4. External Storage Class:**

Features of extern storage class are as follows:

Storage : Memory

Default Initial Value : Zero

Scope : Global

Life : Till the control remains within the program.

* External variables are nothing but global variables.
* Commonly global variables declaration takes place before the main function.
* But if we declare after all functions, then we can’t access those variables in the

functions.

* In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared.
* Use extern storage class for only those variables which are being used by almost all functions in the program.
* Declaring all the variables as extern would amount to a lot of wastage of memory space, because these variables remain active throughout the life of the program.

*Example:*

**Source code: prg1.c**

#includ<stdio.h>

#include”prg2.c”

int y; // by default global variable acts under extern storage class

main()

{

extern int x;

extern int add(int,int);

printf(“x=%d y=%d”,x,y);

printf(“Sum is %d\n”,add(x,y));

}

**Source code: prg2.c**

int x=10, y=20;

int add(int a,int b)

{

return a+b;

}

**OUTPUT:**

x=10 y=20

Sum is 30

***Steps to execute extern storage program:***

1. Compile the external program prg2.c without any errors.
2. Compile and execute internal program prg1.c, such that prg1.c links prg2.c during runtime of the program.

Variables

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language −

* Inside a function or a block which is called **local** variables.
* Outside of all functions which is called **global** variables.
* In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

#include <stdio.h>

int main () {

/\* local variable declaration \*/

int a, b;

int c;

/\* actual initialization \*/

a = 10;

b = 20;

c = a + b;

printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

return 0;

}

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

#include <stdio.h>

/\* global variable declaration \*/

int g;

int main () {

/\* local variable declaration \*/

int a, b;

/\* actual initialization \*/

a = 10;

b = 20;

g = a + b;

printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

return 0;

}

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example −

#include <stdio.h>

/\* global variable declaration \*/

int g = 20;

int main () {

/\* local variable declaration \*/

int g = 10;

printf ("value of g = %d\n", g);

return 0;

}

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

value of g = 10

Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example −

#include <stdio.h>

/\* global variable declaration \*/

int a = 20;

int main () {

/\* local variable declaration in main function \*/

int a = 10;

int b = 20;

int c = 0;

printf ("value of a in main() = %d\n", a);

c = sum( a, b);

printf ("value of c in main() = %d\n", c);

return 0;

}

/\* function to add two integers \*/

int sum(int a, int b) {

printf ("value of a in sum() = %d\n", a);

printf ("value of b in sum() = %d\n", b);

return a + b;

}

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

value of a in main() = 10

value of a in sum() = 10

value of b in sum() = 20

value of c in main() = 30

Difference between recursion and iteration

|  |  |
| --- | --- |
| Recursion | Iteration |
| In recursion, function**calls itself** until the base condition is reached. | Iteration means **repetition of process** until the condition fails. |
| In recursive function, only base condition (terminate condition) is specified | Iterative approach involves four steps, initialization , condition, execution and  updation. |
| Recursion keeps code **short and simple** | Iterative approach makes code longer |
| Recursion is slower than iteration due to overhead of maintaining [stack](http://webrewrite.com/write-program-implement-stack-using-array/) | Iteration is faster than recursion |
| Recursion takes more memory than iteration due to overhead of maintaining [stack](http://webrewrite.com/write-program-implement-stack-using-array/) | Iteration takes less memory than recursion |
| Example :  int fact(int x)  {  if(x==0)||(x==1)  return 1;  return (x\*fact(x-1));  } | Example:  int fact(int x)  {  int i,f=1;  for(i=1;i<=x;i++)  f=f\*i;  return f;  } |