GATE CSE NOTES

by

Joyoshish Saha

Resources : http://cs.yale.edu/homes/aspnes/classes/223/notes.html#overflow-and-the-c-standards
https://cheatography.com/ashlyn-black/cheat-sheets/c-reference/ **** https://cse.iitkgp.ac.in/~pds/ ****
http://www.cs.utsa.edu/~wagner/CS2213/resources/resources.html http://c-faq.com/

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Level	Operators	Description	Associativity
15	() [] -> . ++ --	Function Call Array Subscript Member Selectors Postfix Increment/Decrement	Left to Right
14	++ -- + - ! ~ (type) * & sizeof	Prefix Increment / Decrement Unary plus / minus Logical negation / bitwise complement Casting Dereferencing Address of Find size in bytes	Right to Left
13	* / %	Multiplication Division Modulo	Left to Right
12	+ -	Addition / Subtraction	Left to Right
11	>> <<	Bitwise Right Shift Bitwise Left Shift	Left to Right
10	< <= > >=	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To	Left to Right
9	== !=	Equality Inequality	Left to Right
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left
1	,	Comma Operator	Left to Right

Memorization trick:
PUMA'S REBL TAC
P: Parentheses () [] . ->
U: Unary
M: Multiplicative * / %
A: Addition + -
S: Shift >> <<
R: Relational < <= > >=
E: Equals == !=
B: Bitwise & ^ |
L: Logical && ||
T: Ternary
A: Assignment = += etc.
C: Comma ,

R2L associative: U, T, Assign
[AUTo]

Type	Description	Size	Domain
char	Signed character/byte. Characters are enclosed in **single** quotes.	1	-128..127
double	Double precision number	8	ca. $10^{-308}..10^{308}$
int	Signed integer	4	$-2^{31}..2^{31}-1$
float	Floating point number	4	ca. $10^{-38}..10^{38}$
long (int)	Signed long integer	4	$-2^{31}..2^{31}-1$
long long (int)	Signed very long integer	8	$-2^{63}..2^{63}-1$
short (int)	Short integer	2	$-2^{15}..2^{15}-1$
unsigned char	Unsigned character/byte	1	0..255
unsigned (int)	Unsigned integer	4	$0..2^{32}-1$
unsigned long (int)	Unsigned long integer	4	$0..2^{32}-1$
unsigned long long (int)	Unsigned very long integer	8	$0..2^{64}-1$
unsigned short (int)	Unsigned short integer	2	$0..2^{16}-1$

Pointer size 4B(32b), 8B(64b)
Regardless of data type

Objective-C Data Types Table

Type	Example	Specifier
char	'a', 'O', '\n'	%c
int	14, -14, 780, 0xEEC0, 098	%i, %d
unsigned int	10u, 121U, 0xEFu	%u, %x, %o
long int	18, -2100, 0xfeefL	%ld
unsigned long int	13UL, 101ul, 0xfefeUL	%lu, %lx, %lo
long long int	0xe5e5e5LL, 501ll	%lld
unsigned long long int	10ull, 0xffeeULL	%llu, %llx, %llo
float	12.30f, 3.2e-5f, 0x2.2p09	%f, %e, %g, %a
double	3.1415	%f, %e, %g, %a
long double	3.5e-5l	%Lf, %Le, %Lg, %La
id	Nil	%@

1. **printf (“hello\c”);**

   1.c:4:10: warning: unknown escape sequence: '\c'
   printf ("Hello\c");

   Output- helloc [\ ignored]

2. **int a = 5/9; printf (“%d”, a) //output- 0**
   float a = 5/9; printf (“%f”, a) //output- 0.00000
   float a = 5.0/9; printf (“%d”, a) //output- 0.55556

   In C, as in many other languages, integer division truncates: any fractional part is discarded. If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the integer will be converted to floating point before the operation is done.

3. Following code works fine for character inputs too. K&R pg 18

```
#include <stdio.h>
/* copy input to output; 1st version */
main()
{
        int c;
        c = getchar();
        while (c != EOF) {
                putchar(c);
                c = getchar();
        }
}
```

4. unsigned and signed are used for int and char both types, while short, long used for only int. The qualifier signed or unsigned may be applied to char or any integer. unsigned numbers are always positive or zero, and obey the laws of arithmetic modulo 2n, where n is the number of bits in the type. So, for instance, if chars are 8 bits, unsigned char variables have values between 0 and 255, while signed chars have values between -128 and 127 (in a two's complement machine.) Whether plain chars are signed or unsigned is machine-dependent, but printable characters are always positive.

5. The value of an integer can be specified in octal or hexadecimal instead of decimal. A leading 0 (zero) on an integer constant means octal; a leading 0x or 0X means hexadecimal. For example, decimal 31 can be written as 037 in octal and 0x1f or 0x1F in hex. Octal and hexadecimal constants may also be followed by L to make them long and U to make them unsigned: 0XFUL is an unsigned long constant with value 15 decimal.

6. We can use printf ("%d", '\013') to print the decimal value of 13(which is in octal). Also printf("%d", '\xa') to print the decimal value of a(which is in hex).

set of escape sequences is

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadecimal number
\v	vertical tab		

Certain characters can be represented in character and string constants by escape sequences like \n (newline); these sequences look like two characters, but represent only one.

7. **Enumeration constants**
   enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
   JUL, AUG, SEP, OCT, NOV, DEC };
   /* FEB = 2, MAR = 3, etc. */

8. *On my Linux 64bit system:*
   short int is 2 bytes int is 4 bytes int * is 8 bytes long int is 8 bytes long int * is 8 bytes signed int is 4 bytes unsigned int is 4 bytes float is 4 bytes float * is 8 bytes double is 8 bytes double * is 8 bytes long double is 16 bytes signed char is 1 bytes char is 1 bytes char * is 8 bytes unsigned char is 1 bytes

9. Expressions connected by && or || are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known.

10. the assignment d = c >= '0' && c <= '9' sets d to 1 if c is a digit, and 0 if not.

11. • If either operand is long double, convert the other to long double. • Otherwise, if either operand is double, convert the other to double. • Otherwise, if either operand is float, convert the other to float. • Otherwise, convert char and short to int. • Then, if either operand is long, convert the other to long.

12. Type conversion rule: All the data types of the variables are upgraded to the data type of the variable with the largest data type.

bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double
https://www.geeksforgeeks.org/implicit-type-conversion-in-c-with-examples/
https://www.guru99.com/c-type-casting.html
https://www.includehelp.com/c/type-conversion.aspx

13. **In general an unsigned n-bit type runs from 0 through 2n − 1 while the signed version runs from** -2^{n-1} **through** $2^{n-1}-1$**.** The representation of signed integers uses two's-complement notation, which means that a positive value x is represented as the unsigned value x while a negative value − x is represented as the unsigned value 2n − x. For example, if we had a peculiar implementation of C that used 3-bit ints, the binary values and their interpretation as int or unsigned int would look like this:

bits	as unsigned int	as int
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

The reason we get one extra negative value for an unsigned integer type is this allows us to interpret the first bit as the sign, which makes life a little easier for whoever is implementing our CPU. Two useful features of this representation are:

1. We can convert freely between signed and unsigned values as long as we are in the common range of both, and
2. Addition and subtraction work exactly the same for both signed and unsigned values. For example, on our hypothetical 3-bit machine, 1 + 5 represented as 001 + 101 = 110 gives the same answer as $1 + (-3) = 001 + 101 = 110$. In the first case we interpret 110 as 6, while in the second we interpret it as -2, but both answers are right in their respective contexts.

Note that in order to make this work, we can't detect overflow: when the CPU adds two 3-bit integers, it doesn't know if we are adding $7 + 6 = 111 + 110 = 1101 = 13$ or $(-1) + (-2) = 111 + 110 = 101 = (-3)$. In both cases the result is truncated to 101, which gives the incorrect answer 5 when we are adding unsigned values.

14. Sometimes you may have a running program that won't die. Aside from costing you the use of your terminal window, this may be annoying to other Zoo users, especially if the process won't die even if you close the terminal window or log out.

There are various control-key combinations you can type at a terminal window to interrupt or stop a running program.

ctrl-C

Interrupt the process. Many processes (including any program you write unless you trap SIGINT using the sigaction system call) will die instantly when you do this. Some won't.

ctrl-Z

Suspend the process. This will leave a stopped process lying around. Type jobs to list all your stopped processes, fg to restart the last process (or fg %1 to start process %1 etc.), bg to keep running the stopped process in the background, kill %1 to kill process %1 politely, kill -KILL %1 to kill process %1 whether it wants to die or not.

ctrl-D

Send end-of-file to the process. Useful if you are typing test input to a process that expects to get EOF eventually or writing programs using cat > program.c (not really recommended). For test input, you are often better putting it into a file and using input redirection (./program < test-input-file); this way you can redo the test after you fix the bugs it reveals.

ctrl-

Quit the process. Sends a SIGQUIT, which asks a process to quit and dump core. Mostly useful if ctrl-C and ctrl-Z don't work.

If you have a runaway process that you can't get rid of otherwise, you can use ps g to get a list of all your processes and their process ids. The kill command can then be used on the offending process, e.g. kill -KILL 6666 if your evil process has process id 6666. Sometimes the killall command can simplify this procedure, e.g. killall -KILL evil kills all process with command name evil.

15. **Integer constants**

Constant integer values in C can be written in any of four different ways:

- In the usual decimal notation, e.g. 0, 1, -127, 9919291, 97.
- In octal or base 8, when the leading digit is 0, e.g. 01 for 1, 010 for 8, 0777 for 511, 0141 for 97. Octal is not used much any more, but it is still conventional for representing Unix file permissions.
- In hexadecimal or base 16, when prefixed with 0x. The letters a through f are used for the digits 10 through 15. For example, 0x61 is another way to write 97.
- Using a character constant, which is a single ASCII character or an escape sequence inside single quotes. The value is the ASCII value of the character: 'a' is 97.7 Unlike languages with separate character types, C characters are identical to integers; you can (but shouldn't) calculate 972 by writing 'a'*'a'. You can also store a character anywhere.

Except for character constants, you can insist that an integer constant is unsigned or long by putting a u or l after it. So 1ul is an unsigned long version of 1. By default integer constants are (signed) ints. For long long constants, use ll, e.g., the unsigned long long constant 0xdeadbeef01234567ull. It is also permitted to write the l as L, which can be less confusing if the l looks too much like a 1.

Some examples:

'a'	int
97	int
97u	unsigned int
0xbea00d1ful	unsigned long, written in hexadecimal
0777s	short, written in octal

A curious omission is that there is no way to write a binary integer directly in C. So if you want to write the bit pattern 00101101, you will need to encode it in hexadecimal as 0x2d (or octal as 055). Another potential trap is that leading zeros matter: 012 is an octal value corresponding to what normal people call 10.

16. The **shift operators** << and >> shift the bit sequence left or right: x << y produces the value x · 2^y (ignoring overflow); this is equivalent to shifting every bit in x y positions to the left and filling in y zeros for the missing positions. In the other direction, x >> y produces the value ⌊x · 2^−y⌋ by shifting x y positions to the right. The behavior of the right shift operator depends on whether x is unsigned or signed; for unsigned values, it shifts in zeros from the left end always; for signed values, it shifts in additional copies of the leftmost bit (the sign bit). This makes x >> y have the same sign as x if x is signed.

If y is negative, the behavior of the shift operators is undefined.

Examples (unsigned char x):

x	y	x << y	x >> y
00000001	1	00000010	00000000
11111111	3	11111000	00011111

Examples (signed char x):

x	y	x << y	x >> y
00000001	1	00000010	00000000
11111111	3	11111000	11111111

Shift operators are often used with bitwise logical operators to set or extract individual bits in an integer value. The trick is that (1 << i) contains a 1 in the i-th least significant bit and zeros everywhere else. So x & (1<<i) is nonzero if and only if x has a 1 in the i-th place. This can be used to print out an integer in binary format (which standard printf won't do).

17. The core idea of floating-point representations (as opposed to fixed point representations as used by, say, ints), is that a number x is written as m · b^e where m is a mantissa or fractional part, b is a base, and e is an exponent. On modern computers the base is almost always 2, and for most floating-point representations the mantissa will be scaled to be between 1 and b. This is done by adjusting the exponent, e.g.

1 = 1 · 2^0
2 = 1 · 2^1
0.375 = 1.5 · 2 ^− 2

18. **extern keyword**

extern int var;
int main(void)
{
var = 10;
return 0;
}
	*Compile error.*This program throws an *error in compilation* because var is declared but not defined anywhere. Essentially, the var isn't allocated any memory. And the program is trying to change the value to 10 of a variable that doesn't exist at all.

