C Programming

Notes by: Priyam Garg
Faculty: Sachin Mittal Sir
Books References: *The C programming language by Dennis Ritchie*, *Computer systems: programmers perspective*

Important points

1. How programs in c would be compiled are compiler and system dependent.
2. Programs begin executing at the beginning of `main`
3. `printf` never supplies a newline automatically
4. `\t` for tab, `\b` for backspace, `\"` for double quote, `\\` for backslash itself
5. `printf()` is not a part of C language, its' part of the C standard library, which has very close relation to C language. Parts of C languages are `declarations`, `operators`, `expressions`, `statements`, `pointer`, `arithmetic` etc.
6. Text input or output regardless of its origination is dealt with streams of characters.

Lecture Plan

Unsigned and Signed Integers

Lets' take an example code:

Example 1

```
int y = -9;
```

-9 would be represented in $2's$ complement format.
$9 = 000\ldots0001001$
$-9 = 111\ldots1110111$

> **Note**
>
> By default `int` is signed

```
int y = -9;

printf("%d", y) // -9
printf("%u", y) // huge number (why?) [111...10111] would be treated as unsigned and its huge.
```

- **%d**: This will treat y as `signed` number.
- **%u**: This will treat y as `unsigned` number

> **Important**
>
> `printf()` doesnt' use the information regarding type of data

«~»

Example 2

```
unsigned int y = -9;

printf("%d", y) // -9
printf("%u", y) // huge number again
```


Extension and Truncation

For extra information

```
short int x = 9;
short x = 9;
short signed int x = 9;
```

All the above definitions are same.

Extension

> Copying a lower bit number to higher bit number, e.g. *short* to *int*

Example 1

```
short int x = 9;
```

32 bit extension	
16 bits	16 bits
	0000 0000 0000 1001

we want to copy this x to new variable,

```
int ix = x;
```

32 bit extension	
16 bits	16 bits
0000 0000 0000 0000	0000 0000 0000 1001

copy depends on the `source` and not on what we are extending it to (destination)

«~»

Example 2

```
short int x = -9;
```

32 bit extension	
16 bits	16 bits
	1111 1111 1111 0111

we want to copy this x to new variable,

```
int ix = x;
```

32 bit extension	
16 bits	16 bits
1111 1111 1111 1111	1111 1111 1111 0111

Source is signed so I copied all $1's$

«~»

Example 3

```
short int x = -9;
```

32 bit extension	
16 bits	16 bits
	1111 1111 1111 0111

we want to copy this x to new variable,

```
unsigned int ix = x;
```

32 bit extension	
16 bits	16 bits
1111 1111 1111 1111	1111 1111 1111 0111

Destination (`unsigned`) doesnt' matter.

«~»

Example 4

```
unsigned short int x = -9;
```

32 bit extension	
16 bits	16 bits
	1111 1111 1111 0111

we want to copy this x to new variable,

```
int ix = x;
```

32 bit extension	
16 bits	16 bits
0000 0000 0000 0000	1111 1111 1111 0111

«~»

Example 5

```
unsigned short int x = -9;
```

32 bit extension	
16 bits	16 bits
	1111 1111 1111 0111

we want to copy this x to new variable,

```
unsigned int ix = x;
```

32 bit extension	
16 bits	16 bits
0000 0000 0000 0000	1111 1111 1111 0111

Extension depends on RHS(`source`)

- Promotion always happens according to the source variables' type
  - **Signed**: `sign extension` (copy MSB-0 or 1 to fill new spaces)
  - **Unsigned**: `zero fill` (copy 0's to fill new space)

Truncation

> Copying higher bit number to lower bit number

> **Important**
>
> - Regardless of source/destination or signed/unsigned type, truncation always just truncates.
> - This can cause number to **change drastically in sign and value**

Integer Promotion

> Whenever a small integer type (`char` or `short`) is used in an **expression**, it is implicitly converted to `int`

`char` , `short` and `int` basically belongs to same family but have different bit lengths.

```
char c = 'a';

printf("%d", c) // 97
printf("%c", c) // a
printf("%d", c-1) // 96
```

`char` stores the value as integer only, but for us it shows as character like ("a") when we used `%c` .

Lets' see an example:

```
char c = 90;

printf("%d", c); // 90
```

char c
01011010

printf("%d", c)
0000 0000 0000 0000 0000 0000 0101 1010

It got implicitly converted to 32-bit format.

```
signed char a = 258;

printf("%d", a); // 2
printf("%u", a); // 2
```

a can be represented as $1\ 0000\ 0010$, these are $9-\text{bits}$ so it would be truncated and we get last 8 bits $0000\ 0010$ as a

When a would be used in an expression like `printf("%d", a)` then a will be converted to $32-\text{bit}$ representation. $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010$

> **Note**
>
> `char` , `short` , and `int` are family of integers only

Example 1

```
char a=30, b=40;
char d = a*b;

printf("%d", d); // -80
printf("%d", a*b); // 1200
```

1200 in binary is $0000\ 0100\ 1011\ 0000$, but d is `char` data type, so this binary value will be truncated to $8-\text{bits}$.

We will get $1011\ 0000$, but with `printf("%d", d)` this will promoted to $32-\text{bits}$ binary value. **Note** that `char` is `signed char` hence rest of the bits would be 1.
$1\ 1\ 1\ 1\ 1\ 1\ \underbrace{1}_{-128}\ 0\ \underbrace{11}_{32,16}0000 \rightarrow -128+32+16=-80$

Example 2

```
signed char f = -65;
printf("%d", f); // -65
printf("%u", f); // huge number
```

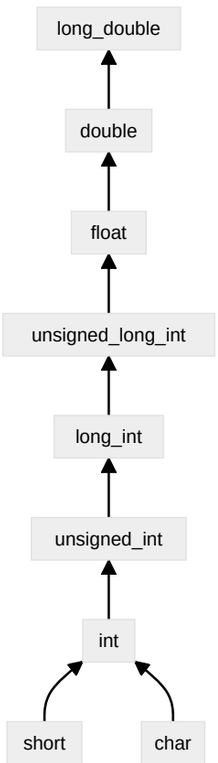
$-65=1011\ 1111$
in $32-\text{bit}$ representation
$-65=1\ 1\ 1\ 1\ 1\ 1\ \underbrace{1}_{-128}\ 0\ \underbrace{111111}_{32,16,8,4,2,1} \rightarrow -128+32+16+8+4+2+1=-65$

Example 3

```
unsigned char f = -65;
printf("%d", f); // 191
printf("%u", f); //191
```

Data Types and Type Conversion

- When `double` is converted to `float`, whether the value is rounded or truncated is implementation dependent.
- (type-name) expression. () this is unary operator named cast.



Important

- Assigning a longer integer type to a shorter or a floating point type to an integer may draw a $warning$ but they are not illegal.

Question 1

```c
int value1 = 10, value2 = 3;
float result;
result  = value1/value2; // 3
printf("%f", result); // 3.00000
```

$\frac{value1}{value2}$ is integer division.

Question 2 (Implicit type conversion)

```c
int value1 = 10;
float value2 = 3;
float result;
result  = value1/value2; // 3.33
printf("%f", result); // 3.333
```

$\frac{value1}{value2}$ is float division, because higher order data type is available. It all happening **implicitly**

Question 3 (Explicit type conversion)

```c
int value1 = 10;
int value2 = 3;
float result;
result  = (float)value1/value2; // 3.33
printf("%f", result); // 3.333
```

$\frac{value1}{value2}$ is float division, because we have explicitly changed the type of result.

Question 4 (Almost GATE PYQ)

```c
int main(){
        unsigned int a = 1000;
        int b = -1;
        if (a>b) printf("a is BIG"); // this is printed
        else printf("a is SMALL");
        return 0;
}
```

- So we are comparing unsigned and signed integer. My question is "*is this possible?*"
- *Can we compare two variables of different types?*

Answer

Check the hierarchy of data types above, `unsigned_int` is placed higher than `int`. Now we doing comparison b/w int and unsigned int so variable with int data type will be implicitly converted to `unsigned int`.

Homework 1

```c
int i = -3 // i → 1..11101 (total 32-bits)
unsigned short u;
u=i; // u → 1..11101 (total 16-bits)

printf("%u", u); // u get integer promoted in 32 bits [000..011..1101] (remember, it depends on source, its' unsigned)
printf("%d", u); // u get integer promoted in 32 bits [000..011..1101] (remember, it depends on source, its' unsigned)
```

Homework 2

```c
signed short ix = -3; // ix → 1..1101 (total 16  bits)
printf("%u", ix); // 11..1101 (total 32 bits) Huge number
printf("%d", ix); // -3
```

Homework 3

```c
signed char a = 100; // 01100100
signed char b = 200; // 11001000
signed int c = a+b; // a,b will be integer promoted to 32-bits, a → 00..0 01100100, b → 11..11 11001000(source is signed and MSB is 1 that shows it negative). a → 100, b → -128+64+8 = -56. a+b = 44
printf("%d %d %d\n", a,b,c); // all of the variables would be integer promoted, since they are being used in expression. 100, -56, 44
```

Homework 4

```c
signed char a = 100; // 01100100
signed char b = 200; // 11001000
signed char c = a+b; // a,b will be integer promoted to 32-bits, a → 00..0 01100100, b → 11..11 11001000(source is signed and MSB is 1 that shows it negative). a → 100, b → -128+64+8 = -56. c = a+b = 44. c will be truncated to 8 bit length, but even after that it would be 44 only.
printf("%d %d %d\n", a,b,c); // all of the variables would be integer promoted, since they are being used in expression. 100, -56, 44
```


Conditional Programming

Switch case

```c
switch(x){ // x should be from family of integers ("A" is alo an integer)
        printf(..) // wont' be printed
        case constant_1: // constant_1 should be unique and not any range.
                ...
                break;
        case constant_2: // constant_2 should not be variable, it has to be constant value
                ...
                break;
        default: // can occur in any order
                ...
                break;
}
```

Example Program

```
int num = 3;
switch(num){
        case 1:
                print("case 1: \n")
        case 3:
                printf("case 3: \n")
        case 2:
                printf("case 2: \n")
                break;
        default:
                printf("Default:")
}
```

It will print

```
case3:
case2:
```

`Switch()` search for the matching case if nothing matches then go to default and fall through till `break;`

Loops

Valid Loop 1

```
void main(){
        int i = 0;
        for (;i<=9;){
                i++;
                stmts...
        }
}
```


Question (Check for Delimiter)

```
for(int i=0;i<=3;i++);
printf("%d", i); // 4
```

do-while

> First do, then check

Example

```
void main(){
        int m=3;
        do{
                printf("%d", m);
                m=0;
        } while(m>0); // 3
}
```

Operators

```
inr main(){
        int a = 10, b=20, c=30;
        if (a<b<c) printf("True\n")
        else printf("False\n")
        return 0;
}
```

$a < b < c$, we take in the associativity of operators. So first $a < b$ will happen and it will give $True$. $True$ in integer would be 1. Now $1 < c$ would be compared and this will $True$ also.
Hence we can conclude that this program tells if c is larger than a, b both.

Now thats' not true, we have have $a = 40, b = 50, c = 30$ and still get $True$. But $c < a, b$.

> Use `a<b && b<c` instead

Illegal Operations

1. $++\underbrace{2}_{const}$
2. $++\underbrace{(a + 2 * b)}_{expression}$

Associativity and Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
2	++ --	Prefix increment and decrement	Right-to-left
2	+ -	unary plus and minus	Right-to-left
2	! ~	Logical NOT and bitwise NOT	Right-to-left
2	(type)	Cast	Right-to-left
2	*	dereference	Right-to-left
2	sizeof	size of	Right-to-left
3	* / %	multiplication division and remainder	Left-to-right
4	+ -	Addition and subtraction	Left-to-right
5	<< >>	Bitwise left shift and right shift	Left-to-right
6	< <=	Relational less than and less than equal to	Left-to-right
6	> >=	Relational more than and more than equal to	Left-to-right
7	== !=	equal to and not equal to	Left-to-right
8	&	bitwise AND	Left-to-right
9	^	bitwise XOR	Left-to-right
10	|	bitwise OR	Left-to-right
11	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right
13	=, +=, -= ..	Assignment	Right-to-left
14	,	Comma	Left-to-right

Bitwise Operators

Bitwise AND

$$\begin{array}{r} 11001 \\ \& \ 01101 \\ \hline 01001 \end{array}$$

Test number is even or odd

```
int test = 1;
if (number & test) printf("Odd");
else printf("Even");
```

test -> 00..001. If number also odd then LSB will be HIGH and "&" will output 1, hence number odd. On the other hand, if number is even, then LSB would be low, hence output would be 0. Hence number even.

