C PROGRAMMING

Data Representation

* Data Types

① int (4 bytes)
- signed
- unsigned.

② char (1 byte)
- signed
- unsigned

③ short (2 bytes)
- signed
- unsigned.

④ float (4 bytes)

⑤ double (8 bytes)

⑥ long double (10 bytes)

- All the data types are represented as 0s & 1s in the memory.

* Basic Number System

* Binary - Decimal Conversion. {Integer}.

* D to B : Repeated Division

Ex. 35

	Remainder	
⇒ $\frac{35}{2}$	1	
⇒ $\frac{17}{2}$	1	∴ 100011
⇒ $\frac{8}{2}$	0	
⇒ $\frac{4}{2}$	0	
⇒ $\frac{2}{2}$	0 ↑	
⇒ 1		

* B to D: Power Multiplication

Ex. 1 0 1 1 0 1

$2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$32 + 0 + 8 + 4 + 0 + 1 = 45$

* Computing 2's Complement:

- Num $\xrightarrow{\text{Flip All Bits}}$ 1's Complement $\xrightarrow{\text{Add 1}}$ 2's Complement

Ex. 1 0 1 1 0 1

1's: 0 1 0 0 1 0

+ 1

2's: 0 1 0 0 1 1

- Method 2: From R to L copy bits till 1st '1' encountered After 1st '1' flip all other bits.

Ex. 1 0 1 1 0 1
(first '1')

2's: 0 1 0 0 1 1

* 2's complement is only calculated for negative numbers. For a pos. no., n, 2's complement is n only.

- +ve number : MSB is 0
  -ve number : MSB is 1.

Ex. 2's complement of :

① $(10)_{10} \Rightarrow (01010)$
② $(-10)_{10} \Rightarrow (10110)$
③ $(-5)_{10} \Rightarrow$
5 is (0101)
-5 is (1011)

Ex: $\underset{\text{-ve\#}}{-28}$ in 16-bits 2's complement representation.

sign bit
28: (11100) ⇒ (0 11100)
-28: 100100 ⇒ 1111 1111 1110 0100
↑ sign bit

Ex: $\underset{\text{+ve\#}}{43}$ in 2's complement form.

43: 0101011

Ex. $\underset{\text{-ve\#}}{-15}$ in 2's complement form.

15: 01111
-15: 10001

* 2's Complement to Decimal

Method 1: 2's $\xrightarrow{\text{2's compl}}$ ± Bin ⟶ ± Decimal.

Ex: 1011

-ve #

2's : 1011

Bin: -0101

Dec. ⇒ -5

Ex: 01101

+ve #

2's : 01101

Bin: +01101

Dec: 13

Method 2: Take sign bit as ~~power~~ -ve power & rest bit as normal & convert to decimal:

Ex. 0101

2^3 2^2 2^1 2^0

$-0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$\Rightarrow +5$

Ex: 10010

2^4 2^3 2^2 2^1 2^0

$\Rightarrow -1\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

$\Rightarrow -16 + 2$

$\Rightarrow -14$

* Copying Number to Higher Bit Representation:

- While copying a number from 'n' bit representation to 'm' bit representation ($m > n$) copy the MSB bit for $(m-n)$ empty bits for signed numbers.

Ex. 1001 m = 8

11111001

Ex: 01101 m = 7

0001101

- For unsigned, just put 0s in $(m-n)$ bits while converting from 'n' bits to 'm' bits.

. This step is called "Sign Extension".

* Range of Signed & Unsigned Representation

. Unsigned: For K bits range is

$$0 \text{ to } (2^K - 1)$$

. 2's Comp: For K bits range is

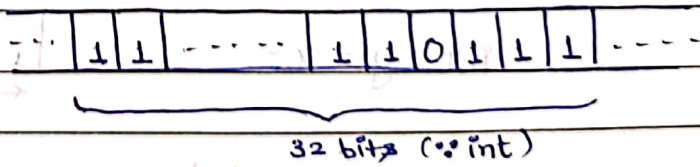
$$-(2^{K-1}) \text{ to } (2^{K-1} - 1)$$

-2^{K-1} is represented as: 1 00...0 (K-1 times)
↳ special representation

* Data Representation

Integer is represented in memory in 2's complement form.

Ex. int y = -9;



. Constant integers are considered to be signed by default.
. In printf, format specifiers are used to clarify how to treat the binary pattern. The type information is ignored.

Ex. printf ("%d", y);

Tells to treat y as signed 32 bit integer.

Ex. printf ("%u", y);

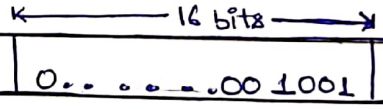
Treats y as 32 bit unsigned integer.

Ex. unsigned int x = -9;

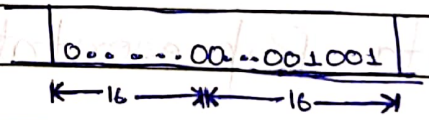
The pattern that was in previous example for 2's complement for -9, will remain same. In memory same binary data will be stored.

* Type Extension

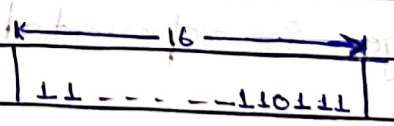
Ex. short int x = 9;



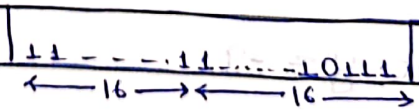
int y = x;



Ex. short int x = -9;



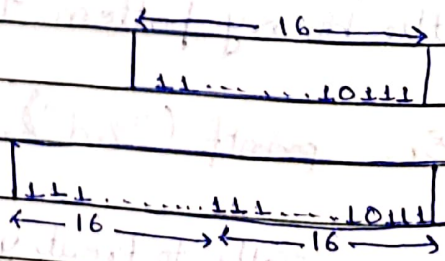
int y = x;



data type

Based on the ~~binary representation~~ of RHS, the extension will extend 1s or 0s.

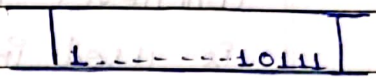
Ex. short int x = -9;
unsigned int y = x;



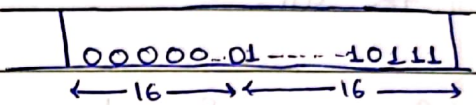
- If the data type of source is signed then the MSB is extended, else 0 is extended.

- If the data type of source is unsigned, then 0 is extended else 1 is extended, if the integer is < 0. (or +ve)

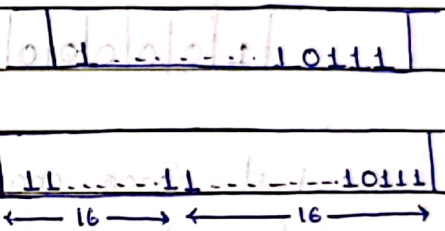
Ex. unsigned short int x = -9



int y = x;



Ex. short int x = -9.
unsigned int y = x;



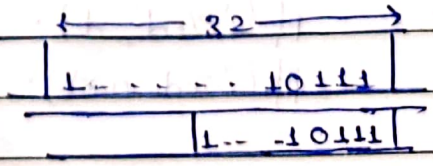
Ex int x = -1;
unsigned int u = x;

pf ("%d", x); -1
pf ("%u", x); $2^{32}-1$
pf ("%d", u); -1
pf ("%u", u); $2^{32}-1$

* Type Truncation

Just cut ~~from~~ starting from right the number of required bits.

