Lec - 1

Date 8/3/22
Page

Every word in C is classified as either keyword or identifiers.

Data Types

int (4 B) S & US	Float (4 B)
char (1 B) S & US	double (8 B)
short (2 B) S & US	long double (10 B)

Format Specifiers

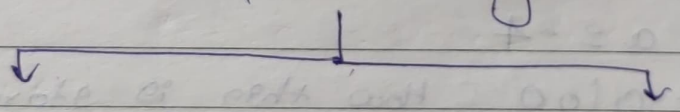
char %c	Hexadecimal Format %x or %X
int %d	addresses %p or %u
unsigned int %u	string %s
float %f	

Signed vs Unsigned.

Signed => 2's complement system.

Unsigned => usual binary

2's complement system



If no is +ve
Represent in usual way

If no is -ve
Represent it using 2's complement

Signed numbers are represented in 2's complement system

MSB will indicate the sign of the number

0 => positive

1 => negative

eg. Represent 5 into 2's cpl ?.
→ 0101
0 is MSB to denote no is +ve

eg Represent 10 in 2's cpl system ?.
a) 01010 c) 11010
b) 1010 d) None of these

eg. 8. -5 in 2's cpl
→ 5 ⇒ 0101
-5 ⇒ 2's cpl of (+5)
⇒ 0101 ⇒ 1010+1 ⇒ 1011
MSB is 1 which denotes -ve no

Memory representation of numbers
int a = 4;
↑32 bits 0 0 0 0000 0100

int a = -4
4 = 0100 How this is stored : later.
-4 = 1100

eg. 16-bit 2's cpl of -28
28 = 011100
-28 in 16 bit 0 0001 1100
(-28) = 1 1110 0011 + 1
= 1 111 00100

eg. 48 on 2's cpl (8 bit)
01010111

eg. 2's cpl of (-19)
19 => 0 1111
(-19) = 10001
also = 11 0001
any no of 1's can be added.

0 11 0 10
00 11 0 10
000 11 0 10
0000 11 0 10
} all are same

2's cpl to decimal

Method 1.
0101 = $-0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 5$

1011 = $-1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = -5$

eg 110001 = $-1\times2^5 + 1\times2^4 + 1\times2^0$
= $-32 + 16 + 1$
= -15

redundant
also 11 0001

sign bit should not change while removing redundant bits
10001 = $-1\times2^4 + 1\times2^0$
= $-16 + 1 = -15$

Date
Page 4

Method 2:

If the no is +ve (MSB = 0), convert the number into decimal format normally

If the number is -ve (MSB = 1).
Take 2's cpl of number and then convert to decimal.

eg $0101 = 0 \cdot 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$

eg 1011 = 2's cpl (1011)
$= 0100 + 1$
$= 0101$
$= -5$

minus sign has to added bcz MSB was 1.

eg. 000000 | 0 11 → This will denote sign.
redundant bits

Only 1 bit is required to denote the sign.

eg 1111111111 | 1010 1
redundant

eg 1010
usual binary $= 1 \times 2^3 + 1 \times 2^1 = 10$
2's cpl $= -1 \times 2^3 + 1 \times 2^1 = -6$

Date
Page 5

Copying Number to higher bit representation

1) Copying unsigned number (zero extension)

eg 01010 → 001010
6 bits 7 bits

eg 1101 → 001101

2) Copying signed number (sign extension)
Need to copy sign bit (0 or 1) as it is

eg 8 bit 1001 0110
9 bit 11001 0110

eg 8 bit 0101 0110
9 bit 00101 0110

eg Smallest integer in 8-bit 2's cpl.
→ 1000 0000
sign = -128

eg range in 2's cpl for n bits

$-2^{n-1} \leq N \leq 2^{n-1} - 1$

HW1

Date
Page 6

1) We have 7 bits. Largest unsigned no and largest 2's cpl signed no.

→ largest unsigned = 111 1111

$= 2^8 - 1 = 127$

For all 1's in k bits, decimal value is $2^{k+1} - 1$

largest 2's cpl = 011 1111

$= 2^7 - 1 = 63$

2) Compute the 2's cpl using fewest no of bits in each case

a) -296 = 1 0000 0000

b) -194

2 | 194 | 0
2 | 77 | 1
2 | 38 | 0
2 | 19 | 1
2 | 9 | 1
2 | 4 | 0
2 | 2 | 0
1 | 0

+ve no

194 = 0 1001 0010

-194 = 1 0110 0101 +1

= 1 0110 0110

c) -107

2 | 107 | 1
2 | 53 | 1
2 | 26 | 0
2 | 13 | 1
2 | 6 | 0
2 | 3 | 1
1

+ve

107 = 0 1101011

-107 = 1001010 1

d) 139

2 | 139 | 1
2 | 67 | 1
2 | 33 | 1
2 | 16 | 0
2 | 8 | 0
2 | 4 | 0
2 | 2 | 0
1 | 0

1 0000 111

above is wrong. MSB is 1 which implies no is -ve. There should be a zero at MSB

0 1000 111

3) a) -67 to 8 bit signed 2's cpl binary

2 | 67 |
2 | 33 | 1
2 | 16 | 0
2 | 8 | 0
2 | 4 | 0
2 | 2 | 0
1 | 0

67 = 0 1000011

-67 = 1 0111100 + 1

= 1 0111101

b) Convert 16 bit signed 2's cpl no to decimal

1001 0001 1111 1010

Method 2: 0110 1110 0000 0101 (10) 01+1 = 10

$= -2^{14} + 2^{13} + 2^{11} + 2^{10} + 2^{9} + 2^{2} + 2^{1}$

c) 8 bit 2's cpl of -23

2 | 23 | 1
2 | 11 | 1
2 | 5 | 1
2 | 2 | 0
1 | 0

23 = 0 1011 1

8 bit = 0001 0111

-23 = 1110 1001

d) 32 bit 2's cpl of -23

23 = 0 0 0 0001 0111

-23 = 1 1 1 1110 1001

4) Assume all numbers are in 2's cpl, which of the following nos are divisible by 11111 011

no = [1111] redundant 1011 -8+2+1 = -5

a) 1110 0111 = 0001 1001 = +16+9 = -29 ✓

b) 1110 0100 = 0001 1011 (100) = 4+8+16 = -28 ✗

c) 1101 0111 = 0010 1001 = -(32+8+1) = -41 ✗

d) 1101 1011 = 0010 0101 = -(32+5) = -37 ✗

5) 16 bit 2's cpl of 1111 1111 1111 0101
= ~~000~~ 1 0 1 0 1 neat redundant
= -16 + 4 + 1 = -11

6) 4 bit no = 0101
→ No. system is not important bcz MSB is 0 so no is +ve and it is 5.

7) 5 bit no most -ve ~~and~~ +ve no in 2's cpl
0 1 1 1 1 = 15
1 0 0 0 0 = -16
In unsigned, no -ve no
most +ve = 31

8) Convert unsigned into decimal
a) 1010 = 10
b) 1010 1010 = 2 + 8 + 32 + 128 = 170
c) 1100 0001 = 1 + 64 + 128 = 193

Convert signed into decimal
a) 1010 = -8 + 2 = -6
b) 0011 0100 = 4 + 16 + 32 = 52
c) 1100 0001 = 0011 1110
= -(1 + 2 + 4 + 8 + 16 + 32)
= -63

Date
Page 9

lec 2

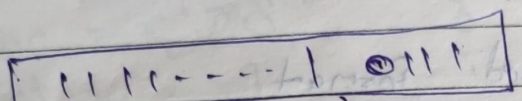
eg. int x = -9

by default int is 32 bits and is considered signed.

32 bits

9 = 01001 — 9 = 0 0 0 0000 1001

-9 = 10111 — -9 = 1 1 1 1111 0111

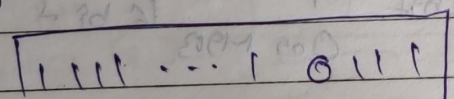
y 

Printf("%d", y) — -9

Pf("%u", y) = $2^{32} - 1 - 8$

printf doesn't use the information about type of variable

eg. unsigned int y = -9.

y 

Pf("%d", y) = -9

Pf("%u", y) = $2^{32} - 1 - 8$

Extension and Truncation

Extension :- copying a lower bit number to higher bit eg short to int.

Truncation :- Copying a higher bit number to lower bit eg int to short.

In extension always look for R+19

eg. short int x = 9;
int ix = x;

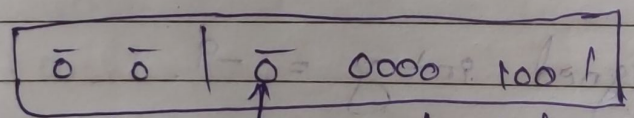
↑ 16 bits

Step1: Fix binary pattern
step2: Format specifier %d %d

9 in signed format
+ve no hence simple binary format 9n 0's cpl

x = 9 = 0 0000 1001

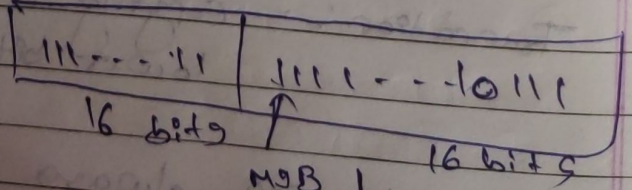
ix = x RHS
RHS is signed hence MSB of RHS will be copied here 0.