#include "somefile.h"
extern int var;
int main(void)
{
var = 10;
return 0;
}
	Success. Assuming that somefile.h contains the definition of var, this program will compile successfully.

extern int var = 0;
int main(void)
{
var = 10;
return 0;
}
	*Success.*If a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated–in other words, that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work.

Declaration of a variable or function simply declares that the variable or function exists somewhere in the program, but the memory is not allocated for them. The declaration of a variable or function serves an important role–it tells the program what its type is going to be. In case of function declarations, it also tells the program the arguments, their data types, the order of those arguments, and the return type of the function. So that's all about the declaration.

Coming to the definition, when we define a variable or function, in addition to everything that a declaration does, it also allocates memory for that variable or function. Therefore, we can think of definition as a superset of the declaration (or declaration as a subset of definition).

A declaration can be done any number of times but definition only once.

The extern keyword is used to extend the visibility of variables/functions.

Since functions are visible throughout the program by default, the use of extern is not needed in function declarations or definitions. Its use is implicit.

When extern is used with a variable, it's only declared, not defined. As an exception, when an extern variable is declared with initialization, it is taken as the definition of the variable as well.

19. **register keyword**

#include<stdio.h>
int main()
{
register int i = 10;
int* a = &i;
printf("%d", *a);
getchar();
return 0;
}
	Compiler error. If you use & operator with a register variable then the compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid.

#include<stdio.h>
int main()
{
int i = 10;
register int* a = &i;
printf("%d", *a);
getchar();
return 0;
}
	Success.
	Register keyword can be used with pointer variables. Obviously, a register can have an address of a memory location. There would not be any problem with the below program.

#include<stdio.h>
int main()
{
int i = 10;
register static int* a = &i;
printf("%d", *a);
getchar();
return 0;
}
	Compile error. Register is a storage class, and C doesn't allow multiple storage class specifiers for a variable. So, register can not be used with static

#include <stdio.h>
register int x = 10;
int main()
{
register int i = 10;
printf("%d\n", i);
printf("%d", x);
return 0;
}
	Compile error. Register can only be used within a block (local), it can not be used in the global scope (outside main).

-->There is no limit on the number of register variables in a C program, but the point is the compiler may put some variables in the register and some not.

20. **auto storage class:**

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables reside. They are assigned a garbage value by default whenever they are declared.

21. **static storage class**

A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over. For example, we can use static int to count a number of times a function is called, but an auto variable can't be used for this purpose.

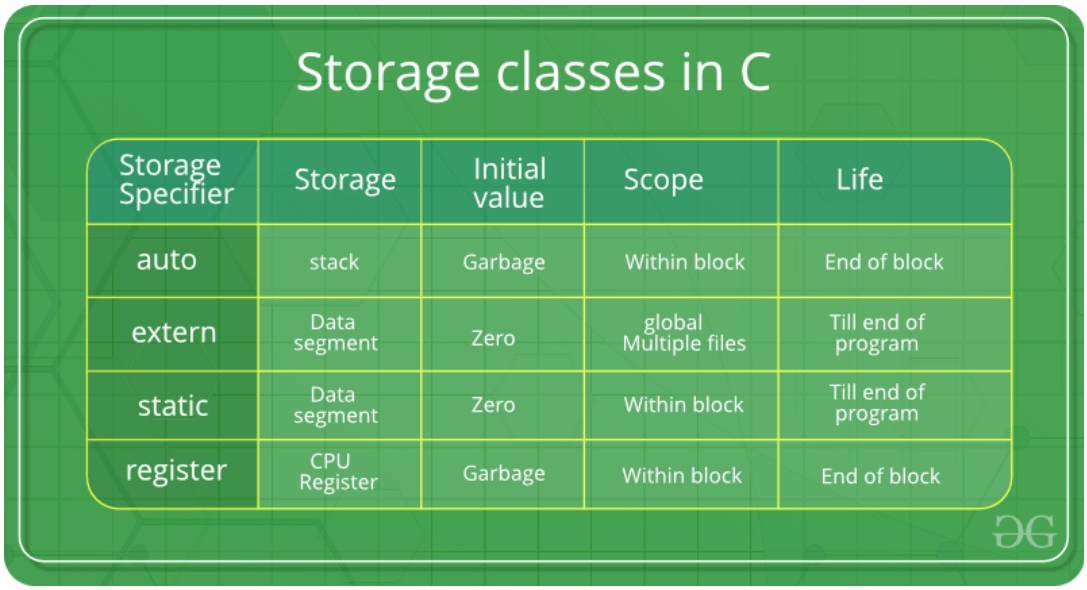
Static variables are allocated memory in the data segment, not the stack segment.

Static variables (like global variables) are initialized as 0 if not initialized explicitly.

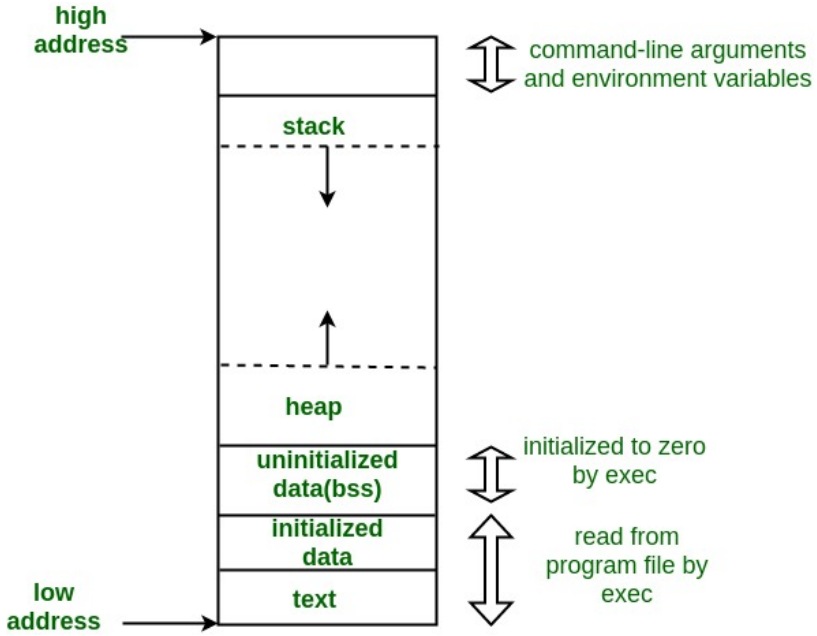
Static variables should not be declared inside a structure. The reason is C compiler requires the entire structure elements to be placed together (i.e.) memory allocation for structure members should be contiguous. It is possible to declare structure inside the function (stack segment) or allocate memory dynamically(heap segment) or it can be even global (BSS or data segment). Whatever might be the case, all structure members should reside in the same memory segment because the value for the structure element is fetched by counting the offset of the element from the beginning address of the structure. Separating out one member alone to a data segment defeats the purpose of static variable and it is possible to have an entire structure as static.

#include<stdio.h>
int initializer(void)
{
return 50;
}
int main()
{
static int i = initializer();
printf(" value of i = %d", i);
getchar();
return 0;
}
	Compile error. In C, static variables can only be initialized using constant literals. For example, the following program fails in compilation.

https://www.guru99.com/c-storage-classes.html



22. **Memory layout of C programs**



https://www.geeksforgeeks.org/memory-layout-of-c-program/

For instance a variable declared static int i; would be contained in the BSS segment.
For instance a global variable declared int j; would be contained in the BSS segment.

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

size a.out command in linux shell to know the memory layout for a.out. Check by taking more static or global variables. BSS, uninitialised data segment size changes.

23. **Pointers to *const***

A pointer to a region that should not be modified should be declared with const type:

```
const char *string = "You cannot modify this string.";
```

The const in the declaration above applies to the characters that string points to: string is not const itself, but is instead a pointer to const. It is still possible to make string point somewhere else, say by doing an assignment:

```
string = "You cannot modify this string either."
```

If you want to make it so that you can’t assign to string, put const right before the variable name:

```
/* prevent assigning to string as well */
const char * const string = "You cannot modify this string.";
```

Now string is a const pointer to const: you can neither modify string nor the values it points to.
Note that *const* only restricts what you can do using this particular variable name. If you can get at the memory that something points to by some other means, say through another pointer, you may be able to change the values in these memory locations anyway:

```
int x = 5;
const int *p = &x;
int *q;

*p = 1; /* will cause an error at compile time */
x = 3;  /* also changes *p, but will not cause an error */
```

-->Difference between const char *p, char * const p and const char * const p

https://www.geeksforgeeks.org/difference-const-char-p-char-const-p-const-char-const-p/

24.

```
int main()
{
        int x[10] = { 0 }, i = 0, *p;
        // p point to starting address of array x
        p = &x[0];
        while (i < 10) {
                *p = 10;

                // intention was to point to integer at x[1]
                p = p + 4;
                i++;
        }
}
```

Although it seems correct as integer is of 4 byte and p is at starting location so adding 4 to it will cause p to point at next integer in array n but pointer arithmetic work according to size of its data type so adding 1 to a pointer of integer then sizeof(int) will get added to it same applies for pointer to any other data type.

```
        int x[10] = { 0 }, i = 0, *p;
        p = &x[0]; // p point to starting address of array x
        while (i < 10) {
                *p = 10;
                p++; // means p = p + sizeof(int)
                i++;
```

25. **Passing array as parameter**

```
#include <stdio.h>
// arr is a pointer even if we have
// use square brackets.
void printArray(int arr[])
{
        int i;
        /* sizeof should not be used here to get number
        of elements in array*/
        int arr_size = sizeof(arr) / sizeof(arr[0]);
        for (i = 0; i < arr_size; i++) {
                printf("%d ", arr[i]);
        }
}
int main()
{
        int arr[4] = { 1, 2, 3, 4 };
        printArray(arr);
        return 0;
}
```

Passing array as parameter When we pass an array to a function, it is always treated as a pointer in the function. That's why we should never use sizeof on an array parameter. We should rather always pass size as a second parameter.
Correct one:

```
void printArray(int arr[], int arr_size)
{
        int i;
        for (i = 0; i < arr_size; i++) {
                printf("%d ", arr[i]);
        }
}
int main()
{
        int arr[] = { 1, 2, 3, 4 };
        int arr_size = sizeof(arr) / sizeof(arr[0]);
        printArray(arr, arr_size);
        return 0;
}
```

26. **Difference between ++*p, *p++ and *++p**

1) Precedence of prefix ++ and * is the same. Associativity of both is right to left.
2) Precedence of postfix ++ is higher than both * and prefix ++. Associativity of postfix ++ is left to right.

```
#include <stdio.h>
int main(void)
{
        int arr[] = {10, 20};
        int *p = arr;
        ++*p;
        printf("arr[0] = %d, arr[1] = %d, *p = %d",
                                                arr[0], arr[1],
*p);
        return 0;
}
```

11 20 11

#include <stdio.h>
int main(void)
{
int arr[] = {10, 20};
int *p = arr;
*p++;
printf("arr[0] = %d, arr[1] = %d, *p = %d",
arr[0], arr[1],
*p);
return 0;
}
	10 20 20

#include <stdio.h>
int main(void)
{
int arr[] = {10, 20};
int *p = arr;
*++p;
printf("arr[0] = %d, arr[1] = %d, *p = %d",
arr[0], arr[1],
*p);
return 0;
}
	10 20 20

27. malloc() allocates memory blocks of given size (in bytes) and returns a pointer to the beginning of the block. malloc() doesn't initialize the allocated memory. If we try to access the content of the memory block(before initializing) then we'll get segmentation fault error(or maybe garbage values). *void* malloc(size_t size);*

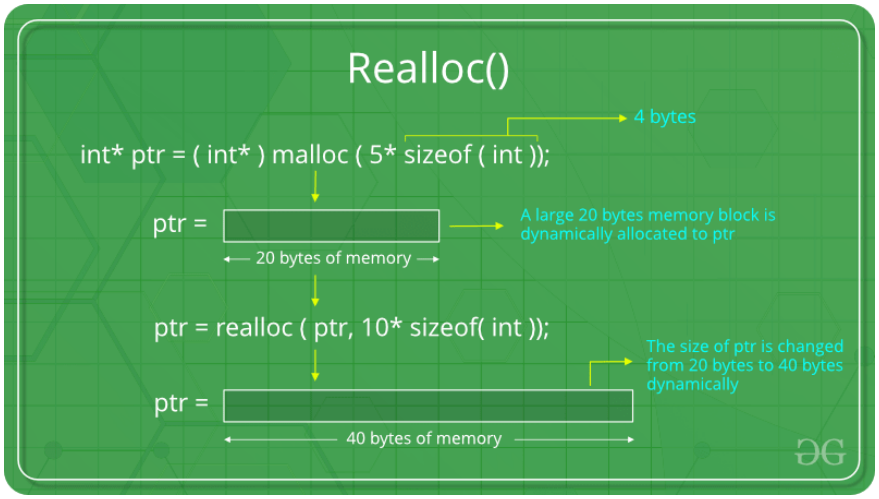
calloc() allocates the memory and also initializes the allocated memory block to zero. If we try to access the content of these blocks then we'll get 0. *void* calloc(size_t num, size_t size);*

We can achieve the same functionality as calloc() by using malloc() followed by memset().

ptr = malloc(size);
memset(ptr, 0, size);

It would be better to use malloc over calloc, unless we want the zero-initialization because malloc is faster than calloc. So if we just want to copy some stuff or do something that doesn't require filling of the blocks with zeros, then malloc would be a better choice.

realloc() *realloc() should only be used for dynamically allocated memory. Otherwise, undefined behaviour.*



free() to deallocate memory

28. Character code values can also be used in place of a char by using their octal or hexa-decimal representation. Octal values are given by a preceding '\' where the next three octal digits are converted to the corresponding character (3 octal digits suffice to get upto 256). Hexa-decimal codes are given by a preceding '\x' where the next two hexa-decimal digits are converted to the corresponding character (2 hexa-decimal digits suffice to get upto 256).