Bitwise XOR

$$\begin{array}{r} 11001 \\ \hat{} \, 01101 \\ \hline 10100 \end{array}$$

Bitwise left shift

$x = 0100\ 1001\ 1100\ 1011 = 18891$
$x << 3 = 0100\ 1001\ 1100\ 1011\ 000 = 151128 = 18891 * 2^3$

Bitwise right shift

$x = 0100\ 1001\ 1100\ 1011 = 18891$
$x >> 3 = 000\ 0\ 1001\ 0011\ 1001\ \boxed{\cancel{011}} = 2361 = \lfloor \frac{18891}{2^3} \rfloor$

Important

- **Unsigned Right shift** - Fill Zeros
- **Signed Right Shift** - Depends on system, either zeros or sign bits. This is system dependent too, so complacency is involved. Rather not go into deep.

Bitwise NOT

$\underbrace{\sim}_{\text{bitwise NOT}} 0 = 111..11$
$\underbrace{!}_{\text{Logical NOT}} 0 = 1$

Note

- `&&` and `||` imply left-to-right evaluation of a truth value. Whereas bitwise operators operate on singular corresponding bits.

Assignment Operators

GATE CSE 2015

```
#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;
}
```

Q. Number of times printf() stmt executed?
A. `*` and `/` have same precedence order and their associativity is Left-to-right. So `*` is executed first. Hence 2*3/4 -> 6/4 -> 1 (since its' integer division). 2.0/5 -> 0.4 (float division), 8/5 -> 1 (integer division).
j = 1+0.4+1 = 2.4 => j = 2 (since j is integer)
k -=--j -> k = k - --j -> k = k - 1 = -1 (pre decrement has higher priority than subtraction)

i = 0; switch(0+(-1)); default -> *1 time* printf executed
i = 1; switch(1+(-1)); default -> *1 time* printf executed
i = 2; switch(2+(-1)); Case 1 -> *3 times* printf executed (fall-through, since no break statement available)
i = 3; switch(3+(-1)); Case 2 -> *3 times* printf executed (fall-through)
i = 4; switch(4+(-1)); Case 3 -> *2 times* printf executed (fall-through)
10 times printf executed

Tip

In twos' complement number system, $x \ \& = \ x - 1$ deletes the rightmost 1-bit in x.

Comma Operator

let $a = expr1, expr2, expr3, \ldots, exprn$
then $a = exprn$ no matter what other expressions are.

```
j = 2
for (int i=0;j>=0, i<=5; i++){
        j--;
} // loop 5 times, or I can say i<=5 is last expression separated by comma, so i<=5 will be True only.
```

```
j = 2
for (int i=0;i<=5, j>=0; i++){
        j--;
} // loop 3 times, or I can say j>=0 is last expression separated by comma, so j>=0 will be True only.
```

Variable Names

1. Made up of only letters and digits
2. first character must be letter
3. "_" counts as a letter
4. Names don't start with "_", since many library developers write their variables from "_".

Declarations

- All variables must be declared before use.

```
int lower, upper, step;
char c, line[1000];
```

- `External` and `static` variables are initialized to zero by default.
- If the variable in question is not `automatic` then the initialization is done once only. (before program starts executing)
- Automatic variables for which there is no external initializer have undefined (i.e. garbage) values.
- You can apply `const` to the declaration of any variable, it specify that the value will not be changed.

```
const double e = 2.71828182845905;
```

- If function prototype (return type) is not declared then C assumes the return type as int for all the input (input arg) types. This is according to C99 standards. It may also send a warning "*implicit declaration warning*".

Order of evaluation

- C, like most languages, does not specify the order in which the operands are evaluated.

Danger

If the order in which the function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(w,n)); // WRONG
```

Sequence Points

- Order of evaluation of any part of any expression, including order of evaluation of function arguments is *unspecified*.
- Compiler can **evaluate** operands and other sub-expressions in any order, and may choose another order when same expression is evaluated again.
- There is no concept of left-to-right or right-to-left evaluation. `Note:` Dont' confuse with left to right and right to left associativity of operators.
  (`Ref:` https://www.enseignement.polytechnique.fr/informatique/INF478/docs/Cpp/en/c/language/eval_order.html)
- Expression $a() + b() + c()$ is parsed as $(a() + b()) + c()$ due to left to right associativity of $+$ operator, but $c()$ may be evaluated first, last, or between $a()$ or $b()$ at run time:

$$a = ++b \; + \; ++c;$$

C standard leaves the details upto compiler writer, we don't know what it does.

Important

This is according to Sachin sir, "*Order of evaluation always from left-to-right*"

Sequence Points

At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effect of subsequent evaluations shall have taken place.

Above expression doesnt' have any sequence point in assignment statement.

$a[i++] = i$ is undefined.

$x = a++$; after the sequence point "a" get incremented. But getting incremented is one behaviour it can show, it may not even increment after sequence point but before itself. So there are two such possibilities that we can see, hence these are called *side effects.*

Now, we should not see any side effects after sequence points for non ambiguous evaluation of expression.

$x = a++ \; + \; a++$; we are trying to modify the value of a multiple times before sequence point. It is undefined, you can get anything based on compiler.

$i = i++$; also undefined.

Some Sequence Points

1. ;
2. $if()$, $for()$, $while()$, $switch()$
3. ? :
4. &&, ||

Question

```c
int main(){
        int a=1,b=1,c=1;
        if (a==b || c++){
                printf("%d", c); // 1
        }
}
```

$a == b$ will be evaluated first and that will return $True$. We have $||$ operator there so we just need one $True$ to return $True$. $c++$ didnt' even executed. Thats' why c wasnt' incremented. This is called **Short circuiting**

Question

```c
int main(){
        int a=1,b=1,c=1;
        if (a==b && c++){
                printf("%d", c); // 2
        }
}
```

Question

```c
int main(){
        int a=1,b=1,c=1;
        if (a!=b && c++){
                printf("%d", c);
        }
        printf("%d", c); // 1
}
```

$a != b$ will be evaluated first and that will return $False$. We have $\&\&$ operator there so we just need one $False$ to return $exit$ from conditional block. $c++$ didnt' even executed. Thats' why c wasnt' incremented.

Question

```c
int main(){
        int i=1;
        if (i++ && (i==1)){
                printf("YES");
        }else{
                printf("NO"); // This will print
        }
} // i++ -> 2 after && which is a sequence point here, now i!=1, so False
```

Question

```c
int main(){
        int i=0, j=1, k=2, m;
        m = i++ || j++ || k++;
        printf("%d %d %d %d", m,i,j,k); // 1, 1,2,2
}
```

$m = (i++ || j++) || k++$;. This expression has truth value operands which is logical OR. If we find $True$ at any point, then we dont' need to evaluate any further.
At first we got $i++$, which remained 0 but after first sequence point which is $||$ first logical OR operand, it will become 1.
$j++$ will become 2 after this subexpression is evaluated but right now its' 1.
Now 0 $||$ 1 is 1 or $True$. Now we don't need to evaluate $k++$, so value for k will remain same.

Question

```c
int x = 1, y = 0, z = 5;
int a = x && y || z++;
printf("%d", z); // 6
printf("%d", a); // 1
```

a is having boolean operands, it will either give 0 or 1.

Question

```c
int main(){
        int x, i=4, j=7;
        x = j || i++ && 1;
        printf("%d", i);
}
```

$x = j \,||\, (i++ \,\&\&\, 1)$; Now $j = 7$ and this itself will return $True$ without evaluating any further expression.

Question

```c
int main(){
        int i = 3, j=2, k=0, m;
        m = ++i && ++j && ++k;
        printf("%d, %d, %d, %d", i, j, k, m); // 4, 3, 1, 1
}
```


Question

```c
int main(){
        int i = 3, j=2, k=0, m;
        m = ++i || ++j && ++k;
        printf("%d, %d, %d, %d", i, j, k, m); // 4, 2, 0, 1
}
```

Question

```c
int k, i=50, j=100, l;
i = i | (j && 100); // i = 51
k = i || (j || 100); // k = 1
l = i & (j && 100); // l = 1
printf("%d %d", i, j);
printf("%d %d", k, l);
```

HOMEWORK 1

```c
void main(){
        int ii = 10;
        ii <<= 1;
        printf("%d", ii);
}
```

10 -> 1010, <<(1010, 1) -> 10100 = 20

HOMEWORK 2

```
main(){
        int var1=1, var2=12, var3=12;
        var1=var2==var3;
        printf("%d", var1);
}
```

$==$ has higher priorty than $=$, so $var1 = (var2 == var3)$, => $var1 = 1$

Note

$--c$ where c is a constant, then its' not legal operation. This is pre-increment and post-increment, same goes for decrement too.

HOMEWORK 3

```
main(){
        int var = - -3;
        printf("%d", var);
}
```

$- -3 = +3$

HOMEWORK 4

```
int x = 7;
int y = 10;
int z = 5;
int result = 0;

result = ++y - 10 || z - 5 && x++;
result += y++ - 11 || z++ - 5 && x++;
result += y+1 > 11 && (z++ >= 6 || x++);

printf("%d, %d, %d, %d", result, x, y, z); // 2, 7, 12, 7
```

$result = (((++y) - 10) \;||\; ((z - 5) \;\&\&\; (x++)))$
$\quad = (1 \;||\; (\text{not evaluated})) = 1$
$result += (((y++) - 11) \;||\; (((z++) - 5) \;\&\&\; (x++)))$
$\quad = (0 \;||\; (0 \;\&\&\; (\text{not evaluated}))) = 1 + 0 = 1$
$result += (((y+1) > 11) \;\&\&\; ((z++ >= 6) \;||\; (x++))$
$\quad = (1 \;\&\&\; (1 \;||\; (\text{not evaluated}))) = 1 + 1 = 2$

HOMEWORK 5

```
main(){
        int a = 0;
        if (0 && a++==0)
                printf("Inside if\n");
        else
                printf("else");
        printf("%d\n", a);
}
```

$==$ has higher priority so, $0 \;\&\&\; (a++ == 0)$, but order of evaluation would be from left to right only. We got 0 followed by $\&\&$, so further sub-expression won't be evaluated.
Hence it will print "else", and a hasnt' been incremented so $a = 0$ only.

HOMEWORK 6

```
main(){
        int a,b,c,d;
        a = 3;
        b = 5;
        c = a,b;
        d = (a,b);
        printf("c=%d, d=%d", c, d);
}
```

$(c = a), b$, so $c = 3$, nothing will happen for b. $d = (a, b)$ so on case of comma operator, we take the last one to be $True$, so $d = b$.
$c = 3, d = 5$

HOMEWORK 7

```
int main(){
        int a = 10, b = 5, c=3;
        b != !a; // True, but doesnt' change values
        c = !!a; // c = !(0) = 1
        printf("%d\t%d", b, c); // 5, 1
}
```

HOMEWORK 8 (For what value(s) program execute printf?)

```
int main(){
        int i;
        scanf("%d", &i);
        if (!i == ~i)
                printf("same this time\n");
}
```

$!i == \sim i$ is $True$ when $!i$ and $\sim i$ are same. ! is boolean operator which gives either 1 or 0.
if $!i == 0 \rightarrow !i = 000..0 \rightarrow \sim i = 00..00 \rightarrow i = 11...1 = -1$
if $!i == 1 \rightarrow !i = 000..01 \rightarrow \sim i = 00..01 \rightarrow i = 11...0 = -2$, but $!-2 \neq 1$, so this cant' be solution. Only $\boxed{-1}$ is solution.

HOMEWORK 9

```
int main(){
        int a = 2;
        if (a >> 1) // 10 → 01
                printf("%d", a); // 2, since a not updated
}
```

Functions

Important

1. **Function prototype** is also called **function declaration**

Some acceptable forms of declarations

1. `int mul (int a, int b);`
2. `int mul (int, int);`
3. `mul (int a, int b);`
4. `mul (int, int)`

Some Important terms to know

Activation Records

An activation record is a data structure that is activated/created when a procedure/function is invoked, and it includes the following data about the function.

It consists of *actual parameters*, *number of arguments*, *return address*, *return value*, *old stack pointer*.

Scope

It is defined as the availability of a variable inside a program, scope is basically the region of code in which a variable is available to use.

There are four types of scope:

- File scope
- Block scope
- Functions scope
- Prototype scope

Visibility

Visibility of a variable is defined as if a variable is accessible or not inside a particular region of code or the whole program