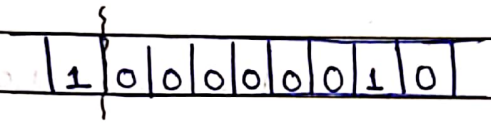
Ex. int x = -9;
short int y = x



* Integer Promotions

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

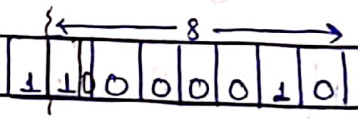
Ex. unsigned char c = 258;



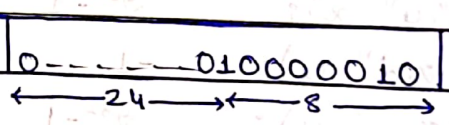
but since char is of 1 byte (8 bit) only the data gets truncated.

pf ("%d", c); & 2

Ex. unsigned char c = 386;



pf ("%d", c);

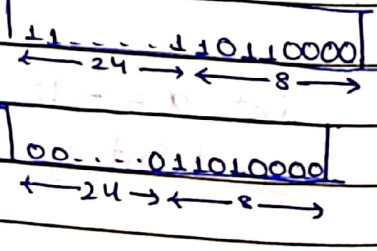


integer promotion

Ex. char a = 30, b = 40;
char d = a * b;

pf ("%d", d); -80
pf ("%d", a*b); 1200

$(1200)_{10}$ =
(0000 0100 1011 0000)



Ex. char a=30, b=40;
unsigned char d= a*b;

pf ("%d", d); 176.

① Integer promotion
a*b 0000 ---- 0000 0100 1011 0000
←16→

② Truncation
d = 1011 0000.

③ Integer promotion.
pf ("%d", d); 0000 --- 0000 0000 1011 0000
←16→

Ex. signed char f= -65;

$(-65)_{10}$
(1111 1111 1011 1111)

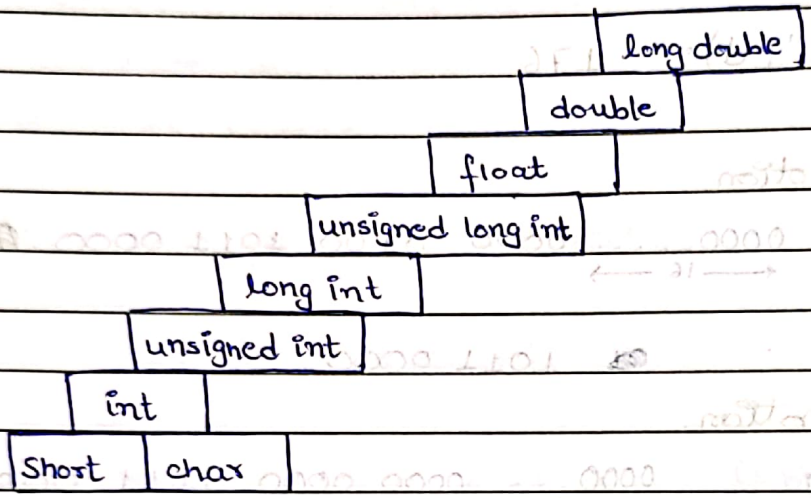
pf ("%d", f); -65
pf ("%u", f); Huge Number

① Truncation
f= -65
←24→ ←8→
1111 ---- 1011 1111

② Integer promotion
pf ("%d", f);
1111 ---- 1111 1011 1111
←24→ ←8→

③ Int promotion
pf ("%u", f);
1111 ---- 1111 1011 1111

* Type Conversion



When values from any two different levels of data types are operated, then the value of lower level gets converted to higher level to match the other value.

The result's data type is also same as the higher of the two levels.

Ex. int v1=10, v2=3;
float res;

res = v1/v2;

pf("%f", res); 3.00...0

res = (float) v1/v2;

pf ("%f", res); 3.33....3

res = v1/(float) v2;

pf ("%f", res); 3.33....3

Ex. int main()
{
unsigned int a = 1000;
int b = -1;

if (a > b) pf ("a is Big");
else pf ("b is Big");

return 0;
}

Output: b is Big.

'b' promoted to unsigned int will be $(2^{32}-1)$, i.e b is (11.....1) in memory.

Ex. int p = -1;
unsigned u = 0;
char c = p + u;

pf ("%u", c);

Output: $2^{32}-1$

Ex. Same question but, u = 1

Output: 0

c = p + u ⇒ 111...1 + 1 will overflow and give 10000...0 (1 + 32)

∴ c = 00000000

Ex. int p = -1;
unsigned u = 1;

```
if (p-u>1)
  pf("YES");
else
  pf("No");
```

Output: YES

Ex. int p = -1;
unsigned u=1;

```
if (p-u > -1)
  pf("YES");
else
  pf("No");
```

Output: No

p-u will be 111...1 - 1 = 111...10
(K ← 32 → 1) (K ← 31 → 1 ↔ 1)

∴ (p-u) > 1 evaluates true.

but -1 will be treated as "unsigned" since (p-u) {the other operand for >} is "unsigned"

∴ (p-u) > -1 ≡ 11...10 > 11...11
evaluates to false.

C PROGRAMMING
CONDITIONALS

* If Statement

Syntax:
```
if (test_expression)
{
    statement_block;
}
```

If { } is not used to enclose the "if statement block" then the scope of the if statement is till the first ';' (semicolon).

- 0: False Other than 0: True.

* Else Clause

Syntax:
```
if (test_expression)
{
    Stmt_block;
}
else
{
    Stmt_block;
}
```

Ex.

```
if (...)          // if 1
  if (...)        // if 2
    pf(...);
else
    pf(...);
```

Else will associate with which if?

"if 2". Since else always associates with the closest ~~else~~ if, when scope isn't specified.

Ex.

```
if (...) {        // if 1
    if (...)      // if 2
        pf(...);
}
else
    pf(...);
```

Else will associate with "if 1".

* Switch Statement

Syntax:

```
switch (expression) {
    case v1 :
        Block 1;
        Break;
```

classmate
Date
Page

:

case V-n:
    Block-n
    Break;
default:
    Block-(n+1);
}

- expression in switch statement should yield an integer value.
- the case statement & default statement can be in any order, but each case value is unique.
- switch starts executing from the first case that matches the expression result either till end or till first break statement encountered.
- case values can't be variable.

Ex:
```
int x=0;
switch(x){
    case x: pf("YES");
}
```
Not Allowed.

Ex. int v = 0;

```
switch (v) {
  default:
    v++;

  case 2:
    pf("A");
    break;
  case 1:
    pf("B");
}
pf("%d", v);
```

Output:

A1

- Default is only visited when no ~~value~~ case value matches the expression, no matter its position.
- condition in switch is only checked once.
- Any stmt within switch block, but not under any case, is never executed.

C PROGRAMMING

CONDITIONALS

* While Loop

Syntax:
```
while ( condition) {
        statement_block;
}
```

- If condition ≠ 0 statement block is executed, otherwise, not executed.

* For Loop

Syntax:
```
for ( expr1; expr2; expr3)
{
    statement_block;
}
```

- In general:
  - expr1 is used for initialization
  - expr2 is used for test condition
  - expr3 is used for incrementation/decrementation

- Execution happens as:

$E_1\ E_2\ (\$\ E_3\ E_2)^{n \text{ times}}$

E_1: Expr1
E_2: Expr2
E_3: Expr3
$\$$: Statement_Block
n: no. of times Expr2 is ≠ 0

- All the three expressions & the statement block are optional.

Ex. for (p=10; p>0;)
    p -= 1;
    pf ("%d", p);

Output: 0.

Ex. int i=0;
    for (; i<=9;)
    {
        i+=1;
        pf ("%d ", i);
    }

Output: 1 2 3 10

Ex. int i;
    for (i=0; i<=3; i++);
        pf ("%d", i);
    return 0;

Output: 4

* Do While Loop:

Syntax: do {
        stmt_block
    }
    while (condition);

classmate
Date
Page

- Execution pattern: $S(CS)^nC$

S: Statement Block
C: Conditional
n: # times condition is true.