#include <stdio.h>
int main()
{
char a, b, c, d, e;
char abc[] = "I \107OT \x49T";
a = 'a';
b = '\0';
c = 0;
d = '\x41'; //41 is a hexadecimal value whose corresponding ASCII char is assigned to d
e = '\101'; //101 is an octal value whose corresponding ASCII char is assigned to e
printf("a =%d, b =%d, c =%d, d =%c e =%c abc = %s\n",a,b,c,d,e,abc);
return 0;
}
	a =97, b =0, c =0, d =A e =A abc = I GOT IT

29. **Implicit type conversion**

One important point about implicit type conversion is that it depends only on the source operands and is independent of the resultant data type. So, if we multiply two integers and store in a long, the result will be calculated as int (usually 4 bytes) and then stored in long (usually 8 bytes). Another common example of this behavior is for division operation. When we divide two integers, the result will be int only, even if we assign it to a float. So, in these cases the programmer has to explicitly cast one operand to the desired output type.

unsigned to signed or vice versa: There is no change in the representation in memory. When casting to signed, the most significant bit is taken as a sign bit which would otherwise be used for representing the number. So, this type casting is necessary during conditional checks as a negative number when type casted to unsigned will give a huge int value.

char to int: If int is 4 bytes- the top 3 bytes are filled with sign bit if character is signed and with zero if character is unsigned, and bottom most byte is the same byte used to represent the char. The same holds for conversion from short int to int or from int to long int

int to char: Only the lowermost byte of the int is taken and made to a char. So, if int value is 511, its char value will be 255.

00000000 00000000 01111111 11111111 //511
//11111111 is 255

int to float or double: The fixed integer is converted to a representable floating point value. So, this might cause a change to entire bits used to represent the integer.

float or double to int: The integral part of the floating point value is saved as integer. The decimal part is ignored. For example, 5.9 will be changed to 5.

float to double: The extra mantissa bits supported in double are filled with 0's so are the extra exponent bits. The bits of float (32 of them) are used without modification.

double to float: The extra mantissa bits supported in double are truncated in floating representation so do the extra exponent fields. If the truncated exponent field were non zeroes, it might cause a change to other mantissa bits as well as the number would then need an approximation to fit in a float size.

30. **Type conversion**

```
int main()
{
 unsigned int a = 1000;
 int b = -1;
 if (a>b) printf("A is BIG! %d\n", a-b);
 else printf("a is SMALL! %d\n", a-b);
 return 0;
}
```

Binary operations between different integral types are performed within a "common" type defined by so called usual arithmetic conversions (see the language specification, 6.3.1.8). In your case the "common" type is unsigned int. This means that int operand (your b) will get converted to unsigned int before the comparison, as well as for the purpose of performing subtraction.

On a typical implementation where int is 32-bit, -1 when converted to an unsigned int is 4,294,967,295 which is indeed ≥ 1000. Even if you treat the subtraction in an unsigned world, 1000 - (4,294,967,295) = -4,294,966,295 = 1,001 which is what you get.

When -1 is converted to unsigned int the result is the maximal possible unsigned int value (same as UINT_MAX). Needless to say, it is going to be greater than your unsigned 1000 value, meaning that a > b is indeed false and a is indeed small compared to (unsigned) b. The if in your code should resolve to the else branch, which is what you observed in your experiment.

The same conversion rules apply to subtraction. Your a-b is really interpreted as a - (unsigned) b and the result has type unsigned int. Such value cannot be printed with %d format specifier, since %d only works with signed values. Your attempt to print it with %d results in undefined behavior, so the value that you see printed (even though it has a logical deterministic explanation in practice) is completely meaningless from the point of view of C language.

```
#include <stdio.h>
int main()
{
  {
    char a = 5;
    int b = 5;
    if(a == b)
      printf("char and int compared equal\n");    }
  {
    int a = 5;
    long int b = 5;
    if(a == b)
      printf("int and long compared equal\n");
  }
  {
    float a = 5.0;
    double b = 5.0;
    if(a == b)
      printf("float and double compared equal\n");
  }
  {
    float a = 5.2;
    double b = 5.2;
    if(a == b)
      printf("float and double again compared equal\n");
  }
  {
    float a = 5.2;
    if(a == 5.2)
      printf("float compared equal with constant\n");
  }
  {
    double a = 5.2;
    if(a == 5.2)
      printf("double compared equal with constant\n");
  }
  return 0;
}
```

```
char and int compared equal
int and long compared equal
float and double compared equal
double compared equal with constant
```

31. **char array, pointer to char array**

#include <stdio.h>
int main()
{
  char *p = "Hello World";
  char q[] = "Hello World";
  printf("%zd %zd", sizeof p, sizeof *p);
  printf("\n");
  printf("%zd %zd", sizeof q, sizeof *q);
  return 0;
}
	8 1
	12 1

32. **Pointer arithmetic**

//Assume p and q are pointers

p+1 //p is incremented to the next address for holding the data type for p. i.e; if p is an int pointer p+2 will add 8 to the content of p (assuming sizeof (int) is 4)

p-1 //p is decremented to the previous address for holding the data type of p

p-q //If p and q are pointers p-q will work as above and thus will
  //return the no. of objects of type of p between the memory addresses of p and q (p and q must be of same data type or else it is compilation error)

p+q //Not allowed and throws compilation error

There are no multiplication or division operators in pointer arithmetic as they have no meaning.

Does a pointer have memory?

Yes, pointer is a data type and so has a memory for it. On a 32 bit system, pointer requires 4 bytes (32 bits) of memory while on a 64 bit system it requires 8 bytes. And it is the same for pointer to any data type. Since pointer has a memory, we can have a pointer to pointer as well and this can be extended to any level.

Assigning values to pointers

Since pointers should hold valid memory address (OS allocates certain memory region for a process and it should use only that region and should not try to access other memory region), we should not assign random values to a pointer variable. Assigning some values to a pointer is allowed, but if that value is not in the memory address for the process, dereferencing that pointer using * operator will cause segmentation fault.

One way of assigning value to a pointer variable is by using & operator on an already defined variable. Thus, the pointer variable will now be holding the address of that variable. The other way is to use malloc function which returns an array of dynamic memory created on the heap. This method is usually used to create dynamic arrays.

-->In C language, a char or void pointer can point to any other data type, but this is not true for any other data type. That is, using an int pointer to point to a char/float type is not valid (can give unexpected results) in C.

Type conversion in pointers

#include<stdio.h>
int main()
{
  int a;
  int* p = &a;
  printf("%zu", sizeof(*(char*)p));
}
	p is typecasted to char pointer and then dereferenced. So, returned type will be char and sizeof(char) is 1.

Check whether your machine is little or big endian: (code 4 works because char is 1 byte and int is 4 bytes. So, char pointer can show what's in the lowest address, say 1000. If little endian it's the LSB(yte)**)**

4. **Is the following code legal?**

```
#include<stdio.h>

int main()
{
  int a = 1;
  if((char*)  &a)
  {
    printf("My machine is little endian");
  }
  else
  {
     printf("My machine is big endian\n");
  }
}
```

- Yes
- No

→ On a little endian machine the lower address will contain the least significant byte. Suppose a is stored at address 1000 and contains 1, then character at 1000 will be 1, if the machine is little endian.

5. **What will be the output of the following code?**

```
#include<stdio.h>

int main()
{
  int *a = (int*) 1;
  printf("%d", a);
}
```

- Garbage value
- 1
- Compile error
- Segmentation fault

→ Assigning int values to pointer variable is possible. Only when we dereference the variable using *, we get a segmentation fault.

#include<stdio.h>
int main()
{
  int a = 1, *p, **pp;
  p = &a;
  pp = p;
  printf("%d", **pp);
}
	Here, p is having address of a and the same is copied to pp. So, *pp will give 1 which is contained in a, but **p will use 1 as an address and tries to access the memory location 1, giving segmentation fault.

#include<stdio.h>
int main()
{
  int a = 1, *p, **pp;
  p = &a;
  pp = p;
  printf("%d", *pp);
}
	Here, p is having address of a and the same is copied to pp. So, *pp will give 1 which is contained in a.

int is 4 bytes. So, 1 will be stored as 00000000 00000000 00000000 00000001. In little endian memory the lower address contains LSB. So, if a is stored at address 1000 (and 1001,1002, 1003 as 4 bytes needed for int in x32 architecture), the lowest address i.e. 1000 will contain the LSB(yte) 00000001. If big endian, 1003 will have the LSB 00000001 and 1000 will have 00000000(MSB for a=1).

33. printf("***%ls***") converts a wide character string to a multibyte character string, and prints it out as a sequence of bytes.
34. **Pointer and array**

To read

```
int (*a)[3];

      a           # "a is"
    (* )          # parentheses, so precedence changes.
                  # "a pointer to"
        [3]       # "an array [3] of"
int       ;       # "int".
```

For

```
int *a[3];

     a            # "a is"
      [3]         # "an array [3] of"
    *             # can't go right, so go left.
                  # "pointer to"
int      ;        # "int".
```

For

```
char *(*(*a[])())()

          a       # "a is"
            []    # "an array of"
         *        # "pointer to"
       (    )()   # "function taking unspecified number of parameters"
     (*       )   # "and returning a pointer to"
                  () # "function"
char *            # "returning pointer to char"
```

35. **Different declarations**

Rules:

Parenthesize declarations as if they were expressions.

1. Locate the innermost parentheses.
2. Say "identifier is" where the identifier is the name of the variable.
    a. Say "an array of X" if you see [X].
    b. Say "a pointer to" if you see *.
    c. Say "A function returning" if you see ();
3. Move to the next set of parentheses.
4. If more, go back to 3.
5. Else, say "type" for the remaining type left (such as short int)

`int i;`	The parenthesization of the above declaration is:
	`int (i); {1}`
	Applying the rules (see above) to the parenthesized expression can be done as follows:
	`The innermost parentheses is (i) {2}`
	`i is the variable name, therefore we say "i is ..." {3}`
	`No more parentheses left so we say "an int". {4,5,6}`
	`That is, "i is an int"`

`int *i;`	The parenthesization of the above declaration is:
	`int (*(i)); {1}`
	Applying the rules (see above) to the parenthesized expression can be done as follows:
	`The innermost parentheses is (i) {2}`
	`i is the variable name, therefore we say "i is ..." {3}`
	`Move to the next set of parenthesis: (*(i)) {4}`
	`Go back to step 3. {5}`
	`We say "a pointer to" since we see a * {3.b}`
	`No more parentheses left so we say "an int". {4,5,6}`
	`That is, "i is a pointer to an int"`

`int *i[3];`	The parenthesization of the above declaration is:

	`int (*((i)[3])); {1} // Note that () and [] have the same`
	`//  precedence but we deal with them from left to right.`
	Applying the rules (see above) to the parenthesized expression can be done as follows:
	`The innermost parentheses is (i) {2}`
	`i is the variable name, therefore we say "i is ..." {3}`
	`Move to the next set of parenthesis: ((i)[3]) {4}`
	`Go back to step 3. {5}`
	`We say "an array of 3 ..." since we see a [3] {3.a}`
	`Move to the next set of parenthesis: (*((i)[3])) {4}`
	`Go back to step 3. {5}`
	`We say "pointers to ..." since we see a * {3,b}`
	`No more parentheses left so we say "ints". {4,5,6}`
	`That is, "i is an array of 3 pointers to ints"`

