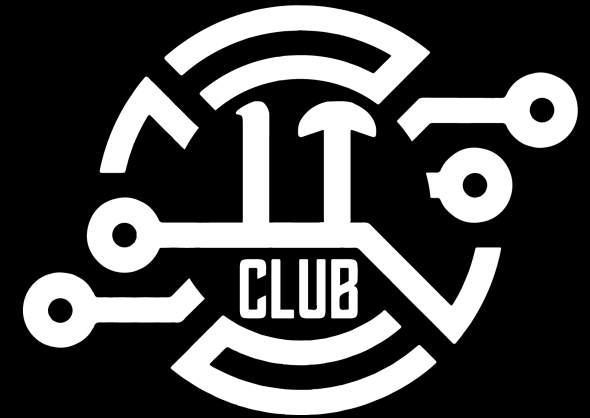


A WORKSHOP ON

# THE C PROGRAMMING LANGUAGE



## LECTURE 0

Now you C me!

**Do you C?**

**Because computers C**



**3 W's of C**

**LECTURE 0**

Now you C me!

# What is C?

C is a programming language, like any other.

It does the job, like any other does. But how it does is where the magic is.

In fact, C is often termed the **lingua franca** of programming languages due to its efficiency for both high level and low level systems.



# Where is C?

C is everywhere.

Databases (Most of the famous ones, including SQL);

Windows' and Linux kernel;

Git;

Doom;

Blender, Browsers' engine;

Most of the programming languages we see today....



# Why C?

- Much control over how our program runs
- Combines the good bits of low level as well as high level languages
- Much older than we are, hence people have already tried many things that we are about to -- meaning, abundant documentation
- Great for starters
- Since programs are directly compiled, they execute faster.

And so on....and on....and on....and on....



# Then, how to C?

That, my friend, is what we have come to see.



**Let's write some code**

## **LECTURE 0**

Now you C me!

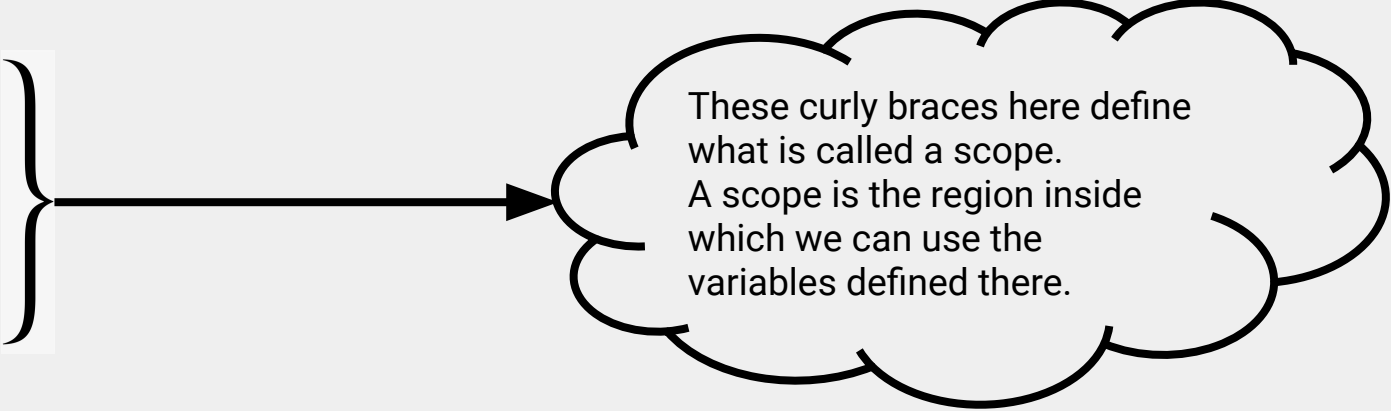


//Why doesn't this line add the two numbers 1 and 2?

```
ungabunga = 1+2
```

//But this does?

```
int main(){  
    int ungabunga = 1+2;  
  
    return 0;  
}
```



These curly braces here define what is called a scope. A scope is the region inside which we can use the variables defined there.



# ENTRY POINT

**The point where the execution of the program starts.**

In C, our entry point is almost always the `main()` function.  
So whatever we have written inside the `main()` is what really gets executed most of the time.



This something is called a *data*.

# VARIABLES

Anything that stores something and can retrieve it for future use is a variable.