Ex:
```
int n=0, m=1;
do {
    pf("%d", m);
    m++;
}
while (m <=n);
```

Output: 1

* Break & Continue Statement

- Break statement can only be used within switch & loop constructs to break from the flow of current execution.

- Continue is used within loop constructs to skip any statements after it till the end of current loop & directly execute expr3 {for loop} or conditional {while & do-while}.

Ex:
```
int x=0, y=10;
for ( ; x<y; x++)
{
```

```
    if (x == 3) continue;
    if (x == 6) break;
    pf ("%d ", x);
}
```

Output: 0 1 2 4 5

Ex. int i=1, j=1;
while (i<10):
    j = j*i;
    i = i+1;
    if (i==y):
        break;

(A) (i==10) or (i==y)
(B) if y>10 then (i==10)
(C) if j=6, then y==4

All of A, B & C are true.

C PROGRAMMING

Operators & Expression

* There are ~~three~~ five types of operators:

1. Relational
2. Logical
3. Assignment
4. Increment & Derement
5. Arithmetic

* Precedence & Associativity:

1.	(), [], ., →, a++, a--	Left to Right
2.	++a, --a, + (unary), - (unary), sizeof, (type), !, ~, &, *	Right to Left
3.	*, /, %	
4.	+, -	
5.	>>, <<	
6.	>=, >, <, <=	Left to Right
7.	==, !=	
8.	&	
9.	^	
10.	|	
11.	&&	
12.	||	
13.	?:	Right to Left
14.	=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=	Right to Left
15.	,	

* Comma Operator: Evaluates & discards the left operand and returns the right operand.

Ex: int a = (3, 4);
pf("%d", a);

Output: 4

Ex. int a = 3;
int b = (a = 1, 2);
pf("%d", a);

Output: 1

* Increment / Decrement operators can't be used with expressions & constants, only variables {∵ variables have address}.

Ex: ++(a+3*b) Wrong
Ex: 9++ Wrong.

Ex. int a=9, b=12, c=3;
int x = a - b/3 + c*2 - 1;
pf("%d", x);

Output: 10

Ex: int j = 2 * 3 / 4 + 2.0 / 5 + 8 / 5;
pf("%d", j);

Output: 2

⇒ ((2*3)/4) + (2.0/5) + (8/5)
↓1.5 ↓0.4 ↓1.6
⇒ (1) + (0.4) + (1)
↓2.4
⇒ (2)

* If power operator is used in pseudocode (^) then its precedence is between '2' & '3' on the chart and its associativity is right to left.

Ex. $2 \hat{} 3 \hat{} 2 = 2 \hat{} (3 \hat{} 2) = 512$ ✓

$\neq (2 \hat{} 3) \hat{} 2$ ✗

Ex.
```
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: pf ("\n%d", i+K);
        case 3: pf ("\n%d", i+K);
        default: pf ("\n%d", i+K);
    }
}
```

Output:

-1	3
0	3
1	
1	
1	
2	
2	
2	

Sol: $j = (2*3/4) + (2.0/5) + (8/5)$

$\Rightarrow (1) + (0.4) + (1)$

$\Rightarrow 2$

$K = K - (--j)$

$\Rightarrow 0 - (1)$

$\Rightarrow -1$

* Bitwise Operators:

- number & 0x1 checks for even & odd.
- Left shift can be risky since it can change MSB from 0 to 1 or 1 to 0, in case of signed no., it leads to sign change.
- ~ takes bitwise complement (same as 1's complement).

* Ternery Operators:

Syntax: expr1 ? expr2 : expr3;

* Ex. int i, j = 2;

```
for (i=0; i<=0, j>=0; i++)
{   pf("%d", i+j);
    j--;
}
```

Output:
2 2 2

Ex. int i, j = 2;

```
for(i=0; i<0 && j>=0; i++
{   pf("%d", i+j);
    j--;
}
```

Output:
No Output

Comma ignores left operand & returns right.
&& returns false, since i>=0 always.

* Assignment operator assigns the value to the lvalue on LHS & returns the assigned value.

* Short Circuiting

Ex. int a=1, b=1, c=1;

```
if (a == b || c++)
    pf("%d", c);
```

Output: 1

(a==b) || (c++)

↑ evaluated first & returns true so (c++) doesn't get evaluated.

* Order of Evaluation: In case of "&&" and "||" operator always evaluate the expression left to right irrespective of operator precedence.

expr1 && expr2 expr1 || expr2

Evaluates expr1 before expr2.

Ex. int i=0, count=0, j;

```
for (j=-3; j<=3; j++)
{
```

```
        if ((j >= 0) && (i++))
            count = count + j;
    }
    count += i;
    pf ("%d", count);
```

Output: 10

for j = -3, -2 & -1, (j>=0) will evaluate false so i++ won't get evaluated.

for j = 0, (j>=0) evaluates true, & and i++ incr. i to 1, but returns 0.

for j = 1, 2, 3, (j>=0) is true & i++ returns 1, 2 & 3 & incr i to 2, 3, & 4

```
                  j  j  j  i
count = 1 + 2 + 3 + 4  = 10.
```

Ex. int k, i = 50, j = 100, l;

```
    l = i | (j && 100);
    k = i || (j || 100);
    l = i & (j && 100);
    pf ("%d %d %d %d", i, j, k, l);
```

Output:

51 100 1 51

- Conditional operator only return 0 or 1.

Ex. e1 || e2 && e3

⇒ e1 || (e2 && e3)

But still due to order of evaluation, e1 will be evaluated first.

Ex. e1 || e2 && e3 || e4 && e5

(e1 || (e2 && e3)) || (e4 && e5)

Order of evaluation: e1, e2, e3, e4 & e5.

```
Ex. int x=1, y=0, z=5;
    int a = x && y || z++;
    pf ("%d", z);
```

Output: 6

```
Ex. int i=1;
    if (i++ && (i==1)) pf("y");
    else pf ("N");
```

Output: N

```
Ex. int i=-3, j=2, K=0, m;
    m = ++i && ++j && ++K;
    pf ("%d %d %d %d", i,j,K,m);
```

Output: -2 3 1 1

Ex. int i=-3, j=2, k=0, m;
m = ++i || ++j && ++k;
pf("%d %d %d %d", i, j, k, m);

Output: -2 2 0 1

C PROGRAMMING
FUNCTIONS

* function_type function_name (parameter list)
{
...
...
...
...
}

* Function Declaration

It is a blueprint of a function, consisting of:
- Function (return) type.
- Function name.
- Parameter list
- Terminating semicolon.

Function declaration helps the compilex to know about the function, so that, if the func. is called in other file by including it, then the compiler doesn't throw error, and allows for dynamic linking later on.

* Function Definition

The actual function body.

* A lot of the time, the func. prototype is put before main function & the definition after main.

* The default return type in C is int & is optional to write.

