



OUR M365 CUSTOMERS RESOLVE BREACHES 200%

Request a demo to see how we can help.

C Programming Language Cheat Sheet



30 min read

By Vineet Choudhary



C Programming

1 What is C?

- C is a programming language developed at AT & T's Bell Laboratories of USA in 1972 by Dennis Ritchie.
- Any programming Language can be divided in to two categories.
 - Problem oriented (High level language)
 - Machine oriented (Low level language)

But C is considered as a **Middle Level Language** .

- C is **modular , portable , reusable** .

1.2 Feature of C Program

- **Structured language**
 - It has the ability to divide and hide all the information and instruction.
 - Code can be partitioned in C using functions or code block.
 - C is a well structured language compare to other.
- **General purpose language**
 - Make it ideal language for system programming.
 - It can also be used for business and scientific application.
 - ANSI established a standard for c in 1983.
 - The ability of c is to manipulate bits,byte and addresses.
 - It is adopted in later 1990.



Share

Tweet

Send

- Portability is the ability to port or use the software written .
- One computer C program can be reused.
- By modification or no modification.
- **Code Re-usability & Ability to customize and extend**
 - A programmer can easily create his own function
 - It can be used repeatedly in different application
 - C program basically collection of function
 - The function are supported by 'c' library
 - Function can be added to 'c' library continuously
- **Limited Number of Key Word**
 - There are only 32 keywords in 'C'
 - 27 keywords are given by ritchie
 - 5 is added by ANSI
 - The strength of 'C' is lies in its in-built function
 - Unix system provides as large number of C function
 - Some function are used in operation .
 - Other are for specialized in their application

1.3 C program structure

```

pre-processor directives
global declarations

main()
{
    local variable deceleration
    statement sequences
    function invoking
}

```

1.4 C Keywords

Keywords are the words whose meaning has already been explained to the C compiler. There are only 32 keywords available in C. The keywords are also called 'Reserved words'.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

1.5 C Character Set

A character denotes any alphabet, digit or special symbol used to represent information. Following are the valid alphabets, numbers and special symbols allowed in C.

- **Alphabets** - A, B,, Y, Z a, b,, y, z
- **Digits** - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Special symbols** - ~ ' ! @ # % ^ & * () _ - + = | \ { } [] : ; " ' < > , . ? /

1.6 Rules for Writing, Compiling and Executing the C program

- C is case sensitive means variable named "COUNTER" is different from a variable named "counter".
- All keywords are lowercased.
- Keywords cannot be used for any other purpose (like variable names).
- Every C statement must end with a ; . Thus ; acts as a statement terminator.
- First character must be an alphabet or underscore, no special symbol other than an underscore, no commas or blank spaces are allowed within a variable, constant or keyword.
- Blank spaces may be inserted between two words to improve the readability of the statement. However, no blank spaces are allowed within a variable, constant or keyword.
- Variable must be declared before it is used in the program.
- File should have the extension .c
- Program needs to be compiled before execution.

1.7 Data types & Placeholders

- C has 5 basic built-in data types.
- Data type defines a set of values that a variable can store along with a set of operations that can be performed on it.
- A variable takes different values at different times.
- General form for declaring a variable is:
type name;
- An example for using variables comes below:

```
#include<stdio.h>
main()
{
    int sum;
    sum=12;
    sum=sum+5;
    printf("Sum is %d",sum);
}
```

printf function will print the following:

Sum is 17

In fact %d is the placeholder for integer variable value that its name comes after double quotes.

- Common data types are:
 - **int** - integer
 - **char** - character

[Share](#)
[Tweet](#)
[Send](#)

- `long` – long integer
- `float` – float number
- `double` – long float
- Other placeholders are:

Placeholders	Format
<code>%c</code>	Character
<code>%d</code>	Signed decimal integer
<code>%i</code>	Signed decimal integer
<code>%e</code>	Scientific notation[e]
<code>%E</code>	Scientific notation[E]
<code>%f</code>	Decimal floating point
<code>%o</code>	unsigned octal
<code>%s</code>	String of character
<code>%u</code>	unsigned decimal integer
<code>%x</code>	unsigned Hexadecimal (lower)
<code>%X</code>	unsigned Hexadecimal (upper)
<code>%p</code>	display a pointer
<code>%%</code>	print a %

1.8 Control characters (Escape sequences)

Certain non printing characters as well as the backslash (`\`) and the apostrophe (`'`), can be expressed in terms of escape sequence.

- `\a` – Bell
- `\n` – New line
- `\r` – Carriage return
- `\b` – Backspace
- `\f` – Formfeed
- `\t` – Horizontal tab
- `\"` – Quotation mark
- `\v` – Vertical tab
- `\'` – Apostrophe
- `\\` – Backslash
- `\?` – Question mark
- `\0` – Null

1.9 Receiving input values from keyboard

`scanf` function used to receiving input from keyboard.

General form of `scanf` function is :

```
scanf("Format string", &variable, &variable, ... );
```

Format string contains placeholders for variables that we intend to receive from keyboard. A `&` sign comes before each variable name that comes in variable listing. Character strings are exceptions from this rule. They will not come with this sign before them.