```c
#include <stdio.h>
int main() {
  int scope; // outer scope variable
  scope = 10;
  // inner block of code
  {
    float scope; // inner scope
    scope = 2.98;
    printf("Inner block scope : %f\n", scope);
  }
  printf("Outer block scope : %d\n", scope);
  return 0;
}
```

`scope` declared and defined outside the inner block is not accessible inside the block because inner block already has `scope` variable, and it has higher priority of getting selected.

Lifetime of a variable

Lifetime of a variable is the time for which the variable is taking up the valid space in the systems' memory, it is of three types:

- **Static lifetime**: Objects/variables having static lifetime will remain in memory until the execution of the program finishes. These type of variables can be declared using `static` keyword, `global variables also have static lifetime`. They survive as long as the program runs
  - `static int count = 0;`

«~»

- **Automatic lifetime**: Objects/variables inside a block have automatic lifetime. Local variables (those defined within a function), have an automatic lifetime by default: they arise when the function is invoked and are deleted once the function execution is finishes.
  - `{ int auto_lifetime_var = 0; }`

«~»

- **Dynamic lifetime**: Objects/variables which are made during runtime of a C program using the *Dynamic Memory Allocation* concept using `malloc()`, `calloc()` function in C or `new` operator in C++ are stored in memory until they are explicitly removed from the memory using the `free()` function in C or `delete` operator in C++ and these variables are said to have dynamic lifetime.
  - `int *ptr = (int *)malloc(sizeof(int));`

«~»

Note

1. Visibility and scope of a variable are very similar to each other but every available variable (in the scope) is not necessarily accessible(visible) in C program

Fake Swap

```c
void swap(int x, int y){
        int temp = x;
        x = y;
        y = temp;
}
int main(){
        int a = 3;
        int b = 4;
        swap(a,b);
        printf("a = %d, b = %d", a,b);
}
```

We will enter the main and there we will get two variables a and b. Activation record will be activated to store these and other important metadata regarding the function.

main()	$\boxed{3} \leftarrow a \; \boxed{4} \leftarrow b$

Now, swap() will be called and new activation record will be added. It will consist of 3 variables.

swap()	$\boxed{3} \leftarrow x \; \boxed{4} \leftarrow y \; \boxed{< empty >} \leftarrow temp$
main()	$\boxed{3} \leftarrow a \; \boxed{4} \leftarrow b$

Now first line of swap will execute and $temp$ will get the value of x.

swap()	$\boxed{3} \leftarrow x \; \boxed{4} \leftarrow y \; \boxed{3} \leftarrow temp$
main()	$\boxed{3} \leftarrow a \; \boxed{4} \leftarrow b$

Now second line will get execute and x will get the value of y.

swap()	$\boxed{4} \leftarrow x \; \boxed{4} \leftarrow y \; \boxed{3} \leftarrow temp$
main()	$\boxed{3} \leftarrow a \; \boxed{4} \leftarrow b$

At the third line, y will get the vale of x.

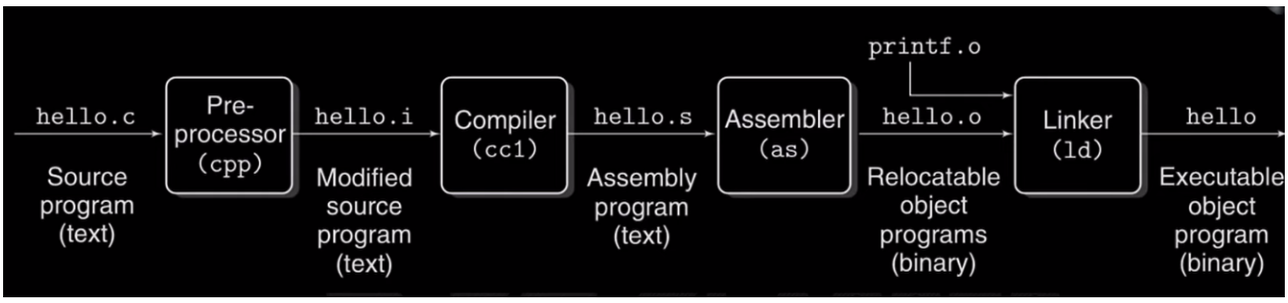
swap()	$\boxed{4} \leftarrow x \; \boxed{3} \leftarrow y \; \boxed{3} \leftarrow temp$
main()	$\boxed{3} \leftarrow a \; \boxed{4} \leftarrow b$

And at this point swap function will get destroyed (Its' activation record gets destroyed), and whatever the parameters were there in activation record they also got destroyed (this is redundant to say, but still). This swap() didnt' affected a, b values. So only activation record that will remain is,

main()	$\boxed{3} \leftarrow a \; \boxed{4} \leftarrow b$

No, values got changed/altered.

CALL (Compiler, Assembler, Linker and Loader)



Memory Layout

Consider a virtual memory where size of stack, heap and static area is fixed.

Stack
Heap
Static Area

Now, we save activation records in stack area, and activation records are formed on the invocation of functions. So lets make a function `main()` with some 3 random functions in it. Such that stack only has enough space to store 2 of the function and not the 3rd one. Even though `heap` is empty and we can use that space but there is a catch.

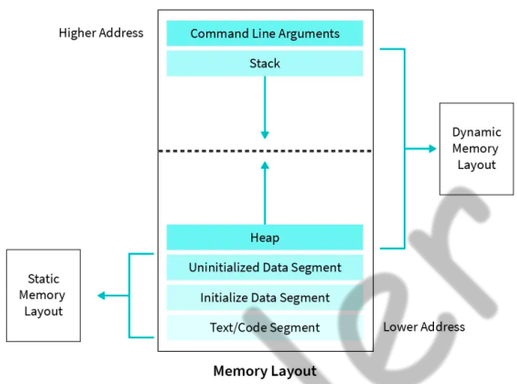
We have created a hard boundary/ hard demarcation between `stack` and `heap` area. So one cannot use the space of other under any circumstance.

```
int main(){
        func1();
        func2();
        func3();
}
```

main()
func1()
func2()
Heap
Static Area

Here, even though we have space in heap section, but this shows that whole virtual memory is filled and can't store any more activation records.

Problem: Fixed size of stack and heap is the problem, they are confining us from optimal usage of space. So we should make these spaces dynamic and flexible.



This is ***virtual memory***

Definition

1. **The Stack**: This is where local variables and functions' activation records are stored. Size changes at runtime.
2. **The Heap**: This is where dynamically allocated data is stored. Size changes at runtime.
3. **The Static Area**: This is where statically declared data (static and global variables) and code is stored. Size is fixed at runtime.

Storage Classes

- Important topic in memory management
- Storage class allow us to determine the scope, visibility, and lifetime of a variable.

What is Storage classes in C?

- Along with definition of variable in the program, we define its; `data type` and its `storage class`.

As for the storage class, it defines the following attributes about the variable:

1. **Scope**

The scope of a variable is *determined at compile time* without creating any function call stack in which it is created, that happens at run-time.

> If you use a variable out of its scope, compiler will throw an error.

«~»

2. **Default definition value**

> Memory allocation does not happen in declaration whereas in definition memory gets allocated to the variable.

The default definition value of a variable depends upon the storage class of variable. It could be random value or zero, varies according to storage classes.

Storage Class	Scope	Lifetime	Initial Value	Storage Location
Auto	Within the block in which it is $\underline{declared}$	Within the block in which it is declared	Garbage Value	RAM
Register	Within the block in which it is $\underline{defined}$	Within the block in which it is declared	Garbage Value	Register
Static	Within the block in which it is defined	Throughout the main program	0	RAM
Extern	Global scope, unbound by any function	Throughout the main program	0	RAM

«~»

Automatic Storage Class in C

- `auto` is the default storage class for the variables defined inside a function or a block, those variables are also called local variable.

```
auto int max; // Explicit declaration
int min; // by default auto
```

All auto variables defined inside a block or function of body.

```
#include <stdio.h>

auto int max; // Error, illegal to declare auto with global scope
int main(){
        auto int min; // Legal Syntax
}
```

min is defined with garbage value. Auto variables are allocated memory at the run time of the program in the activation record of the function.

Register Storage Class in C

- Similar to `auto` storage class with one slight difference, variables of register storage maybe initialized inside a register of the CPU.
- Register variables come in handy if you want to access a variable frequently in the program.
- It *cuts down the time in memory access and results in faster access* of the variable stored in the register.
- It depends on various parameters such as code optimization techniques of compiler or availability of registers decide whether to allocate register or not.
- If the register is not used the variable is stored in the main memory as in **auto variable**.

```
register int variable_x;
```

- Compilers are smart enough to detect whether a variable needs a register or not. Registers are costly and are rarely available for simple programs.

Static Storage Class

- Static variables are initialized inside `static memory during compile time` of the program and are independent of the function call stack in which they are defined.
  - This makes them alive even after function activation records been destroyed
- Once static variable is defined, it lives until the end of the program.
- Scope limited within the parent block which are local static variables.
- We can create global static variables by defining them at the start of the program which can be accessed anywhere in the program.

```c
#include<stdio.h>
static int out;
// Global static variable implicitly initialized to 0.

void main() {
 static int in = 10;//local static variable explicitly ini to 10
 printf("global: %d, local: %d", out, in);
}
```

If I do this, for global static variables

`int g;`	`c static int g;`
can be accessed in other files too	only available to this file
Linker won't ignore this, since this can be used in other file too	Linker would ignore this, since this is not used in any other file than this

Example 1

```c
#include <stdio.h>
void increment(){
        static count = 0; // at compile time, compiler would remove this. So at run time it won't get executed again and again
        count++;
        printf("%d ", count);
}
int main(){
        increment(); // 1
        increment(); // 2
        increment(); // 3
        return 0;
}
```

«~»

Example 2

```c
#include <stdio.h>
static int y = 5; // global static variable

void fun(){
        static int y; // local static vriable
        y++;
        return;
}
int main(){
        fun(); // local static y → 1
        printf("%d", y); // static variable scope is local but lifetime is end of program, so right now we can't access the local static variable y from fun(), but then we can access global static variable y, since it is under scope. so it will print 5
        return 0;
}
```

«~»

Example 3

```c
#include <stdio.h>
static int y = 1; // global static variable

void fun(){
        static int y; // local static variable
        y=5;
        return;
}
int main(){
        static int y = 2;
        fun();
        printf("%d", y); // 2, local definition is preferred over global
        return 0;
}
```

«~»

Example 4

```
main(){
        int x = 0;
        for(int i=1;i<5;i++){
                x += fun1() + fun2(); // 0 + 4 + 1
                                                                // 5 + 3 + 2
                                                                // 10 + 2 + 3
                                                                // 15 + 1 + 4 = 20
        }
        printf("%d", x);
}
inr fun1(){
        static int y = 5;
        y--;
        return y;
}
int func2(){
        static int y;
        y++;
        return y;
}
```

Extern Storage Class

- **Storage** - Static memory

```
extern int a; // It doesnt' mean that a has its own memory, but it is a way to show that we are
referring to some variable a.
```

Example 1

```
#include <stdio.h>
int max;
int main(){
        int len;
        extern int max; // way to tell compiler there is a global variable max
        printf("%d", max);
        max = 5;
}
```

At compile time we will start parsing from beginning but at run time we start at main().

```
#include <stdio.h>
int max;
int main(){
        int len;
        printf("%d", max);
        max = 5;
}
```

Output will still be 0.

Earlier we had extern, later we didn't, but still it outputted same output. Thing is, we already defined `int max` at the top of the program, which is global variable, so using `extern int max` would just be redundant.

Lets check one more example

«~»

Example 2

```
#include <stdio.h>
int main(){
        int len;
        extern int max; // way to tell compiler there is a global variable max, either in the same file
or in other, it is kind of assuring the compiler that there exist this max variable of int type, but
I'm declaring it right now itself.
        printf("%d", max); // 0
        max = 5;
}
int max;
```

> **Common Practice**
>
> Place definitions of all external variables at the beginning of source file and then omit all the extern delarations

Now, if `int max` in same file then why to use extern, exactly, you dont' need to use extern if the global variable is in same file and also if it is defined at the beginning. But if that global variable is not in same file, then you should use extern.

```c
// main.c
#include <stdio.h.
extern int gInt; // just saying to refer global variable gInt, not creating any memory.
void change_extern(void);
int main(){
        printf("main1 gInt %d\n", gInt);
        change_extern();
        gInt = 5;
        change_extern();
        printf("main2 gInt %d\n", gInt);
        return 0;
}
```

Above code will get successfully ***compiled***, since everything is declared correctly, though definition of change_extern() not there, but it is declared. Even we have assured that there is some `gInt`, so no need to worry.

In **Linking**, we will get linker error, since we haven't defined `gInt` or `change_extern()`.