`int (*i)[3];`	The parenthesization of the above declaration is:
	`int ((*(i))[3]); {1} // Note that parentheses are valid tokens`
	`//  in a declaration and therefore must be`
	`//  be left in place when finding the final`
	`//  parenthesization`
	Applying the rules (see above) to the parenthesized expression can be done as follows:
	`The innermost parentheses is (i) {2}`
	`i is the variable name, therefore we say "i is ..." {3}`
	`Move to the next set of parenthesis: (*(i)) {4}`
	`Go back to step 3. {5}`
	`We say "a pointer to ..." since we see a * {3.b}`
	`Move to the next set of parenthesis: ((*(i))[3]) {4}`
	`Go back to step 3. {5}`
	`We say "an array of 3 ..." since we see a [3] {3.a}`
	`No more parentheses left so we say "ints". {4,5,6}`
	`That is, "i is a pointer to an array of 3 ints"`

`int *i();`	The parenthesization of the above declaration is:
	`int (*((i)())); {1} // Note that * has a lower precedence than`
	`//  parentheses, ()`
	Applying the rules (see above) to the parenthesized expression can be done as follows:
	`The innermost parentheses is (i) {2}`
	`// One could argue that () is also the innermost parenthesis but`
	`//  it does not contain anything so we know it must indicate`
	`//  a function`
	`i is the variable name, therefore we say "i is ..." {3}`
	`Move to the next set of parenthesis: ((i)()) {4}`
	`Go back to step 3. {5}`
	`We say "a function returning" since we see a () {3.c}`
	`Move to the next set of parenthesis: (*((i)())) {4}`
	`Go back to step 3. {5}`
	`We say "a pointer to ..." since we see a * {3.b}`

	No more parentheses left so we say "an int". {4,5,6} That is, **"i is a function returning a pointer to an int"**

`int (*i)();`	The parenthesization of the above declaration is:
	`int ((*(i))()); {1} // Note that parentheses are valid tokens`
	`// in a declaration and therefore must be`
	`// be left in place when finding the final`
	`// parenthesization`
	Applying the rules (see above) to the parenthesized expression can be done as follows:
	The innermost parentheses is (i) {2}
	i is the variable name, therefore we say "i is ..." {3}
	Move to the next set of parenthesis: (*(i)) {4}
	Go back to step 3. {5}
	We say "a pointer to" since we see a * {3.b}
	Move to the next set of parenthesis: ((*(i))()) {4}
	Go back to step 3. {5}
	We say "a function returning" since we see a () {3.c}
	No more parentheses left so we say "an int". {4,5,6}
	That is, **"i is a pointer to a function returning an int"**

36. **printf, fprintf** http://www.cplusplus.com/reference/cstdio/printf/

Upon successful return, these functions return the number of characters printed (excluding the null byte used to end output to strings).

scanf, fscanf http://www.cplusplus.com/reference/cstdio/scanf/

On success, these functions return the number of input items successfully matched and assigned; this can be fewer than provided for, or even zero, in the event of an early matching failure.

The value EOF is returned if the end of input is reached before either the first successful conversion or a matching failure occurs. EOF is also returned if a read error occurs, in which case the error indicator for the stream (see ferror(3)) is set, and errno is set to indicate the error.

37. **scanf("%*s") and scanf("%*d") format identifiers**

int a = 12222;
int k = printf("%*d", 9, a);
printf ("\n%d\n", k);
return 0;
	12222 ← 9 field width printed
	9

int a = 12222;
int k = printf("%*d%*d%*d%*d", 9, a, 9, a, 9, a, 9, a);
printf ("\n%d\n", k);
return 0;
	12222 12222 12222 12222
	36

int a; int k = scanf ("%*d %d", &a); printf ("a %*d\nk %*d\n", 5, a, 5, k);	5488 887 a ← ip a 887 k 1

int a;	sgfiugf 12 ← ip

int k = scanf("%*s %d", &a); printf ("a: %d", a); printf ("\nk: %d\n", k);	a: 12 k: 1
int a; char s[10]; int k = scanf("%d %s", &a, s); printf ("%d\n", k);	123 abc 2

38. **printf format identifiers**

https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/format.html

39. **%*s %*d usage in printf and scanf**

https://stackoverflow.com/a/2155564/9428070

https://www.quora.com/What-does-*s-mean-in-printf-in-C

printf

Width field

The Width field specifies a minimum number of characters to output, and is typically used to pad fixed-width fields in tabulated output, where the fields would otherwise be smaller, although it does not cause truncation of oversized fields. The width field may be omitted, or a numeric integer value, or a dynamic value when passed as another argument when indicated by an asterisk *.[3] For example, printf("%*d", 5, 10) will result in 10 being printed, with a total width of 5 characters. Though not part of the width field, a leading zero is interpreted as the zero-padding flag mentioned above, and a negative value is treated as the positive value in conjunction with the left-alignment - flag also mentioned above.

Precision field

The Precision field usually specifies a maximum limit on the output, depending on the particular formatting type. For floating point numeric types, it specifies the number of digits to the right of the decimal point that the output should be rounded. For the string type, it limits the number of characters that should be output, after which the string is truncated. The precision field may be omitted, or a numeric integer value, or a dynamic value when passed as another argument when indicated by an asterisk *. For example, printf("%.*s", 3, "abcdef") will result in abc being printed.

scanf

An optional starting asterisk indicates that the data is to be read from the stream but ignored (i.e. it is not stored in the location pointed by an argument). %*s ignores characters while %*d ignores ints.
It is useful in situations when the specification string contains more than one element, eg.: scanf("%d %*s %d", &i, &j) for the "12 test 34" - where i & j are integers and you wish to ignore the rest.

40. **lvalue rvalue**

https://qr.ae/pNszf1

→ lvalues are “left values”: expressions that can be assigned to, which can be on the left of an assignment

The name of the variable of any type i.e, an identifier of integral, floating, pointer, structure, or union type.
A subscript ([ ]) expression that does not evaluate to an array.
A unary-indirection (*) expression that does not refer to an array
An l-value expression in parentheses.
A const object (a non modifiable l-value).
The result of indirection through a pointer, provided that it isn’t a function pointer.
The result of member access through pointer(-> or .)

→ rvalues are “right values”: everything else, which must be on the right of an assignment

My address ← Kolkata My address is kolkata. Kolkata’s address is ??? Not answerable! We can assign a place to *my address* but nothing can be assigned to Kolkata.

An lvalue (locator value) represents an object that occupies some identifiable location in memory (i.e. has an address).

rvalues are defined by exclusion, by saying that every expression is either an lvalue or an rvalue. Therefore, from the above definition of lvalue, an rvalue is an expression that does not represent an object occupying some identifiable location in memory.

int foo() { return 4; }
int main(void) {
  int* f = &foo();
  return 0;
}
	% clang lvalue.c
	lvalue.c:3:12: error: cannot take the address of an rvalue of type 'int'
	  int* f = &foo();
	  ^~~~~
	1 error generated.

Because rvalues are ones which do not represent an object occupying some identifiable location in memory", we cannot use the & (addressof) operator on them.

int main(void) {
  void* f = &"some string";
  return 0;
}
	It turns out this one is just fine, because … string literals are lvalues! When we write "some string", we reserve memory for that string in an addressable location.

→ The unary & (address-of) operator requires an lvalue as its operand. That is, &n is a valid expression only if n is an lvalue. Thus, an expression such as &12 is an error. Again, 12 does not refer to an object, so it's not addressable.

```
int a, *p;
p = &a; // ok, assignment of address
        // at l-value
&a = p;    // error: &a is an r-value
```

41. **perror**

The C library function void perror(const char *str) prints a descriptive error message to stderr. First the string str is printed, followed by a colon then a space.

#include <stdio.h>
int main () {
  FILE *fp;
 	
  /* first rename if there is any file */
  rename("file.txt", "newfile.txt");
 	
  /* now let's try to open same file */
  fp = fopen("file.txt", "r");
  if(fp == NULL) {
  perror("Error: ");
  return(-1);
  }
  fclose(fp);
  return(0);
}
	Error: : No such file or directory

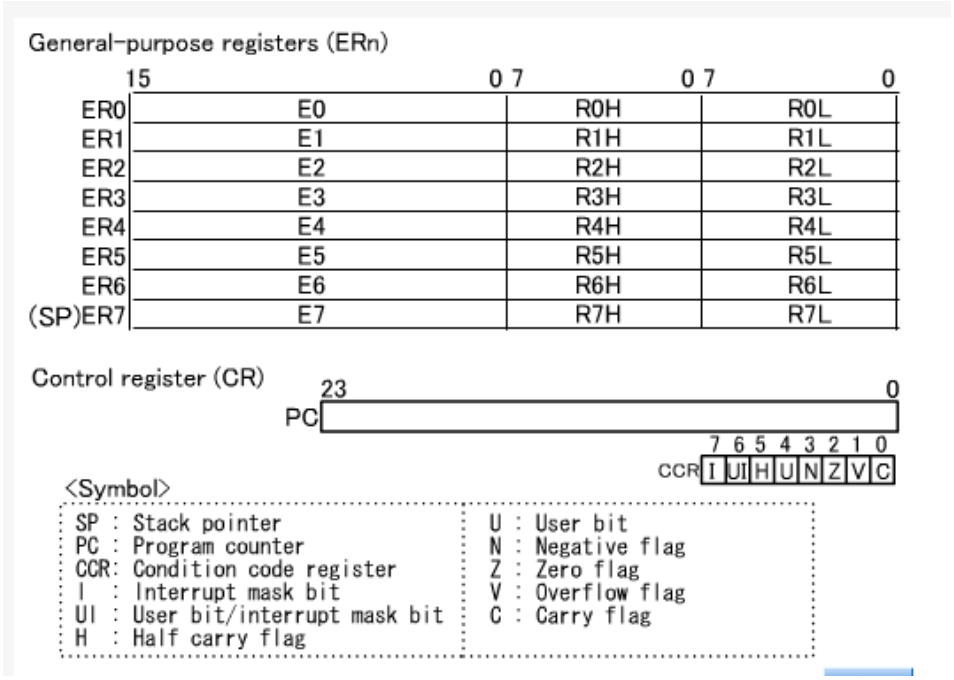
42. **C Program to print environment variables**

```
#include <stdio.h>
// Most of the C compilers support a third parameter to main which
// store all environment variables
int main(int argc, char *argv[], char * envp[])
{
        int i;
        for (i = 0; envp[i] != NULL; i++)
                printf("\n%s", envp[i]);
        getchar();
        return 0;
}
```

```
OS=Windows_NT
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 42 Stepping 7, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=2a07
ProgramData=C:\ProgramData
ProgramFiles=C:\Program Files
PUBLIC=C:\Users\Public
SESSIONNAME=Console
SystemDrive=C:
SystemRoot=C:\Windows
WATCOM=C:\watcom
windir=C:\Windows
```

43. **CPU Internal Registers**

http://resource.renesas.com/lib/eng/e_learnig/h8_300henglish/s04/bf01.html#:~:text=The%20control%20register%20is%20further,code%20register)%20to%20test%20conditions.&text=The%20CPU%20has%208%20general,%2D%20or%208%2Dbit%20data.



- **Accumulator:**

  This is the most frequently used register used to store data taken from memory. It is in different numbers in different microprocessors.
- **Memory Address Registers (MAR):**

  It holds the address of the location to be accessed from memory. MAR and MDR (Memory Data Register) together facilitate the communication of the CPU and the main memory.
- **Memory Data Registers (MDR):**

  It contains data to be written into or to be read out from the addressed location.
- **General Purpose Registers:**

  These are numbered as R0, R1, R2….Rn, and used to store temporary data during any ongoing operation. Its content can be accessed by assembly programming.
- **Program Counter (PC):**

  Program Counter (PC) is used to keep the track of execution of the program. It contains the memory address of the next instruction to be fetched. PC

points to the address of the next instruction to be fetched from the main memory when the previous instruction has been successfully completed.

Program Counter (PC) also functions to count the number of instructions.

- **Instruction Register (IR):**

  It is the register which holds the instruction which is currently being executed.

44. **while((c = getchar()) != EOF)**

Note that the expression inside the while argument both assigns the return value of getchar to c and tests to see if it is equal to EOF (which is returned when no more input characters are available). This is a very common idiom in C programs. Note also that even though c holds a single character, it is declared as an int. The reason is that EOF (a constant defined in stdio.h) is outside the normal character range, and if you assign it to a variable of type char it will be quietly truncated into something else. Because C doesn't provide any sort of exception mechanism for signalling unusual outcomes of function calls, designers of library functions often have to resort to extending the output of a function to include an extra value or two to signal failure.