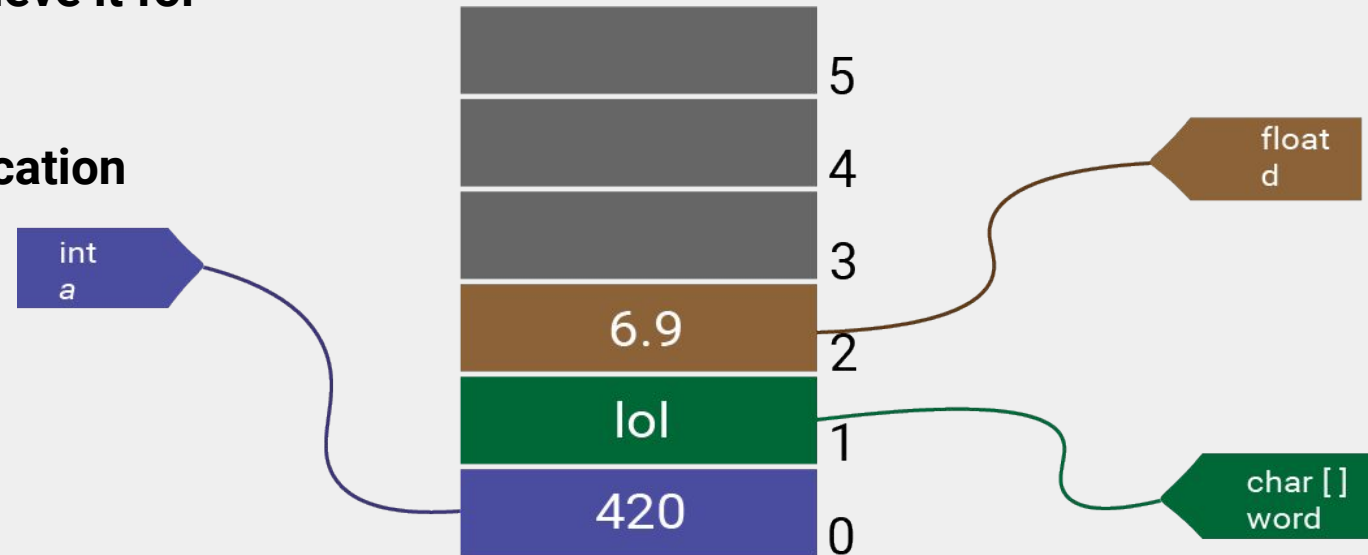
A symbolic representation of the memory location that actually stores the contents.

Examples:

```
int a = 420
```

```
char word[] = "lol"
```

A0vAriAbLe1naMe2cAn\_bE\_this\_BUT\_KeepItReadable



# QUESTION

What would happen if I were an integer in a variable named “int”?

## LECTURE 0

Now you C me

```
int main(){  
    /* Writing this doesn't work because int is a keyword and  
    cannot be used for identifying a literal as a variable*/  
    int int = 76;  
    /*If we replace the variable name with anything else than a keyword,  
    it works without a second thought*/  
    int rollNumber = 76;  
    return 0;  
}
```



# KEYWORDS

The words that are reserved for special purpose by C itself.  
We cannot redefine them. This is the reason we cannot use them as variables

There are a total of 32 basic keywords in C, and we have already seen a couple of them: `int` and `return`

We can always come back to the full list in the [documentations](#) (checking some random posts in StackOverflow is highly discouraged). So in case, you forget them, it will always be your friend.

```
int currentYear = 2021;  
//int here is a keyword (a data type, to be precise)
```



# IDENTIFIERS

Like the name suggests, they help identify the variables or constants we *store* in our program.

So, in a nutshell, the names of variables we have used are all identifiers.  
Example:

```
int currentYear = 2021;  
//currentYear here is an identifier
```

These need to be unique. Else, the compiler would be confused, just like you would if you had two people with the same name at your home.



# CONSTANTS AND LITERALS

The values that are fixed and cannot be changed.  
Literally, literals are the literal values.

Constants may be Integer (Binary, Decimal, Hexadecimal, Octal), Floats, Characters.

```
int currentYear = 2021;  
//2021 here is a literal
```

Generally, strings are associated with literals and we can very often see the term “string literals” but it should be clear that literals are all those values whose meaning cannot be changed during the course of the program as we discussed earlier.





# OPERATORS

I might sound like a broken record but C has a good habit of referring to things just as they are -- operators are literally operators that operate on the values we provide and give us the result.

1. **Arithmetic operators**  
(+, -, \*, /, %)
2. **Assignment operators**  
(=, +=, -=, \*=, /=)
3. **Comparison operators**  
(<, >, <=, >=, ==, !=)
4. **sizeof()**



# OPERATOR PRECEDENCE

The order of operation -- the system which defines what operation gets operated first.

The BODMAS we studied in our early school years is an example of precedence.

## ASSOCIATIVITY

For the operations of same precedence, there needs to be a system that governs what to evaluate first. This is the same system.