↑ 16 bits
eg. short int y = -9
int iy = y
↖ 32 bits

~~y 9~~ = ~~0 x 0~~ 9 in 16 bits = 0 0000 1001
y = -9 = 1 1111 0111
↑ 2's cpl form

iy = y → RHS is signed ∴ MSB will be copied



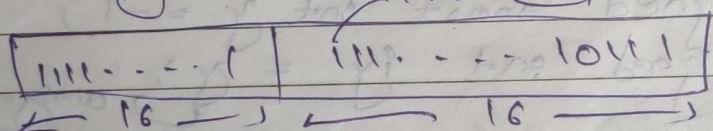
int x ; = signed int x
short int x ; = signed short int x ;
short x ; = short int x = signed short int x ;

eg. short int x = 9
int y = x ; signed
short int y = -9
int iy = y ; signed

0 0000 1001
0 0 0 0000 1001
1 1111 0111
1 1 1 1111 0111

eg short int y = -9
y 9 = 0 0000 1001 in 16 bits
-9 = 1 1111 0111 -9 in 2's cpl in 16 bits

unsigned int iy = y ; signed
RHS is signed hence MSB will be copied



① represent -9 ; Fixed
② Go for extension . Look for RHS type while copying

eg. Lets suppose int is 4 bits (hypothetically)
int x = 9
x 1001
unsigned x = 9
x+2 = 11

x+2 = -7 +2 = -5
step1 : 9 is 1001
step2 : treat bit pattern

eg. unsigned short int y = -9.
int iy = y;

16 bits

y = -9 = 1 1111 0111

iy = y → RHS is unsigned. so 0 will be copied.

iy | 0 0 | 1 1111 0111 |
← 16 → ← 16 →

Unsigned : always write zeros while copying to higher bits.

Sign : always copy MSB while extension

eg. unsigned short int y = -9;
unsigned int iy = y; → unsigned copy 0's

y = 1 1111 0111

iy = | 0 0 | 1 1111 0111 |

Extension depends on RHS

Extensions always happens according to the source variable's type

Signed : "sign extension" (copy MSB 0 or 1 to fill new space)

Unsigned : "zero fill" (copy 0's to fill new space)

Date
Page 13

eg. short int y = -9

unsigned int uy = y;

signed

1	1111	0111
1 1	1 1111	0111

1) Pf ("%d", y); -9
2) Pf ("%u", y) $2^{16} - 1 - 8$
3) Pf ("%d", uy) -9
4) Pf ("%u", uy) $2^{32} - 1 - 8$

eg. unsigned short int y = -9;

int uy = y;

unsigned

0's

1	1111	0111
0 0	1	1111 0111

[4] 1) Pf ("%d", y); -9
2) Pf ("%u", y); $2^{16} - 1 - 8$
3) Pf ("%d", uy); $2^{16} - 1 - 8$
4) Pf ("%u", uy); $2^{16} - 1 - 8$

eg. int x = -1;

unsigned int u = x;

1 1

No extension or truncation required.

1) Pf ("%d", x) -1
2) Pf ("%u", x) $2^{32} - 1$
3) Pf ("%d", u) -1
4) Pf ("%u", u) $2^{32} - 1$

Truncation (higher bits to lower bits)

→ Regardless of source or destination signed/unsigned type, truncation always just truncates

→ This can cause the number to change drastically in sign and value

[i] The concept of integer promotion was not taught at this point.

Date
Page 14

[i]

```
unsigned short int y = -9;
int iy = y;
```

y = 1111 1111 1111 0111

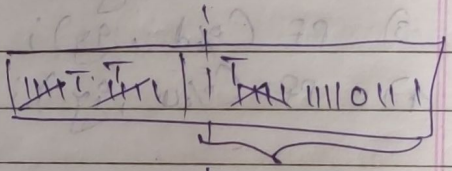
now, pf ("%d", y);

This will cause an integer promotion from 16 bits to 32 bits. Since the source (y) is unsigned 0's will be added in the first 16 bits

0 0	1111 0111

this is no sign bit and the value printed will be $2^{16} - 1 - 8$

eg int x = -9;
short int y = x;



y = 1111 0111

pf ("%d", x) -9
pf ("%u", y) $2^{16} - 1 - 8$

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

char c = 'a' } Both will store same
int x = 97 } value

ascii value of every character is stored
'a' → 97.

char c = 'a';
a+1 = 97 + 1 = 98
int i = 97 + 1 = 98

eg int x = 9 32 bit
short int y = 9 16 bit
char z = 9 8 bits

eg signed char a = 8; | 0000 1000 |
a

unsigned char c = 258; 256+2.
1 | 0000 0010 |
c = 2

pf ("%d", c);
↳ o/p 2

eg. unsigned char c = 386;
256 + 128 + 2 = 386
1 | 1000 0010 |
c

pf ("%d", c); // 130
↳ unsigned hence 0 will copied in integer promotion

| 00 ... 0 | 1000 0010 |
24 bits

eg signed char a = -30, b = 40
signed char d = a * b;

pf ("%d", d);
pf ("%d", a*b);

Remember

a * b will be stored in 32 bits and later the truncated lower 8 bits will be stored in d bcz char stores only 8 bits

1200 = 0 0 0000 0100 | 1011 0000
d

pf ("%d", d) := -80
d is signed MSB copied

| 1 1 1 | 1011 0000 |
redundant

1011 0000 = -128 + 32 + 16
= -80

no truncation → pf ("%d", a * b) = 1200
a * b is 32 bits. Integer promotion to %d is also 32 bits. So the value of a * b will be printed.

eg. char a = 30, b = 40;
unsigned char d = a * b
pf ("%d", d); unsig. 0 will be copied

→ d will contain truncated last 8 bits

| 0 0 0 | 1011 0000 |

o/p = 128 + 32 + 16 = 176

Integer promotion: whenever you have small integers (short or char) then first convert to int (32 bits) and then do anything you want to

eg. signed char f = -69.
pf ("%d", f);
pf ("%u", f);

→ +69 = 64 + 1 = 0100 0001
-69 = 1011 1111

pf ("%d", f) → signed. MSB copied

| 1 1 1 | 1011 1111 | = -69
redundant

pf ("%u", f) = $2^{32} - 64 - 1$.

eg. unsigned char f = -69
pf ("%d", f); → unsig. 0 will be copied

| 0 0 0 | 1011 1111 | ⇒ 2^8 − (64 + 1) = 191

pf ("%u", f); still 191.

Summary

① Printf :- does not care about data type of variable. only cares about format specifiers.

② Extension and Truncation : directly truncate. look for RHS and extend.

③ int promotion → small integers (char or short) in expression are treated as integer.

Date
Page 18

HW-2

1) int i = -3;
unsigned short u;
u = i;
pf ("%u", u);
pf ("%d", u);

0 0 0 0000 0011
1 1 1 1111 1101

u = 1 1111 1101

pf ("%u", u) → unsigned → $2^{16} - 1 - 2$

integer promotion
because of u being short it is promoted to 32 bits bcz of "%u".

pf ("%d", u) → unsigned
0 0 1 1111 1101 → $2^{16} - 1 - 2$

2) signed short ix = -3; 1 1111 1101
pf ("%u", ix); signed
pf ("%d", ix); signed

1 1 1 1111 1101 = $2^{32} - 3$
1 1 1 1111 1101 = -3

4) a = 100; b = 200; c = a + b;
pf ("%d %d %d \n", a, b, c);

→ can't say anything, depends on how we initialize a and b

5) signed short a = 100, b = 200;
signed int c = a + b;
pf ("%d %d %d \n", a, b, c);

a 0110 0100 = 100
b 1100 1000 = -56 (MSB 1)
c = a+b

First a & b both with be converted to int bcz of integer promotion. In both source is signed

a 0 0 0 0110 0100 = 100
b 1 1 1 1100 1000 = -56
redundant
a+b = 100-56 = 44

6) signed char a=100, b=200;
signed char c = a+b.

→ Yal integer promotion steps are same. Now for char only last 8 bits will be truncated. from a+b.

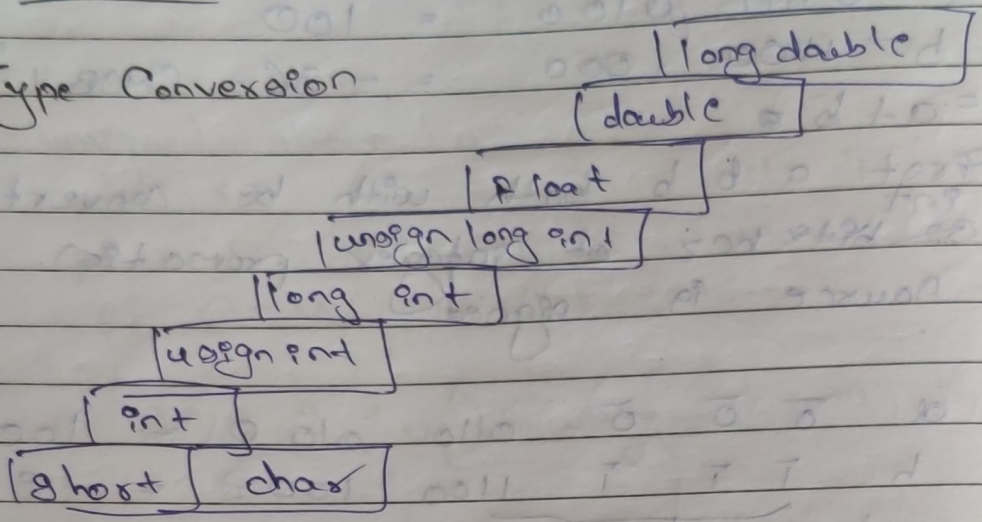
a+b c = 44 = 0000 0010 1100

7) signed char a=100, b=3, c=4;
signed char result = a*b/c;
pf ("%d", result);

→ a*b = 300/4 = 75
a*b will be stored in 32 bits then 300/4 will be stored in 32 bits the last 8 bits will be stored in result, which is 79

Lec - 7

Type Conversion



eg int a = 3; float b = 5.0;
a + b → float. 3.0 + 5.0 = 8.0
↳ int → float

eg. char c = 'A' int a = 3
↳ c + a
char → int.