Note: You are not allowed to insert any additional characters in format string other than placeholders and some special characters. Entering even a space or other undesired character will cause your program to work incorrectly and the results will be unexpected. So make sure you just insert placeholder characters in scanf format string. The following example receives multiple variables from keyboard.

```
float a;
int n;
scanf("%d%f", &n, &a);
```

Pay attention that scanf function has no error checking capabilities built in it. Programmer is responsible for validating input data (type, range etc.) and preventing errors

2. Expression & Operators Precedence

Following table summarizes the rules for precedence and associativity of all operators, including those that we have not yet discussed. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, *, /, and % all have the same precedence, which is higher than that of binary + and -. The ``operator'' () refers to function call. The operators -> and . are used to access members of structures;

DESCRIPTION	OPERATORS	ASSOCIATIVITY
Function Expression	()	Left to Right
Array Expression	[]	Left to Right
Structure Operator	->	Left to Right
Structure Operator	.	Left to Right
Unary minus	-	Right to Left
Increment/Decrement	++, --	Right to Left
One's compliment	~	Right to Left
Negation	!	Right to Left
Address of	&	Right to Left
Value of address	*`	Right to Left
Type cast	(type)	Right to Left
Size in bytes	sizeof	Right to Left
Multiplication	*`	Left to Right
Division	/	Left to Right
Modulus	%	Left to Right
Addition	+	Left to Right
Subtraction	-	Left to Right
Left shift	<<	Left to Right
Right shift	>>	Left to Right
Less than	<	Left to Right
Less than or equal to	<=	Left to Right
Greater than	>	Left to Right
Greater than or equal to	>=	Left to Right
Equal to	==	Left to Right
Not equal to	!=	Left to Right

[Share](#)
[Tweet](#)
[Send](#)

Bitwise exclusive OR	<code>^</code>	Left to Right
Bitwise inclusive OR	<code> </code>	Left to Right
Logical AND	<code>&&</code>	Left to Right
Logical OR	<code> </code>	Left to Right
Conditional	<code>?:</code>	Right to Left
Assignment	<code>=, *=, /=, %=, +=, -=, &=, ^=, =, <=<, >=></code>	Right to Left
Comma	<code>,</code>	Right to Left

Unary `&`, `+`, `-`, and `*` have higher precedence than the binary forms.

3. The Decision Control Structure

C has three major decision making instructions—the `if` statement, the `if-else` statement, and the `switch` statement.

3.1 The if Statement

C uses the keyword `if` to implement the decision control instruction. The general form of `if` statement looks like this:

```
//for single statement
if(condition)
    statement;

//for multiple statement
if(condition)
{
    block of statement;
}
```

The more general form is as follows:

```
//for single statement
if(expression)
    statement;

//for multiple statement
if(expression)
{
    block of statement;
}
```

Here the expression can be any valid expression including a relational expression. We can even use arithmetic expressions in the `if` statement. For example all the following `if` statements are valid

```
if (3 + 2 % 5)
    printf("This works");
```

The expression `(3 + 2 % 5)` evaluates to 5 and since 5 is non-zero it is considered to be true. Hence the `printf("This works");` gets executed.

3.2 The if-else Statement

The if statement by itself will execute a single statement, or a group of statements, when the expression following if evaluates to true. It does nothing when the expression evaluates to false. Can we execute one group of statements if the expression evaluates to true and another group of statements if the expression evaluates to false? Of course! This is what is the purpose of the else statement that is demonstrated as

```
if (expression)
{
    block of statement;
}
else
    statement;
```

Note

- The group of statements after the `if` upto and not including the `else` is called an `if block`. Similarly, the statements after the `else` form the `else block`.
- Notice that the `else` is written exactly below the `if`. The statements in the `if block` and those in the `else block` have been indented to the right.
- Had there been only one statement to be executed in the `if block` and only one statement in the `else block` we could have dropped the pair of braces.
- As with the `if` statement, the default scope of `else` is also the statement immediately after the `else`. To override this default scope a pair of braces as shown in the above "*Multiple Statements within if*" must be used.

3.3 Nested if-elses

If we write an entire `if-else` construct within either the body of the `if` statement or the body of an `else` statement. This is called "*nesting*" of `ifs`. This is demonstrated as -

```
if (expression1)
    statement;
else
{
    if (expression2)
        statement;
    else
    {
        block of statement;
    }
}
```

3.4 The if-else Ladder/else-if Clause

The `else-if` is the most general way of writing a multi-way decision.

```
if(expression1)
    statement;
else if(expression2)
    statement;
else if(expression3)
    statement;
else if(expression4)
{
    block of statement;
}
else
    -----
```

[Share](#)
[Tweet](#)
[Send](#)

The *expressions* are evaluated in order; if an expression is true, the "statement" or "block of statement" associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group of them in braces. The last `else` part handles the "none of the above" or default case where none of the other conditions is satisfied.

3.5 Switch Statements or Control Statements

The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of `constant` integer values, and branches accordingly. The `switch` statement that allows us to make a decision from the number of choices is called a `switch`, or more correctly a `switch-case-default`, since these three keywords go together to make up the `switch` statement.

```
switch (expression)
{
    case constant-expression:
        statement1;
        statement2;
        break;
    case constant-expression:
        statement;
        break;
    ...
    default:
        statement;
}
```

- In `switch...case` command, each `case` acts like a simple label. A label determines a point in program which execution must continue from there. Switch statement will choose one of `case` sections or labels from where the execution of the program will continue. The program will continue execution until it reaches `break` command.
- `break` statements have vital rule in `switch` structure. If you remove these statements, program execution will continue to next case sections and all remaining case sections until the end of `switch` block will be executed (while most of the time we just want one `case` section to be run).
- `default` section will be executed if none of the case sections match switch comparison.

3.6 switch Versus if-else Ladder

There are some things that you simply cannot do with a `switch`. These are:

- A float expression cannot be tested using a switch
- Cases can never have variable expressions (for example it is wrong to say `case a + 3 :`)
- Multiple cases cannot use same expressions.

4. The Loop Control Structure

Sometimes we want some part of our code to be executed more than once. We can either repeat the code in our program or use loops instead. It is obvious that if for example we need to execute some part of code for a hundred times it is not practical to repeat the code. Alternatively we can use our repeating code inside a loop.

There are three methods for `while` and `do-while` which we can repeat a part of a program

[Share](#)
[Tweet](#)
[Send](#)

4.1 while loop

while loop is constructed of a condition or expression and a single command or a block of commands that must run in a loop.

```
//for single statement
while(expression)
    statement;

//for multiple statement
while(expression)
{
    block of statement
}
```

The statements within the `while` loop would keep on getting executed till the condition being tested remains true. When the condition becomes false, the control passes to the first statement that follows the body of the while loop.