45. **Print randomly 0 or 1**

```
srandom(time(0));  /* initialize random number generator */
printf ("%ld\n", random()&(0x1));
```

x & 0x1 : This can be used for selecting bits. For example select the least significant bit of x. (0x for HEX)

46. **break, continue, goto**

The break statement immediately exits the innermost enclosing loop or switch statement. The continue statement skips to the next iteration.

Here is a program with a loop that iterates through all the integers from -10 through 10, skipping 0:

```
#define MAXN (10)
int main(int argc, char **argv)
{
    int n;
    for(n = -MAXN; n <= MAXN; n++) {
        if(n == 0) continue;
        printf("1.0/%3d = %+f\n", n, 1.0/n);
    }
    return 0;
}
```

Occasionally, one would like to break out of more than one nested loop. The way to do this is with a goto statement.

```
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            doSomethingTimeConsumingWith(i, j);
            if(checkWatch() == OUT_OF_TIME) {
                goto giveUp;
            }
        }
    }
giveUp:
    puts("done");
```

47. **Choosing where to put a loop exit**

Choosing where to put a loop exit is usually pretty obvious: you want it after any code that you want to execute at least once, and before any code that you want to execute only if the termination test fails.

If you know in advance what values you are going to be iterating over, you will most likely be using a for loop:

```
for(i = 0; i < n; i++) {
    a[i] = 0;
}
```

Most of the rest of the time, you will want a while loop:

```
while(!done()) {
    doSomething();
}
```

The do..while loop comes up mostly when you want to try something, then try again if it failed:

```
do {
    result = fetchWebPage(url);
} while(result == 0);
```

Finally, leaving a loop in the middle using break can be handy if you have something extra to do before trying again:

```
for(;;) {
    result = fetchWebPage(url);
    if(result != 0) {
        break;
    }
    /* else */
    fprintf(stderr, "fetchWebPage failed with error code %03d\n", result);
    sleep(retryDelay);  /* wait before trying again */
}
```

(Note the empty for loop header means to loop forever; while(1) also works.)

48. **Passing void vs no parameter in a c function**

void helloWorld(void)
{
  puts("hi");
}
int main(int argc, char **argv)
{
  helloworld("sds");
  return 0;
}
	Compilation error

void helloWorld()
{
  puts("hi");
}
int main(int argc, char **argv)
{
  helloworld("sds");
  return 0;
}
	Compiles successfully

49. **fgets vs fscanf**

https://stackoverflow.com/questions/8917550/difference-between-fgets-and-fscanf

50. **Return statement**

Executing a return statement immediately terminates the function. This can be used like break to get out of loops early.

51. By default, **functions have global scope:** they can be used anywhere in your program, even in other files. If a file doesn't contain a declaration for a function someFunc before it is used, the compiler will assume that it is declared like int someFunc() (i.e., return type int and unknown arguments).

52. **static functions:** By default, all functions are global; they can be used in any file of your program whether or not a declaration appears in a header file. To restrict access to the current file, declare a **function static**, like this:

```
static void
helloHelper(void)
{
   puts("hi!");
}
void
hello(int repetitions)
{
   int i;
   for(i = 0; i < repetitions; i++) {
      helloHelper();
   }
}
```

The function hello will be visible everywhere. The function helloHelper will only be visible in the current file. It's generally good practice to declare a function static unless you intend to make it available, since not doing so can cause namespace conflicts, where the presence of two functions with the same name either prevent the program from linking or—even worse—cause the wrong function to be called.

53. **Tail recursion:**

https://www.geeksforgeeks.org/tail-recursion/
https://www.geeksforgeeks.org/tail-call-elimination/

Quick sort is tail recursive but Merge sort is not.
https://www.geeksforgeeks.org/quicksort-tail-call-optimization-reducing-worst-case-space-log-n/

54. **Mechanics of function calls**

Several things happen under the hood when a function is called. Since a function can be called from several different places, the CPU needs to store its previous state to know where to go back. It also needs to allocate space for function arguments and local variables.
Some of this information will be stored in registers, memory locations built into the CPU itself, but most will go on the stack, a region of memory that on typical machines grows downward, even though the most recent additions to the stack are called the "top" of the stack. The location of the top of the stack is stored in the CPU in a special register called the stack pointer.
So a typical function call looks like this internally:

The current instruction pointer or program counter value, which gives the address of the next line of machine code to be executed, is pushed onto the stack.
Any arguments to the function are copied either into specially designated registers or onto new locations on the stack. The exact rules for how to do this vary from one CPU architecture to the next, but a typical convention might be that the first few arguments are copied into registers and the rest (if any) go on the stack.

The instruction pointer is set to the first instruction in the code for the function.
The code for the function allocates additional space on the stack to hold its local variables (if any) and to save copies of the values of any registers it wants to use (so that it can restore their contents before returning to its caller).
The function body is executed until it hits a return statement.
Returning from the function is the reverse of invoking it: any saved registers are restored from the stack, the return value is copied to a standard register, and the values of the instruction pointer and stack pointer are restored to what they were before the function call.

From the programmer's perspective, the important point is that both the arguments and the local variables inside a function are stored in freshly-allocated locations that are thrown away after the function exits. So after a function call the state of the CPU is restored to its previous state, except for the return value. Any arguments that are passed to a function are passed as copies, so changing the values of the function arguments inside the function has no effect on the caller. Any information stored in local variables is lost.

Under very rare circumstances, it may be useful to have a variable local to a function that persists from one function call to the next. You can do so by declaring the variable static.

Static local variables are stored outside the stack with global variables, and have unbounded extent. But they are only visible inside the function that declares them. This makes them slightly less dangerous than global variables—there is no fear that some foolish bit of code elsewhere will quietly change their value—but it is still the case that they usually aren't what you want. It is also likely that operations on static variables will be slightly slower than operations on ordinary ("automatic") variables, since making them persistent means that they have to be stored in (slow) main memory instead of (fast) registers.

55. There are two standard **ways to represent strings**:

As a **delimited string**, where the end of a string is marked by a special character. The advantages of this method are that only one extra byte is needed to indicate the length of an arbitrarily long string, that strings can be manipulated by simple pointer operations, and in some cases that common string operations that involve processing the entire string can be performed very quickly. The disadvantage is that the delimiter can't appear inside any string, which limits what kind of data you can store in a string. C uses a delimited string.

As a **counted string**, where the string data is prefixed or supplemented with an explicit count of the number of characters in the string. The advantage of this representation is that a string can hold arbitrary data (including delimiter characters) and that one can quickly jump to the end of the string without having to scan its entire length. The disadvantage is that maintaining a separate count typically requires more space than adding a one-byte delimiter (unless you limit your string length to 255 characters) and that more care needs to be taken to make sure that the count is correct.

56. **Strings in C** string functions implementation: https://www.cs.umd.edu/~hollings/cs412/s03/prog1/string.c

A string is represented by a variable of type char *, which points to the zeroth character of the string. The value of a string constant has type const char *, and can be assigned to variables and passed as function arguments or return values of this type. Two string constants separated only by whitespace will be concatenated by the compiler as a single constant: "foo" "bar" is the same as "foobar". This feature is not much used in normal code, but shows up sometimes in macros.

Standard C strings are assumed to be in ASCII, a 7-bit code developed in the 1960s to represent English-language text. If you want to write text that includes any letters not in the usual 26-letter Latin alphabet, you will need to use a different encoding. C does not provide very good support for this, but for fixed strings, you can often get away with using Unicode as long as both your text editor and your terminal are set to use the UTF-8 encoding.

char hi[3]; hi[0] = 'h'; hi[1] = 'i'; hi[2] = 'i'; hi[3] = 'i';	Compiled successfully but Stack smashing while running

→ Stack smashing happens whenever we try to store characters in a string of size less than that we are trying to give input. E.g. *char str[3]; fscanf(stdin, "%s", str); fprintf(stdout, "%s", str)* --- when input size is > 3 like *abcd*

57. **strcpy string copy function implementation (famous C idiom!)**

```
void
strcpy2(char *dest, const char *src)
{
    /* This line copies characters one at a time from *src to *dest. */
    /* The postincrements increment the pointers (++ binds tighter than *) */
    /*  to get to the next locations on the next iteration through the loop. */
    /* The loop terminates when *src == '\0' == 0. */
    /* There is no loop body because there is nothing to do there. */
    while(*dest++ = *src++);
}
```

58. **Scanline function implementation- scanning characters from stdin one by one**

```
int scanline (char str [], int lim)                    /* Line will be read in 'str []', while lim is the maximum characters to be read */
{
   int c, len, j;                                      /* 'len' will have the length of the read string */

   j = 0;                                              /* Initializing 'j' */
   for (len = 0; (c = getchar ()) != EOF && c != '\n'; ++len)   /* Reading a character one by one, till the user enters '\n', and checking for failure of 'getchar' */
   {
      if (len < (lim -2))                              /* Checking that string entered has not gone beyond its boundaries. '-2' for '\n' and '\0' */
      {
         str [j] = c;                                  /* Copying read character into 'string [j]' */
         ++ j;                                         /* Incrementing 'j' by 1 */
      }
   }
   if (c == '\n')                                      /* Checking if user has finished inputting the line */
   {
      str [j] = c;                                     /* Copying newline into string */
      ++j;
      ++ len;
   }

   return len;                                         /* Returning number of characters read */
}
```

59. **Overflow problem (Very nice example and explanation)**

```
void f3()
{
  char str1[5], str2[5];

  strcpy(str1, "hello");
  strcpy(str2, "test");
  printf("str1=%s\n", str1);
}

int
main()
{
  f3();
  return 0;
}
```

Let's suppose the compiler decides to place str2 immediately after str1 in memory (although it doesn't have to). The first call to strcpy will write the string "hello" into str1, but this variable doesn't have enough room the the null terminating byte. So it gets written to the next byte in memory, which happens to be the first byte of str2. Then when the next call to strcpy runs, it puts the string "test" in str2 but in doing so it overwrites the null terminating byte put there when str1 was written to. Then when printf gets called, you'll get this as output:

str1=hellotest

When printing str1, printf looks for the null terminator, but there isn't one inside of str1. So it keeps reading until it does. In this case there happens to be another string right after it, so it prints that as well until it finds the null terminator that was properly stored in that string.

60. **strcpy strncpy strlcpy strlcat**

Problem with strcpy(): The strcpy() function does not specify the size of the destination array, so buffer overrun is often a risk. Using strcpy() function to copy a large character array into a smaller one is dangerous, but if the string will not fit, then it will not worth the risk. If the destination string is not large enough to store the source string then the behavior of strcpy() is unspecified or undefined.

The strncpy() function is similar to strcpy() function, except that at most n bytes of src are copied. If there is no NULL character among the first n characters of src, the string placed in dest will not be NULL-terminated. If the length of src is less than n, strncpy() writes additional NULL characters to dest to ensure that a total of n characters are written. Syntax: char *strncpy(char *dest, const char *src, size_t n)

Problem with strncpy(): If there is no null character among the first n character of src, the string placed in dest will not be null-terminated. So strncpy() does not guarantee that the destination string will be NULL terminated. The strlen() non-terminated string can cause segfault. In other words, non-terminated string in C/C++ is a time-bomb just waiting to destroy code.

The **strlcpy()** and **strlcat()** functions copy and concatenate strings with the same input parameters and output result as snprintf(3). They are de- signed to be safer, more consistent, and less error prone replacements for the easily misused functions strncpy(3) and strncat(3). strlcpy() and strlcat() take the full size of the destination buffer and guarantee NUL-termination if there is room. Note that room for the NUL should be included in dstsize. strlcpy() copies up to dstsize - 1 characters from the string src to dst, NUL-terminating the result if dstsize is not 0. strlcat() appends string src to the end of dst. It will append at most dstsize - strlen(dst) - 1 characters. It will then NUL-terminate, unless dstsize is 0 or the original dst string was longerthan dstsize (in prac- tice this should not happen as it means that either dstsize is incorrect or that dst is not a proper string). If the src and dst strings overlap, the behavior is undefined.

61. **strcat string concatenation function implementation (famous C idiom!)**

```
void
strcat2(char *dest, const char *src)
{
   while(*dest) dest++;
   while(*dest++ = *src++);
}
```

62. **strlen implementation**

```
int
strlen(const char *s)
{
   int i;
   for(i = 0; *s; i++, s++);
   return i;
}
```

63. **strcmp implementation**

```
int
strcmp(const char *s1, const char *s2)
{
   while(*s1 && *s2 && *s1 == *s2) {
      s1++;
      s2++;
   }
   return *s1 - *s2;
}
```

64. **strdup strndup**

char *strdup(const char *s);

This function returns a pointer to a null-terminated byte string, which is a duplicate of the string pointed to by s. The memory obtained is done dynamically using malloc and hence it can be freed using free().