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

res = v1 / v2; 10/3 = 3.33 but
int/int = int = 3.
pf ("%f", res); // 3.0

res = (float) v1 / v2; v1 (float)
pf ("%f", res) // 3.33 v2 (int) → float typecast

res = v1 / (float) v2;
pf ("%f", res); // 3.33

eg. unsigned int a = 1000;
int b = -1;

if (a > b) pf ("a big");
else pf ("b big");

1000	1111....1
a	b

-1 on 2's cpl is stored as all 1's

a > b is False
10^3 $2^{32}-1$

eg. int P = -1; unsigned u = 1;
char c = P + u;

111....1	000...01	00...000
P	u	P+u

c will have only the last 8 bits of P+u. which is all 0's.
so c = 0;

Note:

eg. ~~We~~ We don't have unsigned floats

eg int P = -1; unsigned int u = 1
if (P - u > 1). pf ("huge");

P = all 1's u = 00...01
P - u = 1111...1110.

before: P - u.
↳ int
↳ unsigned
↳ unsigned > 1
unsigned

so p-u >= 1 is true

eg p - u >= -1.
1111...10 (11111...-1)
unsigned → signed → unsigned
type casted

so p-u >= -1 is false

If statement

eg. if (grade <= 10) ;
① pf ("<=10") ;
② pf ("something...") ;

Only the first statement immediately following "if" is in the same block. i.e.

if (grade <= 10) ; {
pf ("<= 10") ;
}
pf ("something...") ;

Both are same

If there are no braces ("{ }") following the if statement then only the next immediate statement belongs to the if statement.

eg if (0)
pf ("1") ; o/p : 23
pf (";") ;
pf ("3") ;

0 is false in C prog.
Any other value than zero is true

eg
```
if (1)
if (2)
if (3)
else { }
```
→
```
if (1) {
    if (2) {
        if (3) {
        } else {
        }
    }
}
```

By default else is associated with closest if

eg
```
if (1)
    if (2)
    else { }
    if (3)
        if (4)
        else { }
```
→
```
if (1) {
    if (2) {
    } else { }
}
if (3) {
    if (4) {
    } else {
    }
}
```

eg
```
if (1)
    if (2)
        if (3)
        else
    else {}
    if (4)
    else
```
→
```
if (1)
{
}
if (4) {
} else {
}
```

eg
```
if (a < 10)
    if (a % 2 == 0)
        pf ("a is even and less than 10 \n");
else
    pf ("mystery");
```

Switch statement.

Rules for switch statement

→ The expression (after switch keyword) must yield an integer value i.e the expression should be an integer or a variable or an expression that evaluates to an integer

→ The case statements and default statement can occur in any order in the switch statement.

→ The case label values must be unique.

→ The case label must end with a colon (:).

eg
```
no = 2;
switch (no + 2)
{
    pf ("hey");
    case 1:
        pf ("1");
    case 2:
        pf ("2");
    default:
        pf ("default");
}
```

→ will be ignored bcz this statement is not in any case or default.

switch (i)
integer expression: int, char, short.

eg. no = 2; error.
switch (no) { → case cannot have variable
case no: error.
pf ("no");
}

eg. int val = 0;
switch (val) {
default:
val++;
case 2:
pf ("case 2\n");
break;
case 1;
pf ("case 1");
}
pf ("%d", val);
O/P - case 2
1

flow.
case 2 → case 1 → default (val=1 now) then case 2 pf() break; then pf val which is 1.

Step1: look for matching case. If no case is matching then go to default. If case is matching then go to that case.
After [if] you will be inside case/default keep on executing everything till you finds break.

HW 3

2) int x = 9;
int y = x / 3.0 / 6 * 2;
~~y~~ pf ("%d", y);
→ (x / 3.0) = (9 / 3.0) = 3.0
→ (3.0 / 6) = 0.5
→ 0.0 * 2 = 1
y = 1

Note · case 1:
pf ("Four\n");; → Two semicol doesn't produc
error

Lec - 4

For loop
For (initialization ; test ; update) {
// loop body
}

initialization happens only once in entire life
All three init, test and update are optional

eg. for (p=10; p>0;).
p = p-1;
pf ("%d", p); o/p: 0

No braces means only next immediate statement after for is its body.

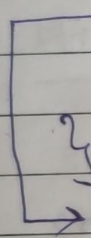
Date
Page 27

Do while. loop.

```
do
{
    // statements
} while (condition);
```

Break statement:

```
while (test expression) {
    // codes
    if (cond to break) {
        break;
    }
}
```



Same with do while & for loop

Logical Operators

Precedence of the relational and logical operators

- Highest !
  > >= < <=
  == !=
  &&
  lowest ||

Assignment Operators:

v op= exp;
v = v op (exp);

x += y + 1
x = x + (y + 1);

[1] For all other combinations. Do addition by C rules and then take its floor.

Date
Page 28

a *= b+2

① a = (a*b)+2 ✗

② a = a*(b+2) ✓

Post increment:
n = 5
x = n++;
① assignment x = 5
② increment n = 5+1

Pre increment:
x = ++n
① Increment n = 5+1
② assignment x = 6

++2 (constant)
++(a+b*2) (expression)
} both are not allowed.

Precedence of Arithmetic Operators
High: *, /, % Low: +, -

eg 4/2*2 = (4/2)*2 = 4 ✓
= (4/(2*2)) = 1 ✗

[1] eg. ~~x = a - b/g + c*2 - 1~~
int d = 2*3/4 + 2.0/9 + 8/5;
= (6/4) + (2.0/9) + (8/5)
= 1.9 + 0.4 + 1.6
int div ↓ ↓ ↓
1 + 0.4 + 1
2.4 ↳ Floating division
int j = 2;

Unsigned Right shift: Fill zeros
signed Right shift: Depends on system either zeros or sign bit

Date
Page 29

One's complement operator ~

x	= 1001	0110	1100	1011
~x	= 0110	1001	0011	0100

x = 8 ~x = -9 x in 2's cpl = ~x + 1

Simple logical way

-8 in 2's cpl = ~(8) + 1 → is cpl of 8

-8 = ~(8) + 1

~(8) = -8 - 1 = -9.

HW 3.4

a)
```
int ii = 10;
ii <<= 1;
pf ("%d", ii);
```

The above code is error free. expression is valid.

var op= exp; ii <<= 1;

var = var op exp; ii = ii << 1;

ii = 20;

20 is o/p.

4)
```
int var = - -3;
pf ("%d", var);
```
// 3 is op

This is not --3; Here we have space between two minus symbol.

Similarly, - - -3 = -3

- - - -3 = 3

11) int a=0;
if (0 && a++ == 0)
pf ("inside if");
else
pf ("inside else");
pf ("%d", a);

→ Correct Flow of program.
== has higher priority than &&
so the like
if (0 && (a++ == 0))
now bcz of short circuit RHS of && is not executed and a value stays the same

17) int a = 10, b = 5, c = 3;
b != !a; b != 0 True but not assigning
c = !!a; c = !!True, c = True
pf ("%d %d", b, c);

We treat '!' as not. So if a no is non-zero it returns zero. If it is zero, it returns 1.

O/P 0 5 c = 1

'^' operator in c is XOR and not exponent

18) int i;
scanf ("%d", &i);
if (!i == ~i) pf ("same this time");

Mistake:
if i=0; !i !(False) = True = 1
~i = ~(all zeros) = all 1's
But all 1's is -1. Value of all ones is not true or false but a integer value. all 1's = -1
so 1 == -1 is False

But if $i = 2^{32} - 1$ or -1 then !i and ~i will be same

if i = -1 i.e. !i = false = 0 same
~i = ~(all one's) = all zeros = 0

Conditional Operator or Ternary Operator.

exp1 ? exp2 : exp3;
(exp1 → condition; exp2 → True; exp3 → False)

Comma Operator
evaluated left to right and the value of right-most expression is the value of the combined expression

value = (x=10, y=9, x+y);
x=10, y=9, x+y
value = 10+9

eg int i, j=2;

for (i=0; i<=5, j>=0; i++) {
    pf ("%d", i+j);
    j--;
}

i<=5, j>=0 is same as j>=0

O/P	O/P
i=0 j=2	2
1 1	2
2 0	2
3 -1	break

eg. Same eg but j>=0, i<=5.
The above is same as i<=5

i	j	O/P
0	2	2
1	1	2
2	0	2
3	-1	2
4	-2	2
5	-3	2

* j>=0 && i<=5
Then in '&&' both needs to be evaluated

[] 1) = —> highest
2) += -= 3) ^= |=
4) *= /= 5) <<= >>=
6) %= &= Date Page 33

Associativity and Precedence of Operators

Number	Operator	Associativity
1.	() . [] . -> ++ --(Post fix)	L to R
2.	sizeof & * + - ~ ! typecasts ++ --(Prefix)	R to L
3.	* / % } Arithmetic	L to R
4.	+ - }	"
5.	<< >> } shift	"
6.	< <= > >= } Comparison	"
7.	== != }	"
8.	& }	"
9.	^ } bitwise	"
10.	| }	"
11.	&& } logical	"
12.	|| }	"
13.	? } Ternary	R to L
[] 14.	All. var op= exp } shorthand	R to L
15.	, } comma operator	L to R

eg a=1, b=1, c=1 MUASCBL TSCM
if (a == b || c++)
pf ("%d", c);

-> ++ (post incr) has higher priority than == but still (a==b) will be executed first because of order of evaluation

for eg main() {
a = a+3 line 1
b = b*5 line 2
}

even though * has higher priority line 1 will get executed first bcz of order of evaluation

So on exp1 || exp2 or exp1 && exp2
exp1 will be evaluated first.

eg
```
int k, i = 50, j = 100, l;
i = i | (j && 100);
k = i || (j || 100);
l = i & (j && 100);
pf ("%d %d %d %d", i, j, k, l);
```

→ i = i | (j && 100)
= i | 100 = T && 100 = True = True = 1
= 50 | 1 50 0011 0010
= 51 1 1
50 | 1 0011 0011

True (no dont care) ← x T

k = i || (j || 100);
logical so value of k will be 0 or 1 for sure
k = 1

l = 51 & (i && 100)
= 1 51 & 1 51 0011 0011
1 0000 0001
51 & 1 0000 0001
= 1

eg. i = 0
i++ || i
0 here i = 1

&&, ||, ?, comma - are sequence points in C.

Date
Page 39

Side effects:

```
i = 5
i++;
p = p + i;
```

(6)

This line is not changing the value of i some other line is changing it. This is known as side effects.

