

C - FUNCTIONS

http://www.tutorialspoint.com/cprogramming/c_functions.htm

Copyright © tutorialspoint.com

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 usually is so 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, function **strcat** to concatenate two strings, function **memcpy** to copy one memory location to another location and many more functions.

A function is known with various names like 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 language 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.

Example:

Following is the source code for a function called **max**. This function takes two parameters num1 and num2 and returns the maximum 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, following is the function declaration:

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

Parameter names are not important in function declaration only their type is required, so following is also 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.

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

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

```
}
```

I kept max function along with main function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

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.

The 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 that arguments can be passed to a function:

Call Type	Description
Call by value	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.
Call by reference	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, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max function used the same method

Loading [MathJax]/jax/output/HTML-CSS/fonts/TeX/fontdata.js

C - POINTERS

http://www.tutorialspoint.com/cprogramming/c_pointers.htm

Copyright © tutorialspoint.com

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, 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. Consider the following example, which will print the address of the variables defined:

```
#include <stdio.h>

int main ()
{
    int var1;
    char var2[10];

    printf("Address of var1 variable: %x\n", &var1 );
    printf("Address of var2 variable: %x\n", &var2 );

    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

So you understood what is memory address and how to access it, so base of the concept is over. Now let us see what is a pointer.

What Are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. 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 * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch;    /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. *a* we define a pointer variable *b* assign the address of a variable to a pointer and *c* finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following

example makes use of these operations:

```
#include <stdio.h>

int main ()
{
    int var = 20; /* actual variable declaration */
    int *ip;      /* pointer variable declaration */

    ip = &var; /* store address of var in pointer variable*/

    printf("Address of var variable: %x\n", &var );

    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip );

    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );

    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>

int main ()
{
    int *ptr = NULL;

    printf("The value of ptr is : %x\n", ptr );

    return 0;
}
```

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

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null *zero* value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

C Pointers in Detail:

Pointers have many but easy concepts and they are very important to C programming. There are following few important pointer concepts which should be clear to a C programmer:

Concept	Description
C - Pointer arithmetic	There are four arithmetic operators that can be used on pointers: ++, --, +, -
C - Array of pointers	You can define arrays to hold a number of pointers.
C - Pointer to pointer	C allows you to have pointer on a pointer and so on.
Passing pointers to functions in C	Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.
Return pointer from functions in C	C allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.

Loading [MathJax]/jax/output/HTML-CSS/jax.js