

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens –

```
printf("Hello, World! \n");
```

The individual tokens are –

```
printf
```

```
(
```

```
"Hello, World! \n"
```

```
)
```

```
;
```

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Given below are two different statements –

```
printf("Hello, World! \n");
```

```
return 0;
```

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminate with the characters */ as shown below –

```
/* my first program in C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, \$, and % within identifiers. C is a case-sensitive programming language. Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers –

```
mohd      zara      abc      move_name  a_123
```

```
myname50  _temp      j        a23b9      retVal
```

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned

const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement –

```
int age;
```

there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement –

```
fruit = apples + oranges; // get the total fruit
```

no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish to increase readability.

s in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens –

```
printf("Hello, World! \n");
```

The individual tokens are –

```
printf  
(  
    "Hello, World! \n"  
)  
;
```

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Given below are two different statements –

```
printf("Hello, World! \n");  
return 0;
```

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with `/*` and terminate with the characters `*/` as shown below –

```
/* my first program in C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore `'_'` followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as `@`, `$`, and `%` within identifiers. C is a case-sensitive programming language. Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers –

```
mohd      zara      abc      move_name  a_123
myname50  _temp      j        a23b9      retVal
```

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

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

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as `int`, ends and the next element begins. Therefore, in the following statement –

```
int age;
```

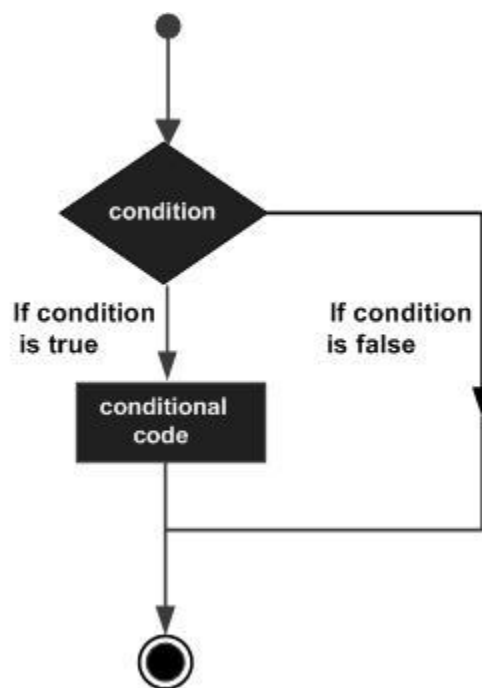
there must be at least one whitespace character (usually a space) between `int` and `age` for the compiler to be able to distinguish them. On the other hand, in the following statement –

```
fruit = apples + oranges; // get the total fruit
```

no whitespace characters are necessary between `fruit` and `=`, or between `=` and `apples`, although you are free to include some if you wish to increase readability.

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Show below is the general form of a typical decision making structure found in most of the programming languages –



C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

C programming language provides the following types of decision making statements.

Sr.No.	Statement & Description
1	<p>if statement</p> <p>An if statement consists of a boolean expression followed by one or more statements.</p>
2	<p>if...else statement</p> <p>An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.</p>
3	<p>nested if statements</p> <p>You can use one if or else if statement inside another if or else if statement(s).</p>
4	<p>switch statement</p> <p>A switch statement allows a variable to be tested for equality against a list of values.</p>
5	<p>nested switch statements</p> <p>You can use one switch statement inside another switch statement(s).</p>

The ? : Operator

We have covered conditional operator `? :` in the previous chapter which can be used to replace `if...else` statements. It has the following general form –

```
Exp1 ? Exp2 : Exp3;
```

Where `Exp1`, `Exp2`, and `Exp3` are expressions. Notice the use and placement of the colon.

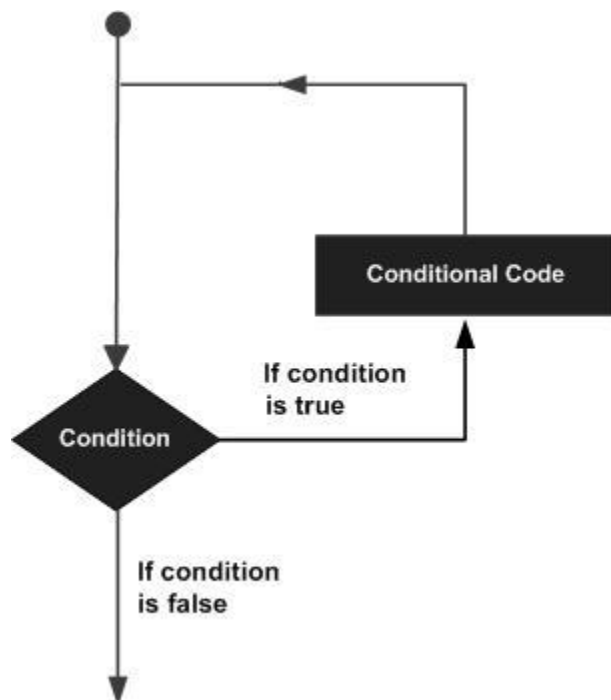
The value of a `? :` expression is determined like this –