```c
// extern.c
#inclde <stdio.h>
int gInt; // in static memory so initial value = 0
void change_extern(void){
        printf("change_extern1 gInt %d\n", gInt);
        gInt = 10;
        printf("change_extern2 gInt %d\n", gInt);
}
```

```
main1 gInt: 0 # gInt static variable as value 0
change_extern1 gInt: 0
change_extern2 gInt: 10 # gInt = 10
change_extern1 gInt: 5  # gInt = 5 locally happened
change_extern2 gInt: 10 # inside change_extern() gInt = 10
main2 gInt: 10 # since all thse changes happeinng on one static variable so this will be printed at it
is.
```

«~»

Example 3

```c
#include <stdio.h>
main(){
        extern int num;
        num = 20;
        printf("%d", num);
}
// No compilation error but linker error, since linker would not be able to resolve the reference of
num.
```

«~»

Example 4

```c
#include <stdio.h>
extern int i; // it says there exist some variable i, so refer to it
int main(){
        int i = 5; // this is locally defined so it will be refered by expression just below.
        printf("%d", i);
        return 0;
}
// no compilation or linker error. Linker won't throw error because it is not used here at all, it
said to refer i that exist somewhere outside or in this file, but there was no such i in expression,
and the i that was there, it refered local definition only.
```

«~»

Example 5

```c
#include <stdio.h>
extern int i;
int main(){
        printf("Hello");
        return 0;
}
// extern int i is useless, its not getting used any how, because no expression is there to refer this i, hence linker won't even need to resolve the refernce. So no error and hence this extern not used here.
```

«~»

Example 6

```c
#include <stdio.h>
void func(void);
int main(){
        printf("Hello");
        return 0;
}
// No compile time error, since everything is declared but we won't get linker error too. Since we not using func() hence linker won't even need to link it too, so we won't get any linker error. If we would have been using this func() anywhere, then there would be linker error since there no definition to link to, or reference the definition.
```

Homework 1

```c
extern int s;
int t;
static int u;
main(){ }
```

Q. Assume above program compiles and runs successfully. Which of `s`, `t` and `u` are available to a function present in another file?
A. `u` is static so its scope is this file only, `t` is global variable, so it can be defined as global variable to another file too. `s` is a reference and not in memory itself. It needs to be defined, then only it can be used on any other function outside of file scope. So, answer is **only t**

Homework 2

```c
int main(){
        extern int i; // This will say that refer i that is defined globally, so compiler get assurance that i is defined, so its' okay, pass
        printf("%d", i); // this won't ask for value at compile time, because compiler has assurance of value of i
        int i = 50; // at this point, compiler will get the value of i as 50
        return 0;
}
int i=100; // at this point compiler got the value of i=100, and i=50 will be overwritten
// Now at linking, i would be 100, and 100 will get printed.
```

Important

In C you can't define multiple times but declare multiple times. At multiple definition it will give error else warning.

```c
#include <stdio.h>
extern int i;
extern int i;
int i = 10;
i=5;   // compile time error, you can't do something like this outside block, not even printf
int main(){ }
```

Homework 3

```c
#include <stdio.h>
extern int i;
void func(){
        i++;
}
int main(){
        func();
        i++;
        printf("%d", i); // 2
}
int i;
```

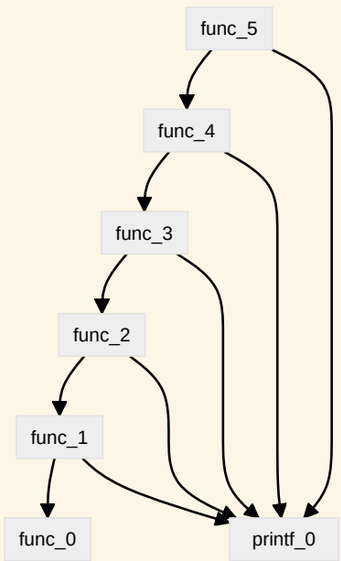
Note

- Linker only available to extern
- By default functions are extern

Recursion

Question

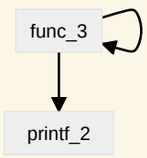
```c
#include <stdio.h>
void func(){
        static int n = 5;
        if(n==0)
                return;
        n--;
        func();
        printf("%d", n);
}
int main(){
        func();
}
```



A. 0 0 0 0 0

Question

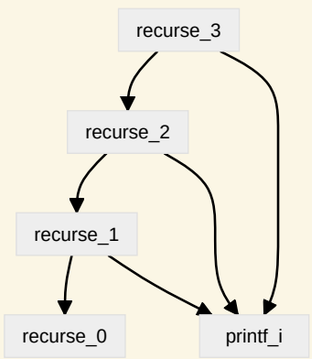
```
void fun(int n){
        if(n==0) return;
        fun(n--);
        printf("%d", n);
}
```



A. This keeps on going, but eventually would hit the recursion limit, and nothing would be printed. Stackoverflow

Question

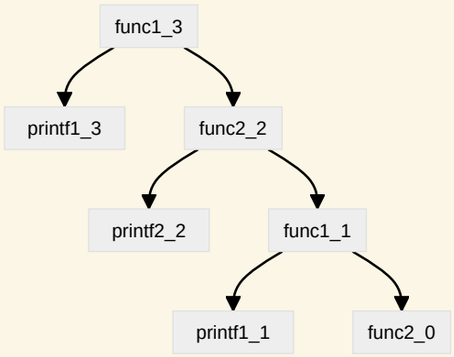
```
void recurse(){
        static int i = 4;
        if (--i){
                recurse();
                printf("%d", i);
        }
}
```



A. i = 0 at the end, hence 0 0 0 0

Question

```
#include <stdio.h>
void func2(int n);
void func1(int n){
        if (n>0){
                printf("func1 %d\n",n);
                func2(n-1);
        }
}
void func2(int n){
        if (n>0){
                printf("func2 %d\n", n);
                func1(n-1);
        }
}
int main(){
        func1(3);
        return 0;
}
```



A. func1 3
func2 2
func1 1

GATE CSE 2004

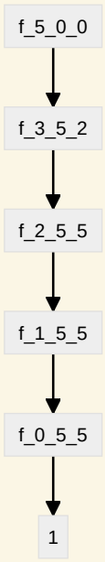
```
int f(int n){
        static int i = 1;
        if (n>=5) return n;
        n = n+i;
        i++;
        return f(n);
}
```

$f(1)$
$f\_n\_i \downarrow$

GATE CSE 2007

```
int f(int n){
        static int r=0;
        if (n<=0) return 1;
        if (n>3){
                r=n;
                return f(n-2)+2;
        }
        return f(n-1) + r;
}
```

$f(5)$
$f\_n\_r\_a \downarrow$

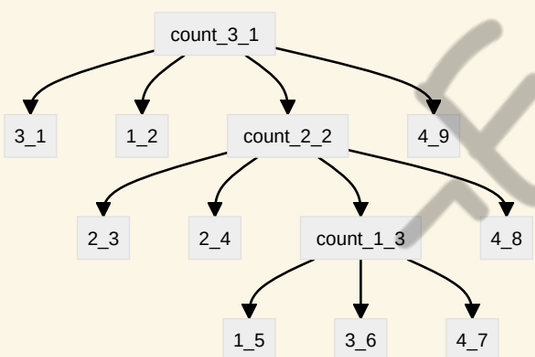


A. 0 + 2 + 5 + 5 + 5 + 1 = 18

Question

```
void count(int n){
        static int d=1;
        printf("%d", n);
        printf("%d", d);
        d++;
        if (n>1) count(n-1);
        printf("%d", d);
}
```

$count(3)$



A. 3 1 2 2 1 3 4 4 4 (last two would be 4 because d is static)

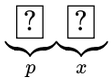
Pointers

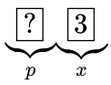
`int a = 5;`

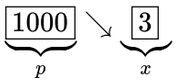
`a = 5`
`&a = 1000`
`int *p = &a`

`p` -> it is a variable that holds the address of integer

```
int *p, x;
x=3;
p=&x;
```







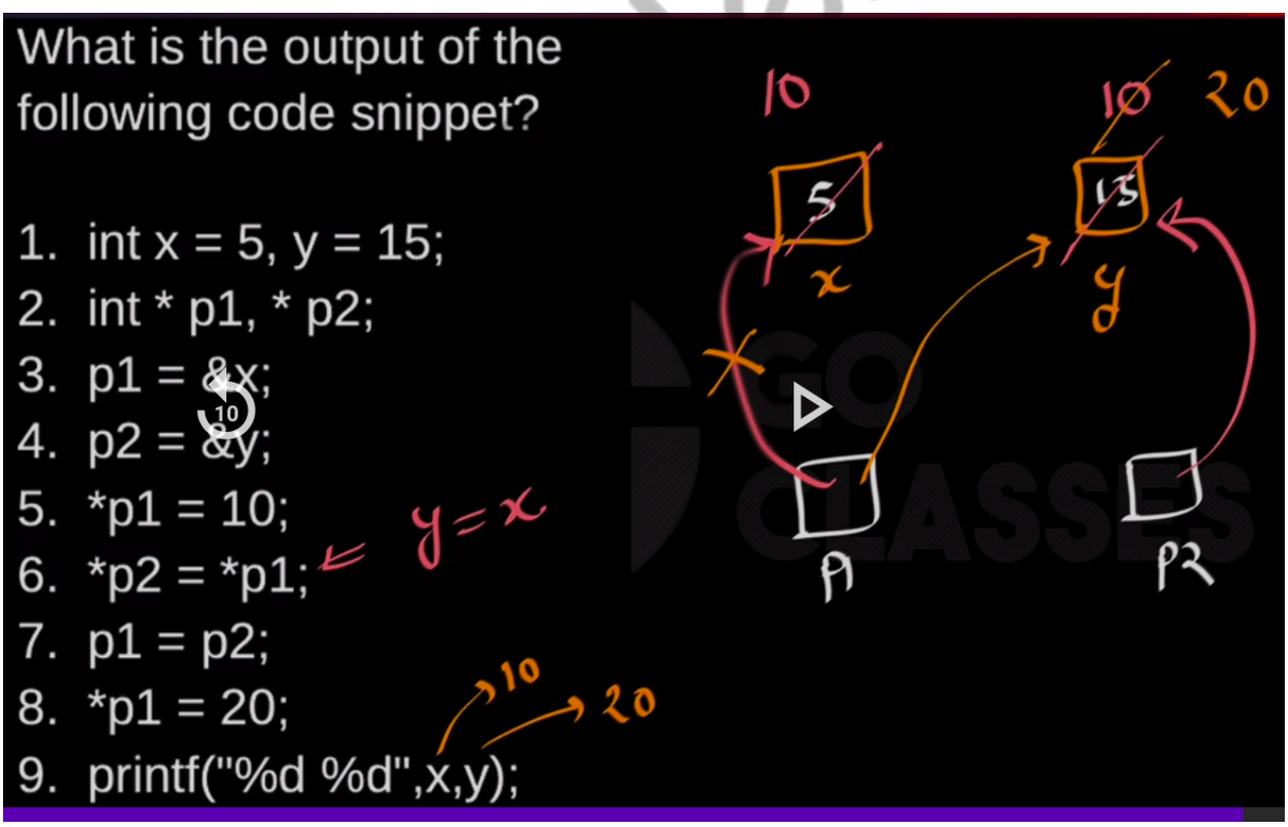
Illegal Uses

1. &125 (pointing at constants)
2. &(x+y) (pointing at expression)
3. `register int x; int *p = &x;`

Undertanding

Example

```c
#include <stdio.h>
int main(){
        int a, b;
        int c = 5;
        int *p; // pointer to variable that holds address
        a = 4*(c+5);
        p = &c; // p is a variable holding address of variable c.
        b = 4*(*p+5); // since p is variable holding address, then *p would be pointing to that address. And that address of c, so *p would be exactly same as c. So this expression we can write as 4*(5+5) = 40
        return 0;
}
```



Passing pointers to functions

```c
main(){
        int x;
        x = 20;
        change(&x); // if we passing address to the variable, then parameter type should be pointer to some type to catch this address
        printf("%d", x);
}
change(int *p){ // thats' why we have pointer to integer, this cam catch the address
        *p = *p + 10; // update the value on the address &x
}
```

GATE 2010

```c
#include <stdio.h>
void f(int *p, int *q){
        p=q; // addr p → addr q
        *p=2; // value at addr p = 2
}
int i=0; j=1; // addr i = 100, addr j = 200
int main(){
        f(&i, &j); // f(100,200)
        printf("%d %d\n", i, j); // 0 2 (j got updated, not i)
        return 0;
}
```

FAKE SWAP

```c
swap(int *a, int *b){
        int *t;
        t=a; // t points to a
        a=b; // a points to b
        b=t; // b points to t
}
main(){
        int x, y;
        x = 100;
        y = 200;
        printf(Before call x, y); // 100, 200
        swap(&x, &y); // earlier a was pointing to x and b was pointing to y, now a is pointing to
y and b pointing to x, but x and y values are not swapped, they are same
        printf(After call x, y); // 100, 200
}
```

SWAP

```c
swap(int *a, int *b){
        int t;
        t=*a; // t gets a value
        *a=*b; // a gets b value
        *b=t; // b gets t value
}
main(){
        int x, y;
        x = 100;
        y = 200;
        printf(Before call x, y); // 100, 200
        swap(&x, &y); // now value has been swapped instead of addresses, hence real swap took
place
        printf(After call x, y); // 100, 200
}
```

Arrays

```c
1. char t[5] ={'a', 'b', 'c', 'd'}; // last one would be "\0"
2. char t[4] = {'a', 'b', 'c', 'd'}; // no null char
3. char s[5] = "abcd" // short form of 1
4. char s[4] = "abcd" // short form of 2
// Just by printing we can't tell if character sequence has null character at the end
```

Dimension not specified

The compiler will deduce the dimension from the initializer list