';' is the end of statement and side effect will be reflected immediately after it

eg

```
i = 6
x = i--;   } side effect
x = i + 1;
```

(5)

side effect is taking place on 2nd line

After every sequence point side effect takes place

In C prog ;, &&, || are sequence points

This means after && or || the sideeffect will be reflected.

Immediately after any of the sequence point, side effect take place.

eg

```
i=0, j=1, k=2, m
m = i++ || j++ || k++
```

i++ → side eff → i=1 here

r=1.

eg exp1 || exp2 || exp3

⇓

(exp1 || exp2) || exp3

bcz || is L to R

eg → exp1 && exp2 && exp3

(exp1 && exp3) && exp3

→ exp1 && exp2 || exp3

(exp1 && exp3) || exp3

exp1 || exp2 && exp3

exp1 || (exp2 && exp3)

&& || exp1 will be evaluated first bcz of order of evaluation.

eg

```
i=1;
if (i++ && (i==1));
    pf("yes");
else
    pf("No");
```

i=2 now

i++ && (i==1)

i=1 ↑ 2==1 False

True sequence point

T && F = False

O/P : No

Functions

Function declaration also known as function prototype.

```
int mul (int m, int n); /* Fun prototype */
```

Either def the fun before its usage. Or declare it first and define wherever you want.

Equally acceptable forms of declaration of mul fun are:

```
int mul (int, int);
    mul (int a, int b);
    mul (int, int);
int mul (int a, int b);
```

In return type is not given by default int is assumed

eg.

```
int Fun (int x, int y); // Fun declaration
main ()
{
    int a;
    a = Fun (2, 3);
    pf ("%d", a);
}
```

No compilation error. But there will be runtime error.

Q. Will this work?

```
main() {
    int a;
    a = fun(2,3);
    pf("%d",a);
}
int fun(int x, int y) {
    return x+y;
}
```

→ Since fun is not declared before the return type of fun is assumed to be int. In def of fun its return type is int too hence no error and program will run successfully

There will be a compiler warning implicit declaration of fun.

Q. Will this work?

```
main() {
    char a;
    a = fun(2,3);
    pf("%c",a);
}
char fun(int x, int y) {
    return x+y;
}
```

Compiler assump int as return type

→ but return type is char

No. The above program wont work.
Compilation error

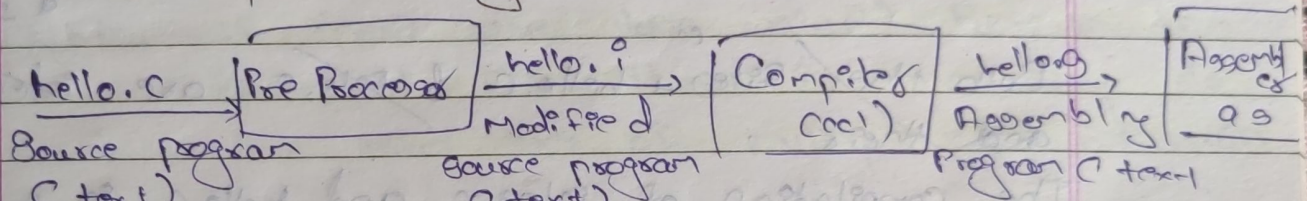
Q. Will this work? No

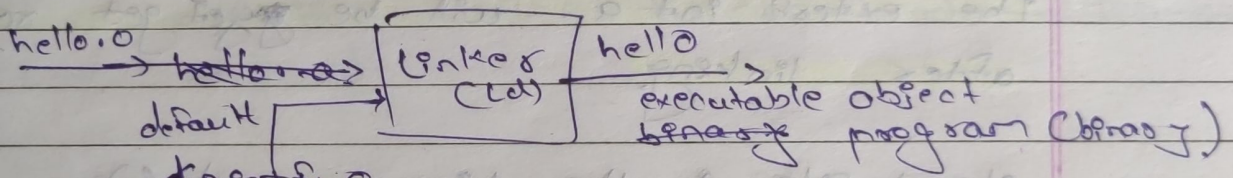
```
main() {
    float a = foo();
}
```

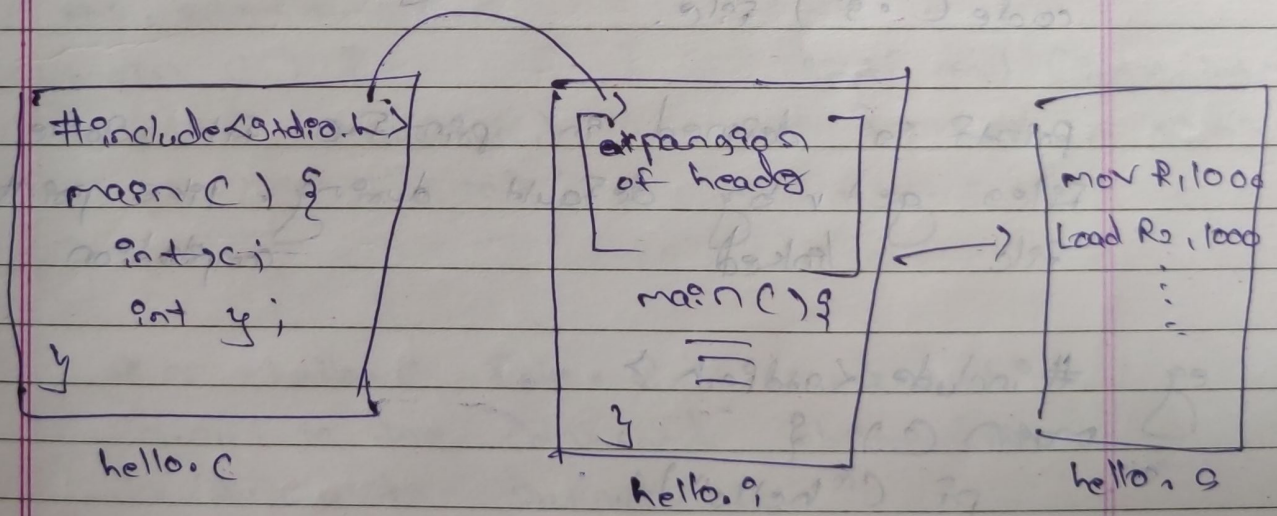
```
float foo() {
    return 3.9;
}
```

Compiler assumes int as return type but later found float Compiler error

The Compilation System



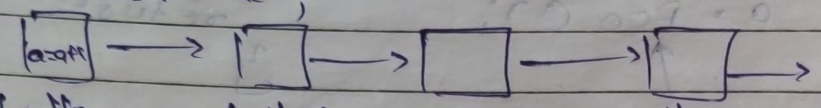




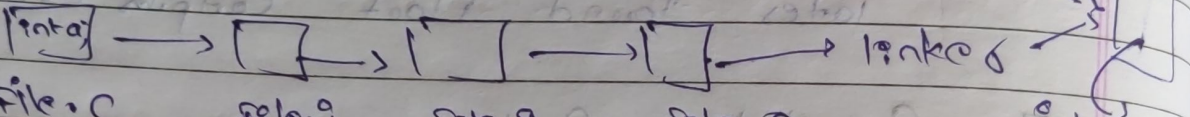
hello.c & hello.i are same file. Just so that we know header gets expanded after pre-processing we have named it hello.i.

hello.i can be renamed as hello.c

① extern int a;



② int a;



Both the files hello.c & file.c are independent and their.

The compilation of files.c and hello.c is independent

The extern int a will be ~~not~~ get resolved after linking.

Object file is the binary code (bo) for assembly code (".s") file.

printf is defined in printf.o file. Some files get by default during the compilation file linked

eg
```
#include <stdio.h>
main() {
    pf ("hello");
}
```

If header is removed [i] then the return type of pf will be considered int as default.
During the linking phase the return type will be compared with the function prototype

of printf. If return types are same then no error will be generated during linking phase.

prototype: [int printf (char *, arg,)]

[1] if header is removed. still program will be compiled successfully.

Note: If there are two main() fun in two different files. there is no way to link them bcz of multiple def of main

For compilation, main() fun of ~~header files are~~ not required

eg

hello.c

```
#include <stdio.h>
main ( ) {
  int x = mul(2,3);
  pf ("%d", x);
}
```

file.c

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

gcc hello.c file.c .a.out

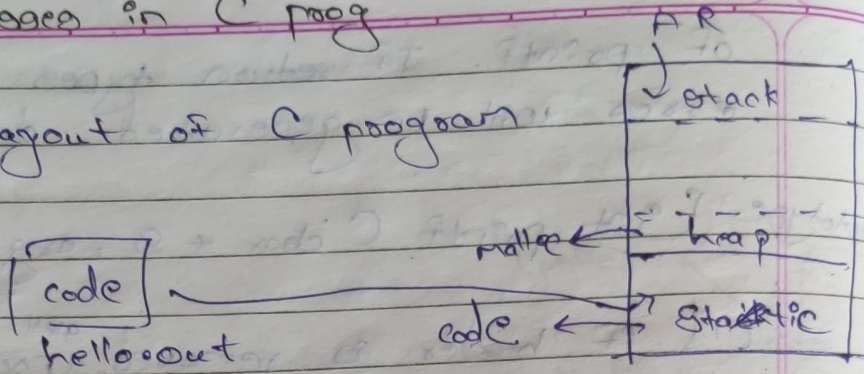
a.out is the combined object file.

./a.out Output: 6.