- `Exp1` is evaluated. If it is true, then `Exp2` is evaluated and becomes the value of the entire `? :` expression.
- If `Exp1` is false, then `Exp3` is evaluated and its value becomes the value of the expression.

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages –



C programming language provides the following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	<p>while loop</p> <p>Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.</p>
2	<p>for loop</p> <p>Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.</p>
3	<p>do...while loop</p> <p>It is more like a while statement, except that it tests the condition at the end of the loop body.</p>
4	<p>nested loops</p> <p>You can use one or more loops inside any other while, for, or do..while loop.</p>

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

Sr.No.	Control Statement & Description
1	<p>break statement</p> <p>Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.</p>
2	<p>continue statement</p> <p>Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.</p>
3	<p>goto statement</p> <p>Transfers control to the labeled statement.</p>

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>

int main () {

    for( ; ; ) {
        printf("This loop will run forever.\n");
    }

    return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the `for(;;)` construct to signify an infinite loop.

NOTE – You can terminate an infinite loop by pressing Ctrl + C keys.

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

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

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

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

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

Defining a Function

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

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

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –

- **Return Type** – A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the `return_type` is the keyword `void`.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

Example

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

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Function Declarations

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

A function declaration has the following parts –

```
return_type function_name( parameter list );
```

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

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

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

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

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

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

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

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. For example –

[Live Demo](#)

```
#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
```

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 –

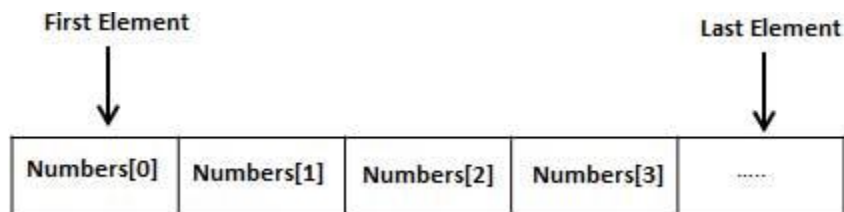
Sr.No.	Call Type & Description
1	<p>Call by value</p> <p>This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.</p>
2	<p>Call by reference</p> <p>This method copies the 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.</p>

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.

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The `arraySize` must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called `balance` of type `double`, use this statement –

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows –

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces `{ }` cannot be larger than the number of elements that we declare for the array between square brackets `[]`.

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

[Live Demo](#)

```
#include <stdio.h>

int main () {

    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;

    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ ) {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }
```



```

    /* output each array element's value */
    for (j = 0; j < 10; j++ ) {
        printf("Element[%d] = %d\n", j, n[j] );
    }

    return 0;
}

```

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

```

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer –

Sr.No.	Concept & Description
1	<p>Multi-dimensional arrays</p> <p>C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.</p>
2	<p>Passing arrays to functions</p>

	You can pass to the function a pointer to an array by specifying the array's name without an index.
3	Return array from a function C allows a function to return an array.
4	Pointer to an array You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

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 prints the address of the variables defined –

[Live Demo](#)

```
#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 the following result –

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

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 using 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 * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
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 a 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. The following example makes use of these operations –

```
#include <stdio.h>
```

```
int main () {
```

[Live Demo](#)

```

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 the following result –

```

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

```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an 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 –

[Live Demo](#)

```

#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
```

In 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 */
```

Pointers in Detail

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer –

Sr.No.	Concept & Description
1	<p>Pointer arithmetic</p> <p>There are four arithmetic operators that can be used in pointers: ++, --, +, -</p>
2	<p>Array of pointers</p> <p>You can define arrays to hold a number of pointers.</p>
3	<p>Pointer to pointer</p> <p>C allows you to have pointer on a pointer and so on.</p>

4	<p>Passing pointers to functions in C</p> <p>Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.</p>
5	<p>Return pointer from functions in C</p> <p>C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.</p>