The general form of `while` is as shown below:

```
initialise loop counter;
while (test loopcounter using a condition)
{
    do this;
    and this;
    increment loopcounter;
}
```

4.2 for loop

`for` loop is something similar to `while` loop but it is more complex. `for` loop is constructed from a control statement that determines how many times the loop will run and a command section. Command section is either a single command or a block of commands.

```
//for single statement
for(control statement)
    statement;

//for multiple statement
for(control statement)
{
    block of statement
}
```

Control statement itself has three parts:

```
for ( initialization; test condition; run every time command )
```

- **Initialization** part is performed only once at `for` loop start. We can initialize a loop variable here.
- **Test condition** is the most important part of the loop. Loop will continue to run if this condition is valid (true). If the condition becomes invalid (false) then the loop will terminate.
- **Run every time command** section will be performed in every loop cycle. We use this part to reach the final condition for terminating the loop. **For example** we can increase or decrease loop variable's value in a way that after specified number of cycles the loop condition becomes invalid and `for` loop can terminate.

4.3 do-while loop

The `while` and `for` loops test the termination condition at the top. By contrast, the third loop in C, the `do-while`, tests at the bottom after making each pass through the loop body; the body is always executed at least once.

The syntax of the `do` is

```
do
{
    statements;
}while (expression);
```

The statement is executed, then `expression` is evaluated. If it is true, statement is evaluated again, and so on. When the expression becomes false, the loop terminates. Experience shows that `do-while` is much less used than `while` and `for`. A `do-while` loop is used to ensure that the statements within the loop are executed at least once.

4.4 Break and Continue statement

We used `break` statement in `switch ... case` structures in previously. We can also use "break" statement inside loops to terminate a loop and exit it (with a specific condition).

In above example loop execution continues until either `num ≥ 20` or entered score is negative.

```
while (num<20)
{
    printf("Enter score : ");
    scanf("%d",&scores[num]);
    if(scores[num]<0)
        break;
}
```

`Continue` statement can be used in loops. Like `break` command `continue` changes flow of a program. It does not terminate the loop however. It just skips the rest of current iteration of the loop and returns to starting point of the loop.

```
#include<stdio.h>
main()
{
    while((ch=getchar())≠'\n')
    {
        if(ch=='.')
            continue;
        putchar(ch);
    }
}
```

In above example, program accepts all input but omits the `'.'` character from it. The text will be echoed as you enter it but the main output will be printed after you press the enter key (which is equal to inserting a `"\n"` character) is pressed. As we told earlier this is because `getchar()` function is a buffered input function.

4.5 Goto and labels

C provides the infinitely-abusable `goto` statement, and `labels` to branch to. Formally, the `goto` statement is never necessary, and in practice it is almost always easy to write code without it. We have not used `goto` in this book.

Nevertheless, there are a few situations where `gotos` may find a place. The most common is to abandon processing in

directly since it only exits from the innermost loop. Thus:

```
for ( ... )
{
    for ( ... )
    {
        ...
        if (disaster)
        {
            goto error;
        }
    }
}
...
error:
/* clean up the mess */
```

This organization is handy if the error-handling code is non-trivial, and if errors can occur in several places.

A `label` has the same form as a variable name, and is followed by a colon. It can be attached to any statement in the same function as the `goto`. The scope of a label is the entire function.



Note - By uses of `goto`, programs become unreliable, unreadable, and hard to debug. And yet many programmers find `goto` seductive.

5. Arrays

Arrays are structures that hold multiple variables of the same data type. The first element in the array is numbered 0, so the last element is 1 less than the size of the array. An array is also known as a subscripted variable. Before using an array its type and dimension must be declared.

5.1 Array Declaration

Like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want.

```
int marks[30] ;
```

Here, `int` specifies the type of the variable, just as it does with ordinary variables and the word `marks` specifies the name of the variable. The `[30]` however is new. The number 30 tells how many elements of the type `int` will be in our array. This number is often called the "*dimension*" of the array. The bracket `([])` tells the compiler that we are dealing with an array.

```
int num[6] = { 2, 4, 12, 5, 45, 5 } ;  
int n[] = { 2, 4, 12, 5, 45, 5 } ;  
float press[] = { 12.3, 34.2 -23.4, -11.3 } ;
```

5.2 Accessing Elements of an Array

Once an array is declared, let us see how individual elements in the array can be referred. This is done with subscript, the number in the brackets following the array name. This number specifies the element's position in the array. All the array elements are numbered, starting with 0. Thus, marks [2] is not the second element of the array, but the third.

```
int valueOfThirdElement = marks[2];
```

5.3 Entering Data into an Array

Here is the section of code that places data into an array:

```
for(i = 0; i ≤ 29; i++)  
{  
    printf("\nEnter marks ");  
    scanf("%d", &marks[i]);  
}
```

The `for` loop causes the process of asking for and receiving a student's marks from the user to be repeated 30 times. The first time through the loop, `i` has a value 0, so the `scanf()` function will cause the value typed to be stored in the array element `marks[0]`, the first element of the array. This process will be repeated until `i` becomes 29. This is last time through the loop, which is a good thing, because there is no array element like `marks[30]`.

In `scanf()` function, we have used the "address of" operator (&) on the element `marks[i]` of the array. In so doing, we are passing the address of this particular array element to the `scanf()` function, rather than its value; which is what `scanf()` requires.