Example: + and - have the same precedence, and its associativity is left-to-right so in `int a = 10+5-9;` , first + operation happens, then only -.



```
int main(){
    int num1=5, num2=10, sum;
    float pointNumber;
    char firstChar;
    int sizeofInt, sizeofChar, sizeofFloat;
    pointNumber = 2.5;
    sum = num1 + num2;
    sizeofInt = sizeof(sum);
    sizeofChar = sizeof(firstChar);
    //Why does this give off the value 4 but
    sizeofFloat = sizeof(pointNumber);
    //but this gives the value 8?
    sizeofAnotherFloat = sizeof(2.5);
    return 0;
}
```



# A Reference (1byte = 8bits)

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615





# Another Reference

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters		
32	space	
33	!	
34	"	
35	#	
36	\$	
37	%	
38	&	
39	'	
40	(	
41	)	
42	*	
43	+	
44	,	
45	-	
46	.	
47	/	
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[	
92	\	
93	]	
94	^	
95	_	
96	`	
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	
124		
125	}	
126	~	

Extended ASCII characters			
128	Ç	160	á
129	ü	161	í
130	é	162	ó
131	â	163	ú
132	ä	164	ñ
133	à	165	Ñ
134	á	166	ª
135	ç	167	º
136	ê	168	¿
137	ë	169	®
138	è	170	¬
139	ï	171	½
140	î	172	¼
141	ì	173	¡
142	Ä	174	«
143	Å	175	»
144	É	176	
145	æ	177	
146	Æ	178	
147	ô	179	
148	ö	180	
149	ò	181	À
150	û	182	Á
151	ù	183	Â
152	ÿ	184	©
153	Ö	185	
154	Ü	186	
155	ø	187	
156	£	188	
157	Ø	189	
158	×	190	¥
159	f	191	
192	Ł	224	Ó
193	ł	225	Ô
194	Ł	226	Õ
195	ł	227	Ö
196	Ł	228	ø
197	ł	229	Ő
198	Ł	230	μ
199	Ł	231	þ
200	Ł	232	ƒ
201	Ł	233	ú
202	Ł	234	û
203	Ł	235	ü
204	Ł	236	ý
205	Ł	237	Ý
206	Ł	238	ˆ
207	Ł	239	˙
208	Ł	240	≡
209	Ł	241	±
210	Ł	242	≡
211	Ł	243	¼
212	Ł	244	¶
213	Ł	245	§
214	Ł	246	÷
215	Ł	247	˙
216	Ł	248	˚
217	Ł	249	ˆ
218	Ł	250	˙
219	Ł	251	¹
220	Ł	252	³
221	Ł	253	²
222	Ł	254	■
223	Ł	255	nbsp



# A MISCELLANEOUS TOPIC

## LECTURE 0

Now you C me!

Sometimes, we need to change how we operate in the values in between an operation. Some of them, the compiler does them automatically.

Example:

If we divide a float by an int, the result is automatically a float.

**But what would happen if I were to divide an int by an int?**



Compile and see the different values the lines return and try to understand what exactly is going on here

```
int main(){
    int aInt = 1;
    float bFloat = 2.5;

    int cMul = aInt * bFloat;
    float dMul = aInt * bFloat;
    float cDiv = aInt / bFloat;
    int div = aInt / bFloat;

    return 0;
}
```

What is happening here is known as type casting. A cast can be thought of as an operator to convert one type of variable into another -- sometimes by compiler itself, if it makes sense to it, the other times by user, by explicitly stating so.





# CONDITIONALS

Have you ever thought to yourself, “If I could just code a program that would tell me if I could vote or not after I type in my age”?

Unless you’re an old school computer teacher, the chances of it being true are quite slim, but well...

Conditionals are used wherever we need to specify a condition, as its name implies (yet again), and make a block of code run based on the fact that it is true or false.

Since they can alter the flow of execution of a program, they are one of the control-flow statements.

The most famous, which we have most probably heard of, is the **if** conditional.



# IF CONDITIONAL

The most common type of conditional, and the one which you will use the most (I hope).

It governs the program such that, certain code is executed only if certain condition is true.

General Syntax:

```
if (expression){  
    statement-true  
}  
else{  
    statement-false  
}
```



```
int main(){
    int birthYear, age;
    int drinkingAge;
    birthYear = 1999;
    age = 2021 - birthYear;
    if (age <= 18)
        drinkingAge = 0;
    else
        drinkingAge = 1;
    return 0;
}
```



But what if you have to check multiple conditions at once -- something like checking if an integer lies between 5 and 15.  
Well, we resort to boolean operators.

```
int main(){
    int num = 6;
    int toggle;
    if(num > 5 && num < 15)
        toggle =1;
    else
        toggle =0;

    return 0;
}
```



Imagine, someone has you on gunpoint and says he will let you walk if you could check if the number he gives you lies in between 500 and 5000 and is divisible by 177. Fret not, for nest conditionals can be your live saving friend at that moment. And no, nest conditionals do not mean birds inside your conditional scope.

Nesting simply refers to using or placing something one inside the other.

The syntax can go something like:

```
if (expression){  
    statement-true  
    ...  
    if (expression){  
        statement-true  
        ...  
    }...  
}
```



```
int main(){
    int hisNumber = 3540;
    int toggle;
    //first condition
    if (hisNumber>500 && hisNumber<5000){
        //second condition
        if (hisNumber%177==0){
            toggle = 1;
        }
    }
    else
        toggle = 0;
    return 0;
}
```

Try pulling this code in one line with something called ternary operator



Find out how in our tomorrow's lecture, where we will learn if we could hack your friend's computer with loops.

## LECTURE 0

Now you C me!