Ex.

```
int mul(int x, int y)
{
    return x*y;
}
```

≡

```
mul(int x, int y)
{
    return x*y;
}
```

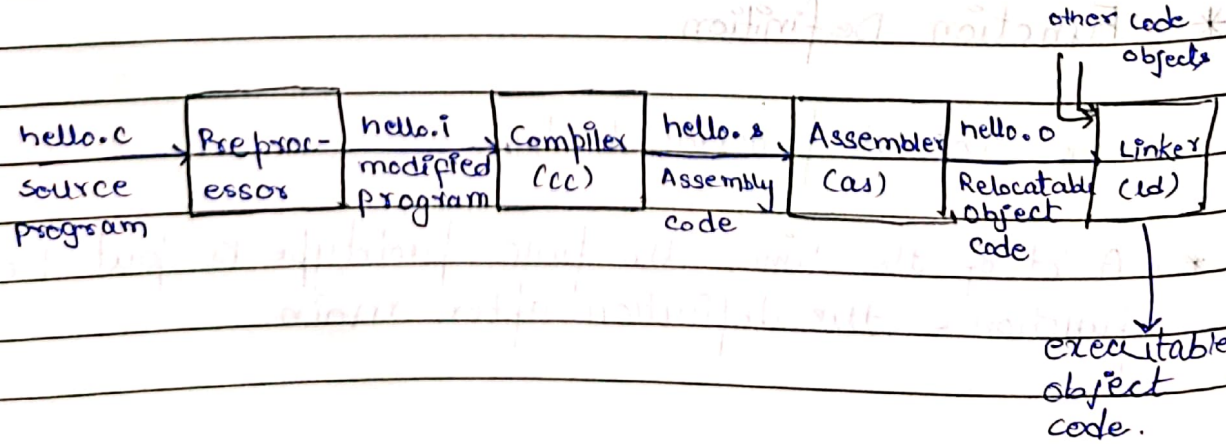
Ex.

```
#include <stdio.h>

main () {
    int x = fun (2,3);
    pf ("%d", x);
}

int fun (int x, int y)
{
    return x+y;
}
```

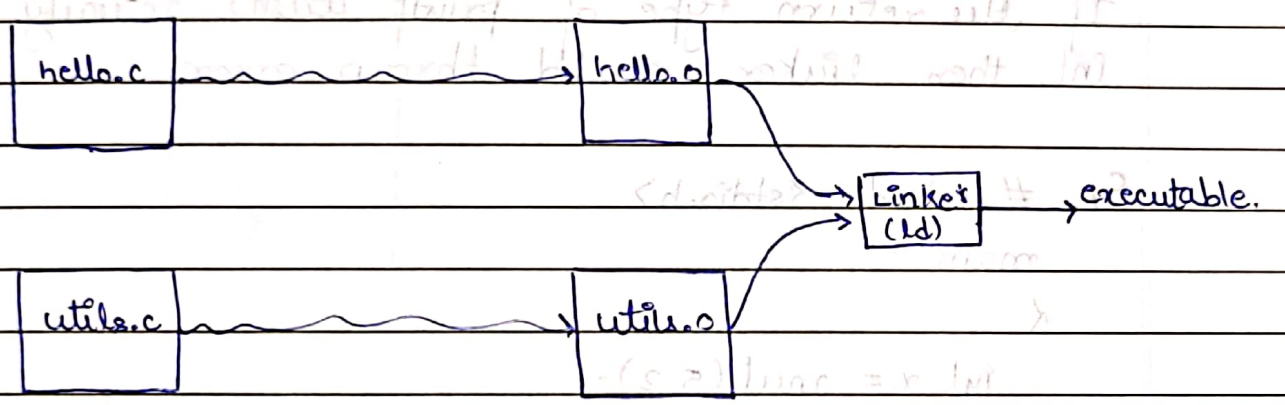
This works fine. Compiles assumes the return type of fun as the default int.

During runtime, the returntype matches to "int".

* The Compilation System



- Multiple independent compilation units can be run separately to obtain the object code, then they can be linked together.



For example a func. can be used in "hello.c" by putting just its prototype in "hello.c" & its definition in "utils.c".

Similarly a variable from "hello.c" can be used in "utils.c" by declaring it in "utils.c" as, "extern int x".

- Similarly, "stdio.h" only contains the prototype. The object code is automatically linked for the standard functions.

Ex.

```
main() {
    pf("hello");
}
```

Compiler assumes that printf has default return type. int, since prototype not included. Successfully gets compiled. At

classmate
Date
Page

linking phase, compiler verifies that the assumption is true or not. Since true, linker links printf successfully.

If the return type of printf wasn't actually int, then linker would throw error.

Ex. #include <stdio.h>
main()
{
int x = mul(5,2);
pf("%d", x);
}

This gets compiled successfully, with the compiler assuming the return type of mul as int (default).

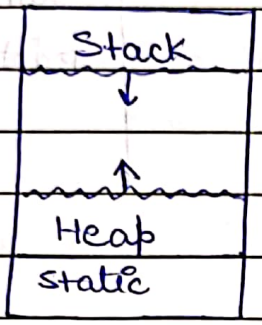
At linking phase, linker looks for the definition of mul with return type int to resolve reference. Since it can't find the definition, it throws error.

* Two main functions are not allowed within single executable.

C PROGRAMMING

Memory Layout

* Broad Overview of Memory



The lifecycle of a program can be broadly divided into two parts:

(1) Compilation Time
(2) Running Time

* When the compiled program is run, the compiled code is loaded into the static memory space.

* Stack & Heap grow in opposite direction as per their memory need.

- Since static area consists of fixed data (code/text, static variables, etc), it doesn't grow or shrink.

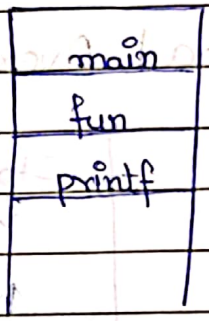
* Stack Memory

Stack memory is used to store activation records,

local variables, etc, during run-time.

Ex.

```
main ()
{
  fun();
}

fun ()
{
  int a=5;
  pf("%d", a);
}
```



* Heap Memory

Used for runtime allocation of memory as requested, using, malloc() & free() function.

* Static Memory

Used for compile time allocation for code, global & static variables.

* Storage Classes

Storage classes tell the location, scope, lifetime of a variable.

(i) Auto
(ii) static
(iii) Extern
(iv) Register

* Auto

- Storage space: Stack
- Initial value: Garbage
- Scope: within Block
- Lifetime: End of the block

- auto is the default storage class for local variables.

Ex:

```
f() {                    f() {
  auto int a;      ≡       int a;
}                        }
```

	Scope	Lifetime	Possible?
①	T	T	Yes. When the block is executing
②	F	T	Yes. Func. called from the scope. The var. is alive but inacessible from function's scope.
③	T	F	Not possible.
④	F	F	Yes. After end of the block.

- Auto variables are local variables.
- Allocated locally on stack, so runtime allocation.
- Auto variables are not available to the linker. Since they are local, linker can't use ~~local~~ auto vars. in another file.

- auto vars must always be within a block {}.

Ex.

```
auto int a;
main ()
{...}
```

Invalid.

* Static

- Storage Location: Static Memory
- Initial value: zero.
- Scope: within block.
- Lifetime: Till end of program
- Static variables aren't available to the linker.

Ex.

```
:
{
  int a=5;
}
:
```

← is 'a' alive? No
is 'a' visible? No

```
:
{
  static int a=5;
}
:
```

← is 'a' alive? Yes
is 'a' visible? No.

- static variables can be local as well as global.

Ex. static int a=5;

```
main ()
{
  --
  --
}
```

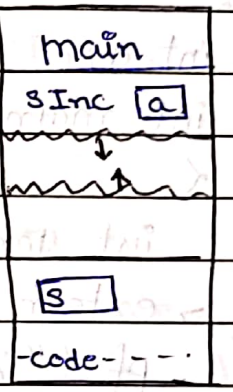
Since static vars are not available to linker, 'a' is limited to the file only & can't be used in other files.

Global variables without static ~~register~~ keyword can be used in other files.

Ex.

```
void sInc()
{
    int a=2;
    static int s;
    s = s+a;
    pf("%d", s);
}

int main()
{
    sInc();
    sInc();
    sInc();
}
```

Output: 2 4 6



'a' is stored in ~~Heap~~ stack, whereas 's' is stored in static area.

`static int s;` line gets executed only once during entire runtime of program.

* Extern

- Storage location: Static Memory
- Initial value: zero
- Lifetime: Till end of program.
- Scope: Global
- Extern vars. are available to the linker.

* extern int a;

The extern Keyword doesn't allocate memory, but is used to refer a variable/function from same file or other file. → global

Ex.

```
int a;
int main()
{
  int ten;
  extern int a;
  pf ("%d", a);
}
```

refers to the global variable 'a'.

This line is useless, since the global variable 'a' is in the same file, and declared before use.