```
int myArray[] = {1,2,3,4,5,6,7,8,9};
char s[] = "abcd"; // equivalent to char s[5] = "abcd", one for nnull character
char s[] = {'a', 'b', 'c', 'd'}; // equivalent to char s[4]
```

Note

- There is no string concept in C language, you can treat the character array as a string
- If you put something like `char s[] = {'a', 'b'}` then compiler won't treat it as string, if you are given `char s[] = ab` then compiler treats it as string and add extra null character

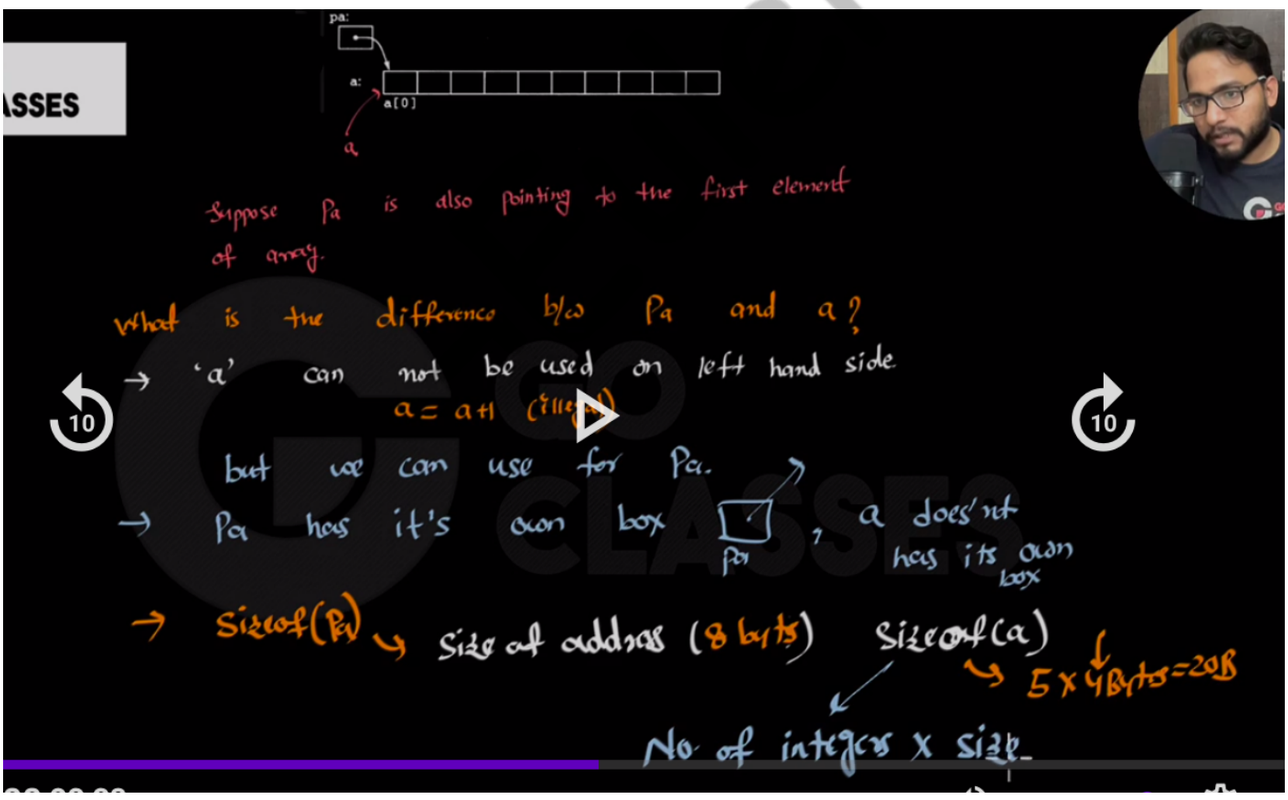
Arrays and pointers

$$
\begin{array}{ll}
a[0] & \boxed{2} \leftarrow a \\
a[1] & \boxed{4} \leftarrow a+1 \\
a[2] & \boxed{6} \leftarrow a+2 \\
a[3] & \boxed{8} \leftarrow a+3 \\
a[4] & \boxed{22} \leftarrow a+4
\end{array}
$$

$a \equiv \&a[0]$
$a+1 \equiv \&a[1]$

hence,

$*(a+i) \equiv a[i]$



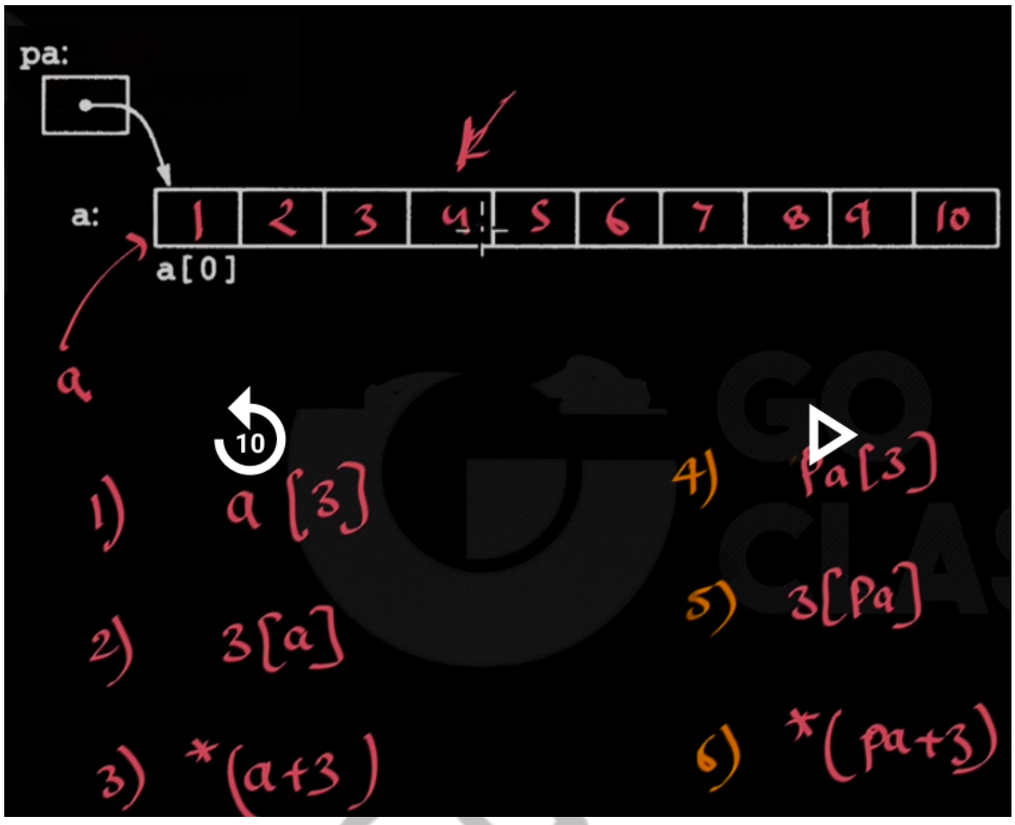
Question

```
int a[] = {3,6,9,12,15};
int *addr = &(a[0]); // addr has address of first element of a
(*addr)+4 = // this will point to the value at that address for first element of a. [3 + 4 = 7]
*(addr+4) = // 4th address from addr, which means from starting, that is a+4, now we want the value
at this addr *(a+4) → [15]
```

Question

Given the declaration `int A[20], *p, x; p=A;`
which of the following statements is ***NOT TRUE.***

✓ `p = A`; is equivalent to `p=&A[0]`
✓ `x=p[4];` is equivalent to `x=*(p+4);`
✓ `x=p[3];` is equivalent to `x=*(A+3);`
❌ `x = *A+7;` is equivalent to `x=A[7];` // NOT TRUE



6 ways to access 4^{th} value

`sizeof()` operator

> The `sizeof` operator is the only one in C, which is evaluated at compile time.

Interesting

```c
#include <stdio.h>
int main(){
        int i=1;
        printf("%d\n", sizeof(i++)); // 4
        printf("%d", i); // 1
}
```

A. Compiler cant' evaluate anything inside `sizeof`. Thing is, its' not compiler job to run it even, we need ALU, registers and all to evaluate, hence compiler just can only evaluate the **size** of its argument.
Compiler will know that `i++` is going to be integer only, thats' all it needs to know, and on the basis of that it gave the size as `4`.

Question

```c
#include <stdio.h>
int main(){
        char c='a';
        char d='b';
        printf("%d\n", sizeof(c+d)); // 4
}
```

A. Because of integer promotion

Question

```c
#include <stdio.h>
int main(){
        char *t;
        char c;
        int i;
        int *p;
        printf("%d %d %d %d\n", sizeof(t), sizeof(&c), sizeof(&i), sizeof(p)); // 8 8 8 8
}
```

A. It doesnt' matter what type of pointer it is, the size of address would always be same and it is 8 bytes (if we have 64-bit architecture).

Question

```c
int main(){
        int a[10];
        int *p;
        p=a; // a is the address of first element, and p holds the address itself, so p will hold the address of first element of a. Since its address, so size of address would be 8
        printf("%d %d\n", sizeof(p), sizeof(a)); // 8 40
}
```

Question

```c
int main(){
        int a[] = {1,2,4,8};
        int *b = a+2; // b would be pointing to the address of value 4
        int *c = b-- +1; // c would be pointing to the address of value 8, then b would point to the address of 2
        printf("%d %d", *b, *c); // 2 8
}
```


Pointer Arithmetic

> $pointer \equiv address$

```c
int *p          p+1 = p + sizeof(int)

char *t         t+1 = t + sizeof(char)

long int *z     z+1 = z + sizeof(long int)
```

Valid Arithmetic on pointers

1. Adding or subtracting an integer to the pointer
2. Subtracting two pointers from each other (*tells the distance in terms of number of elements*)
3. Comparing pointers

Note

Generally we do pointer arithmetic only on elements of arrays

Difference between two pointers

```
int a[10];
int *p1 = a;
int *p2 = p1+3;
printf("%d", p2-p1); // in general arithmetic its (P2-P1)/sizeof(int) = 12/4 = 3
```

Important

- When two pointers are subtracted, both shall point to elements of the same array object.
  - If not, then result can be any garbage value or error

Invalid arithmetic

1. p1+p2
2. p1*p2
3. p1 % p2
4. p1 / p2

Tricky Question 1 (Source: Stanford)

```
char str[] = "Stanford University";
char a = str[1];
char b = *(char*)((int*)str+3); // str is character pointer to first character of "Stanfo...". Now we have explicitly type casted it to integer pointer, but integer pointer has sizeof of 4 bytes and character pointer has sizeof of 1 byte, this means earlier(before typecasting) 'str+1' would have been the address of letter 't', but after type casting now 'str+1' would be pointing to 'f'. So 'str+3' would point to the character that is 12 bytes from 'str', i.e. 'v'. Now we want this in char pointer so we explicitly typecasted with char*.
printf(a); // t
printf(b); // v
```

Tricky Question 2

