**UNIT-III**

**FUNCTIONS & POINTERS**

**What is a Function?**

* 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.
* In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by **{}.**

**There are two types of functions in C programming:**

****

**Standard library functions**

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

* The **printf()** is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the **stdio.h** header file.
* Hence, to use the **printf()** function, we need to include the stdio.h header file using **#include <stdio.h>.**
* The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.

**User-defined function**

* You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

**Passing Values between Functions:**

#include <stdio.h>

int calsum (int x, int y, int z);

int main()

{

int a, b, c, sum;

printf ("Enter any three numbers");

scanf ("%d %d %d", &a, &b, &c);

printf ("Sum = %d\n", sum);

return 0;

}

int calsum (int x, int y, int z)

{

int d;

d=x+y+z;

return (d);

}

**Scope Rule of Functions:**

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.**

**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;

}

**There are basically 4 scope rules:**

* **File Scope:** These variables are usually declared outside of all of the functions and blocks, at the top of the program and can be accessed from any portion of the program. These are also called the global scope variables as they can be globally accessed.
* **Block Scope:** A Block is a set of statements enclosed within left and right braces i.e. ‘{‘ and ‘}’ respectively. Blocks may be nested in C(a block may contain other blocks inside it). A variable declared inside a block is accessible in the block and all inner blocks of that block, but not accessible outside the block. Basically these are local to the blocks in which the variables are defined and are not accessible outside.
* **Function Prototype Scope:** These variables range includes within the function parameter list. The scope of the these variables begins right after the declaration in the function prototype and runs to the end of the declarations list. These scopes don’t include the function definition, but just the function prototype.
* **Function Scope:** A Function scope begins at the opening of the function and ends with the closing of it. Function scope is applicable to labels only. A label declared is used as a target to go to the statement and both goto and label statement must be in the same function.

**Function Declaration and Prototype:**

**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.

**Function Prototype:**

Function prototype in C is used by the compiler to ensure whether the function call matches the return type and the correct number of arguments or parameters with its data type of the called function. A prototype declares the function name, its parameters, and its return type to the rest of the program prior to the function's actual declaration.

**Syntax of function prototype in C programming**

**return\_type function\_name( type argument1, type argument2, ...);**

**Difference between function prototype and function definition in C:**

**The only difference between the function definition and its function prototype is the addition semicolon (;) at the end of prototype declaration.**

But, the parameter identifier could be different in function prototype and function definition because the scope of parameter identifier in a function prototype is limited within the prototype declaration.

Actually, the compiler ignores the name of the parameter list in the function prototype. Having said that, it is good programming practice to include parameter names which increase program clarity.

Let’s consider following **function definition:**

**int area( int length, int breadth ) //function definition**

**{**

 **.... //function body**

**}**

Now, the corresponding **prototype declaration** of the above function is:

**int area( int length, int breadth ); //function prototype**

**Call by Value and Call by Reference:**



**Call by value:**

The call by value method of passing arguments to a function 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. By default, C programming uses call by value to pass arguments.

* In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
* In call by value method, we can not modify the value of the actual parameter by the formal parameter.
* In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
* The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

**Example:**

#include<stdio.h>

void change(int num) {

 printf("Before adding value inside function num=%d \n",num);

 num=num+100;

 printf("After adding value inside function num=%d \n", num);

}

int main() {

 int x=100;

 printf("Before function call x=%d \n", x);

 change(x);//passing value in function

 printf("After function call x=%d \n", x);

return 0;

}

**Output**

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=100

**Call by reference in C**

* In call by reference, the address of the variable is passed into the function call as the actual parameter.
* The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
* In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

**Example:**

#include<stdio.h>

void change(int \*num) {

 printf("Before adding value inside function num=%d \n",\*num);

 (\*num) += 100;

 printf("After adding value inside function num=%d \n", \*num);

}

int main() {

 int x=100;

 printf("Before function call x=%d \n", x);

 change(&x);//passing reference in function

 printf("After function call x=%d \n", x);

return 0;

}

**Output**

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200

**Difference between call by value and call by reference in c**

|  |  |
| --- | --- |
| **Call by value** | **Call by reference** |
| A copy of the value is passed into the function | An address of value is passed into the function |
| Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters. | Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters. |
| Actual and formal arguments are created at the different memory location. | Actual and formal arguments are created at the same memory location |

**Introduction to Pointers:**

* A pointer is a variable that stores the memory address of another variable as its value.
* A pointer variable points to a data type (like int ) of the same type, and is created with the \* operator.
* Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

**The general form of a pointer variable declaration is −**

**type \*var-name;**

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk \* used to declare a pointer is the same asterisk used for multiplication.

**Declaring Pointers (Creating Pointers)**

In c programming language, declaration of pointer variable is similar to the creation of normal variable but the name is prefixed with \* symbol. We use the following syntax to declare a pointer variable...