It returns a pointer to the duplicated string s.

This function is similar to strdup(), but copies at most n bytes.

65 . Global variables and static local variables are guaranteed to be initialized to an all-0 pattern, which will give the value 0 for most types.

66. **Pointers in C**

On a typical 64-bit machine, each pointer will be allocated 8 bytes(64 bits), enough to represent an address in memory. sizeof (pointer_var) = 8

* acts as a dereference operator and is also used to declare pointer variables.

*(dereference) generally works opposite of &(address-of).

The * operator binds very tightly, so you can usually use *p anywhere you could use the variable it points to without worrying about parentheses. However, a few operators, such as the -- and ++ operators and the . operator used to unpack structs, bind tighter. These require parentheses if you want the * to take precedence.

```
(*p)++;          /* increment the value pointed to by p */
*p++;            /* WARNING: increments p itself */
```

You can print a pointer value using printf with the %p format specifier. To do so, you should convert the pointer to the generic pointer type void * first using a cast, although on machines that don't have different representations for different pointer types, this may not be necessary. printf("&a = %p\n", (void *) &a);

In C, you cannot take the address of a variable with *register* storage class. Precisely because it's a register. An address is an address of a memory location. If something is resident in a register it is by definition not in main memory.

In C, the *register* keyword retains a meaning. Here, we are not, for example, allowed to take the address of an object (see the quoted footnote). An implementation may ignore the register hint, and place the object in memory anyways, but the keyword does restrict what you can do with the object. These restrictions should enable a compiler to better optimize access to the object, but it's also possible (like it is suggested in the C++ case), that the compiler will be able to infer this anyways.

In C++, the *register* keyword has no meaning. It does function as a compiler hint, but it's suggested that most compilers will ignore that hint anyways.

If you want to write a function that takes a pointer argument but promises not to modify the target of the pointer, use const, like this:

```
void
printPointerTarget(const int *p)
{
   printf("%d\n", *p);
}
```

If you really want to modify the target anyway, C lets you "cast away const":

```
void
printPointerTarget(const int *p)
{
   *((int *) p) = 5;  /* no compile-time error */
   printf("%d\n", *p);
}
```

***Note that while it is safe to pass pointers down into functions, it is very dangerous to pass pointers up. The reason is that the space used to hold any local variable of the function will be reclaimed when the function exits, but the pointer will still point to the same location, even though something else may now be stored there. So this function is very dangerous:

```
int *
dangerous(void)
{
   int n;

   return &n;          /* NO! */
}
...
   *dangerous() = 12;  /* writes 12 to some unknown location */
```

An exception is when you can guarantee that the location pointed to will survive even after the function exits, e.g. when the location is dynamically allocated using malloc (see below) or when the local variable is declared static:

```
int *
returnStatic(void)
{
   static int n;

   return &n;
}
...
   *returnStatic() = 12;       /* writes 12 to the hidden static variable */
```

Usually returning a pointer to a static local variable is not good practice, since the point of making a variable local is to keep outsiders from getting at it. If you find yourself tempted to do this, a better approach is to allocate a new block using malloc (see below) and return a pointer to that. The downside of the malloc method is that the caller has to promise to call free on the block later, or you will get a storage leak.

- Because pointers are just numerical values, one can do arithmetic on them. Specifically, it is permitted to

Add an integer to a pointer or subtract an integer from a pointer. The effect of p+n where p is a pointer and n is an integer is to compute the address equal to p plus n times the size of whatever p points to (this is why int * pointers and char * pointers aren't the same).
Subtract one pointer from another. The two pointers must have the same type (e.g. both int * or both char *). The result is a signed integer value of type ptrdiff_t, equal to the numerical difference between the addresses divided by the size of the objects pointed to.
Compare two pointers using ==, !=, <, >, <=, or >=.
Increment or decrement a pointer using ++ or --.

- int a[10] here *a* and *&a* both point to the first element of array *a.* Also, *a[n]* means **(a+n).*
- Curious feature of the definition of a[n] as identical to *(a+n) is that it doesn't actually matter which of the array names or the index goes inside the braces. *So all of a[0], *a, and 0[a] refer to the zeroth entry in a.*

int a[4] = {1, 2, 3, 4}; printf ("%d\n", a[0]); printf ("%d\n", *a); printf ("%d\n", 0[a]) <---------check! printf ("%d\n", *a); printf ("%d\n", 1[a])	1 1 1 2 2

- Note that C doesn't do any sort of bounds checking. Given the declaration int a[50];, only indices from a[0] to a[49] can be used safely. However, the compiler will not blink at a[-12] or a[10000]. If you read from such a location you will get garbage data; if you write to it, you will overwrite god-knows-what, possibly trashing some other variable somewhere else in your program or some critical part of the stack (like the location to jump to when you return from a function). It is up to you as a programmer to avoid such buffer overruns, which can lead to very mysterious (and in the case of code that gets input from a network, security-damaging) bugs. The valgrind program can help detect such overruns in some cases.

- **Pointers to void**

A special pointer type is void *, a "pointer to void". Such pointers are declared in the usual way:

```
void *nothing;      /* pointer to nothing */
```

Unlike ordinary pointers, you can't dereference a void * pointer or do arithmetic on it, because the compiler doesn't know what type it points to. However, you are allowed to use a void * as a kind of "raw address" pointer value that you can store arbitrary pointers in. It is permitted to assign to a void * variable from an expression of any pointer type; conversely, a void * pointer value can be assigned to a pointer variable of any type. An example is the return value of malloc or the argument to free, both of which are declared as void *.

```
int *block;
block = malloc(sizeof(int) * 12);  /* void * converted to int * before assignment */
free(block);                       /* int * converted to void * before passing to free */
```

- **Dangling, Void , Null and Wild Pointer in C**
  https://aticleworld.com/dangling-void-null-wild-pointers/
  https://www.geeksforgeeks.org/dangling-void-null-wild-pointers/

- A **function pointer**, internally, is just the numerical address for the code for a function. When a function name is used by itself without parentheses, the value is a pointer to the function, just as the name of an array by itself is a pointer to its zeroth element. Function pointers can be stored in variables, structs, unions, and arrays and passed to and from functions just like any other pointer type. They can also be called: a variable of type function pointer can be used in place of a function name.

int
main(int argc, char **argv)
{
  int (*say)(const char *);
  say = puts;
  say("hello world");
  return 0;
}
	hello world

- **Callbacks, Dispatch tables, restrict keyword : cs.yale**
- void BadPointer() {
  int* p;
  // allocate the pointer, but not the pointee
  *p = 42;
  // this dereference is a serious runtime error
  }
  // What happens at runtime when the bad pointer is dereferenced…
- **A pointer may only be dereferenced after it has been assigned to refer to a pointee. Most pointer bugs involve violating this one rule.**

67. **What and where are the stack and heap?**

https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap

The OS allocates the stack for each system-level thread when the thread is created. Typically the OS is called by the language runtime to allocate the heap for the application.

The stack is faster because the access pattern makes it trivial to allocate and deallocate memory from it (a pointer/integer is simply incremented or decremented), while the heap has much more complex bookkeeping involved in an allocation or deallocation. Also, each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast. Another performance hit for the heap is that the heap, being mostly a global resource, typically has to be multi-threading safe, i.e. each allocation and deallocation needs to be - typically - synchronized with "all" other heap accesses in the program.

Stack:

- Stored in computer RAM just like the heap.
- Variables created on the stack will go out of scope and are automatically deallocated.
- Much faster to allocate in comparison to variables on the heap.
- Implemented with an actual stack data structure.
- Stores local data, return addresses, used for parameter passing.
- Can have a stack overflow when too much of the stack is used (mostly from infinite or too deep recursion, very large allocations).
- Data created on the stack can be used without pointers.
- You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.
- Usually has a maximum size already determined when your program starts.

Heap:

- Stored in computer RAM just like the stack.
- In C++, variables on the heap must be destroyed manually and never fall out of scope. The data is freed with delete, delete[], or free.
- Slower to allocate in comparison to variables on the stack.
- Used on demand to allocate a block of data for use by the program.
- Can have fragmentation when there are a lot of allocations and deallocations.
- In C++ or C, data created on the heap will be pointed to by pointers and allocated with new or malloc respectively.
- Can have allocation failures if too big of a buffer is requested to be allocated.
- You would use the heap if you don't know exactly how much data you will need at run time or if you need to allocate a lot of data.
- Responsible for memory leaks.

68. In high level languages such as Java, there are functions which **prevent you from accessing arrays out of bound** by generating an exception such as java.lang.ArrayIndexOutOfBoundsException. But in case of C, there is no such functionality, so programmers need to take care of this situation.

69. **Structured data types:**

C has two kinds of structured data types: **struct** s and **union** s. A **struct holds multiple values in consecutive memory locations**, called fields, and implements what in type theory is called a **product type**: the set of possible values is the Cartesian product of the sets of possible values for its fields. In contrast, a **union has multiple fields but they are all stored in the same location**: effectively, this means that **only one field at a time can hold a value**, making a union a sum type whose set of possible values is the union of the sets of possible values for each of its fields.

struct sample strptr;*
struct sample str;
strptr = &str;
*(*strptr).data*
strptr->data It is an error to write *sp.data in this case; since . binds tighter than *

- You can find the position of a component within a struct using the **offsetof** macro, which is defined in stddef.h . This returns the number of bytes from the base of the struct that the component starts at, and can be used to do various terrifying non-semantic things with pointers.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
int
main(int argc, char **argv)
{
    struct foo {
        int i;
        char c;
        double d;
        float f;
        char *s;
    };
    printf("i is at %lu\n", offsetof(struct foo, i));
    printf("c is at %lu\n", offsetof(struct foo, c));
    printf("d is at %lu\n", offsetof(struct foo, d));
    printf("f is at %lu\n", offsetof(struct foo, f));
    printf("s is at %lu\n", offsetof(struct foo, s));
    return 0;
}
```

```
i is at 0
c is at 4
d is at 8
f is at 16
s is at 24
```

- It is possible to specify the exact number of bits taken up by a member of a struct of integer type. This is seldom useful, but may in principle let you pack more information in less space. Bit fields are sometimes used to unpack data from an external source that uses this trick, but this is dangerous, because there is no guarantee that the compiler will order the bit fields in your struct in any particular order (at the very least, you will need to worry about endianness (http://en.wikipedia.org/wiki/Endianness).

```
struct color {
        unsigned int red : 2;
        unsigned int green : 2;
        unsigned int blue : 2;
        unsigned int alpha : 2;
}
```

This defines a struct that (probably) occupies only one byte, and supplies four 2-bit fields, each of which can hold values in the range 0-3.

- A **union** is a special data type available in C that allows to *store different data types in the same memory location*. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional.

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union −

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

Memory size occupied by data : 20

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming

Here, we can see that the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

data.i : 10
data.f : 220.500000
data.str : C Programming

- C provides the enum construction for the special case where you want to have a sequence of named constants of type int , but you don't care what their actual values are, as in

enum color { RED, BLUE, GREEN, MAUVE, TURQUOISE };

This will assign the value 0 to RED , 1 to BLUE , and so on. These values are effectively of type int , although you can declare variables, arguments, and return values as type enum color to indicate their intended interpretation. Despite declaring a variable enum color c (say), the compiler will still allow c to hold arbitrary values of type int .

70. **typedef:** typedef struct string *String;

The syntax for typedef looks like a variable declaration preceded by typedef , except that the variable is replaced by the new type name that acts like whatever type the defined variable would have had. You can use a name defined with typedef anywhere you could use a normal type name, as long as it is later in the source file than the typedef definition. Typically typedef s are placed in a header file (.h file) that is then included anywhere that needs them. You are not limited to using typedef s only for complex types. For example, if you were writing numerical code and wanted to declare overtly that a certain quantity was not just any double but actually a length in meters, you could write typedef double LengthInMeters; typedef double AreaInSquareMeters; AreaInSquareMeters rectangleArea(LengthInMeters height, LengthInMeters width);

71. **Macros:**

To create a macro with arguments, put them in parentheses separated by commas after the macro name, e.g. #define Square(x) ((x)*(x)) Now if you write Square(foo) it will expand as ((foo)*(foo)) . Note the heavy use of parentheses inside the macro definition to avoid trouble with operator precedence; if instead we had written #define BadSquare(x) x*x then BadSquare(3+4) would give 3+4*3+4 , which evaluates to 19 , which is probably not what we intended. The general rule is that macro arguments should always be put in parentheses if you are using them in an expression where precedence might be an issue.

We can have *multiple arguments* too.

#define Average(x,y) (((x)+(y))/2.0)

Multiple expressions in a macro:

#define NoisyInc(x) (puts("incrementing"), (x)++)
#define Max(a,b) ((a) > (b) ? (a) : (b))

Non syntactic macros:

#define UpTo(i, n) for((i) = 0; (i) < (n); (i)++)
#define TestMalloc(x) ((x) = malloc(sizeof(*x)), assert(x))

Multiple statements in one macro:

#define HiHi() do { puts("hi"); puts("hi"); } while(0)

Note that no construct except do..while will work here. Just using braces will cause trouble with the semicolon before the else , and no other compound statement besides do..while expects to be followed by a semicolon in this way.

String expansion:

#define NoisyInc2(x) (puts("Incrementing " #x), x++)

Here #x expands to whatever the value of x is wrapped in double quotes. The resulting string constant is then concatenated with the adjacent string constant according to standard C string constant concatenation rules.

To concatenate things that aren't strings, use the ## operator, as in

#define FakeArray(n) fakeArrayVariableNumber ## n

This lets you write FakeArray(12) instead of fakeArrayVariableNumber12 . Note that there is generally no good reason to ever do this.

Where this feature does become useful is if you want to be able to refer to part of the source code of your program. For example, here is short program that includes a macro that prints the source code and value of an expression:

