

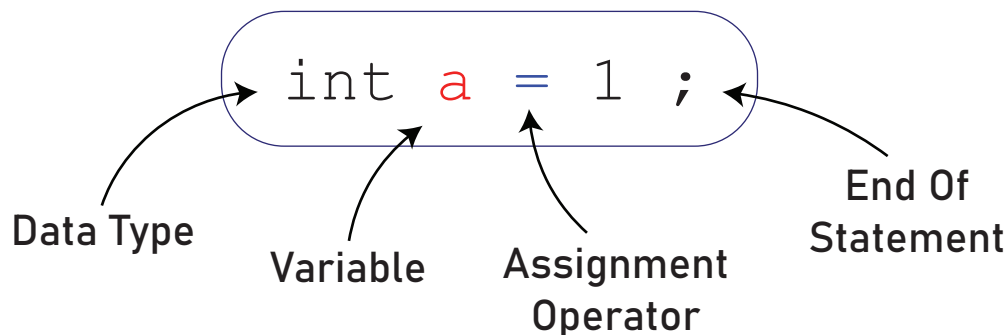
A Quick Intro To C

Lalith Kumar Reddy
ENIGMA, IIT B

C is a programming language. Programming is all about writing a logically executable code.

C is a language with specific syntax, which can only be learned through practice, practice and practice.

This is an example of a simple statement in C.



This type of statement is called initialization. We call it declaration if the variable is created but no value is assigned to it.

A data type specifies the type of data you are storing.

A variable is the name you assign to store the value.

Data type can be char, int, or float.

```
char c = 'g';      // 1 byte
int a = 5;         // 4 bytes
float b = 10.25;   // 8 bytes
```

The char data type stores 1 character in a variable. They are stored in the form of ASCII values (American Standard Code for Information Interchange). Each character is assigned a unique value. Here are few of them for your reference.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

The integer data type can be represented using byte(1), short(2), int(4) and long(8). Actually depending on compiler size (32/64bit), size of int differs between 4 or 8 bytes. By default, assume 4.

The integer data types can not represent fractional values. This is why we have two other data types: float(4) and double(8).

Depending on their sizes(in bytes), their value ranges also vary. Also, int can be of signed or unsigned type.

Like if int is 4 bytes => 32 bits, range of signed int type is between -2^{31} to $2^{31} - 1$, and unsigned int type is between 0 to $2^{32} - 1$.

The statements are written in functions.

```
int main() {
    // main is the name of the main function
    int val = 4;
    return val;
    // return value is of same data type
}
```

Anything written within braces "{ }" is one block.

In order to interact with the code, you will have to give inputs and wait for your code to give outputs.

This can be established with the help of a library provided in C.

```
#include <stdio.h>
// stdio is a library provided in C
// This helps with input/output

int main() {
    printf("Hello, World!\nI am alive.");
    // "\n" is an escape sequence
    // it takes the cursor to the next line

    return 0;
}
```

This will give you the output ->

```
Hello, World!
I am alive.
```

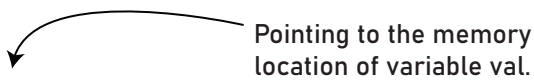
Now to take inputs and operate on them:

```
#include <stdio.h>

int main() {
    int val;
    scanf("%d", &val);
    // %d is input type decimal (int)
    // &val is the address of val

    val = val*2;
    printf("Twice of val is %d", val);

    return val;
}
```



scanf is used to take input from the user.

We use ->

%d for int type

%f for float type

%c for char type

%s for string

Compute the output of the following code.

```
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    printf("%d %d\n", a, b);

    b = a;
    a = 0;
    printf("%d %d\n", a, b);

    return 0;
}
```

This will give you the output ->

```
5 10
0 5
```

Note: When you assign one variable to the other, you just copy that value and not link their memories.

So when we do “b = a;” we only copy the value of a to b.
When a is changed to 0, b remains 5.

Now check out these arithmetic operations.

```
#include <stdio.h>
int main() {
    int a = 5/2;
    float b = 5/2;
    float c = 5.0/2, d = 5/2.0;
    float e = 5.0/2.0;

    printf("a=%d, b=%f, c=%f, d=%f, e=%f\n",
        a, b, c, d, e);

    return 0;
}
```

What do you think will be the output of this?

This is the output of the code->

a=2, b=2.000000, c=2.500000, d=2.500000, e=2.500000

Why? Because integer operations round down the decimal value and produce the output. In case of float operations, even one float value will produce the exact fractional value without rounding it down.

Lets observe something interesting. Check this code segment.

```
printf("%f\n", 10*5/2);  
printf("%f\n", 5/2*10);
```

This will give you the output ->

25.000000
20.000000

How so? In C, it is not simple BODMAS. It is following an order of precedence. In this case, * and / have the same priority. So it is just executed from left to right.