5.4 Reading Data from an Array

The balance of the program reads the data back out of the array and uses it to calculate the average. The for loop is much the same, but now the body of the loop causes each student's marks to be added to a running total stored in a variable called `sum`. When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

```
for ( i = 0 ; i ≤ 29 ; i++ )  
    sum = sum + marks[i] ;  
avg = sum / 30 ;  
printf ( "\nAverage marks = %d", avg ) ;
```

5.5 Example

Let us try to write a program to find average marks obtained by a class of 30 students in a test.

```
#include<stdio.h>
main()
{
    int avg, i, sum=0;
    int marks[30] ; /*array declaration */
    for ( i = 0 ; i ≤ 29 ; i++ )
    {
        printf ( "\nEnter marks " ) ;
        scanf ( "%d", &marks[i] ) ; /* store data in array */
    }
    for ( i = 0 ; i ≤ 29 ; i++ )
        sum = sum + marks[i] ; /* read data from an array*/
    avg = sum / 30 ;
    printf ( "\nAverage marks = %d", avg ) ;
}
```

6 Strings

6.1 What is String?

Strings are arrays of characters. Each member of array contains one of characters in the string.

Example

```
#include<stdio.h>
main()
{
    char name[20];
    printf("Enter your name : ");
    scanf("%s",name);
    printf("Hello, %s , how are you ?\n",name);
}
```

Output Results:

```
Enter your name : Vineet
Hello, Vineet, how are you ?
```

If user enters "Vineet" then the first member of array will contain 'V', second cell will contain 'i' and so on. C determines end of a string by a zero value character. We call this character as `NULL` character and show it with `\0` character. (It's only one character and its value is 0, however we show it with two characters to remember it is a character type, not an integer).

Equally we can make that string by assigning character values to each member.

```
name[0]='B';
name[1]='r';
name[2]='i';
name[3]='a';
name[4]='n';
name[5]='\0';
```

As we saw in above example placeholder for string variables is `%s`. Also we will not use a `&` sign for receiving string values.

6.2 Point to Note

While entering the string using `scanf()` we must be cautious about two things:

- The length of the string should not exceed the dimension of the character array. This is because the C compiler doesn't perform bounds checking on character arrays.
- `scanf()` is not capable of receiving multi-word strings. Therefore names such as "Vineet Choudhary" would be unacceptable. The way to get around this limitation is by using the function `gets()`. The usage of functions `gets()` and its counter part `puts()` is shown below.

```
#include<stdio.h>
main( )
{
    char name[25] ;
    printf ("Enter your full name ") ;
    gets (name) ;
    puts ("Hello!") ;
    puts (name) ;
}
```

And here is the output...

```
Enter your name Vineet Choudhary
Hello!
Vineet Choudhary
```

The program and the output are self-explanatory except for the fact that, `puts()` can display only one string at a time (hence the use of two `puts()` in the program above). Also, on displaying a string, unlike `printf()`, `puts()` places the cursor on the next line. Though `gets()` is capable of receiving only one string at a time, the plus point with `gets()` is that it can receive a multi-word string.

If we are prepared to take the trouble we can make `scanf()` accept multi-word strings by writing it in this manner:

```
char name[25] ;
printf ("Enter your full name ") ;
scanf ("%^\n)s", name) ;
```

Though workable this is the best of the ways to call a function, you would agree.

6.3 Standard Library String Functions

With every C compiler a large set of useful string handling library functions are provided in `string.h` file.

- `strlen` - Finds length of a string
- `strlwr` - Converts a string to lowercase
- `strupr` - Converts a string to uppercase
- `strcat` - Appends one string at the end of another
- `strncat` - Appends first n characters of a string at the end of another
- `strcpy` - Copies a string into another
- `strncpy` - Copies first n characters of one string into another

- `strncmp` - Compares first n characters of two strings
- `strncmpi` - Compares two strings without regard to case ("i" denotes that this function ignores case)
- `stricmp` - Compares two strings without regard to case (identical to `strncmpi`)
- `strnicmp` - Compares first n characters of two strings without regard to case
- `strdup` - Duplicates a string
- `strchr` - Finds first occurrence of a given character in a string
- `strrchr` - Finds last occurrence of a given character in a string
- `strstr` - Finds first occurrence of a given string in another string
- `strset` - Sets all characters of string to a given character
- `strnset` - Sets first n characters of a string to a given character
- `strrev` - Reverses string

7. Functions

7.1 What is a Function?

A function is combined of a block of code that can be called or used anywhere in the program by calling the name. Body of a function starts with `{` and ends with `}`. This is similar to the main function. Example below shows how we can write a simple function.

```
#include<stdio.h>

/*Function prototypes*/
myfunc();

main()
{
    myfunc();
}

/*Function Defination*/
myfunc()
{
    printf("Hello, this is a test\n");
}
```

In above example we have put a section of program in a separate function. Function body can be very complex though. After creating a function we can call it using its name. Functions can also call each other. A function can even call itself.

By the way pay attention to `function prototypes` section. In some C compilers we are needed to introduce the functions we are creating in a program above the program. Introducing a function is being called `function prototype`.

Note:

- Any C program contains at least one function.

- If a C program contains more than one function, then one (and only one) of these functions must be `main()`, because program execution always begins with `main()`.
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in `main()`.
- After each function has done its thing, control returns to `main()`. When `main()` runs out of function calls, the program ends.

7.2 Why Use Functions

Why write separate functions at all? Why not squeeze the entire logic into one function, `main()`? Two reasons:

- Writing functions avoids rewriting the same code over and over.
- Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

What is the moral of the story? Don't try to cram the entire logic in one function. It is a very bad style of programming. Instead, break a program into small units and write functions for each of these isolated subdivisions. Don't hesitate to write functions that are called only once. What is important is that these functions perform some logically isolated task.