```
int main(){
        int a[] = {1,2,3,4,5,6,7,8,9,10};
        int *ip = a+2; // point to 3
        int *p1, *p2;
        p1 = (int *)((short*)ip + 4); // short is 2 bytes, but this is integer array, so every element is 4 byte, so we need to skip 2*4/4 elements = 2 elements. It will skip 3 and 4. So it will land on 5.
        p2 = (int *)((short*)(ip-2)+2);
        printf(p1); // 5
        printf(p2); // 2
}
```

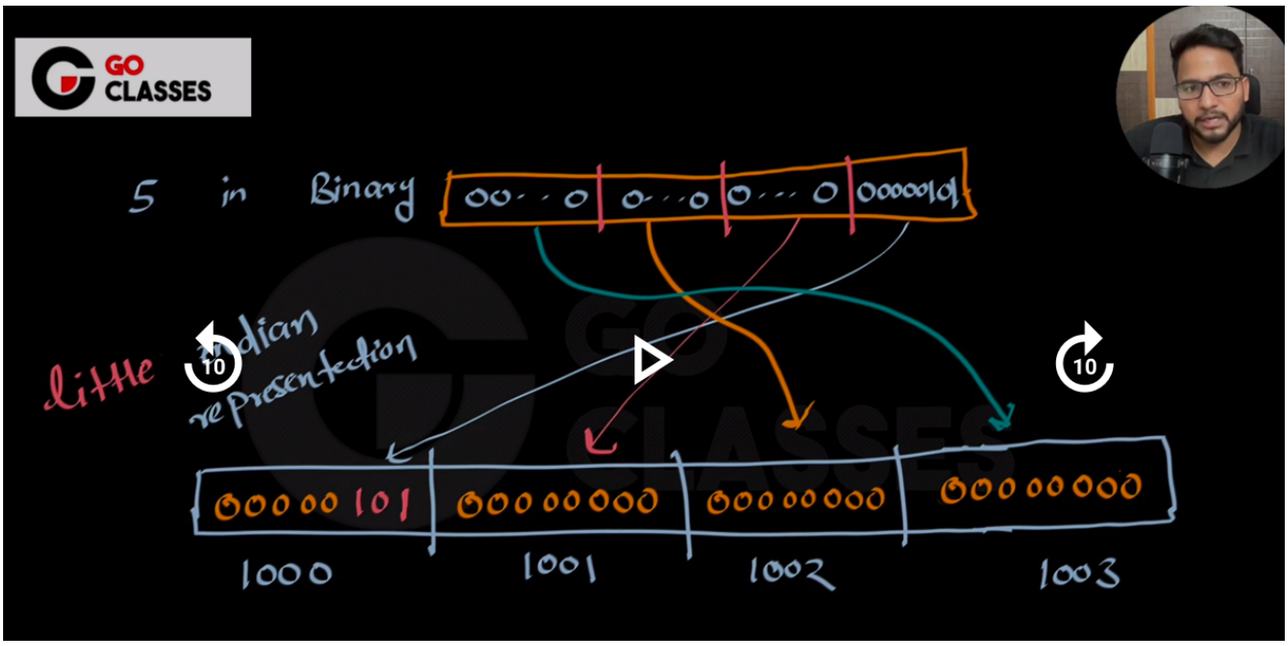
Tricky Question 3

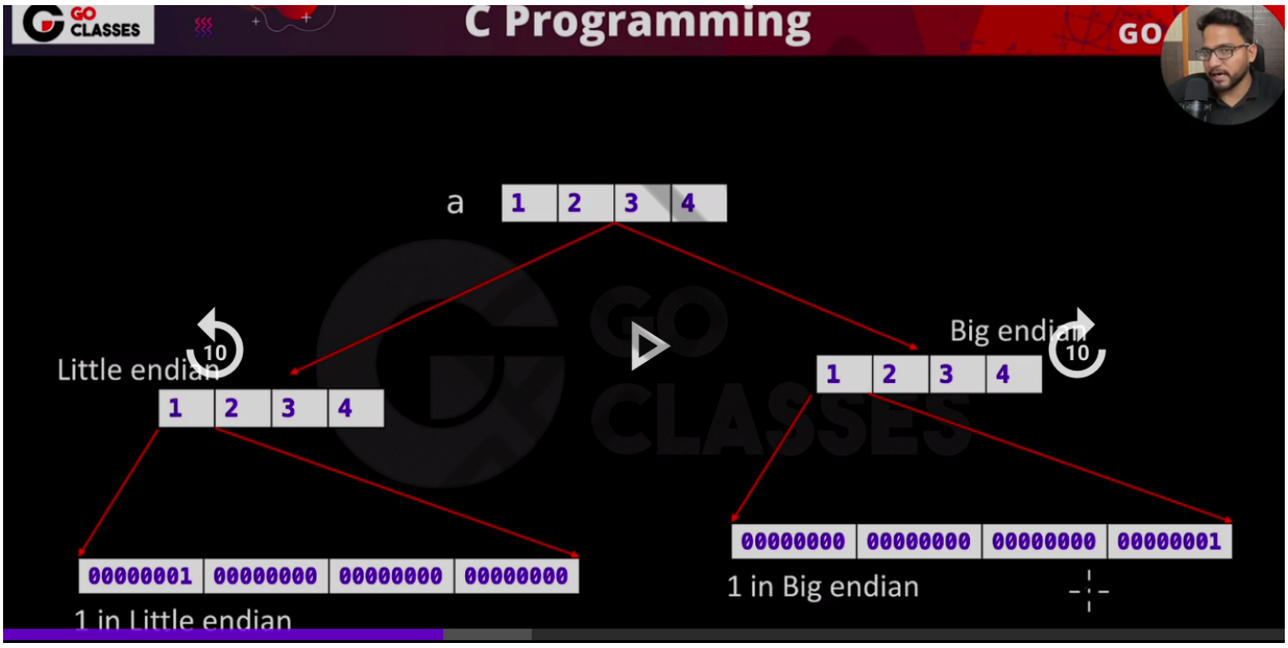
```
int a[4] = {6, 7, 8, 9};
printf("%d\n", (int)(&a[2]-&a[0])); // (int) (2) → 2
printf("%d\n", (int)((char*)&a[2]-(char*)&a[0]); // (int)((char*)(a+2) - (char*)(a)) → (a+i) is integer pointer, converting to char pointer would need us to multiply them with 4 → (int)(4*(a+2) - 4*(a)) → (int)(8) → 8

// We can think of it another way too, every element is integer so needs 4 bytes. now we type casting them to character, hence every 4 byte integer would break down to 4 1-byte characters, now we take difference between address of these two values 6 and 8, it would be 8 bytes, then type cast to integer again, and answer would be 8
```

Little Endian and Big Endian

- Endianess is only applicable to single variables, i.e. only to each array element value, not to the order of elements int the array
- 1-byte variables can't have their byte reversed, so no need to do anything with them





Example

```
int i = 5;
printf("i = %d\n", i);
char *c = (char *)&i;
printf("The first byte= %d\n", *(c));
printf("The second byte= %d\n", *(c+1));
printf("The third byte= %d\n", *(c+2));
printf("The fourth byte= %d\n", *(c+3));
```

A. In big endian

```
i=5
The first byte = 0
The second byte = 0
The third byte = 0
The fourth byte = 5
```

In little endian

```
i=5
The first byte = 5
The second byte = 0
The third byte = 0
The fourth byte = 0
```

Tricky

```c
#include <stdio.h>
int main(){
        int i = 511; // 00000000 00000000 00000001 11111111
        printf("i=%d\n", i); // i = 511
        char *p = (char *)&i; // if (little endian), then p is pointing to 8 bits which are all ones
        printf("The first byte= %d\n", *p); // Integer promoted signed 8 bit to 32 bits, so all would be ones, and it is -1, so -1 is printed
}
```

Passing 1D array to function

Example 1

```c
main(){
        int a[5] = {1,2,3,4,5};
        func(a); // we passing the "address" of the first integer to function and not the whole array itself.
}
void func(int *p){ // we need something that can catch the address and it is pointer to variable, since it is integer array so we will have integer pointer.
        stmts..
}
```

Example 2

```c
main(){
        int a[] = {1,2};
        func(a);
}
void func(int a[]){ // internally int a[] would be converted to int *a.
        stmts...
}
```

Infact, `int a[50]`, `int a[2]` .. gets converted to `int *a`. Compiler ignore the parameter(50, 2, ..) you passing

Strings in C programming

We dont' have `string s;` in C programming language. But only character array `char c[10];` in C.
We can treat this character array as string

```c
char c[10] = "Hello";
printf("%s", c); // It will print everything from adress 100 till it gets the address where it finds the first null character.
printf("%s", c+1); // will start printing from address 101
```

H	e	l	l	o	\0	\0	\0	\0	\0
100	101	102	..3	..4	5	6	7	8	9

```c
char c[5] = "Hello";
printf("%s", c); // This can print "hello" or it can even print more than "hello", thing is, we are dependent on null character "\0".
// By default every location is null character, but what if at address 105 we would be having non null character, then printf would have printed that too until first null charcter we come acress.
```

my_strlen()

```c
#include <stdio.h>
int my_strlen(char *c){
        int i=0;
        while (c[i++]!='\0'); // we are inherently depending on null character, if the character array doesnt' have null chracter then our program will break.
        return i; // it will give one extra value for size, so it should be (return i-1;)
}
int my_strlen(char *c){
        int i=0;
        while(*c != "\0"){
                c++;
                i++;
        }
        return i; // this will give correct value
}
int main(){
        char c[] = "hello";
        printf("%d", my_strlen(c));
}
```

Important

```c
char c[] = "Hello";
c[0] = 'g'; // Valid
char *t = "Hello"; // can't be modified, this is constant (stored in ROM in static area)
t[0] = 'g'; // Invalid
```

Double Pointers

```c
main(){
        int x, y, *p1, **p2;
        x = 100;
        y = 200;
        p1 = &x; // p1 will point to x
        p2 = &p1; // p2 will point to p1
        printf("%d ", **p2); // *(*p2) -> *(p1) -> x = 100
        *p2 = &y; // p1 = &y
        printf("%d ", **p2); // *(*p2) -> *(p1) -> y = 200
}
```

Example

```c
int k, x[3] = {5,7,9};
int *myptr, **ourptr;
myptr = x;          // myptr points to address x
ourptr = &myptr;    // ourptr points to address myptr
k = *myptr;         // k = 5
k = (**ourptr) + 1; // k = *(*ourptr) + 1 -> *(myptr) + 1 -> 5+1 = 6
k = *(*ourptr + 1); // k = *(myptr + 1) -> *(x + 1) -> 7
```

GATE 2008

```
int f(int x, int *py, int **ppz){
        int y,z;
        **ppz += 1; // update made [ 4 → 4+1 = 5]
        z = **ppz; // z = 5
        *py += 2; // update made [ 5 → 5+2 = 7 ]
        y = *py; // y = 7
        x += 3; // local update made [ 4 → 4+3 = 7 ]
        return x+y+z; // 7 + 7 + 5 = 19
}
void main(){
        int c, *b, **a;
        c = 4; b = &c; a= &b; // b is pointer to address c, a is pointer to address b [ a → b → c(4) ]
        printf("%d", f(c,b,a));
}
```

Question

```
void inc_ptr(int **h){
        *h = *h + 1; // *h = q → q+1 (like q++)
}
int A[3] = {50,60,70};
int *q = A; // q pointing to address A
inc_ptr(&q); // since q is pointer itself, and we passing address of it, so we  need double pointer to catch this, such that in one dereference we would be able to reach q and in another dereference we can reach to where q is pointing.
printf("*q = %d\n", *q); // after q++, it would be pointing 60, hence *q → 60
```

Berkeley

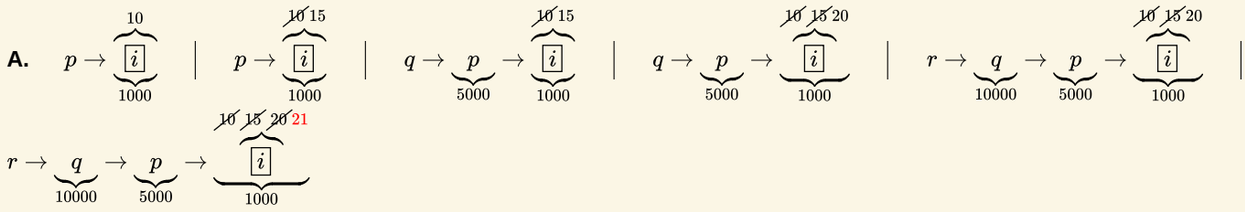
```
int x[] = {2,4,6,8,10};
int *p = x; // addr 2
int **pp = &p; // addr p
(*pp)++; // (p)++ → p points to 4
(*(*pp))++; // (*(p))++ → increment the value at address p by 1 [ 4 → 5 ]
printf("%d\n", *p); // 5
```

Question

```
#include <stdio.h>
void foo(int **p){
        int j = 11;
        *p = &j; // this p is in activation record of foo, and has nothing to do with p of main action record. p is pointing to address j. But this local *p is same as pointer variable p of main activation record, such that p of main activation record is now having value of &j, in other words, its' pointing to j now. Hence the earlier link with value 10 will break and no one would be pointing to that 10 anymore.
        printf("%d", **p); // 11
}
int main(){
        int i = 10;
        int *p = &i; // p pointing to address i, this p is in activation record of main
        foo(&p); // we passed the address of this variable p
        printf("%d", *p); // This is p of main activation record, and in foo() we broke the old connection of it and made a new one to value 11, but that activation record existed only uptil that function was in execution. Now that activation record is destroyed and so is that address of 11, so p is now pointing to null, hence dereference of p(*p) would give runtime error. Since null can't be printed.
        return 0;
}
```

Question

```
int i=10, *p, **q, ***r;
p=&i;
*p=15;
q=&p;
**q=20;
r=&q;
***r=(*p)+1;
printf("%d", i);
```

A. 

GATE 2011

```
char c[] = 'GATE2011';
char *p = c;
printf("%s", p+p[3]-p[1]); // p+E-A → p+5-1 → p+4 → 2011
```

Illegal

```
main(){
        int *p; // declared a pointer variable (not in memory yet)
        *p = 5; // if its not in memory, then we can't set the value its pointing to as 5, this
"may" lead to runtime error
        printf("%d", *p);
}
```


2D Arrays

addr 1000 (1)	**2**	**3**	**10**
4	5	6	11
7	8	9	12
16	15	14	13

Think of this as a 2d array:-

1000 is address of =>

1. 1^{st} element
2. 1^{st} row
3. 1^{st} byte
4. entire 2d array

Let the above 2d array be denoted with a

$a \equiv 1000$	$a[1][2] = 6$	$*(*(a+1)+6) = 9$
$*a \equiv 1000$	$*(*(a+1)+2) = 6$	$*(*(a+3)-2) = 9$
$**a \equiv 1000$	$*(*a+6) = 6$	$a[3][-2] = 9$
$\&a \equiv 1000$	$a[0][6] = 6$	

- a -> pointer to first row
- $a+i$ -> pointer to i^{th} row

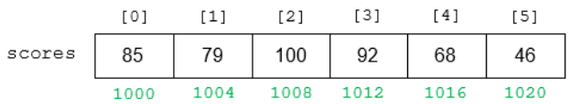
- $*(p+i)$ -> pointer to first element in the i^{th} row
- $*(p+i)+j$ -> pointer to j^{th} element in the i^{th} row.
- $*(*(p+i)+j)$ -> value stored in the cell (i,j)

How arrays are stored in memory

Reference: https://faculty.cs.niu.edu/~mcmahon/CS241/Notes/arrays_and_pointers.html

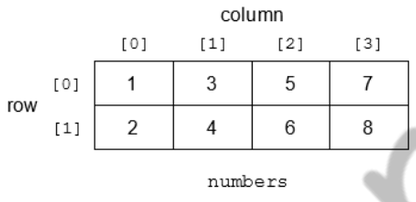
> the `address of an array` is the address of its first element, which is the address of the first byte of memory occupied by the array.