Ex.

```
int main()
{
  extern int a;
  pf ("%d", a);
}

int a;
```

Here the line is useful. It tells the compiler to not throw error, since 'a' is somewhere else in the program.

If the line was removed, there would be error thrown by the compiler.

- This is avoidable by putting all global variables at top & then use, if using single file.
- Extern is useful when multiple files are there & a variable from one file is used by other file.

Ex: // main.c

```
extern int counter;   —(1)
void incC();          —(2)

int main()
{
  pf("%d", counter);
  counter = 5;
  pf("%d", counter);
  incC();
  pf("%d", counter);
}
```

// utils.c

```
int counter;
void incC()
{
  counter += 10;
  pf("%d", counter);
}
```

Line (1) tells compiler to not worry about counter which will either come later in same file or from another file (while linking).

Line (2) tells compiler either the declared func. will come later in same file or in another file.

* Functions are by default extern, so,

extern void incC (); ≡ void incC ();

* Declaration vs Definition

- Declaration is what the compiler need to accept references to the identifiers.

Ex.

```
extern int a;
extern int g (int, float);
double f (float, char);
```

- Definition is what the linker needs to know in order to link references to those entities.

```
int a;
int g (int x, float y) { return x+y; }
double f (float k, char c) { return k*c; }
```

- Therefore,
declaration is like: "somewhere there exist foo."

definition is like: "here it is".

Ex.

(1) Declaring a variable without defining it.

extern dtype var-name;

{No mem. allocated}

(2) Define variable with extern keyword.

extern dtype var-name = value;

{ Mem. allocated }
{extern keyword is useless}

(3) Will this work?

```
main () {
  extern int a=5;
}
```

No. Extern variables must be global variable, i.e extern declaration should refer to global variable.

* Functions themselves are always external, since C doesn't allow func. to be defined inside another function. Declaration is allowed.

* Register

- Storage: CPU register or Memory
- Initial Value: Garbage
- Scope: Within block
- Lifetime: End of scope.
- Used only with local variables. Similar to auto.

- If CPU has empty register then the variable will be stored there, else will be stored in memory (in Stack like auto variables).

- '&' (address of) operator can't be used with register variable to get the m/m address.

* By default, global variables are not static, since they can be used in another file (so it can't be static).

i.e.

```
int a;          ≠    static a;
main() {..}          main() {..}
```

By default global variables aren't extern, since they get memory allocated.

i.e.

```
int a;          ≠    extern int a;
```

Global vars can't have auto or register, since they are for local vars.

Therefore by default there is no keyword for global variables.

They are available to the linker to be referenced in other files & they are stored in "static memory space".

Ex. If we have ~~with~~ just this single file, then which error will it throw?

```
#include <stdio.h>
int main()
{
    extern int a;
    a = 10;
    pf("%d", a);
}
```

(A) Compilation Error
(B) Linker Error ✓
(C) Runtime Error
(D) No Error.

Linker error. Compilation has access to only this file. Compiler sees ~~and~~ extern declaration. & doesn't worry about 'a'.
Linker checks for a in the file but doesn't find it, therefore throws error.

Ex. #include <stdio.h>
extern int i;

int main() {
pf("%d", 5);
}

No error from linker, since i isn't being used anywhere.

* The only storage class that can be specified for func. parameters is "register".

Ex. int fun(register int);

* A func when specified static becomes limited to the file of declaration.

Ex int i = 10;
i = 5; ← Not allowed. Only initialization is allowed globally in C.
int main()
{ pf("%d", i);
}
Compile Time Error.

C PROGRAMMING
Recursion

Functions defined in terms of itself are called Recursive functions.

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

f(1) = ?

f(1)
↓
f(1+1)
⇒ f(2)
↓
f(2+2)
⇒ f(4)
↓
f(4+3)
= f(7) ↑ 7

∴ f(1) = 7.

★ Solve questions.

C PROGRAMMING

Pointers & Arrays

Pointers are special variables which store address of other variables/ functions, etc.

Ex.

```
int quantity = 5;
int *p = &quantity;
```

Var name		Address
quantity	5	5000
	⋮	
p	5000	52084
	⋮	

'&' is address of operator which is used with variables to get its m/m address.

* Declaring a pointer:

~~datatype *~~ dt *p

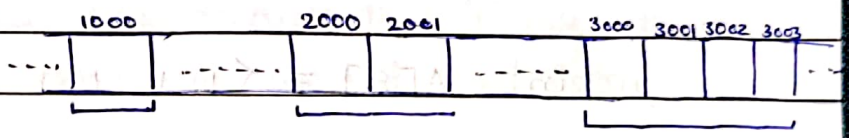
Read: p is a pointer to the datatype dt.

Initially p might hold garbage value.

Ex: int *ptr;

• '*' is dereference operator, when used with pointer var. name, and returns the value stored at the location whose addr. is stored in the ptr variable.

Ex. Suppose a, b & c point to addr. 1000, 2000 & 3000, resp.




```
char *a;
short *b;
int *c;
```

What is the value stored in a, b & c?

1000
2000 i.e. starting addr. of the variable.
3000

• The preferable format specifier for pointers is "%p".

Ex.
```
main ()
{
    int x = 100, y = 200
    swap (&x, &y);
    pf ("%d %d", x, y);
}
```

```
swap (int *a, int *b)
{
    int *temp;
    temp = a;
    a = b;
    b = temp;
}
```

Output: 100 200

No swap of values performed. Only addresses swapped

* Array

A collection of similar data.

* Initialization:

```
int A[5] = {1, 1, 1, 1, 1}

int A[5] = {1, 2,}        // initializes to 1,2,0,0,0.

int A[5] = {0}            // initializes all 0s. {: not assigned;

static int A[5];          // all 0s since static

int A[5];                 // garbage if local
                          // 0s if global.
```

In case of character array default value is null character.

```
char C[5] = {'a', 'b', 'c', 'd'};   // 'a','b','c','d','\0'

char C[4] = {'a', 'b', 'c', 'd'};   // 'a','b','c','d'

char S[5] = "abcd";                 // 'a','b','c','d','\0'

char S[4] = "abcd";                 // 'a','b','c','d'
```

1 & 3 are same.

2 & 4 are same.

Note: When an array var. 'arr' is used in expressions, it decays into a pointer to the first element.

Ex. int a[5] = {0};

int *p = a + 1; ← a behaves as "int (*)".

* What is string in C Programming?

There is no string datatype in C.

A string in C is just a character array, with '\0' character at the end. {Not because that is any standard construct. Rather, many library func. use that convension}

Ex.

```
char A[5] = "abcd";
char B[4] = "abcd";
```

A is both char array & string, but B is just char array.

What will happen if size is not specified?

Ex.

```
char A[] = "abcd";
```

Compiler interprets it as A[5] = "abcd".

Ex. char A[] = {'a', 'b', 'c', 'd'}

Interpreted as A[4] only.

* Array is stored in static area in memory, so assignment like this is not valid:

```
int A[2];
int B[3];
A = B;
```

Error!!

* sizeof with Array:

Ex. short i = 20;
char c = 97;

pf ("%d, %d, %d", sizeof(i), sizeof(c), sizeof(c+i));

Output: 2 1 4
{Assuming sizeof short: 2 int: 4 char: 1}

c+i will lead to both c & i being integer promoted.