Date 21/3/22
Page 42

Storage Classes in C prog

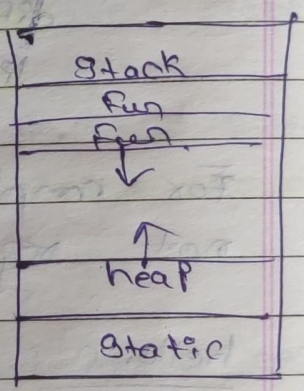
Memory layout of C program

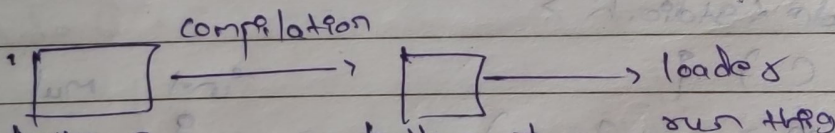


At the compile time we know size of static area.

eg
```
main()
{
  for(i=0; i<=5; i++)
    fun();
}
```

hello.out





1) Compilation
2) running loaded in MM

Stack Memory
- Run time allocation
- Local variables / function calls
  activation record

1) Virtual mem - details will be in OS
2) Activation record - details will be in CD

Heap Memory.
- Run time allocation.
- malloc() and free() are used to allocate and deallocate

Static Memory (code)
- Compile time allocation (Deciding about size of memory)
- Static Variables / Global variables
  variables outside main

C Storage classes
- Where is my variable located?
auto, register, static, extern

eg
```
int b = 3;
main() {
  int a;
  Fun();
}
```
Where "a" is located?
1) static
2) Stack
3) heap
4) Nothing

① Auto Storage Class
Storage - Stack
Initial value - garbage
Scope - within block
Lifetime - end of block

auto int a;
→ here a is having auto storage class

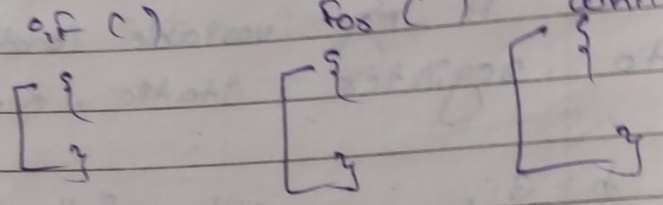
eg. main () {
int a;
{
int b;
}
}

At 1
Can I see a ? Yes
Can I see b ? Yes

At 2.
Can I see a ? Yes
Can I see b ? No

Scope: Can I see that variable

Block
{
}



Lifetime: Is the variable still in memory

eg main ( ) {
{
int a;
}
→ ①
}

At ①
Can I see a ? No
Is a in memory ? No

Bcz all local variables have auto storage class by default.

And lifetime of auto storage class var's is till end of block.

Auto storage class cont..

- Run time allocation (From stack)
- The auto storage class is the default storage class for all local variable:
  int a = auto int a;
- Objects with the auto class are not available to the linker.

eg.

File.c

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

Second.c

```
Fun ()
{
    can I use 'a' here?
}
```

No

Bcz a has auto storage class whose scope is till end of block. So accessing it outside the file is no brainer.

eg

```
1  auto int a;
2  main () {
3      auto int b;
4          for (b = 0; b < 10; b++) {
5              auto int a = b + a;
6          }
7  }
```

1. Illegal auto must be within a scope
3. Valid auto declaration
5. Valid inner block declaration

② Static storage class

Storage - static memory
Initial value - zero
Scope - within block
Lifetime - Till end of program

Objects with the static class are also not available to the linker.

eg

```
static int a;
main ( ) {
}
```
→ ①

Is this valid: Yes
Can I access a at ①: Yes

```
main ( )
{
  static int a;
}
```
→ ②

Is this valid: Yes
Can I access a at ②: No

reason: Scope of static is within block.

eg

```
static int a;
main ( )
{
}
```
File.c

another file

[Can we access 'a' here?
→ No]

File 2.c

eg

```
int a;
main ( )
{
}
```
File.c

[Can we access 'a' here?
→ Yes]

File 2.c

```
{                          {
  int a;                     static int a;
}                          }
  → ①                        → ①
```

Can I see a? No — Can I see a? No
Is a alive? No — Is a alive? Yes

③ Extern storage Class

storage	-	static Memory
Initial Value	-	Zero
Scope	-	Global
Life Time	-	Till end of program

Objects with the extern class are available to the linker.

diff storages of a.

int a; → box

static int a; → box (static a)

auto int a; → box (stack a)

extern int a; → no box for a

This keyword does not allocate memory. It is used to refer variable / function

eg

```
1 int max;          → max = 0 bcz global
2 main ( ) {
3   int len;
4   extern int max;   → which max is these ?
5   pf ("%d", max);
6   max = 5;
7 }
```

At front line max is allocated memory in static area bcz it is global.

At 4th line, extern is saying it should expect var max some where else. This some where else is same as global max; so this line is useless

eg.
```
main ()
{
    int len;
    extern int max;   ← This is not useless
    pf ("%d", max);
    max = 5;
}
int max;
```

Here extern is saying Please don't worry, Max is some where else. In this case at the end of function.

We can even remove extern keyword if max is declared at the top.

Common practice is to place definitions of all extern variables at the beginning of source files and then omit all the extern declarations.

Extern is not at all IMP if we are using single files.

eg1

```
int a,b,c=0;
void prtFun();

main()
{
static int a=1;
prtFun();
a+=1;
prtFun();
pf("\n %d %d",a,b);
}
```

```
void prtFun()
{
static int a=2;
int b=1;
a+=++b;
pf("\n %d %d",
   a,b);
}
```

Remember even though both a are static they do not refer to same variable. Both are unique in their own scope and are valid till lifetime of program.

eg2 Parameters cannot be given a storage class except register.

eg

```
int Fun1(static int para)
{
    pf("%d", para);
}
```

```
int main()
{
int q=200;
pf("%d", q);
Fun1(q);
}
```

This will throw an error. storage class defined for parameter 'para'.

Explicitly giving register storage class to params is allowed

Lec - 8

Date 24/3/22
Page 50

Extern storage class

```
int max;

main() {
  int len;
  extern int max;
  pf("%d", max);
  max = 5;
}
```

Scope

```
main() {
  int len;
  makes sense
    ↳ extern int max;
  pf("%d", max);
  max = 9;
}
int max
```

Extern storage class

Storage - Static Memory
Initial value - zero
Scope - Global
Life time - Till end of program.

all these bcz of global var

Objects with the extern class are available to the linker.

Declaration vs Definition

Declaration is what the compiler needs to accept references to that identifier.

```
extern int bar;
extern int g(int, int);
double f(int, double);
```

↳ extern by default.

Date
Page 51

Definition → what the linker needs to know in order to link references to those ~~definit~~ entities.

```
int bar;          // This will create a box
int g(int lhs, int rhs) {
    return lhs * rhs;
}
double f(int i, double d) {
    return i + d;
}
```

Declaration: "Somewhere, there exists a Foo."
Definition: "... and here it is!"

Q How to define a fun without declaring it first?
→ Define it above the main.

Q. How to declare a variable without defining it?
→ Use extern. eg extern int i;

Q How to define a variable using extern keyword?
→ extern int i; . declaration
int i; . definition
extern int i = 3; definition

Note: Function declaration is by default extern.

Prog 1.

```
int x;          will refer
extern int x;
main () {
    pf (x);
}
```

Prog 2

```
int x;
                will refer
main () {
    extern int x;
}
```

eg

```
int x;  ——> global
main () {
    int x;  ——> local
    extern int x;
}
```

Inside main, we cannot access global x

Compile time error.

This is bcz x should be refering to global variable and inside main global variable is not accessible.

We can write extern inside main or outside main but it should refer to global variable

Q. Is this declaration or definition ?
extern int x; declaration

Q. In the program where is the var 'a' getting defined and where it is getting declared?

```
int main () {

    extern int a;   ← declaration
    pf ("%d\n", a);
    return 0;
}
int a = 20;   ← definition
```

Q. Is there any diff b/w following declarations?
1. extern int fun().
2. int fun();

→ Both are identical

Note: Functions themselves are always extern, because C does not allow functions to be defined inside other functions.

Defining a function inside another function is not allowed.

eg

```
main () {
    → by default extern
    int fun();

    int a = fun();
}

int fun () {
    return 1;
}
```

```
by default extern;
main ()
    int fun ();
    int main ()

    int a = fun();
}

int fun {
    return 1;
}
```

Register Storage Class

Cannot be used for global variables

Storage	- CPU register	
Initial Value	- Garbage	} same as auto
Scope	- Within block	
Lifetime	- End of block	

eg
```
void main() {
    register int i = 0;
    pf("%d", i);
}
```

register keyword is a hint to the compiler that the variable will be heavily used and that you recommend it be kept in a processor register if possible

Most modern compilers do that automatically and are better at picking them than us humans

Q. What is default for global variables? Are they static or extern by default in C?

Argument:
If global vars are by default static then it means we would be able to access them in a single file, but we can use global vars in different files as well.
Does this imply that th

by default ?

→ auto and register cannot be used for global variables so they are out.

If static, then we cannot use them in another file

Are they global ? The problem is

1) int ⇏ static int a;
2) int a ⇏ extern int a;
   ↳ getting a box ↳ no box

Means a global variable by default is int global extern. "int a" & "extern int a" are different.

Soln By default, global variables gets stored in static memory and are available to linker. (static storage and external linkage).

eg.
```
main ( ) {
    extern int num ; [1]
    num = 20;
    pf ("%d", num);
}
```

Output ?

1) Compile error
2) Linker "
3) Run "
4) No "

→ Compile : Yes
Link : No

[1] to the compiler, don't worry num is either in same file or another file.

[1] We are trying to assign a value to a var which has not been allocated memory
extern
Simply put, extern var cannot be initialized inside a fun

Date
Page 56

But during linking no definition of num is found. Hence linker error

def
eg ~~int~~ extern int i = 10; //useless extern
int main() {
extern int i;
printf("%d", i);
return 0;
}

a) Garbage value
b) 10
c) Compiler error
d) Linker error

i is global and i inside main() will refer to global i.

eg defining not declaring. No global var found for extern
int main() {
[1] extern int i = 10; Compile error.
pf("%d", i);
}
No global var i found.

eg extern int i; // No i def somewhere

int main() {
i = 5; // For compiler, i def somewhere
pf("%d", i);
}
So no compiler errors.
Linker error bcz no def of i found.

eg.
```
extern int i;
int main() {
    int a = 5;
    pf("%d", a);
}
```
Not using this extern i anywhere.

If we are not using extern var anywhere in program then there will not be any linker errors.