7.3 Function arguments

Functions are able to accept input parameters in the form of variables. These input parameter variables can then be used in function body.

```
#include<stdio.h>
/* use function prototypes */
sayhello(int count);
main()
{
    sayhello(4);
}

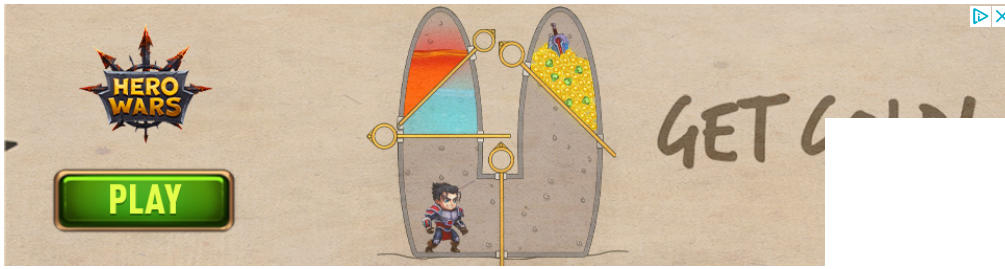
sayhello(int count)
{
    int c;
    for(c=0; c<count; c++)
        printf("Hello\n");
}
```

In above example we have called `sayhello()` function with the parameter `4`. This function receives an input value and assigns it to `count` variable before starting execution of function body. `sayhello()` function will then print a hello message `count` times on the screen.

7.4 Function return values

In mathematics we generally expect a function to return a value. It may or may not accept arguments but it always returns a value.

This `return` value has a type as other values in C. It can be `int` , `float` , `char` or anything else. Type of this `return` value determines type of your function.



Default type of function is `int` or integer. If you do not indicate type of function that you use, it will be of type `int` . As we told earlier every function must return a value. We do this with `return` command.

```
int sum()
{
    int a,b,c;
    a=1;
    b=4;
    c=a+b;
    return c;
}
```

Above function returns the value of variable `c` as the return value of function. We can also use expressions in return command. **For example** we can replace two last lines of function with `return a+b;` If you forget to return a value in a function you will get a warning message in most of C compilers. This message will warn you that your function must return a value. Warnings do not stop program execution but errors stop it.

In our previous examples we did not return any value in our functions. For example you must return a value in `main()` function.

```
main()
{
    .
    .
    .
    return 0;
}
```

Default return value for an `int` type function is 0. If you do not insert `return 0` or any other value in your `main()` function a 0 value will be returned automatically. If you are planning to return an `int` value in your function, it is seriously preferred to mention the return value in your function header and make.

7.5 void return value

There is another `void` type of function in C language. Void type function is a function that does not `return` a value. You can define a function that does not need a return value as `void` .

```
void test ()
{
    /* fuction code comes here but no return value */
}
```

```
a=test();
```

Using above command will generate an error.

7.6 Recursive Function

In C, it is possible for the functions to call themselves. A function is called **recursive** if a statement within the body of a function calls the same function.

Following is the recursive function to calculate the factorial value.

```
#include<stdio.h>
int rec(int);
main()
{
    int a, fact ;
    printf("\nEnter any number ");
    scanf ("%d", &a);
    fact = rec(a);
    printf ("Factorial value = %d", fact);
}
int rec (int x)
{
    int f;
    if (x == 1)
        return (1);
    else
        f = x * rec (x - 1) ;
    return (f) ;
}
```

Output-

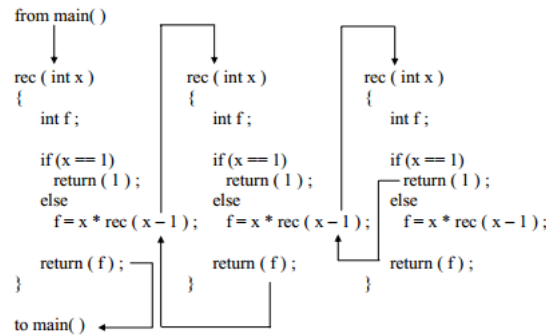
```
Enter any number 5
Factorial value = 120
```

Let us understand this recursive factorial function.

what happens is,

```
rec(5) returns(5 * rec(4),
which returns (4 * rec(3),
which returns (3 * rec(2),
which returns (2 * rec(1),
which returns (1))))
```

Foxxed? Well, that is recursion for you in its simplest garbs. I hope you agree that it's difficult to visualize how the control flows from one function call to another. Possibly Following figure would make things a bit clearer.



7.7 Multiple Parameters

In fact you can use more than one argument in a function. Following example will show you how you can do this.

```

#include<stdio.h>
int min(int a,int b);
main()
{
    int m;
    m=min(3,6);
    printf("Minimum is %d",m);
    return 0;
}
int min(int a,int b)
{
    if(a<b)
        return a;
    else
        return b;
}
  
```

As you see you can add your variables to arguments list easily.

7.8 Call by value

C programming language function calls, use call by value method. Let's see an example to understand this subject better.

```

#include<stdio.h>
void test(int a);
main()
{
    int m;
    m=2;
    printf("\nM is %d",m);
    test(m);
    printf("\nM is %d\n",m);
    return 0;
}
void test(int a)
{
    a=5;
}
  
```

In `main()` function, we have declared a variable `m`. We have assigned value 2 to `m`. We want to see if function call is able to change value of `m` as we have modified value of incoming parameter inside `test()` function.

Program output result is:

Share

Tweet

Send

```
M is 2
M is 2
```

So you see function call has not changed the value of argument. This is because function-calling method only sends value of variable `m` to the function and it does not send variable itself. Actually it places value of variable `m` in a memory location called stack and then function retrieves the value without having access to main variable itself. This is why we call this method of calling "call by value".

7.9 Call by reference

There is another method of sending variables being called "Call by reference". This second method enables function to modify value of argument variables used in function call.

We will first see an example and then we will describe it.

```
#include<stdio.h>
void test(int *ptr);
main()
{
    int m;
    m=2;
    printf("\nM is %d",m);
    test(&m);
    printf("\nM is %d\n",m);
    return 0;
}
void test(int *ptr)
{
    printf("\nModifying the value inside memory address %ld", &m);
    *ptr=5;
}
```

To be able to modify the value of a variable used as an argument in a function, function needs to know memory address of the variable (where exactly the variable is situated in memory).