Ex. ~~sizeof~~
int a[5] = {1, 2, 3, 4, 5};

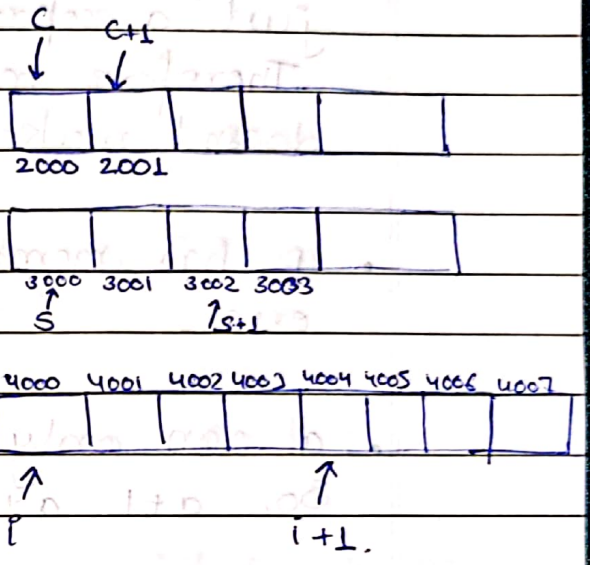
pf ("%d, %d", sizeof(a), sizeof(a[0]));

Output: 20, 4

* Pointer Arithmetic

Ex. char *c;
short *s;
int *i;

```
c += 1;
s += 1;
i += 1;
```



Would lead to:

c = 2001

s = 3002

i = 4004

Pointers increment (or decrement) by offset of the datatype size.

```
d_type * p;
    p += 1
will be
    p = p + 1 x sizeof (d_type);
```

* Pointers & Array.

```
int *pa;          // pointer to an integer
int a[10];        // an array of 10 integers.
pa = &a[0];
```

- a stores the address of element at 0th index.
  *a gives the 0th element.
- &a gives the address of whole array since the type of a is int [10].

- a by itself doesn't have any memory, its just a representative of the array. Therefore modifying a or assigning to a doesn't make sense.
- p has memory in which it stores addresses.
- a can only give the base base address. So, a+1, a+2, etc. make sense.
- a is not a pointer.

* Little Endian & Big Endian

If data spans over multiple bytes, like int, float, short, etc, then there are two variations in which they can be stored in memory.

Little Endian: Least significant Byte starts from Little u/m location.

Ex. int x = 0x FE AB 10 D5

∴ x =

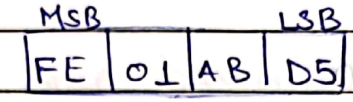
MSB			LSB
FE	AB	10	D5

2500	2501	2502	2503	2504	2505	
D5	10	AB	FE			...

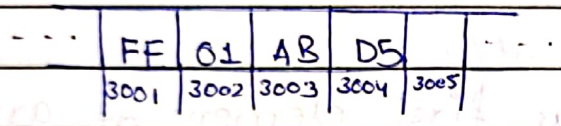
Big Endian: Stores MSB in lower m/m address & so on.

Ex:

int x = FE01AB D5.



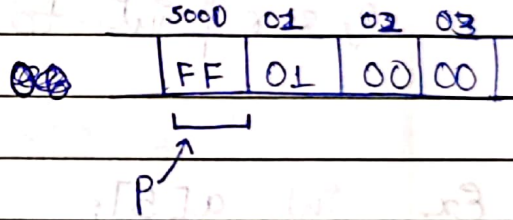
∴ x =



Ex. int i = 511;

char *p = (char *) &i;

pf ("%d", *p);

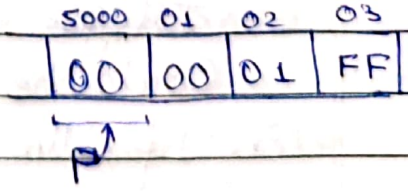


① {Assuming Little Endian}

∵ p is char pointer it will dereference only a single byte, then it gets converted to int because of "%d".

∴ Output: 255

② Big Endian:



Output: 0.

* a[i] is same as i[a] which gives, *(a+i).

Ex. int a[10] = {0};
int *p = a;

int **q = &a;
int (*r)[10] = &a;

p points here to the first element of array.

q & r point to entire array.

Ex. int a[5];

which is illegal?

① a+10
Not illegal, just adding offset to address.

② *(a+10)
~~Illegal~~
Could be illegal.
Ex. If the addr. is not in process's memory segment

* size of array vs size of pointer

int a[10];
sizeof(a) give
10 x sizeof(int)

int *p;
sizeof(p) will give
size of address in
system: 4 or 8 byte

Ex. char c[] = "apple";

sizeof(c);
6

~~sizeof~~ strlen(c);
5

sizeof will give size based on the datatype.

strlen counts size upto but not including '\0' character

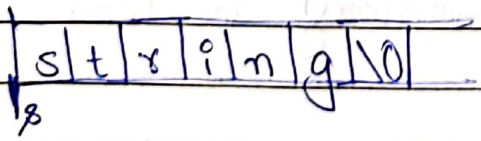
* Passing 1D Array to Function

fun(int *a) ≡ fun(int a[]) ≡ fun(int a[10])

All the versions get converted to fun(int *a) only. Other two are just beginner friendly constructs.

* char s[] = "string"

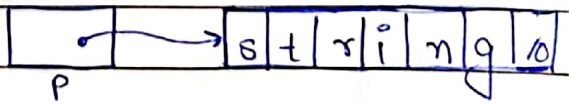
This initialises a char array in local scope.



stack

char *p = "string"

This first stores the string constant in static area, then its address in local scope in a variable p.



stack m/m area.

static m/m area

- Since 'p' points to string constant, which is stored in read-only section of static m/m area, the string can't be modified.

Ex.

```
p[2] = 'K'
```

is illegal.

whereas, 's' which is a char array is modifiable.

Ex.

```
s[5] = 'l'
```

is allowed.

- On the other hand, since 'p' is just a pointer in local memory, reassignment is valid.

Ex.

```
char *p = "Hello";
p = "New";
```

is valid.

But, s is referring to an array & doesn't have its own memory location, reassignment is illegal.

Ex.

```
char s[] = "Hello";
s = "New";
```

is invalid.

* Double Pointer

A pointer which points to a pointer.

Ex. int x, *p, **q;
x = 5;
p = &x;
q = &p;

So. x == *p == **q

Ex. int K, a[3] = {5, 7, 9};
int *p, **q;

p = a;
q = &p;

K = *p; pf("%d", K);
K = **q + 1 —"—
K = *(*q + 1); —"—

Output: 5 5 7

Ex. int ***p = 1000;
pf("%u", p + 1);

{Assume size of m/m addr is 8 bytes}

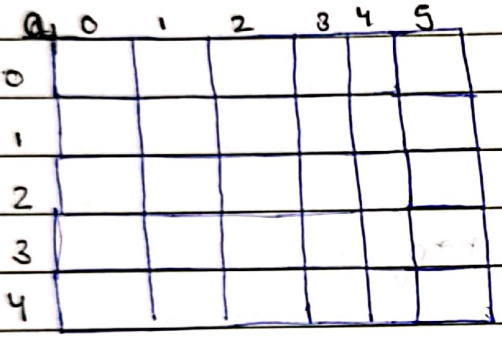
Output: 1008 {p+1 == p + sizeof(*p)}

{This works a little diffe for array}

* 2D Arrays:

A collection of 1D arrays.

Ex. int a[5][6]



a stores the reference to the [0][0]th element.

- a[0] points to 0th row.
- *(a+3) points to 3rd row.
- (a+3) gives addr. of 3rd row's ~~0th element~~.
- **(a+3) gives 0th element of 3rd row.
- *(*(a+3)+2) gives 2nd element of 3rd row.
- *(a+3)+2 gives addr. of 2nd element of 3rd row.

Ex. int a[4][3]; // a points to addr 1000
// int is 4 bytes.

pf("%u", a+1);

Output: 1012

classmate
Date
Page

a+1 ≡ a + sizeof (int[3])
≡ 1000 + [3 x sizeof (int)]
≡ 1000 + 12
≡ 1012

{ Hence skips based on its content's size }
{ content of a are 4 int[3] }
{ content of a[i] are 3 ints }

* Say,

```
int **t;
int a[3][2];

t = a;      —(1)
t = *a;     —(2)
*t = *a;    —(3)
```

(1) Is invalid, since t is a pointer to (int*) type, whereas a is addr of entire row of an array, i.e. (int [2]).