Output : 5

If the above file were

Prog 1
```
fun() {
    a = 10;
}
```
Now it will be linker error bcz no def of i found.

eg.
```
extern int a;
extern int b;
extern int Foo();
main() {
    pf("Foo");
}
```
No error

If main was
```
main() {
    foo();
    pf("Foo");
}
```
Now linker error.

eg. extern int i;
extern int i;
} Multiple declarations of a variable are allowed

Initialization int i = 10; ⟶ definition of i
i = 5; ⟶ This is not allowed in C.

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

We cannot write any statement (apart from initialization) without being inside of any function.

Compile Time Error.

Lec-9

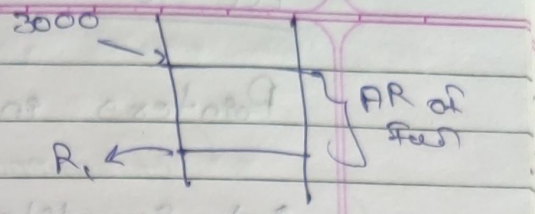
Recursion

eg
```
int f(int n) {
    static int i = 0;
    i = i + n;
    f(--n);
}
```

Static variables have lifetime till lifetime of program because in assembly code they refer to an address directly. They don't have any relative address

For other variables

Load R_1, 1000.



For static variables

Load R_1, 0x fff000. some address which refers to the static variable.

eg.
```
fun (int n)
{
  if (n == 0) return 0;
  else fun (n--);
}
```

fun (3) [3] 2 Stackoverflow

↓ n

fun (3) [3] 2

↓ n

fun (3)

⋮

Note: fun (n--) + fun (--n)

This is undefined in C. The order of execution is undefined.

For formal parameters only register storage class is allowed. No other storage class is allowed.

Date 29/3/20
Page 60

Lec - 10

Pointers in C

	Quantity	← Variable
int quantity = 179	179	← Value
	5000	← address

p = &quantity.

quantity is a variable.
Type of quantity : int

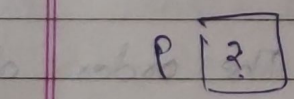
p is also a variable
Type of p : pointer.

Pointer is a variable which holds some address

Declaring pointer variables

```
int *p; /* integer pointer */

float *x; /* float pointer */
```



Contains garbage points to unknown location

Different Pointer Declaration style.

```
int* p;
int *p;        Most preffered.
int * p;
```

Illegal Uses

&125 (pointing at constants)
&(x+y) (pointing at expressions)

```
register int x;
int *p = &x;
```

you cannot take address of register variable, even if compiler decides to keep variable in memory rather than in register

int *ptr => to tell compiler that ptr is a pointer

*ptr is a pointer ✗
ptr is a pointer ✓

```
int *ptr
ptr = &x;
```

Just to tell compiler

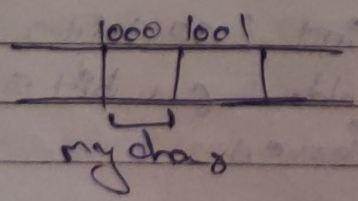
```
int *ptr = &x;
```

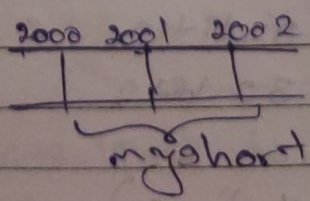
⇓ ⇓

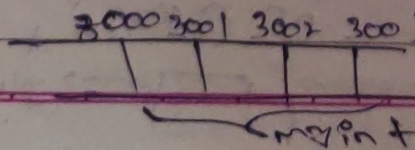
```
*ptr = &x;      ptr = &x
```

✗ ✓

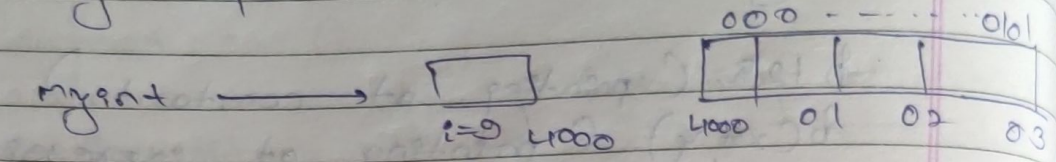
```
char *mychar;
short *myshort;
int *myint;
```



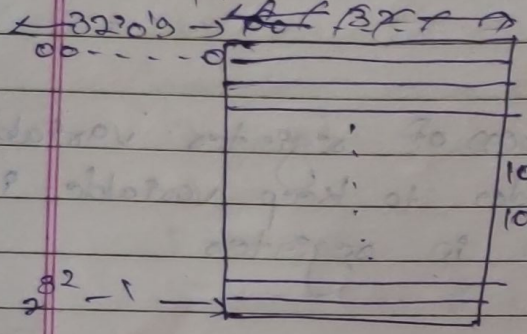




let myint point to variable i;



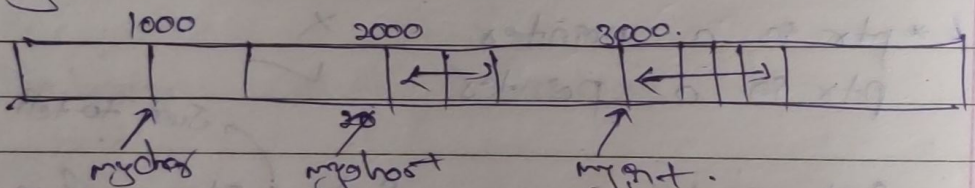
- Suppose the memory is 32 bits



mychar will be of 32 bits

myint will be of 32 bits

Any address is of 32 bit



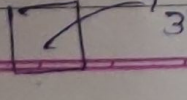
Every pointer size will be same

int *p1 — p1 → some addr
char *p2 — p2 → some addr
short *p3 — p3 → some addr.

Q Suppose

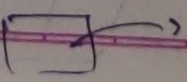
int : 32 bits
addr : 32 bits

pf("%u", p)



int : 32 bits
addr : 64 bits
→ server

pf("%u", p);



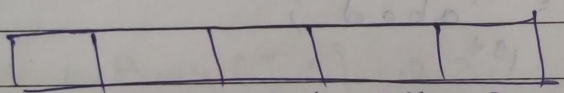
%p is a ptr which holds address. For server code addr is of 64 bits. Since "%u" is supposed to take only 32 bits it will lead to truncation of lower 32 bits.

To overcome above problem %p is used. Then we don't need to care about ptr size or data type size

Array

Array is a collection of similar data.

```
int a[5];
```



a[5] = ?

↓

May give runtime error.

Array Initialization

1) `int myArray[10] = {5,9,9,...5};`

2) `int myArray[10] = {1,2};` // initialize to 1,2,0,0,0,...

3) `int myArray[10] = {0};` all elements 0

First element is initialized to 0 bcz of this Rest all will be 0.

Same thing happening in 2nd.

4) static int myArray [10]; // all elements 0

local int myarr [10]; → garbage values
global int myarr [10]; → zero

In case of character array default value as NULL character '\0' (ASCII value zero).

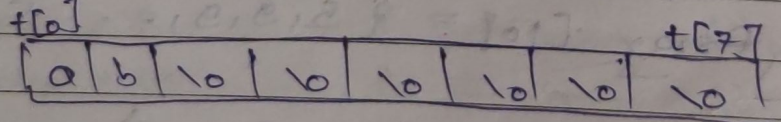
1) char t[5] = { 'a', 'b', 'c', 'd' }
init. to 'a', 'b', 'c', 'd', '\0'

2) char t[4] = { 'a', 'b', 'c', 'd' };
initialize to 'a', 'b', 'c', 'd'

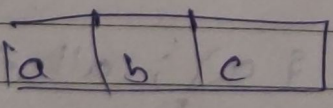
3) char s[5] = "abcd";
Same as 1st

4) char s[4] = "abcd";
Same as 2nd.

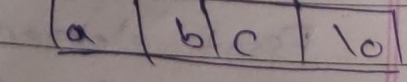
eg char t[8] = { 'a', 'b' };



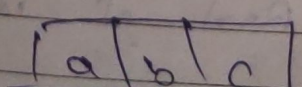
eg char t[3] = { 'a', 'b', 'c' }



eg. char t[4] = "abc"



eg char t[3] = "abc"



there will not be any null character at the end

Q. What is string in C programming?

→ There is NO string as datatype in C programming

String is nothing but a character array.

char s[5] = "abcd"

a	b	c	d	\0

} char array.

ending with NULL then we can treat as string.

Used by strlen, strcpy, in %s, strcmp.

Q. If dimension of array is not specified.

→ The compiler will deduce the dimension from the initializer list.

int myArray[] = {1, 2, 3, ..., 9};

↑ 9

char s[] = "abcd";

equivalent to char s[5] = "abcd";

① char a[5] = "abcd" Null at the end.

② char a[4] = "abcd" No Null

③ char a[] = "abcd" Null at the end

④ char a[] = {'a', 'b', 'c', 'd'} No null

⑤ char a[10] = {'a', 'b', 'c'} Null ~~at the end~~

⑥ char a[4] = {'a', 'b', 'c', 'd'} No null

eg
```
char s1[4] = "abc";
char s2[4];
s2 = s1;    /* Error */
```

An array name can't be used in left side of assignment

sizeof()

int	char	float	double
4	1	4	8

eg.
```
int a[10];
sizeof(a)  // 40
sizeof(a[0]);  // 4
```

eg.
```
short i = 20;
char c = 97;
```

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

o/p: 2, 1, 4

c + i
↳ char, short

during addition integer promotion will happen

eg.
```
int i = 3;
pf(sizeof(i++));
pf(i)
   ↳ 3
```

Inside sizeof, expression is not evaluated.