In C programming language `&` operator gives the address at which the variable is stored. For example if `m` is a variable of type `int` then `&m` will give us the starting memory address of our variable. We call this resulting address a **pointer**.

```
ptr=&m;
```

In above command `ptr` variable will contain memory address of variable `m`. This method is used in some of standard functions in C. For example `scanf` function uses this method to be able to receive values from console keyboard and put it in a variable. In fact it places received value in memory location of the variable used in function. Now we understand the reason why we add `&` sign before variable names in `scanf` variables.

```
scanf("%d",&a);
```

Now that we have memory address of variable we must use an operator that enables us to assign a value or access to value stored in that address.

As we told, we can find address of variable `a` using `&` operator.

```
ptr=&a;
```

```
val=*ptr; /* finding the value ptr points to */
```

We can even modify the value inside the address:

```
*ptr=5;
```

Let's look at our example again. We have passed pointer (memory address) to function. Now function is able to modify value stored inside variable. If you run the program, you will get these results:

```
M is 2
Modifying the value inside memory address 2293620
M is 5
```

So you see this time we have changed value of an argument variable inside the called function (by modifying the value inside the memory location of our main variable).

8. Pointers

8.1 What is a Pointer?

A pointer is a variable that contains the address of a variable. The main thing is that once you can talk about the address of a variable, you'll then be able to goto that address and retrieve the data stored in it.

8.2 C Pointer Syntax

Pointers require a bit of new syntax because when you have a pointer, you need the ability to both request the memory location it stores and the value stored at that memory location. Moreover, since pointers are some what special, you need to tell the compiler when you declare your pointer variable that the variable is a pointer, and tell the compiler what type of memory it points to.

The pointer declaration looks like this:

```
<variable_type> *<name>;
```

For example, you could declare a pointer that stores the address of an integer with the following syntax:

```
int *points_to_integer;
```

Notice the use of the *. This is the key to declaring a pointer; if you add it directly before the variable name, it will declare the variable to be a pointer.

8.3 Pointer Notation

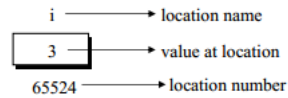
Consider the declaration,

```
int i = 3;
```

This declaration tells the C compiler to:

- Reserve space in memory to hold the integer value.
- Associate the name `i` with this memory location.
- Store the value 3 at this location.

We may represent `i`'s location in memory by the following memory map.



We see that the computer has selected memory location 65524 as the place to store the value 3. The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is, `i`'s address in memory is a number. We can print this address number through the following program:

```
#include<stdio.h>
main()
{
    int i = 3 ;
    printf("\nAddress of i = %u",&i);
    printf("\nValue of i = %d",i);
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
```

Look at the first `printf()` statement carefully. `&` used in this statement is C's "address of" operator. The expression `&i` returns the address of the variable `i`, which in this case happens to be 65524. Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using `%u`, which is a format specifier for printing an unsigned integer. We have been using the `&` operator all the time in the `scanf()` statement.

The other pointer operator available in C is `*`, called "value at address" operator. It gives the value stored at a particular address. The "value at address" operator is also called "indirection" operator.



Observe carefully the output of the following program:

```
#include<stdio.h>
main()
{
    int i = 3 ;
    printf("\nAddress of i = %u",&i);
    printf("\nValue of i = %d",i);
}
```

Share

Tweet

Send

```
printf("\nValue of i = %d",*(&i));
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
Value of i = 3
```

Note that printing the value of `*(&i)` is same as printing the value of `i`. The expression `&i` gives the address of the variable `i`. This address can be collected in a variable, by saying,

```
j = &i ;
```

But remember that `j` is not an ordinary variable like any other integer variable. It is a variable that contains the address of other variable (`i` in this case). Since `j` is a variable the compiler must provide it space in the memory. Once again, the following memory map would illustrate the contents of `i` and `j`.



As you can see, `i`'s value is 3 and `j`'s value is `i`'s address. But wait, we can't use `j` in a program without declaring it. And since `j` is a variable that contains the address of `i`, it is declared as,

```
int *j ;
```

This declaration tells the compiler that `j` will be used to store the address of an integer value. In other words `j` points to an integer. How do we justify the usage of `*` in the declaration,

```
int *j ;
```

Let us go by the meaning of `*`. It stands for "value at address". Thus, `int *j` would mean, the value at the address contained in `j` is an int.

8.4 Example

Here is a program that demonstrates the relationships we have been discussing.

```
#include<stdio.h>
main( )
{
    int i = 3 ;
    int *j ;
    j = &i ;
    printf ("\nAddress of i = %u",&i) ;
    printf ("\nAddress of i = %u",j) ;
    printf ("\nAddress of j = %u",&j) ;
    printf ("\nValue of j = %u",j) ;
    printf ("\nValue of i = %d",i) ;
    printf ("\nValue of i = %d",*(&i)) ;
    printf ("\nValue of i = %d",*j) ;
}
```

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3
```

9. Structures

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

9.1 Declaring a Structure

The general form of a structure declaration statement is given below:

```
struct <structure name>
{
    structure element 1;
    structure element 2;
    structure element 3;
    .....
    .....
    structure element n;
};
```

Once the new structure data type has been defined one or more variables can be declared to be of that type.

For example the variables b1, b2, b3 can be declared to be of the type struct book,

```
struct book
{
    char name;
    float price;
    int pages;
};
```

as,

```
struct book b1, b2, b3 ;
```

This statement sets aside space in memory. It makes available space to hold all the elements in the structure—in this case, 7 bytes — one for name, four for price and two for pages. These bytes are always in adjacent memory locations.

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initiate arrays.

```
struct book
{
    char name[10];
    float price;
    int pages;
};
```



```
struct book b1 = { "Basic", 130.00, 550 } ;  
struct book b2 = { "Physics", 150.80, 800 } ;
```

9.2 Accessing Structure Elements

In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. So to refer to `pages` of the structure defined in `book` structure we have to use,

```
b1.pages
```

Similarly, to refer to `price` we would use,

```
b1.price
```

Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

9.3 Example

The following example illustrates the use of this data type.