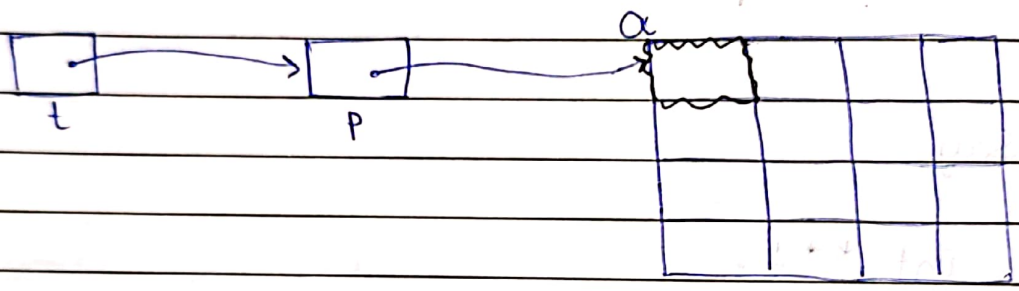
(2) Is again invalid. *a stores addr. of integer, or *a is a pointer to int type.

(3) Is valid, since *t stores addr of int or *t is pointer to int type.

Ex. int a[4][4];
int *p, **t;

- p = *a;
  t = &p;

This is valid.



* Pointer to Array & Array of Pointers

* Array of pointers to type ~~int~~ dtype.

dtype* A[n];

A is an array of n pointers to dtype.

* Pointer to an array of dtype of size n.

dtype (*p)[n]

p is a pointer to an array of n dtype elements.

Ex. int (*p)[10]

p+1 ≡ p + sizeof (*p)
⇒ p + sizeof (int[10])

Ex. int * A[10]

A+1 ≡ A + sizeof (content of A)
⇒ A + sizeof (int*)

Note: n in declaration of pointer to array is not optional.

Ex. int a[5] = {1,2};
int (*p)[5];

p = ?

What should ? be for p to point to entire array?

(A) p = a; ✓
(B) p = &a; ✓
(C) p = *a;
(D) p = (int(*)[5])a; ✓

a gives correct base addr, & typecasting changes it to correct type.

Ex. char * sports[5] = {
"golf",
"football",
"cricket",
"badminton",
"tennis"
};

(1) *((sports+2)[1])

will give character 'b'.

(2) *(sports[1])

will give character 'f'

(3) (sports+3)[-1]

will be: *(sports+3-1)

will give pointer to "cricket".

(4) Is this valid?

sports[2][3] = 'l'

No. Since each string literal is stored in read-only section of static m/m.

* Passing 2D Array to Function

* int a[5][6];
  fun (a)

then, it should have declaration like
fun (int (*p)[6]);

* fun (int p[5][6]);

would internally convert to fun(int (*p)[6]) only.

So, it doesn't matter if it is:

fun (int p[500][6]); is still converted to

fun (int (*p)[6]) only.

* fun (int **p) isn't the right way. There is no size info.

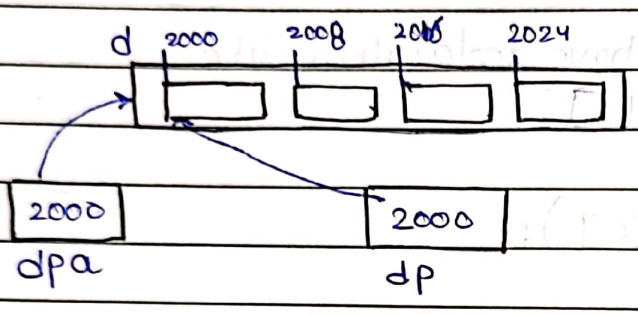
p is just a pointer to (int *) type, but a ~~actually~~ is actually a pointer to a row of 6 ints.

p+1 would be, p + 1 * sizeof (*p)
≡ p + 8bytes

a+1 would be, a + 1 * sizeof (content of a)
⇒ a + 1 * sizeof (int[6])
≡ a + 24 bytes.

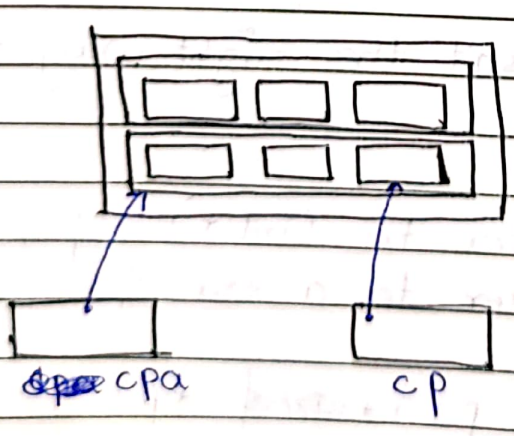
Ex. double d[4] = {1.0, 2.0, 2.2, 3.4};

double (*dpa)[4] = &d;
double * dp = d;



Ex. char C[2][3] = {
{'a', 'b', 'c'},
{'d', 'e', 'f'},
}

char * cp = &C[1][2];
char (*cpa)[3] = C+1;



(1) Get • C[1][1] using cpa & cp.

C[1][1] = *(*cpa+1) = *(cp-1)

(2) Get c[0][i] using cpa & cp.

c[0][i] ≡ *(*(cpa-1)+1) ≡ *(cp-4)

* Pointer Arithmetic:

* When two pointers say,

```
dt1 *p1;
dt2 *p2;
```

are subtracted, then if

C1: dt1 is same as dt2:
Then it gives the no. of locations of size = sizeof(dt1) b/w p1 & p2.

C2: dt1 is not same as dt2.
Then it has undefined behaviour. May give garbage value.

* Void Pointer

A void pointer variable can hold any type of address.

The value can be accessed by typecasting to a particular type based on which the length of data in bytes depends.

44	00	10	FF
1000	01	02	03

Eg. void *p = 1000;
pf("%d %c", *((unsigned *)p), *((char*)p));
Output: 4279238724 D

C PROGRAMMING

Complex Declarations

* Precedence of Declaration Operator

1. ( ) [ ] { Left to Right }
2. * { Right to Left }
3. Data Type

* Rule of Translation.

1. Find the identifier. This is the starting point.

   Right

2. Go [illegible] until you find right parenthesis or end of declaration. If right parenthesis, goto step 3.

3. Go Left until you find left paranthesis or end of declaration.

Ex. int *a[10];

a is an array of 10 pointers to int.

Ex.

int (*pf)( );

pf is a pointer to func. returning int.

Ex. `int* (*(*fp)(int))[10]`

fp is a pointer to a func. which takes an int and returns a pointer to an array of 10 pointers to int.

Ex. `int (*apa[10])[5];`

apa is an array of 10 pointers to array of 5 int.

Ex. Pointer to the following func :

```
int func (int *x);
int (*p) (int *);
```

C PROGRAMMING

DYNAMIC Memory

* Dynamic memory is allocated on the heap.

* Malloc

Allocates m/m on the heap & returns the starting address.

Ex.

```
int * p;

p = (int *) malloc (sizeof (int) * 10);
```

will allocate blindly 40 bytes of m/m. Since the starting addr. is stored in int pointer p, the 40 bytes can be used to store 10 ints.

p is local variable stored on stack in same activation record, pointing to addr. on the heap

* Free

Deallocates m/m on heap allocated using malloc. Implementation details depend on the OS.

classmate
Date
Page

* **Dangling Pointer:** A pointer pointing to a m/m location which is already freed.

Ex.
```
int *p = malloc (sizeof(int));
free(p);
```

p now points to m/m location which has already freed.