Pointer Arithmetic

Base addr

1000 mychar = mychar + 1 = 1000 + 1*1 = 1001

2000 myshort = myshort + 1 = 2000 + 1*2 = 2002

3000 myint = myint + 1 = 3000 + 1*4 = 3004

eg int *p ~~~ 3000
char *c ~~~ 1000
short *s ~~~ 2000
} assume

c+1 = 1000 + 1*sizeof(char)
= 1000 + 1*1 = 1001

p+1 = 3000 + 1*sizeof(int)
= 3000 + 1*4 = 3004

eg type *p;

p+1 = p + 1*sizeof(type) ✓
= p + 1*sizeof(type*) ✗

eg char *p
p+1 = 1000 + 1*sizeof(char) = 1001
= 1000 + 1*sizeof(char*) = 1004
Assuming pointer are 32b (4B)

eg char **p size of char = 1
addresses are 8B long
p+1 = 1000 + 1*sizeof(char*)
= 1000 + 1*8
= 1008

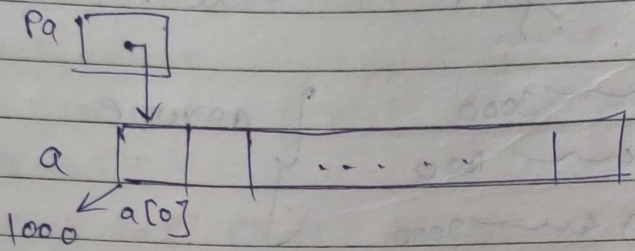
Note: In case of array (*p) may not be 8B.

Pointers and Arrays

int *pa; → Pa is a pointer to integer.

int a[10]; → a is an int array of size 10.

Pa = &a[0];



eg Pa++ = ? Pa = Pa + 1

= Pa + 1 * sizeof(int)

= 1004

eg int arr[10];

Is arr a pointer? No

Is arr a const pointer? No

arr is just an alias for the base address.

arr does not have its own box.

eg int *p; a is 1000

p = &a[0]; } &a[0] is also 1000

p = a; } both same

Little endian vs Big endian

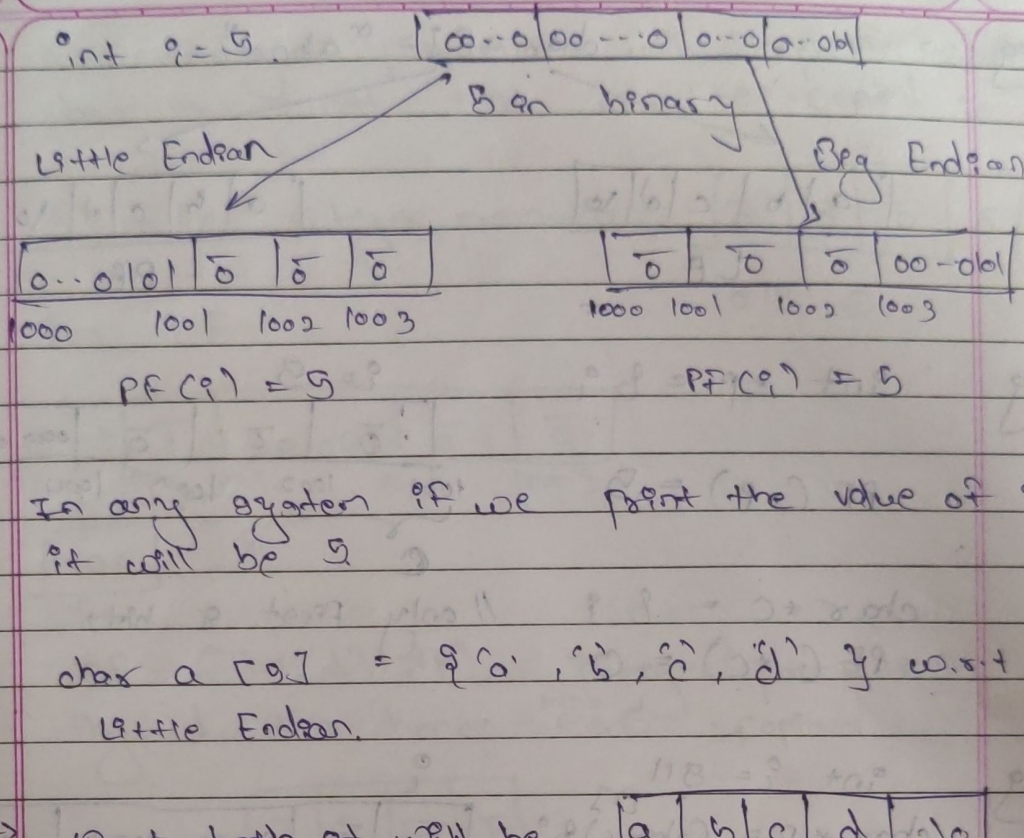
If data is multiple bytes then there are 2 ways

1) Little endian

2) Big endian

LE - lower addr lower byte
BE - higher addr lower byte

int i = 5;



PF(i) = 5 PF(i) = 5

In any system if we print the value of i it will be 5

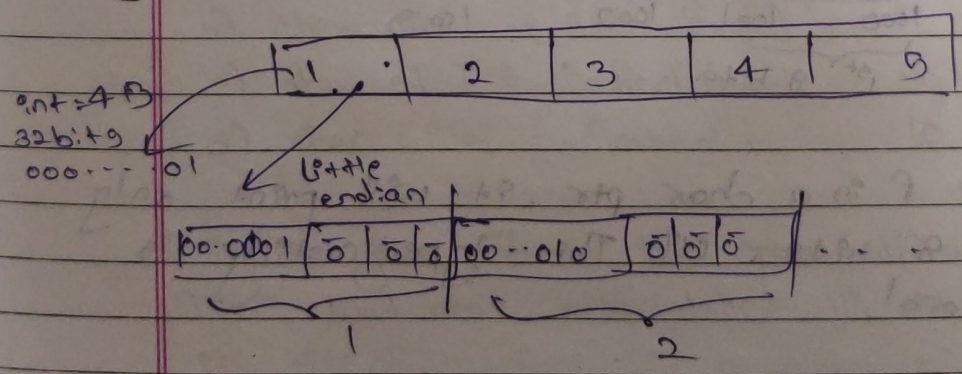
eg. char a[5] = {'a', 'b', 'c', 'd'}; w.r.t Little Endian.

→ w.r.t both of it will be | a | b | c | d | \0 |

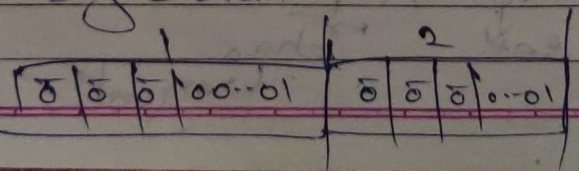
LE, BE is for how a particular data type will be stored. If Array will be stored as usual.

Another eg for clarification

eg. int a[5] = {1, 2, 3, 4, 5};



Big endian

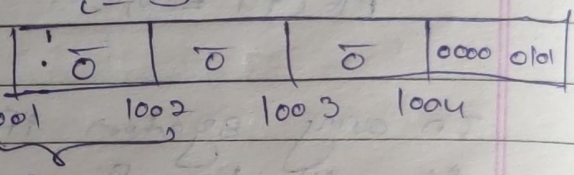


eg

```
char a[] = "abcd"
```

eg

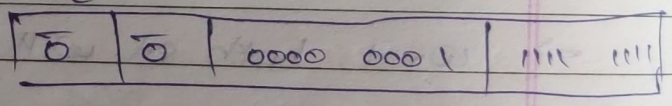
```
int *p = &i;
PF (*p) = 5
```




```
char *c = &i;   // only first 8 bits
PF (*c) = 0
```

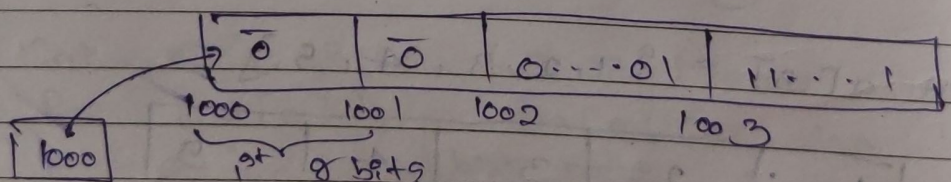
eg

```
int i = 511
char *p = (char*)&i;
PF ("%d", *p);
```



This depends how int is stored LE or BE.

For BE

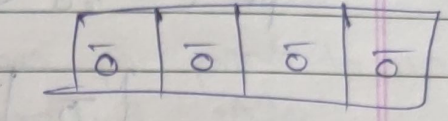


Since P is a char ptr, it will point only 1st 8 bits, [1] This type casting is optional.

Now, PF ("%d", *p)
int ← char
↳ signed

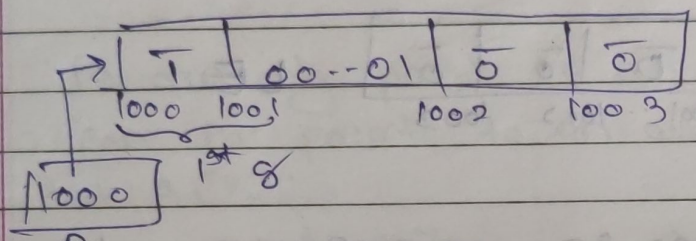
Now integer promotion will happen. since char is signed MSB will be copied ie 0

*p after integer promotion

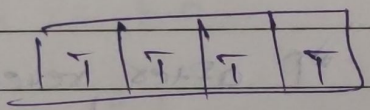


o/p: 0

For LE



pf ("%d" *p) Integer promotion. signed MSB hence MSB copied



o/p: -1

HW-11

Q.2 sizeof(char) on each case

a) char c[] = "abcd"; 5

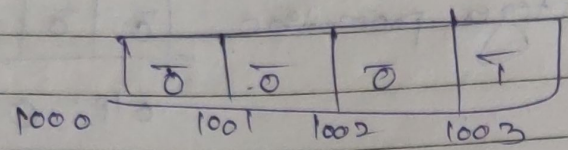
b) char c[] = {'a', 'b', 'c', 'd'}; 4

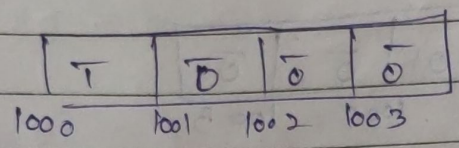
c) char c[5] = "abcd"; 5

d) char c[10] = "abcd"; 10

size of gives/returns the space allocated to a variable since Null char also takes 1B it is included in size of.