```
#define PrintExpr(x) (printf("%s = %d\n", #x, (x)))
int
main(int argc, char **argv)
{
PrintExpr(2+2);
return 0;
}
```

```
2+2 = 4
```

assert macro:

```
#include <assert.h>
void assert(int expression);
```

assert.h defines the macro assert. The macro can be used to verify assumptions made by the program and print a diagnostic message if this assumption is false.

When executed, if the expression is false (that is, compares equal to 0), assert writes information about the call that failed on stderr and then calls abort(). The information it writes to stderr includes:

the source filename (the predefined macro __FILE__)

the source line number (the predefined macro __LINE__)

the source function (the predefined identifier __func__) (added in C99)

the text of expression that evaluated to 0

Programmers can eliminate the assertions without changing the source code: if the macro NDEBUG is defined before the inclusion of <assert.h>, the assert macro is defined simply as:

```
#define assert(ignore)((void) 0)
```

and therefore has no effect on the program, not even evaluating its argument. Therefore expressions passed to assert must not contain side-effects since they will not happen when debugging is disabled. For instance:

```
assert(x = gets());
```

will not read a line and not assign to x when debugging is disabled.

```
/* CELEBA08

   In this example, the assert() macro tests the string argument for a
   null string and an empty string, and verifies that the length argument
   is positive before proceeding.

 */
#include <stdio.h>
#include <assert.h>

void analyze(char *, int);
int main(void)
{
   char *string1 = "ABC";
   char *string2 = "";
   int length = 3;

   analyze(string1, length);
   printf("string1 %s is not null or empty, "
         "and has length %d \n", string1, length);
   analyze(string2, length);
   printf("string2 %s is not null or empty,"
         "and has length %d\n", string2, length);
}

void analyze(char *string, int length)
{
   assert(string != NULL);     /* cannot be NULL */
   assert(*string != '\0');    /* cannot be empty */
   assert(length > 0);         /* must be positive */
}
```

```
String1 ABC is not NULL or empty, and has length 3
Assertion failed: *string != '\0', file: CELEBA08 C       A1, line: 26 in function
analyze
```

Conditional compilation:

In addition to generating code, macros can be used for conditional compilation, where a section of the source code is included only if a particular macro is defined. This is done using the #ifdef and #ifndef preprocessor directives. In its simplest form, writing #ifdef NAME includes all code up to the next #endif if and only if NAME is defined. Similarly, #ifndef NAME includes all code up to the next #endif if and only if NAME is not defined.
Like regular C if statements, #ifdef and #ifndef directives can be nested, and can include else cases, which are separated by an #else directive.

#if directive:

The preprocessor also includes a more general #if directive that evaluates simple arithmetic expressions. The limitations are that it can only do integer arithmetic (using the widest signed integer type available to the compiler) and can only do it to integer and character constants and the special operator defined(NAME) , which evaluates to 1 if NAME is defined and 0 otherwise. The most common use of this is to combine several #ifdef -like tests into one.

4.13.8 Can a macro call a preprocessor command?

E.g., can you write something like

```
#define DefinePlus1(x, y)  #define x ((y)+1)
#define IncludeLib(x)      #include "lib/" #x
```

The answer is **no**. C preprocessor commands are only recognized in unexpanded text. If you want self-modifying macros you will need to use a fancier macro processor like m4 (http://en.wikipedia.org/wiki/M4_(computer_language)).

72. **C library function - clock()**

The C library function clock_t clock(void) returns the number of clock ticks elapsed since the program was launched. To get the number of seconds used by the CPU, you will need to divide by CLOCKS_PER_SEC.

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

int main () {
   clock_t start_t, end_t, total_t;
   int i;

   start_t = clock();
   printf("Starting of the program, start_t = %ld\n", start_t);

   printf("Going to scan a big loop, start_t = %ld\n", start_t);
   for(i=0; i< 10000000; i++) {
   }
   end_t = clock();
   printf("End of the big loop, end_t = %ld\n", end_t);

   total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
   printf("Total time taken by CPU: %f\n", total_t  );
   printf("Exiting of the program...\n");

   return(0);
}
```

```
Starting of the program, start_t = 0
Going to scan a big loop, start_t = 0
End of the big loop, end_t = 20000
Total time taken by CPU: 0.000000
Exiting of the program...
```

73. **isalpha, isdigit, isspace**
https://grandidierite.github.io/looking-at-the-source-code-for-function-isalpha-isdigit-isalnum-isspace-islower-isupper-isxdigit-iscntrl-isprint-ispunct-isgraph-tolower-and-toupper-in-C-programming/

74. **atol, atoll, atof, atoi**
https://www.geeksforgeeks.org/atol-atoll-and-atof-functions-in-c-c/
https://www.codevscolor.com/c-atof-atoi-atol-function

75. **switch statement example:** No break used. Every line executed sequentially.

```
char c = 'A';
switch (c) {
    case 'A':
        printf ("A\n");
    case 'B':
        printf ("B\n");
    default :
        printf ("Default!\n");
}
```

```
A
B
Default!
```

76. **strstr : Locate substring** http://www.cplusplus.com/reference/cstring/strstr/

Search and replace:

```
#include <stdio.h>
#include <string.h>
int main ()
{
  char str[] ="This is a simple string";
  char * pch;
  pch = strstr (str,"simple");
  strncpy (pch,"sample",6);
  puts (str);
  return 0;
}
```

```
This is a sample string
```

77. **strtok : string tokenizer** http://www.cplusplus.com/reference/cstring/strtok/

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

int main ()
{
  char str[] ="- This, a sample string.";
  char * pch;
  printf ("Splitting string \"%s\" into tokens:\n",str);
  pch = strtok (str," ,.-");
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok (NULL, " ,.-");
  }
  return 0;
}
```

```
Splitting string "- This, a sample string." into tokens:
This
a
sample
string
```

78. **sprintf sscanf** http://www.cplusplus.com/reference/cstdio/sprintf/
http://www.cplusplus.com/reference/cstdio/sscanf/

79. Static and dynamic scoping: https://www.geeksforgeeks.org/static-and-dynamic-scoping/

80. Pointers to arrays: https://www.youtube.com/watch?v=z_2-9o-l_p8&feature=youtu.be
** https://www.oreilly.com/library/view/understanding-and-using/9781449344535/ch04.html
https://www.learncpp.com/cpp-tutorial/6-14-pointers-to-pointers/
** https://www.geeksforgeeks.org/multidimensional-pointer-arithmetic-in-cc/

1D int a[10];
a[i] is equivalent to *(a+i);
→ int (*p)[10] Pointer to array of 10 elements (Remember by precedence **(),[]** > *)
→ int *p[10] Array of 10 int type pointers (Remember by precedence **(),[]** > *)

2D int a[10][20]
a[i][j] is equivalent to *(*(a+i)+j)

3D int a[10][20][30]
a[i][j][k] is equivalent to *(*(*(a+i)+j)+k)

- **IMPORTANT!**
Pointer to int is *int**
Think of (i) pointer to 1D array as *int*** int a[5] *a* means a pointer to 1D array
(ii) pointer to 2D array as *int**** int a[5][6] *a* means a pointer to 2D array
(iii) pointer to 3D array as *int***** ...

Dereferencing the *int**** type we get type *int**** . Think of it like dereference operator * is cancelling the star from int****.
(int**) → int****
(int*) → int***
(int) → int**

81. **Static and Dynamic Scoping**
https://www.geeksforgeeks.org/static-and-dynamic-scoping/

82. **Scope rules in C**: https://www.geeksforgeeks.org/scope-rules-in-c/

83. **Element location in memory for 2D array**

In C, arrays are always stored in row-major form.

Formula to evaluate 2-D array's location is:----

$$loc(a[i][j]) = BA + [(i - lb_1) \times NC + (j - lb_2)] \times c$$

Where,

BA - Base Address
NC - no. of columns
c - memory size allocated to data type of array

$a[lb_1 \cdots ub_1][lb_2 \cdots ub_2]$

Great questions!

#include<stdio.h>
int main()
{
  int i, j, k = 0;
  j=2 * 3 / 4 + 2.0 / 5 + 8 / 5;
  k-=--j;
  for (i=0; i<5; i++)
  {
  switch(i+k)
  {
  case 1:
  case 2: printf("\n%d", i+k);
  case 3: printf("\n%d", i+k);
  default: printf("\n%d", i+k);
  }
  }
  return 0;
}
	10
	
	https://gateoverflow.in/8557/gate2015-3-48

Consider the following pseudo code. What is the total number of multiplications to be performed?
 	
D = 2
for i = 1 to n do
  for j = i to n do
  for k = j + 1 to n do
  D = D * 3
	One-sixth of the product of the 3 consecutive integers.
	
	https://gateoverflow.in/1920/gate2014-1-42
	
	*Applied/C/2.Control statements/Gate 2014.mp4

The following function computes X^Y for positive integers X and Y.
 	
int exp (int X, int Y) {
  int res =1, a = X, b = Y;
  while (b != 0) {
  if (b % 2 == 0) {a = a * a; b = b/2; }
  else {res = res * a; b = b - 1; }
  }
  return res;
}
https://gateoverflow.in/39578/gate2016-2-35
	X^Y = res * (a^b)

int main()
{
  int n = 3;
  {
  n = 6;
  printf ("%d\n", n);
  }
  printf ("%d\n", n);
 	
  return 0;
}
	6
	6

int main()
{
int n = 3;
{
int n = 6;
printf ("%d\n", n);
}
printf ("%d\n", n);
return 0;
}
	6
	3

int main()
{
printf ("%d\n", x);
return 0;
}
void f1()
{
printf ("%d\n", x);
}
int x = 20;
void f2()
{
printf ("%d\n", x);
}
	Compile error
	
	X variable is accessible only in f2, not f1 or main.

void f1();
void f2();
int main()
{
extern int x;
printf ("%d\n", x);
f1();
f2();
return 0;
}
void f1()
{
extern int x;
printf ("%d\n", x);
}
int x = 20;
void f2()
{
printf ("%d\n", x);
}
	20
	20
	20

int a = 7;
printf ("%d\n", (10>5) || (a = 10));
printf ("%d\n", a);
 	
int m, i = 0, j = 1, k =2;
m = i++ || j++ || k++;
printf (“%d %d %d %d”, m, i, j, k)
	1
	7
	
	1 1 2 2

int a = 3, 1;
int b = (5, 4);
printf (“%d”, a+b)
	7

int a;
int b = 5;
a = 0 && --b;
printf (“%d %d”, a, b);
	0 5

int a, b=5; a = 0 && --b print a, b	**0 5 (not 0 4 even though -- has higher precedence than &&)**
int i = 5, j =10, k =15; printf (“%d ”, sizeof(k /= i+j)); printf (“%d”, k);	**4 15** https://www.geeksforgeeks.org/why-does-sizeofx-not-increment-x-in-c/ https://discuss.codechef.com/t/working-of-the-sizeof-operator-containing-expressions/5598

int a = 6;
a -= (a--) - (--a);
print a
	2

int a[10];
int i = 0;
a[i] = ++i;
printf (“%d %d %d”, a[0], a[1], a[2])
	0 1 0

int i = 0, j = 1, k = 0;
if (++k, j, i++) {
  printf (“%d %d %d”, i, j, k);
}
	Prints nothing

char c = ‘a’;
switch(c) {
  case ‘a’ && 1: printf (“A”);
  case ‘b’ && 1: printf (“B”);
  break;
  default: printf (“C”);
}
	Compile time error
	2 identical cases 1 and 1

int fun();
int main {
  for (fun(); fun(); fun())
  printf ("%d\n", fun());
  return 0;
}
int fun()
{
  int static n=10;
  return n--;
}
	8
	5
	2

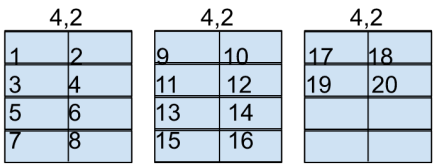
```
main() {
   printf (“anti”);
   main(); }

char str[] = “%d %c”;
char arr[] = “APPLIEDCOURSE”;
printf(str, 0[arr], 3[arr+4]);
```

antiantiantanti… until stack overflow happens in call stack

65 C