First case will be $50/2 = 25$

Second case will be $2*10 = 20$ (Remember integer operations!)

Now lets see some other basic operators.

```
int x = 10;  
int y = 4;  
int r = x % y; //Remainder when x is divided by y  
// Read as x modulo y  
printf("Rem = %d\n", r);
```

This will give you the output ->

Rem = 2

If we want to increase or decrease values by 1, we can simply use the increment(++) or decrement(--) operators.

```
int x = 2;  
int y = 3 * x++;  
int z = --y;  
z--;  
printf("x=%d, y=%d, z=%d\n", x, y, z);
```

So the output of the code will look like->

x=3, y=5, z=4

Confused? No worries. It is quite simple.

If `x++` is written, this means the current value of `x` is used and then `x` is increased. This is called post-increment.

If `++x` is written, this means the first the value of `x` is increased and the new value is used. This is called pre-increment.

Similarly, we can workout the same for decreasing the value.

Just keep in mind these priorities:

Prefix > paranthesis > mul/div/mod > add/sub > assign > postfix

Lets revisit functions. A function is just a group of statements which are used to execute a specific task.


Every code has the “main” function. Whenever we run any code, the “main” function is executed.

You can create your own functions. Each function with various functionalities and data types. The advantage of functions is, we don't have to rewrite the code segments for multiple operations.

Here is a simple example of a square function.

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

```
int square(int x) {  
    // x is called an argument  
    return x*x;  
}
```



Return type is int

```
int main() { // arguments of main is void (empty)  
    int a = 3;  
    int a_square = square(a);  
    // Called the function square and saved the  
    value in a_square  
    printf("Square of 3 = %d\n", a_square);  
    return 0;  
}
```

Output->

Square of 3 = 9

Note: If you write the square function after the main function, the code will not execute as the square function was not declared before main.

Alternatively, you can declare square first and write the function anywhere else.

```
#include <stdio.h>
// Declaring the function square
int square(int num); // Observe that argument
names are different, but not a problem

int main() {
    int a = 3;
    int a_square = square(a);

    printf("Square of 3 = %d\n", a_square);
    return 0;
}

int square(int x) {
    return x*x;
}
```

We can choose to have no return types. Such functions will be of void data type.

```
#include <stdio.h>

void square(int x) {
    int a_square = x*x;
    printf("Square of 3 = %d\n", a_square);
}

int main() {
    square(3);
    return 0;
}
```

Output of both these cases will be the same->

Square of 3 = 9

Also know that functions can have multiple parameters in the list of arguments. In fact, printf is also a function. It has varying num of arguments depending on number of variables to be displayed.

What do you think will be the output of this?

```
#include <stdio.h>

void square(int x) {
    x = x*x;
    printf("x in square = %d\n", x);
}

int main() {
    int x = 3
    square(x);
    printf("x in main = %d\n", x);
    return 0;
}
```

Output->

```
x in square = 9
x in main = 3
```

Reason: As mentioned much earlier, we only give it the value and not the address of the memory. So when x is modified in square, it only changes the value inside and not the original value.

This is called pass by value/call by value.

If we pass the address of the variable, we are passing/calling by reference.

Lets say we are interested in using only one variable throughout the program, we can define the variable globally.

The following code shows a global variable being used in multiple functions.


```
#include <stdio.h>

int n = 10; // n is global
void square() {
    n = n*n;
    printf("n in square = %d\n", n);
}

int main() {
    int x = 4; // x is local to main
    n = 11;
    square();
    printf("n in main = %d\n", n);
    return 0;
}
```

Output->

```
n in square = 121
n in main = 121
```

But do keep in mind that global variable is not the solution to change values in other functions. You could choose to return the new value or pass pointers (which you will learn much later). It is risky to operate with global variables in large codes.

Here I will introduce you to conditional statements, along with relational and logical operators.

Just remember true ~ 1 and false ~ 0.

```
int a=10, b=10;

if(a == b){
    printf("a = b\n");
}
else {
    printf("a != b\n");
}
```

Output->

```
a = b
```

'==' is called a conditional operator. It compares if a is equal to b and the output will be 1 or 0. '>', '<', '>=', '<=', '!=' are some of the other conditional operators.

'!(not), '||(or), '&&'(and) are logical operators.

As you might have noticed, we are using the 'if' statement for comparison of values. If that turns out to be true, everything inside of the if block is executed.

If that turns out to be false, everything inside the 'else' block will be executed.