```
#include<stdio.h>  
main()  
{  
    struct book  
    {  
        char name;  
        float price;  
        int pages;  
    };  
    struct book b1, b2, b3 ;  
  
    printf("\nEnter names, prices & no. of pages of 3 books\n");  
    scanf("%c %f %d", &b1.name, &b1.price, &b1.pages);  
    scanf("%c %f %d", &b2.name, &b2.price, &b2.pages);  
    scanf("%c %f %d", &b3.name, &b3.price, &b3.pages);  
  
    printf("\n\nAnd this is what you entered");  
    printf("\n%c %f %d", b1.name, b1.price, b1.pages);  
    printf("\n%c %f %d", b2.name, b2.price, b2.pages);  
    printf("\n%c %f %d", b3.name, b3.price, b3.pages);  
}
```

And here is the output...

```
Enter names, prices and no. of pages of 3 books  
A 100.00 354  
C 256.50 682  
F 233.70 512
```

```
And this is what you entered  
A 100.000000 354  
C 256.500000 682  
F 233.700000 512
```

- A structure is usually used when we wish to store dissimilar data together.
- Structure elements can be accessed through a structure variable using a dot (.) operator.
- Structure elements can be accessed through a pointer to a structure using the arrow (→) operator.
- All elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.
- It is possible to pass a structure variable to a function either by value or by address.
- It is possible to create an array of structures

10. Files

10.1 File Pointers

There are many ways to use files in C. The most straight forward use of files is via a file pointer.

```
FILE *fp;
```

`fp` is a pointer to a file.

The type `FILE`, is not a basic type, instead it is defined in the header file `stdio.h`, this file must be included in your program.

10.2 Opening a File

```
fp = fopen(filename, mode);
```

The filename and mode are both strings.

The mode can be

- `r` - read
- `w` - write, overwrite file if it exists
- `a` - write, but append instead of overwrite
- `r+` - read & write, do not destroy file if it exists
- `w+` - read & write, but overwrite file if it exists
- `a+` - read & write, but append instead of overwrite
- `b` - may be appended to any of the above to force the file to be opened in binary mode rather than text mode
- `fp = fopen("data.dat", "a");` - will open the disk file `data.dat` for writing, and any information written will be appended to the file.

The following useful table from the ANSI C Rationale lists the different actions and requirements of the different modes for opening a file:

	r	w	a	r+	w+	a+
file must exist before open	✓			✓		
old file contents discarded on open		✓			✓	
stream can be read	✓			✓	✓	✓
stream can be written		✓	✓	✓	✓	✓
stream can be written only at end			✓			✓

`fopen` returns `NULL` if the file could not be opened in the mode requested. The returned value should be checked before any attempt is made to access the file. The following code shows how the value returned by `fopen` might be checked. When the file cannot be opened a suitable error message is printed and the program halted. In most situations this would be inappropriate, instead the user should be given the chance of re-entering the file name.

```
#include <stdio.h>
int main()
{
    char filename[80];
    FILE *fp;
    printf("File to be opened? ");
    scanf("%79s", filename);
    fp = fopen(filename, "r");
    if (fp == NULL)
    {
        fprintf(stderr, "Unable to open file %s\n", filename);
        return 1; /* Exit to operating system */
    }
    //code that accesses the contents of the file
    return 0;
}
```

Sequential file access is performed with the following library functions.

- `fprintf(fp, formatstring , ...)` – print to a file
- `fscanf(fp, formatstring , ...)` – read from a file
- `getc(fp)` – get a character from a file
- `putc(c, fp)` – put a character in a file
- `ungetc(c, fp)` – put a character back onto a file (only one character is guaranteed to be able to be pushed back)
- `fopen(filename , mode)` – open a file
- `fclose(fp)` – close a file

The standard header file `stdio.h` defines three file pointer constants, `stdin`, `stdout` and `stderr` for the standard input, output and error streams. It is considered good practice to write error messages to the standard error stream.

Use the `fprintf()` function to do this:

```
fprintf(stderr, "ERROR: unable to open file %s\n", filename);
```

The functions `fscanf()` and `getc()` are used for sequential access to the file, that is subsequent reads will read the data following the data just read, and eventually you will reach the end of the file. If you want to move forwards and backwards through the file, or jump to a specific offset from the beginning, end or current location use the `fseek()` function (though the file must be opened in binary mode). The function `ftell()` reports the current offset from the beginning of the file.

If you wish to mix reading and writing to the same file, each switch from reading to writing (or vice versa) must be preceded by a call to `fseek()`, `fsetpos()`, `rewind()` or `fflush()`. If it is required to stay at the same position in the file then use `fflush()` or `fseek(fp, 0L, SEEK_CUR)` which will move 0 bytes from the current position!

11. Command Line Arguments

The arguments that we pass on to `main()` at the command prompt are called command line arguments. The full declaration of `main` looks like this:

```
int main (int argc, char *argv[])
```

The function `main()` can have two arguments, traditionally named as `argc` and `argv`. Out of these, `argv` is an array of pointers to strings and `argc` is an `int` whose value is equal to the number of strings to which `argv` points. When the program is executed, the strings on the command line are passed to `main()`. More precisely, the strings at the command line are stored in memory and address of the first string is stored in `argv[0]`, address of the second string is stored in `argv[1]` and so on. The argument `argc` is set to the number of strings given on the command line.

For example, in our sample program, if at the command prompt we give,