Ex.
```
char* func ( ) {
    char str[5] = "appl";
    return str;
}

int main ()
{
    char *c = func();
}
```

c points to m/m which has been freed, since activation record no longer exists.

* **Memory Leak:** Memory which hasn't been freed & there is no way to free it.

Ex.
```
void func () {
    char *c = malloc ( sizeof (char) * 4);
}
main() {
    func ();
}
```

Ex. int main()
{
malloc(1);
}

classmate
Date
Page

C PROGRAMMING

Other Topics

* Macros :

Ex. #define plus_one(x) x+2.

```
i = 3 * plus_one(2);
```

becomes,

```
i = 3 * 2 + 2
```

& give result i = 8.

Ex. #define plus_one(x) (x+2);

```
i = 3 * plus_one(2);
```

becomes

```
i = 3 * (2+2);
```

& gives i = 12.

* Static vs Dynamic Scoping

Scope is the m/m that a func. can access. This can include local vars, global vars, along with other.

- **Static Scoping :** A func. can access its local m/m, along with the m/m in which the func. is declared & its scope.

(a.k.a. Lexical Scoping)

- Dynamic Scoping: A func. can access its local m/m along with the m/m in which the func. is called & its scope.

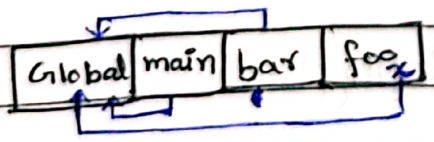
Ex.
```
x = 1;
foo()
{
    y = x+1;
    print y;
}

bar()
{
    x = 2;
    foo();
}

main()
{
    foo();
    bar();
}
```

Output:

Static Scoping

2

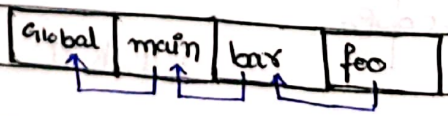
2



Dynamic Scoping

2

3



Since, foo, bar & main are all declared in global scope.

Since, foo is called in bar, bar is called in main & main called in global scope.

Dynamic scoping is called so since the scope is resolved during runtime dynamically.

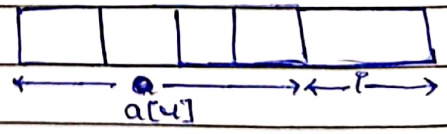
C Programming
Structs

Struct is a collection / group of logically related data of heterogeneous type.

Ex.

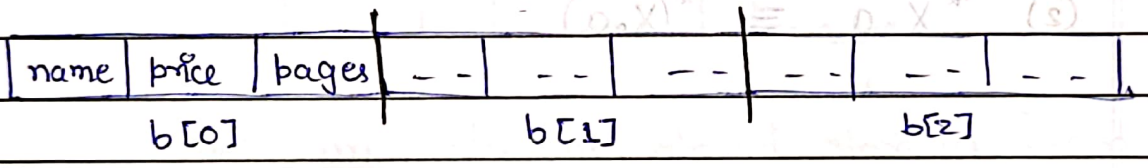
```
struct Box                    struct Circle
{                             {
  int W;                        int R;
  int H;                      };
};

int main() {
  struct Box b;
  struct Circle c;
}
```

Ex.
```
struct rec {
  int a[4];
  long i;
};

struct rec r1;
r1.a[0] = 2;
r1.i = 5;
```



* Array of Structs:

Ex.
```
struct book {
    char name;
    float price;
    int pages;
};

struct book b[3];
// { b is an array of 3 struct book }.
```



* Struct name is not a pointer to anything, unlike array names which basically point to index 0 of the array.

Ex:
```
void f (struct book B)
{
    B.a = 'K';
}

int main ()
{
    struct book b;
    b.a = 'A'; pf ("%c", b.a); // 'A'
    f (b);
    pf ("%c", b.a);            // 'A'
}
```

Ex. struct book b;
b.a = 'x';
pf ("%c", *b);

Will throw error!!

* (1) (*X).a ≡ X→a
↑ arrow operator.

(2) *X.a ≡ *(X.a)

* Copying Structs:

Ex.
Struct Book b1, b2;
b1.a = "x";
b1.p = 25;
b2 = b1;

This will copy all members one-by-one from b1 to b2.

* Comparing Structs:

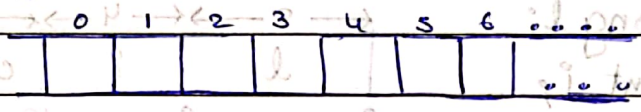
Direct comparision isn't allowed. Individual members have to be compared.

Direct comparision gives compile time error.

* Storing Struct in memory - aligned manner:

* M/M aligned storage means that the variable will be stored in m/m loc. with starting addr. divisible by the size of variable.

Ex. Say int (4B) is to be stored.

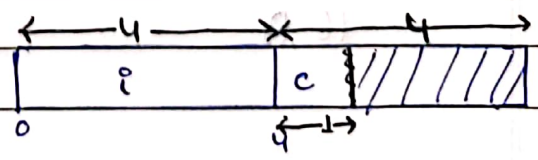


int can only be stored starting from either 0 or 4 here, since they are divisible by 4.

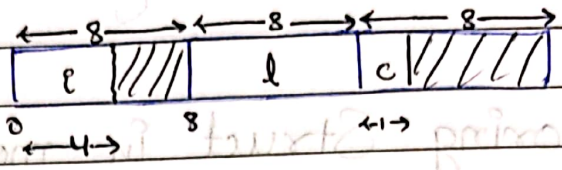
It is just a performance measure, no language specific reason. (Architectural)

* To find size of structs stored in m/m aligned fashion:
  (1) Find the size of largest member of the struct, 'c'
  (2) For each m/m member allocate the same space 'c'.
  (3) Store in aligned fashion, by checking division by size.

Ex.
```
struct a {
    int i;
    char c;
};
sizeof (struct a); // 8
```



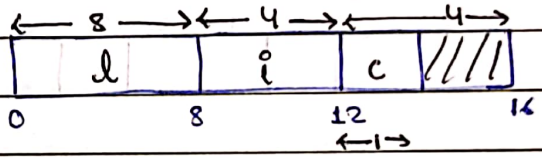
Ex. struct b {
int i;
long l;
char c;
}



24 B.

{∵ l is to be stored starting from x|8}

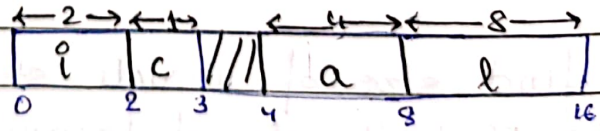
Ex. struct c {
long l;
int i;
char c;
}



16 B.

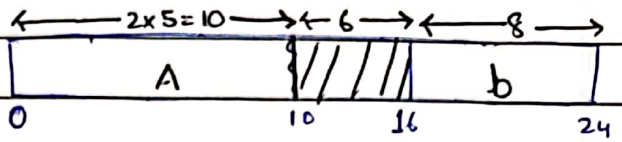
All elements are m/m aligned.

Ex. struct x {
short i;
char c;
int a;
long l;
};



16 B

Ex. struct Y {
    short A[5];
    long b;
};



24 B.

Q. Malloc vs Calloc.

Calloc can be thought of as Clean Malloc. The idea of malloc is to get a memory buffer. The buffer may or maynot be zero-initialised.

Calloc ensures zero-initialised buffers.
Calloc can manually do malloc + memset (0) or request an already clean OS page.

Q. How does free know how much memory to free?

When you call malloc, with size n, some additional memory is used to store the size info, which can't be directly accessed.
Free uses this info. to clear memory.

Q. Segmentation fault & Causes

When a program tries to access a location out of its address space, it causes segfault.
Common causes:
(1) Null pointer deref.
(2) Refering to uninitialized (wild) pointer.
(3) Referencing a freed (dangling) pointer.
(4) Stack & buffer overflows.