We can also have multiple conditions as shown below.

```
int a=10, b=10;

if(a == b){
    if(a > 10 || a < 10)// simply a != 10
        printf("a = b != 10\n");
    else
        printf("a = b = 10\n");
}
else if(a > b){
    printf("a > b\n");
}
else if(a < b){
    printf("a < b\n");
}
```

Output->

a = b = 10

Observe that if there is only one statement to execute, paranthesis are not required.

The first block is structured in a way that there are conditions again inside it. This is called "Nested if-else".

Now workout the output of this.

```
if(!0 || 1 && 0)
    printf("True\n");
else
    printf("False\n");
printf("Last line\n");
```

Output->

True

Last line

Here again there is an order of precedence. && has higher priority over ||, and ! has higher priority over both of these.

If there are many other conditions for checking just one factor, we can use "Switch-case". It is far simpler for large comparisons.

```
int day_of_week=3;

switch(day_of_week) {
    case 0:
        printf("Sunday\n");
        break;
    case 1:
        printf("Monday\n");
        break;
    case 2:
        printf("Tuesday\n");
        break;
    case 3:
        printf("Wednesday\n");
        break;
    case 4:
        printf("Thursday\n");
        break;
    case 5:
        printf("Friday\n");
        break;
    case 6:
        printf("Saturday\n");
        break;
    default:
        printf("Invalid\n");
        break;
}
```

Output->

Wednesday

Note: It is important you put the “break” statement at the end of every case. Or else it will “Fall through” and execute the next case statements as well, till it finds a break or till it reaches the end.

Also see the end of the switch block. There is a default case. This is useful when there maybe an invalid input and no case matches it.

Now we'll discuss loops. There are three kinds:-

- While
- Do While
- For

Advantage of using loops is that we can reduce the size of code we write significantly. To perform 100 same operations, we don't have to write the same piece of code 100 times.

First, lets see while loop.

```
int a=0;

while(a < 10){
    printf("%d ", a);
    a++;
}
```

Output->

0 1 2 3 4 5 6 7 8 9

IMPORTANT NOTE: Never forget increment/decrement. Otherwise it will enter into an infinite while loop. (In this case, printing all 0s).

This could also be written using do while.

```
do{
    printf("%d \n", a);
    a++
}
while(a < 10);
```

Output remains exactly the same.

Now we can make it little easier using for loop.

```
int a;

for(a=0; a<10; a++){
    printf("%d ", a);
}
```

Output is exactly the same for this too.

The format of for is->

for (<assignment>; <condition>; <increment/decrement>)

It is not necessary to have all three fields filled. Maybe the value was assigned earlier. Maybe there is a termination condition inside the loop. Maybe we increase/decrease depending on certain output.

But it is preferred to have all filled together.

Finally, lets discuss arrays.

Arrays are a set of values corresponding to the same data type. Following are examples of declaration and initialization of certain data types.

```
int arr1[5];    // Declaring array of size 5
arr1 = {1, 2, 3, 4, 5}
```

```
float arr2[4] = {2.0, 3.5, 1.29, 5.5};
// Initialized array of size 4
```

```
char s[] = "A string"; // This is the same as->
// s = {'A', ' ', 's', 't', 'r', 'i', 'n', 'g'}
```

It is not allowed to assign more data than the size allotted. But it is allowed to initialize the values without specifying the size.

Character array is a special case. A string is the same as an array of characters. Yet its elements are accessed using the same way as an array is.

Here is a basic way to access and assign or read values of an array.

```
int n;
scanf("%d", &n);
int a[n];

for(int i=0; i<n; i++){
    scanf("%d", &a[i]);
    // Remember to pass address to save input
}

for(int i=0; i<n; i++){
    printf("%d ", a[i]);
}
```

Input->

10

0 6 3 -3 4 1 -4 7 8 12

Output->

0 6 3 -3 4 1 -4 7 8 12

I would suggest you to play around this to get a grip on I/O methods and arrays.

Note: When you scan for string "%s", you don't have to pass the add of the variable as arrays themselves are memory addresses. You access the elements with arr[i] and not arr. [Pointers... ;) ...]

Now that you've reached the end of basic C, you are ready to learn arduino concepts.

Here are a few video links:-

Command line and Vim: <https://youtu.be/UYu9Iklqprl>

Printf: <https://youtu.be/O9-5HLgprwc>

Variables: <https://youtu.be/MaLt2glPqc0>

Arithmetic: <https://youtu.be/Nhn7iVa2V7g>

Functions: <https://youtu.be/al-07bn7iHU>

Scope and scanf: <https://youtu.be/Z8gcuJtFUc0>

Logic, If-else: <https://youtu.be/I9ixcVmZKCE>

Loops and arrays: <https://youtu.be/b00tN5VTTFw>

Practice: <https://www.hackerrank.com/domains/algorithms?filters%5Bsubdomains%5D%5B%5D=implementation>