```
int scores[6] = {85, 79, 100, 92, 68, 46};
```

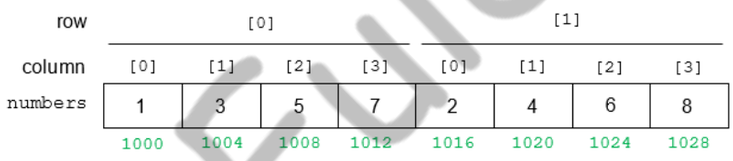


- `&scores[0]` would be 1000

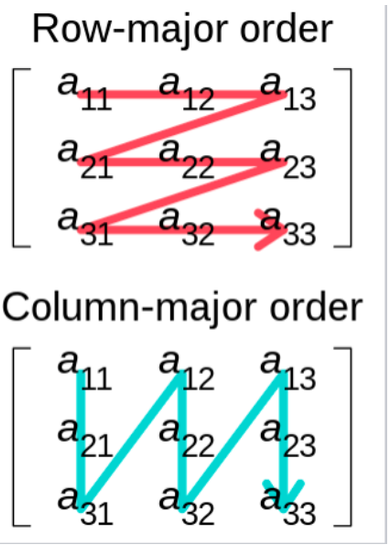
```
int numbers[2][4] = {{1, 3, 5, 7}, {2, 4, 6, 8}};
```



2D arrays in C are stored in on dimension, like this:



All of the elements of row 0 come first, followed by the elements of row 1, then row 2, etc. This is known as *row-major order* .

Illustrarion 1	Illustration 2
$A = a_{y,x} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$ could be stored in two possible ways:	

Address	Row-major order	Column-major order
0	a_{11}	a_{11}
1	a_{12}	a_{21}
2	a_{13}	a_{12}
3	a_{21}	a_{22}
4	a_{22}	a_{13}
5	a_{23}	a_{23}

- The address of the array and the address of the first element of the array are in fact the same number, but they are different data types. The data type of the address of the first element of the array is `int*` ("pointer to an int"), while the data type of the address of the array itself is `int (*)[6]` ("pointer to an array of 6 ints").

Pointer to Array

```
int (*p)[4];
```

p is a pointer to an integer array of size 4.
"here, pointer that points to 0^{th} element and pointer that points to whole array is totally different."

```
// C++ program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include <iostream>
using namespace std;
int main(){
        // Pointer to an integer
        int *p;
        // Pointer to an array of 5 integers
        int (*ptr)[5];
        int arr[5];
        // Points to 0th element of the arr.
        p = arr;
        // Points to the whole array arr.
        ptr = &arr;
        cout << "p =" << p <<", ptr = "<< ptr<< endl;
        p++;
        ptr++;
        cout << "p =" << p <<", ptr = "<< ptr<< endl;
        return 0;
}

// This code is contributed by SHUBHAMSINGH10
Output


p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50
p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64


+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
^               ^
|               |
*(&arr+0)      *(&arr+1)
```


It is different from

```
int *p[4];
```

which is read as, p is array of size 4, and p[0:4] are pointer to integer. This is read so because of precedence order of $*$ and $[]$.

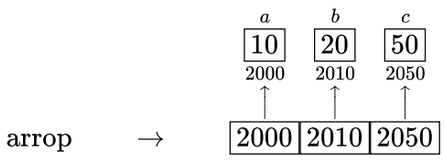
Question

```
#include <stdio.h>
int main(){
        int (*p)[4];
        int a[4] = {1,2,3,4};
        p = &a; // p pointing to address of array a
        for (int i = 0; i<4;++i){
                printf("%d\n", *(*p+i));
        }
}
// i=0 → *(*p+0)) → *(*(&a) + 0) → *(a+0) → 1
// i=1 → *(*p+1)) → *(*(&a) + 1) → *(a+1) → 2
// i=2 → *(*p+2)) → *(*(&a) + 2) → *(a+3) → 3
// i=3 → *(*p+3)) → *(*(&a) + 3) → *(a+3) → 4
```

Question

```c
#include <stdio.h>
int main(){
        int a[] = {1,2,3,4,5,6};
        int *ptr = (int *)(&a+1); // this points to next array starting byte
        printf("%d ", *(ptr-1)); // element at(next array starting byte address - sizeof(int)) = 6
        return 0;
}
```

Array of pointers




```c
int *arrop[3];
int *(arrop[3]); // both same
```

How to access value at b?

1. `b`
2. `*(arrop[1])`
3. `*arrop[1]`
4. `*(arrop+1)` = 2010
5. `**(arrop+1)` = b = 20

```c
char *sports[5] = {
        "golf",
        "hockey",
        "football",
        "cricket",
        "shooting"
}
```

`sports[1][0] = 't';` is invalid, its read only memory, since we have array of character pointers, hence "golf" and all other would be treated as string literals and they can't be modified.

```c
char sports[5] = {
        "golf",
        "hockey",
        "football",
        "cricket",
        "shooting"
}
```

`sports[1][0] = 't';` is valid.

Question

```c
int main(){
        int a[] = {1,2,3}; // addr a → 1000
        int b[] = {10, 20 30}; // addr b → 2000
        int c[] = {5,6,7}; // addr c → 3000
        int *arr[] = {a,b,c}; // {1000→a,2000→b,3000→c}
        int **pp = arr; // pp will be pointing to what arr is pointing to
        pp++; // pp pointing to b in arr
        printf("%d, %d, %d, $d", pp[1][2], **(pp++), (**pp)++, **pp); // *(*(pp+1)+2) → *(c+2) → 7, *(*(b)+0) → 10, (*(*pp+0))++ → (*(c+0))++ → 5, 6
        return 0;
}
```

Passing 2D array to function

- Copies address of first element of 2D array, in case of 2D arrays, first element is first row.

`int a[x][y]`

VALID

```
func(int (*a)[y]){ }
func(int a[][y]){ }
func(int a[x][y]){ }
func(int a[z][y]){ }
```

INVALID

```
func(int **a) { }
func(int a[x][]){ }
```

> **Important**
>
> Inside function all representations would be converted to `int (*a)[y]` format, which would be pointing to first row of array, hence number of columns are important to pass. Number of rows become optional parameter.

Complex pointer declarations

Pointer Arithmetic (difference bw two pointers)

```
int a[10];
int *p1 = a;
int *p2 = p1+3;
printf("%d", p2-p1); // p2=p1+3-p1 = 3
```

> When two pointers are subtracted, both **shall point to elements of the same array object**.
> If not then result can be garbage value or error.

Example

```
int a[2][3][4];
```

a will be pointing to address of first $2D$ array in $3D$ array.

- `a[1]-a[0]` : `*(a+1)-*a`
  - -> both pointing to first row in their respective 2d arrays. Hence total rows in between are **3**.
- `a[1][2]-a[0][1]` : `*(*(a+1)+2) - *(*(a)+1)`
  - -> `*` (pointing to First row of second 2d array + 2) - `*` (pointing to first row of first 2d array + 1)
  - -> `*` (pointing to 3rd row of second 2d array) - `*` (pointing to second row of first 2d array)
  - -> ***pointing to first element of 3rd row of second 2d array - pointing to first element of second row of first 2d array***
  - -> Here when we are saying pointing to row, then it actually means we pointing to first element of that row.
  - -> This subtraction will give difference in terms of integer size, rather than array like we did above, we need to find how many integers are in between.
  - So $2 * 4$ elements in second 2d array, and total $2 * 4$ elements in 1st 2d array, So 8+8 = **16**

Translation to English

declaration	English
*	pointer to
[]	array of
()	function returning

Steps of translation

Step 1

Find the identifier. This is your starting point. Then say to yourself, ***"identifier is"***. You have started your declaration.

e.g. `int *p[10];` -> "p is".

Step 2

Look at the symbols on the right of the identifier, If, say, you find a ***"[ ]"*** there, you would say ***"identifier is array of"***. Continue right until you run out of symbols OR hit a RIGHT parenthesis ***")"***. In case if you fin ***")"*** - see Step 3.

`int *p[10];` -> "p is array of 10"

Step 3

Keep going left (and keep translating to english) until you run out of symbols OR hit a left parentesis ***"("***

`int *p[10];` -> "p is array of 10 integer pointers"

Question

```
int (*p)[10]; // p is pointer to array of 10 integers
int* f(); // f is returning integer pointer
int (*pf)(); // pf is pointer returning integer
int (*pf)(int); // pf is pointer to function taking int argument and returning integer
int (*f) (int *); // f is pointer to function taking pointer to integer as argument and returning integer
int *(*(*fp1)(int))[10]; // fp1 is pointer to function taking arguments of int type and returning pointer to array of 10 integer pointers
```

Some Questions on pointers

Question 1

Meaning of following declaration?

```
int (*p[5])();
```

Answer p is size 5 array of pointer to function that returns integer value

«~»

Question 2

```
int (*x[10])(char *);
```

Answer x is size 10 array of pointer to function that take strings and return integer value

«~»

Question 3

```
#include <stdio.h>
void swap(int **x, int i, int j){
        int *temp;
        temp = x[i];
        x[i] = x[j];
        x[j] = temp;
}
int main(){
        int a[6] = {1,2,3};
        int (*p)[6] = &a;
        int *arr[3] = {a, a+2, a+4};
        p[0][2] = 5;
        swap(arr, 1,2);
        printf("%d\n", *arr[2]);
        return 0;
}
```

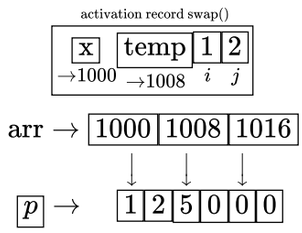
Answer `int (*p)[6]` is pointer to array of 6 integers, and its pointing to a. $p \rightarrow \boxed{1}\boxed{2}\boxed{3}\boxed{0}\boxed{0}\boxed{0}$. `arr` is array of pointers such that $\text{arr} \rightarrow \boxed{1000}\boxed{1008}\boxed{1016}$

$$\begin{array}{l} \text{arr} \rightarrow \boxed{1000}\boxed{1008}\boxed{1016} \\ \qquad\qquad\quad \downarrow \quad \downarrow \quad \downarrow \\ \boxed{p} \rightarrow \quad \boxed{1}\boxed{2}\boxed{3}\boxed{0}\boxed{0}\boxed{0} \end{array}$$

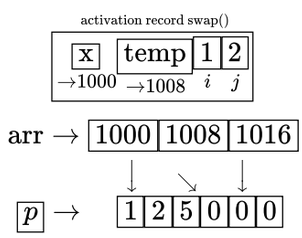
`arr` is pointing to `1000` and that is pointing to `1`, hence to catch this we need double pointer, hence in swap we have double pointer.

`*(*(p)+2) = 5 -> *(*p+2)=5 -> *(addr(1)+2) = 5` => $p \rightarrow \boxed{1}\boxed{2}\boxed{5}$. In swap() `x` is pointing to `1000`.

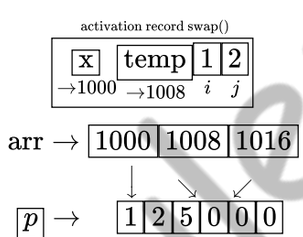
```
temp = x[i]; // temp pointing to where x[i] pointing
```




```
x[i] = x[j]; // x[i] pointing to where x[j] pointing
// x[1] pointing to 0 or *arr[1] pointing to 0
```




```
x[j] = temp; // x[j] will point to where temp pointing to
// x[2] pointing to 5 or *arr[2] pointing to 5
```



Those diagonal arrows are just showing that these address are now pointing to swapped values and not that both are pointing to 0. After swap has happened, activation record is destroyed.