```
filecopy PR1.C PR2.C
```

then,

`argc` would contain 3

- `argv[0]` - would contain base address of the string `"filecopy"`
- `argv[1]` - would contain base address of the string `"PR1.C"`
- `argv[2]` - would contain base address of the string `"PR2.C"`

[Complete C Programming Language Tutorials.](#)

#Primary

#C Programming


← Prev article


Strings - C Programming

Next article →

Watch Microsoft Build 2016 Keynote

C Programming	36
Swift	33
Tech	24





**OUR M365 CUSTOMERS
RESOLVE BREACHES
200% FASTER**

Request a demo to see
how we can help.

Learn More

Newsletter

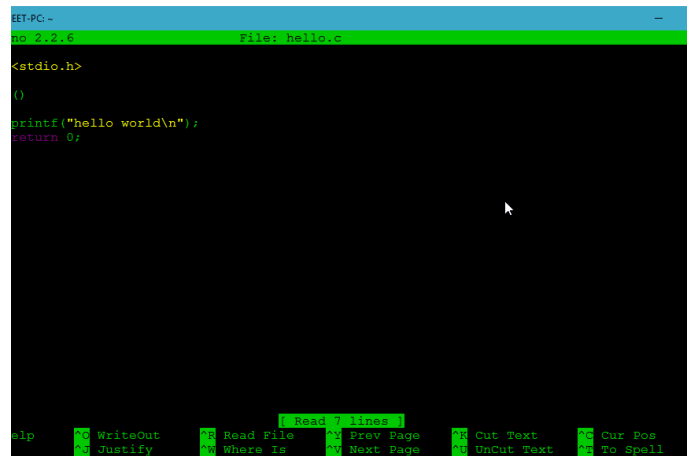
Stay up to date! Get all the latest & greatest posts delivered straight to your inbox

SUBSCRIBE



WIKI

C/C++ Compiler (gcc) for Android - Run C/C++ programs on Android



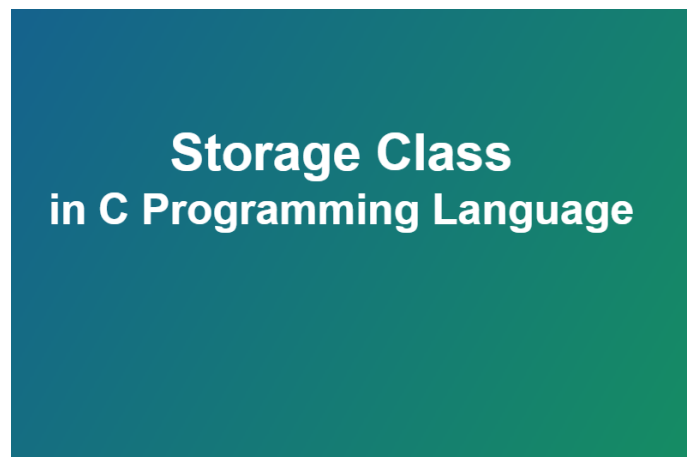
WIKI

Compile C program with gcc compiler on Bash on Ubuntu on Windows 10



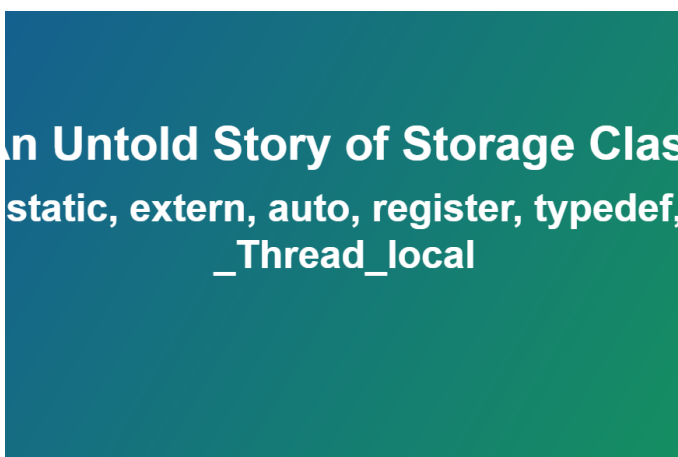
PRIMARY

What is Digraphs, Trigraphs and Tokens? - C/C++ Programming Language



PRIMARY

Storage Classes in C Programming Language



PRIMARY

An Untold Story of Storage Class in C Programming Language



PRIMARY

Simple and Static Assertion (assert) in C Programming Language

[Share](#)[Tweet](#)[Send](#)

Sign Up To The Newsletter

Stay up to date! Get all the latest & greatest posts delivered straight to your inbox

Subscribe

[C Programming](#) [C++](#) [Data Structure](#) [Downloads](#) [Interview Q&A](#) [iOS](#) [Microsoft](#) [Programming Puzzles](#) [Swift Programming](#) [Tech Wiki](#) [Xcode](#) [Interview Question](#) [Interview Assignment](#) [Android News](#) [iOS News](#) [Windows News](#) [Games News](#) [Tips and Tricks](#)

Copyright 2020, Developer Insider. All Rights Reserved.



AM
EX

DON'T
do business
WITHOUT IT™

Prodotto
BUSINESS

Con **Carta Business**
American Express, per te:

> **Quota gratuita**
il 1° anno.

> **Fino a 48 giorni**
per pagare le spese.



Scopri di più

Messaggio pubblicitario con finalità promozionali. F
americanexpress.it/terminiecondizioni. Si applicano 1

LATEST ARTICLES

- 1

What's new in Xcode 12?
- 2

Download Xcode and other Developer Tools up to 16 times faster
- 3

Best way to check if your iOS App is running on a Jailbroken Phone
- 4

Best iOS Development Tips and Tricks - Part 3
- 5

Install SonarQube on Ubuntu 18.04
- 6

Install Jenkins on Ubuntu 18.04
- 7

iOS 13 - How to Integrate Sign In with Apple in your Application?
- 8

What's new in Xcode 11? [Updated for 11.1, 11.2, 11.3, 11.4, 11.5, & 11.6]
- 9

Download Visual Studio 2019 Web Installer / ISO (Community / Professional / Enterprise)
- 10

iOS12 - Password AutoFill, Automatic Strong Password, and Security Code AutoFill

Primary73

IOS57

ShareTweetSend