**datatype \*pointerName ;**

A variable declaration prefixed with \* symbol becomes a pointer variable.

**Example Code**

**int \*ptr ;**

**Advantage of pointer**

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can return multiple values from a function using the pointer.

3) It makes you able to access any memory location in the computer's memory.

**Program:**

**#include <stdio.h>**

**int main()**

**{**

 **int num = 10;**

 **printf("Value of variable num is: %d", num);**

 **/\* To print the address of a variable we use %p**

 **\* format specifier and ampersand (&) sign just**

 **\* before the variable name like &num.**

 **\*/**

 **printf("\nAddress of variable num is: %p", &num);**

 **return 0;**

**}**

**Output:**

**Value of variable num is: 10**

**Address of variable num is: 0x7fff5694dc58**



**Pointer Notation:**

Once a pointer is declared, you can refer to the thing it points to, known as the **target** of the pointer, by **"dereferencing the pointer"**. To do this, use the unary \* operator:

 int \* ptr; // ptr is now a pointer-to-int

 // Notation:

 // ptr refers to the pointer itself

 // \*ptr the dereferenced pointer -- refers now to the TARGET

Suppose that ptr is the above pointer. Suppose it stores the address 1234. Also suppose that the integer stored at address 1234 has the value 99.

 cout << "The pointer is: " << ptr; // prints the pointer

 cout << "The target is: " << \*ptr; // prints the target

 // Output:

 // The pointer is: 1234 // exact printout here may vary

 // The target is: 99

Note: the exact printout of an addres may vary based on the system. Some systems print out addresses in hexadecimal notation (base 16).

**Note:** The notation can be a little confusing.

* + If you see the \* in a *declaration statement*, with a type in front of the \*, a pointer is being declared for the first time.
	+ AFTER that, when you see the \* on the pointer name, you are dereferencing the pointer to get to the target.

Pointers **don't always have valid targets**.

* + A pointer refers to some address in the program's memory space.
	+ A program's memory space is divided up into **segments**
	+ Each memory segment has a different purpose. Some segments are for data storage, but some segments are for other things, and off limits for data storage
	+ If a pointer is pointing into an "out-of-bounds" memory segment, then it does **NOT** have a valid target (for your usage)
	+ **IMPORTANT**: If you try to dereference a pointer that doesn't have a valid target, your program will crash with a *segmentation fault* error. This means you tried to go into an off-limits segment

Don't dereference a pointer until you know it has been initialized to a valid target!

**Back to function Call:**

* The callback is basically any executable code that is passed as an argument to other code, that is expected to call back or execute the argument at a given time.
* Callback function technique is useful where you need to write a function that’s able to perform several different functions at a point or can perform a function specified only by the caller. This technique requires to pass a pointer-to-function to another routine, which calls back the user function to perform some task.

**Example:**

#include<stdio.h>

void my\_function() {

 printf("This is a normal function.");

}

void my\_callback\_function(void (\*ptr)()) {

 printf("This is callback function.

");

 (\*ptr)(); //calling the callback function

}

main() {

 void (\*ptr)() = &my\_function;

 my\_callback\_function(ptr);

}

**Output**

**This is callback function.**

**This is a normal function.**

**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.

**void recursion() {**

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

**}**

**int main() {**

 **recursion();**

**}**

**Flowchart of Recursion:**

****

**Types of Recursion in C**

There are two types of recursion in the C language.

* **Direct Recursion**
* **Indirect Recursion**

1. Direct Recursion

Direct recursion in C occurs when a function calls itself directly from inside. Such functions are also called direct recursive functions.

Following is the structure of direct recursion.

**function\_01()**

**{**

 **//some code**

 **function\_01();**

 **//some code**

**}**

**2. Indirect Recursion**

Indirect recursion in C occurs when a function calls another function and if this function calls the first function again. Such functions are also called indirect recursive functions.

**Following is the structure of indirect recursion.**

**function\_01()**

**{**

 **//some code**

 **function\_02();**

**}**

**function\_02()**

**{**

 **//some code**

 **function\_01();**

**}**

**Recursion and Stack:**

1. "Recursion" is technique of solving any problem by calling same function again and again until some breaking (base) condition where recursion stops and it starts calculating the solution from there on. For **eg. calculating factorial of a given number**
2. Thus in recursion last function called needs to be completed first.
3. Now Stack is a LIFO data structure i.e. ( Last In First Out) and hence it is used to implement recursion.
4. The High level Programming languages, such as Pascal , C etc. that provides support for recursion use stack for book keeping.
5. In each recursive call, there is need to save the
	1. current values of parameters,
	2. local variables and
	3. the return address (the address where the control has to return from the call).
6. Also, as a function calls to another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.



7. Recursion is extremely useful and extensively used because many problems are elegantly specified or solved in a recursive way.

8. The example of recursion as an application of stack is keeping books inside the drawer and the removing each book recursively.