```
printf("%d\n", *arr[2]); // 5
```

«~»

Pointer Arithmetic Combining * and ++/--

Operator	Associativity
`() [] → . ++ -- (postfix)`	Left-to-Right
`sizeof & * + - ~ ! (type) ++ -- (prefix)`	Right-to-Left

`*p++ -> *(p++)`

Question 1

```
int n = 3;
int *p = &n;
printf("n=%d\n", ++*p);
printf("n=%d\n", n++);
```

Answer `p` pointing to where `n` is pointing. `++*p -> ++(*p)` -> 4. This will update `n` also. `n++` -> 4

«~»

Question 2

```
int main(){
        char data = 'a';
        char *ptr = &data;
        printf("%c", ++*ptr++);
        return 0;
}
```

Answer `ptr` pointing to where `data` pointing to. `+++*ptr++ -> +++*(ptr++) -> ++(*(ptr++)) -> ++(a) -> b`

«~»

Void Pointer

```
int main(){
        int a = 7;
        void *p; // can store address of any type
        p = &a;
        printf("Integer variable is = %d", *((int *)p));
        printf("%d\n", sizeof(void*)); // 8
}
```

Since `void` pointer doesn't have well defined type, so we first need to explicitly typecast it and then dereference.

- So developers are using `void*` for creating generic functions.

You think you know pointers?

What is?	`int (*t)[10];`	`int t[10][20];`	`int *t[10]`	`int t[10]`
`t` ?				
`*t` ?				
`(*t)[1]` ?				
`*t[5]` ?				
`*(t+1)` ?				
`t[5]` ?				
`*(t[1]+3)` ?				
`t[1][2]` ?				
`**(t+1)` ?				

Malloc and free

```
int *p;
ip = (int *) malloc(sizeof(int)*10); // allocate 10 ints
ip[6] = 42; // set the 7th element to 42
```

Important

1. malloc returns the void type pointer, we can use that without typecast
2. malloc returns the `void*` because it doesn't know for what purpose this memory is being used.
3. On fail, malloc returns NULL.

Question

```
int foo(){
        int a = 9;
        int b[3] = {2,7,8};
        char *c = (char*)malloc(100);
}
// a,b,c are all in stack. c is also in stack and pointing to memory location in heap but residing
in stack only
```

`free(p)` : it is a way to tell OS that we(programmer) don't need the memory pointed by p.
after `free(p)` -> what OS does with the memory? thats' none of our business.
`*p` -> program may crash.

Question

```
struct node *ptr = (struct node *) malloc(sizeof(struct node));
printf("%p", ptr); // ASSUME THIS PRINTS 0x522f1c0
free(ptr); // this is a way to tell OS that this memory is free and we are not using it, nothing
more or less free does
printf("%p", ptr); // 0x522f1c0
```

Memory Leak

> Memory leak is a memory which hasn't been freed, there is no way to access(or free it) now.

Example 1

```
void func(){
        char *ch = malloc(10);
}
main(){
        func();
        // ch is not valid outside, no way to access malloc-ed memory
}
```

Example 2

```
main(){
        int *x;
        x = malloc(8);
        x = malloc(4);
        // there is no way to free the former memory we allocated in the heap (8 bytes), since we have allocated new memoy block of (4 bytes) and x is pointing to it now.
}
```

Example 3

```
main(){
        int *x;
        x = malloc(8);
        y = x;
        x = malloc(4);
        // No memory leak now, since both can be freed using y and x
}
```

Example 4

```
main(){
        malloc(8);
        // This is memory leak, since we have allocated memory but no way to access it, so no way to free it too.
}
```

Dangling Pointer

> A Dangling pointer points to memory that has already been freed.

Example 1

```
char *func(){
        char str[10] = {'H', 'e', 'l', 'l', 'o'};
        return str;
}
main(){
        char *c;
        c = func();
        *c = 'a';
}
// suppose the address of the str is starting from 1000, and func returned this address, but the moment this activation record of func is destroyed after returning the address, all the allocations of func are also destroyed. Hence at 1000 there will be no value to access. And now c would be pointing to 1000. But *c can't access anything, there exist no useful data. Runtime error
```

Question 1

```
int *p, *q, *r;
p = malloc(8);
q = p;
free(p);
r = malloc(8);
*q = 5; // dangling pointer
```

Question

```
int *q;
void foo(){
        int a;
        q = &a; // q pointing to where a pointing to
}
int main(){
        foo(); // after execution of this, activation record of foo removed. hence q is now pointing to NULL
        *q = 5; // q pointing to NULL so can't assign value, won't make sense.
}
```

Question

```
#include <stdio.h>
void we(void){
        int *ptr;
        {
                int x;
                ptr = &x;
        } // This would be an activation record, with x as variable ptr pointing to x, but after execution, this activation record deleted, hence ptr points to NULL
        *ptr = 3; // dangling pointer, since ptr now pointing to NULL.
}
void main(){
        we();
}
```


Struct

- Used for handling group of logically related data items.
  - e.g. Student name, roll number, and marks
  - real part and complex part if a complex number
- Helps in organizing complex data in a more meaningful way
- Individual structure elements are called members

Example_1

```
struct Box{
        int width;
        int length;
        int height;
};
struct circ{
        double radius;
};
int main(){
        struct Box b;
        struct circ c;
        return 0;
}
```

Declaration

Method 1

```
struct employee{
        int id;
        char name[50];
        float salary;
};
struct employee e1, e2;
```

Method 2

```
struct employee{
        int id;
        char name[50];
        float salary;
}e1,e2;
```

typedef

> typedef is a way in C to give name to a custom type

```
typedef type newname;
typedef int dollors;
typedef unsigned char Byte;

dollors d;
Byte b,c;
```

- We usually use `typedef` with struct.

```
typedef struct bk Books;
Books book1, book2;
```

or

```
typedef struct bk{
        char title[50];
        char author[50];
        char subject[100];
        int book_id;
}Books;
Books book1={"title", "author1", "sub", 123};
Books book2;
book1.author = "author";
```

Array of structures

```
typedef struct bk{
        char title[50];
        char author[50];
        char subject[100];
        int book_id;
}Book;
Book b[10];
b[4].title = "book_4";
```

Pointer as member of struct

```
struct test{
        char name[20];
        int *ptr_mem;
};
struct test t1;
t1.ptr_mem = &x // since its a pointer so takes address
```

Pointer to struct

```c
struct Box{
        int width;
        int length;
        int height;
};
struct Box b;
struct Box *pBox;
pBox = &b;
b.width = 3;
printf("%d", (*pBox).width); // same as printf("%d", b.width)
printf("%d", pBox→width); // same as above
```

Example (last printf not tested, could be wrong)

```c
struct stud{
        int roll;
        char dept_code[25];
        float cgpa;
} class[10], *ptr;
ptr = class; // ptr pointing to class[0], similar to normal arrays, only thing is we now have different type.
printf("%s", ptr → dep_code); // class[0].dept_code
ptr++;
printf("%s", ptr → dep_code); // class[1].dept_code
printf("%s", (int*)ptr+1); // class[1].dept_code
printf("%f", *((char*)((int*)ptr+1)+25)); // class[1].cgpa
```

Question 1

```c
#include <stdio.h>
typedef struct{
        char *a;
        char *b;
} t;
void f2(t *p);
main(){
        t s = {"A", "B"}; // In activation record of main(), variable 's' of type 't' would be present, where s.a would be pointing to "A" in ROM and s.b would be pointing to "B" in ROM.
        printf("%s %s\n", s.a, s.b); // its a %s format specifier, hence it will print string from the address thats' passed to it, here, s.a is address to "A", same for "B". So it will print, A B
        f2(&s); // f2() takes address to the variable of type 't'
        printf("%s %s\n", s.a, s.b); // in f2() we have updated the connections of s.a and s.b, so updated values will be printed, V W
}
void f2(t *p){
        // in activation record of f2 in stack, p would be pointing to s (s of main() activation record), in other words, it stores the value of address to 's'.
        p→a = "V"; // p→a would be an address, which will point to where "V" pointing to in ROM. Hence s.a will be updated to new address, which will be pointing to new location in ROM.
        p→b = "W"; // same will happen with this
        printf("%s %s\n", p→a, p→b); // now, old connections of p→a or s.a to "A" have been broken, and new connection is "V". Hence it will print, V W
        return;
}
```

Question 2

```
struct student{
        char *name;
};
struct student s; // global static variable
struct student fun(void){
        s.name = "newton"; // s.name points to address where "newton" is stored in ROM
        printf("%s\n", s.name); // newtom
        s.name = "alan"; // s.name points to address where "alan" is stored in the ROM, and
disconnects with previous address
        return s; // return object with a member name that is pointing to "alan"
}
void main(){
        struct student m = func(); // main activation record has m, which gets the return value of
type struct student, that has name member pointing to "alan"
        printf("%s\n", m.name); // alan
        m.name = "turing"; // main activation record variable m.name points to address in ROM where
"turing" stored, this doesn't affect s at all
        printf("%s\n", s.name); // alan
}
```

Question 3

```
struct student{
        int x;
};
struct student s[2];
void main(){
        s[0].x = 10;
        s[1].x = 1;
        struct student *m;
        m = s;
        printf("%d\n", m->x++); // (m->x)++ = 10
        printf("%d\n", m->x); // 11
        printf("%d\n", m++->x); // (m++)->x = 11
        printf("%d\n", m->x); // 1
        printf("%d\n", ++m->x); // ++(m->x) = 2
        printf("%d\n", m->x); // 2
}
```


Question 4

```
typedef struct Point {int x; int y} Point;
void changeX1 (Point p1) {pt.x = 11;}
void changeX2 (Point *pt) {pt->x = 33;}
void changeX3 (Point *pt) {(*pt).x = 44;}

int main(){
        Point my_pt = {1,2};
        changeX1(my_pt); // this will update my_pt.x locally, and then activation record will be
deleted, hence no changes made to my_pt in main
        printf("%d", my_pt.x); // 1

        my_pt.x = 1, my_pt.y = 2;
        changeX2(&my_pt); // changes were reflected on the address of &my_pt
        printf("%d", my_pt.x); // 33

        my_pt.x = 1, my_pt.y = 2;
        changeX3(&my_pt); // changes were reflected on the address of &my_pt. (*p).x same as p->x
        printf("%d", my_pt.x); // 44
}
```

Miscellaneous

getchar()

`c = getchar()` is equivalent to `scanf("%c", &c)`

putchar()

`putchar()` is equivalent to `printf("%c", &c)`

```c
#include <stdio.h>

int main(){
        char ch;
        ch = getchar(); // 6
        putchar(ch); // 6
}
```

Similar to GATE CSE 2008

```c
void fun(void){
        int c;
        if((c = getchar()) != '\n')
                fun();
        putchar(c);
}
main(){
        printf("Enter Text\n");
        fun();
}
```

characters will be printed in reverse order in next line

Remember

Assignment operators have low priority order than equality operator

Macros

- All lines that start with `#` are processed by preprocessor

```c
#define MAXLINE 120
char buf[MAXLINE + 1];
becomes
char buf[120+1];
```

```c
#define plusone(x) x+1
i = 3*plusone(2);
becomes
i = 3*2+1 // and not i = 3*(2+1); i = 7
```

```c
#define plusone(x) (x+1)
i = 3*plusone(2);
becomes
i = 3*(2+1) // 9
```

Question

```c
#include <stdio.h>
#define MULTIPLY(a,b) a*b
int main(){
        printf("%d", MULTIPLY(2+3, 3+5)); // 2+3*3 + 5 = 16
}
```

Call by reference and call by value

> ⚠ **Warning**
>
> C uses call by value only and not call by reference. Call by reference not in GATE syllabus right now.

Solution(s)

What is?	`int (*t)[10];`	`int t[10][20];`	`int *t[10]`	`int t[10]`
`t` ?	Pointer to 1-D array of 10 integers	Pointer to first 1-D array	Address of first pointer of array	Address of first element of array
`*t` ?	Points to first element	Pointer to first element of first 1-D array	Pointer to first element of first pointer	Value of first element of array
`(*t)[1]` ?	value at index 1	Value of 2nd element of first 1-D array	Value of 2nd element of first pointer	INVALID
`*t[5]` ?	value of first element of 6th array	Value of first element of 6th row	Value of first element of 6th pointer	INVALID
`*(t+1)` ?	Pointer to first element of the 2nd array	Pointer to first element of the 2nd array	Pointer to first element of second pointer	Value of second element
`t[5]` ?	pointer to first element of 6th array	pointer to first element of 6th array	Pointer to first element of sixth pointer	Value of sixth element
`*(t[1]+3)` ?	Value of 4th element of 2nd array	Value of 4th element of 2nd array	Value of 4th element of second pointer	INVALID
`t[1][2]` ?	Value of 3rd element of 2nd array	Value of 3rd element of 2nd array	Value of 3rd element of 2nd pointer	INVALID
`**(t+1)` ?	Value of first element of 2nd array	Value of first element of 2nd array	Value of first element of 2nd pointer	INVALID