```
int a[3][4][2] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,
                  15,16,17,18,19,20};
printf (“%d ”, *((*(a+2)-3))[2]  );
```

Very very important!
Check @
https://www.youtube.com/watch?v=z_2-9o-I_p8&feature=youtu.be
Below question too !!



a → Pointer to 2D array
a+2 → Pointer to 3rd 2D array / address of 17
*(a+2) → Pointer to 1st 1D array of 3rd 2D array
*(a+2)-3 → 3 1D arrays back / address of array[11, 12]
(*(a+2)-3)[2] → *((*(a+2)-3) + 2) move two 1D arrays and points to 15
(((a+2)-3))[2] → 15

So, answer is 15

```
katai zeher!!

char arr[2][3][3] = {‘A’, ‘P’, ‘P’, ‘L’, ‘I’, ‘E’, ‘D’, ‘ ’, ‘C’, ‘O’,      ‘U’, ‘R’, ‘S’, ‘E’};
char (*p) [2][3][3] = &arr;
printf (“%d ”, (*(*(*p)[0] + 5)) - (*(*(*p)[1] - 3))    );
```

Answer is 1 … First draw the 3D array like the upper one

p → ptr to 3D array | char****
*p → ptr to 2D array / address of ‘A’ | char***
(*p)[0] → *((*p) + 0) → *(*p) → ptr to 1D array | char**
*(*p)[0] → ptr to char ‘A’ | char*
*(*p)[0] + 5 → ptr to char ‘E’ | 5 chars after ‘A’
((*p)[0] + 5) → lastly dereferencing. So, we get ‘E’.

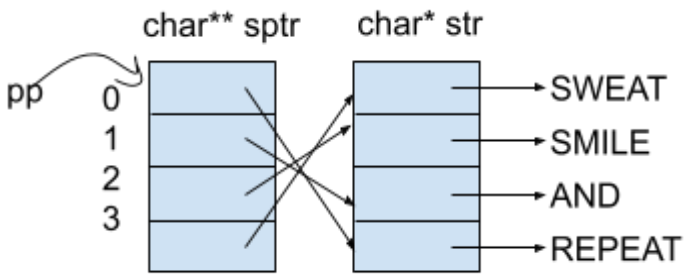
Similarly, (*(*(*p)[1] - 3)) gives ‘D’. So, ‘E’ - ‘D’ = 1

```
char str1[100];
char *fun(char str[]) {
  static int i =  0;
  if (*str) {
    fun(str+1);
    str1[i] = *str;
    i++;
  }
  return str1;
}
main() {
  char str[] = “males laL”;
  printf (“%s”, fun(str));
}
```

Lal selam

```
int a[5][6][7] = {0};
----------missing statement
printf (“%d” a[1][2][3]);
```
→ Missing statement(s) that can be used to print 4:
a) a[1][2][3] = 4;
b) *(*(*(a+1)+2)+3) = 4;
c) (*(*(a+1)+2))[3] = 4;
d) *((int*)a+1*6*7+2*7+3) = 4;

a, b, c, d

https://youtu.be/7AGyjX0u-ZU	
Very good question! char* str[] = {“SWEAT”,”SMILE”, “AND”, “REPEAT”}; char** sptr[] = {str+3, str+2, str+1, str}; char*** pp; pp = sptr; ++pp; printf (“%s “, **++pp); --pp; printf (“%s “, **pp); --pp; printf (“%s “, **pp)+3; https://youtu.be/7AGyjX0u-ZU Timestamp 42:00	**SMILE AND EAT** 

Very good question!
void fun(int ptr1, int *ptr2, int **ptr3){
ptr1 = 2;
*ptr2 = ptr1;
**ptr3 = *ptr2;
ptr1 = 1;
}
main(){
int a=3, b=3, *c=&b;
fun(a, &b, &c);
printf(“%d %d %d”, a, b, *c);
}
	3 2 2

char p[] = “ASDFGHJKL”;
char t;
int i, j;
for (i=0,j=strlen(p); i<j; i++,j--){
t = p[i];
p[i] = p[j];
p[j] = t;
}
printf (“%s”, p);
	Prints nothing
	
	At the outset, ‘A’ is replaced with ‘\0’. So, while printing ‘\0’ is seen first. Hence nothing gets printed.

char c[] = “APPLIED COURSE”; char *p = c; printf (“%s”, p+p[4]-p[0]);	**COURSE** **‘I’ - ‘A’ = 8**

int *p = (int*)malloc(sizeof(int));
p=NULL;
free(p);
	Memory leak
	We have nothing to free as we have removed the pointer by pointing p to NULL. Memory is wasted.
	Correct order to free the memory would be:
	free(p);
	p=NULL;

int* (*(*A[10])(int* a, int b, int c))[32];	**A is an array of 10 pointers to a function that accepts a pointer to an int and two int variables b**

	and c and returns a pointer to an array of 32 pointers to an int.

Tricky!
struct list {
  int a; char *ch;
}s[]={18, “Applied”, 19, “Gate”, 20, “Course”};
main(){
  struct list *ptr = s;
  printf (“%s ”, ptr++->ch);
  printf (“%s ”, (++ptr)->ch);
  printf (“%s”, (--ptr)[1].a);
  printf (“%s”, ptr[0].a);
}
	Applied Course 2019

struct address {
  char city[10];
  char street[30];
  int pin;
};
struct {
  char name[30];
  int gender;
  struct address locate;
} person, *id = &person;
 	
Now, *(id->name+2) is equivalent to:
  a) person.name+2
  b) id->(name+2)
  c) *((*id).name+2)
	Only option c

int i=5, j=2;
void fun(int x) {
  printf ("%d\n", x+10);
  i = 10;
  j = 1;
  printf ("%d\n", x);
}
int main() {
  fun (i+j);
  return 0;
}
	17
	7

int main()
{
 	
  int x = 1, y = 2, z = 3;
 	
  printf("x = %d, y = %d, z = %d\n",
  x, y, z);
  {
 	
  int x = 10;
  float y = 20;
 	
  printf("x = %d, y = %f, z = %d\n",
	x = 1, y = 2, z = 3
	x = 10, y = 20.000000, z = 3
	x = 10, y = 20.000000, z = 100
	x = 1, y = 2, z = 3

```
                x, y, z);
                {


                        int z = 100;
                        printf("x = %d, y = %f, z = %d\n",
                        x, y, z);
                }
        }
        printf("x = %d, y = %d, z = %d\n",
        x, y, z);
        return 0;
}
```

```
int main()
{
{
        int x = 10, y = 20;
        {
                printf("x = %d, y = %d\n", x, y);
                {
                        int y = 40;
                        x++;
                        y++;
                        printf("x = %d, y = %d\n", x, y);
                }
                printf("x = %d, y = %d\n", x, y);
        }
}
return 0;
}
```

x = 10, y = 20
x = 11, y = 41
x = 11, y = 20

```
int main()
{
{
        int x = 10;
}
{
        // Error: x is not accessible here
        printf("%d", x);
}
return 0;
}
```

???

The following program fragment is written in a programming language that allows global variables and does not allow nested declarations of functions.

```
global int i=100, j=5;
void P(x) {
        int i=10;
   print(x+10);
   i=200;
   j=20;
   print (x);
}
main() {P(i+j);}
```

error

https://gateoverflow.in/43575/gate2003-74?show=338119#a338119

If the programming language uses dynamic scoping and call by name parameter passing mechanism, the values printed by the above program are	

???
The following program fragment is written in a programming language that allows global variables and does not allow nested declarations of functions.
 	
global int i=100, j=5;
void P(x) {
int i=10;
print(x+10);
i=200;
j=20;
print (x);
}
main() {P(i+j);}
If the programming language uses static scoping and call by need parameter passing mechanism, the values printed by the above program are:
	https://gateoverflow.in/960/gate2003-73

#include<stdio.h>
void fun1(char* s1, char* s2){
char* temp;
temp = s1;
s1 = s2;
s2 = temp;
}
void fun2(char** s1, char** s2){
char* temp;
temp = *s1;
*s1 = *s2;
*s2 = temp;
}
int main(){
char *str1="Hi", *str2 = "Bye";
fun1(str1, str2); printf("%s %s", str1, str2);
fun2(&str1, &str2); printf("%s %s", str1, str2);
return 0;
}
	Hi ByeBye Hi

int main() {
double da, db;
db = foo(da);
}
int foo (int a) {
return a;
}
	test.c: In function ‘main’:
	test.c:32:10: **warning**: implicit declaration of function ‘foo’ [-Wimplicit-function-declaration]
	db = foo(da);
	^~~

int main() {
double da, db;
db = foo(da);
}
double foo (double a) {
	test.c: In function ‘main’:
	test.c:32:10: **warning**: implicit declaration of function ‘foo’ [-Wimplicit-function-declaration]
	db = foo(da);
	^~~

return a;
}
	test.c: At top level:
	test.c:34:8: **error**: conflicting types for ‘foo’
	double foo (double a) {
	^~~
	test.c:32:10: note: previous implicit declaration of ‘foo’ was here
	db = foo(da);
	^~~

#include <stdio.h>
struct test {
int i;
char *c;
}st[] = {5, "become", 4, "better", 6, "jungle", 8, "ancestor", 7, "brother"};
main ()
{
struct test *p = st;
p += 1;
++p -> c;
printf("%s,", p++ -> c);
printf("%c,", *++p -> c);
printf("%d,", p[0].i);
printf("%s \n", p -> c);
}
	etter, u, 6, ungle
	
	https://gateoverflow.in/3592/gate2006-it-49

char b[3][2] = {1}; printf ("%p\n", b); printf ("%p\n", &b); printf ("%p\n", *b); printf ("%d\n", **b);	**0x7ffefd373ad2** **0x7ffefd373ad2** **0x7ffefd373ad2** **1**

int main () {
unsigned int x [4] [3] =
{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};
printf ("%u, %u, %u", x + 3, *(x + 3), *(x + 2) + 3);
}
	2036, 2036, 2036

int i, j, k;
for (i = 2; i++;)
{
printf ("%d\n", i);
if (i == 5) break;
}
	3
	4
	5

***
 	
int i = 0;
while (i++, i<=8)
printf ("%d\n", i);
	1
	2
	3
	4
	5
	6
	7
	8

int i = 0;
while (i++, i<=8);
printf ("%d\n", i);
	9

int i = 0;
do {
printf ("%d ", i++);
continue;
} while(i <=10);
	0 1 2 3 4 5 6 7 8 9 10

int i = 9, k =8;
const int *ptr = &i; //pointer to const variable
ptr = &k;
	Compiles successfully

int i = 9, k =8;
const int *ptr = &i;
ptr = &k;
*ptr = 0;
	error: assignment of read-only location '*ptr'
	***ptr = 0;**

int i = 9, k =8;
int *const ptr = &i; //const pointer to variable
ptr = &k;
	error: assignment of read-only variable 'ptr'
	ptr = &k;

int i = 9, k =8;
const int *const ptr = &i; //const ptr to const var
ptr = &k;
*ptr = 7;
	test.c:31:9: error: assignment of read-only variable 'ptr'
	ptr = &k;
	^
	test.c:32:10: error: assignment of read-only location '*ptr'
	***ptr = 7;**

Is it possible to change/modify a character in: **char *p="ravindra"** ?	We cannot modify the string at later stage in program. We can change str to point something else but cannot change value present at *p*. https://www.geeksforgeeks.org/char-vs-stdstring-vs-char-c/?fbclid=IwAR3pmXnq0xtgKPtoFI17PkHnGn3ZVIgPaYoy7mADLGMSxIV6QU681qTh73Y

Zehrila!
int main(void)
{
int *p = (int *)malloc(sizeof(int));
*p = 0x8F7E1A2B;
printf ("%X\n", *p);
unsigned char *q = p;
printf ("%X\n", *q++);
printf ("%X\n", *q++);
printf ("%X\n", *q++);
printf ("%X\n", *q++);
return 0;
}
	https://stackoverflow.com/questions/59137031/output-after-using-malloc
	
	Little endian system,
	
	8F7E1A2B
	2B
	1A
	7E
	8F