Q.8
```
int i = 299;
int *p = &i;
pf ("%d \n", *p);
```





Little endian stores in reverse way, also when we fetch some value, it first reverse and convert to decimal

So decimal value of *p after reversing 1 0 0 1 0 1 1 is 299 signed or unsigned.

In case of BE its usual method. 299

Lec-12

Discovering a[i]

a represents address of first element in array
a+0 " " "
a+1 " second " "
a+i " i th " "

a[i] is same as *(a+i)

Two ways to access first element of an array.
1) a[0] 2) *a.

Q int *pa;
pa = &a[0]; (or)
pa = a.

Does pa point to an array? No

Pa can only point to an integer.
Can Pa point to an array of integers. No
int a[5];

eg char *c = &a[0]
short *s = &a[0]
int *p = &a[0]
? t = &a

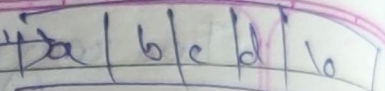
c+1 = 2001
s+1 = 2002
p+1 = 2004
t+1 = &a+1
= 2000+1*5*4
= 2020

eg. int a[8];
a+0 → No error
a+8 → No error
a+10 → still no error
*(a+10) → may lead to runtime error.

eg point *p = &a;
p[i] is equivalent to *(p+i)

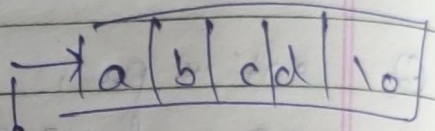
eg int a[5] = {1,2,3,4,5};
int *p = a; }
int *p = &a[0]; } both are same

char c[] = "abcd"



here c = c+1 is not allowed c

char *c = "abcd"



here c = c+1 is allowed

size of array vs size of pointer

int a[10]; size of (a) = 40 // size of array
int *p = &a; size of (p) = 4 // size of ptr

size of (*p) = 4 // first element of array
size of (&a) = 4 // size of an addr, 4B

eg long int *i; char *ch; short int *st;
int *a; float *f; double *d;

size of each of these is same
size of (*i) = size of (ch)
4 4

eg T / F.
int *p
char *c
size of (*p) == size of (*c) ? False
int char

Passing 1D array to function

* copies the base address of array to function parameter.

```
void func1 (int *a) { }
void func2 (int a[]) { }
void func3 (int a[10]) { }
```

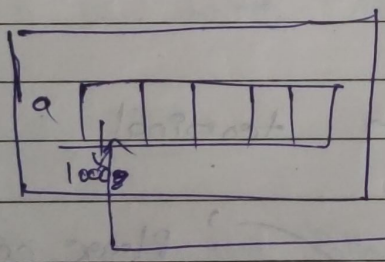
} All are same

eg.

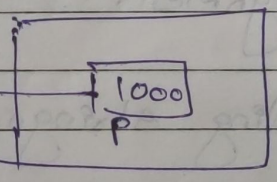
```
Main () {
    int a[5];
    Fun (a) ;
}
```

```
Fun (int *p) {
    // do something
}
```

AR of main



AR of fun



eg.

```
main ( ) {
    int a[5];
    // a = a + 1 ? ①
    fun (a) ;
}
```

internally → int *a

```
func (int a[5] ) {
    // a = a + 1 ②
}
```

① is not allowed as 'a' is an array name

② is allowed because now this local a is a pointer

eg

```
int main() {
    int a[5]
    fun(a)
}

fun(int a[10]) {     -> internally int *a
}
```

This will run without any error.

int a[10] or int *a or int a[]
all internally works as int *a;

Passing 1D array to function

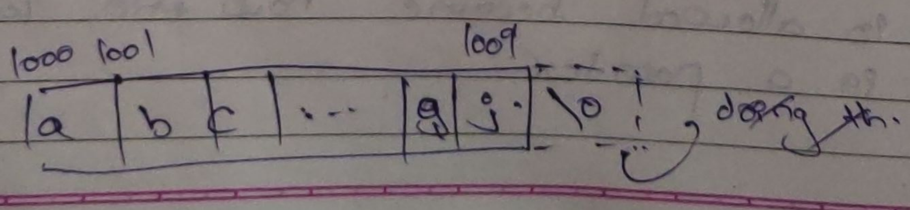
1) What really gets passed is a pointer to the array's first element.
2) We can use parameter name on the left side of assignment.

Reading strings from terminal

char address[10]
scanf("%s", address); -> Please enter string with 9 characters

But if we write 10 characters then compiler will try to place '\0' outside the allocated address which may lead to runtime error bcz that memory may not be accessible.

scanf("%s", 1000) -> at addr 1000
o/p : ab...j



int arr[10]; int arr2;
Remember array name is an address so doing scanf("%d, arr) is allowed, but doing so for other variables is not allowed sf("%d", arr2)
↳ error

Date
Page 77

Scanf will put Null char after 9. But doing so may result in runtime error.

Strlen function

```
int strlen(char *s){
    int n;
    for(n=0; *s != '\0'; s++)
        n++;
    return n;
}
```

eg pf("%d", strlen("Hello"));
o/p: 5

pf("%d", sizeof("Hello"));
o/p: 6

bcz '\0' takes 1B as space.

HW0-12

Q.6 Which of the following is a correct way to use the scanf function.

a) int i=0; sf("%d", i);
b) int i=0; int *p=&i; sf("%d", *p);
c) char s[10] = "1234567"; sf("%s", s);
d) char c; sf("%c", c);

only C. sf expects address (i.e pointer) as an argument.

Lec-13

Note: strlen() doesn't count Null Characters
sizeof() counts null character

eg
```
main() {
  char s[] = "string"
}
```

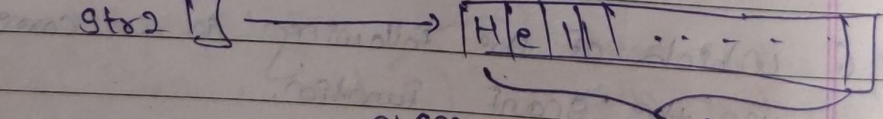
This will be stored on stack and not in static area

To strlen() a char pointer is passed

Character string array vs Character Pointers.

```
char str1[] = "Hello World!";   → Stack (static)
char *str2 = "Hello World!";
```

both assumed global

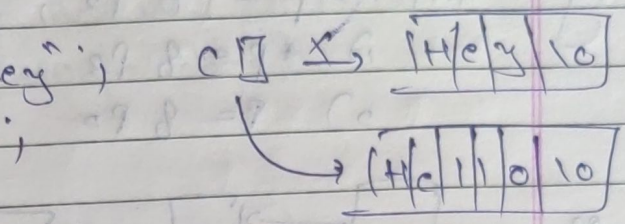


Even in static, the string str2, "Hello World" will be in ROM

```
str1[1] = 'u';   // Valid
str2[1] = 'u';   // Invalid - May crash program.
```

```
char *c = "Hey"
c[1] = 'x'   cannot modify, not allowed
pf("%c", c[1])  Reading is allowed
```

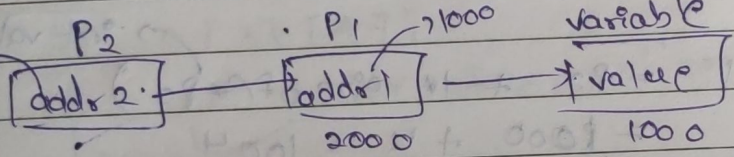
```
main()
{
  char *c = "Hey";
  c = "Hello";
}
```



both are static

Double Pointer

is a pointer which points to pointer which points to a variable




```
int x, *p1, **p2;
x = 100;
p1 = &x;
p2 = &p1;
pf ("%d", **p2);
```

eg
```
int ***p;
p = 1000
p+1 = p + 1*sizeof(**p)
```

addr = 8B.
int 4B

*p means what p points to. P point to addr of an int. so *p will be an addr
*p addr, **p addr, ***p value

1000 + 1 * (8)
= 1008

Things change in case of arrays

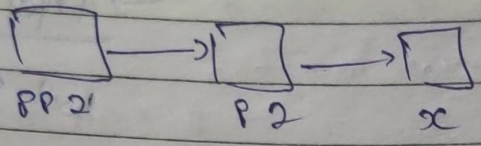
Date
Page 80

eg `int **p = &p2;` means ?

i) `*p = &p2` ✗

ii) `p = &p2` ✓

eg



`int **pp2, *p2, x;`

`pp2+1 = pp2 + 1*sizeof(*pp2)` → int ptr

`= 1000 + 1*8 = 1008`

`p2+1 = p2 + 1*sizeof(*p2)` → int value

`= 1000 + 4 = 1004`

eg `int a[10];`

`&a+1` ⟶ It will skip one array

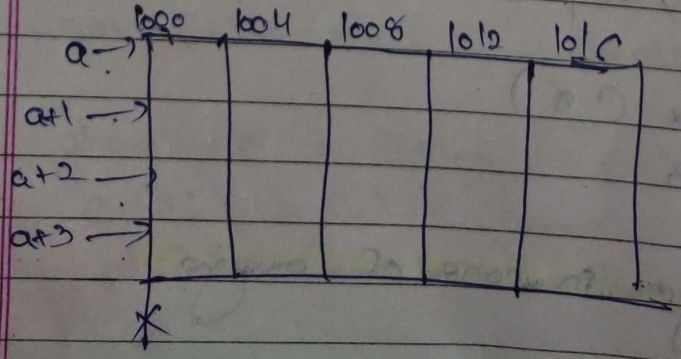
`&a+1 = 1000 + 1*sizeof(*&a)`

`= 1000 + 1*sizeof(a)`

`= 1000 + 1*40 = 1040`

2D arrays

`a[4][5];` `int *p = &x;`



value of a = 1000

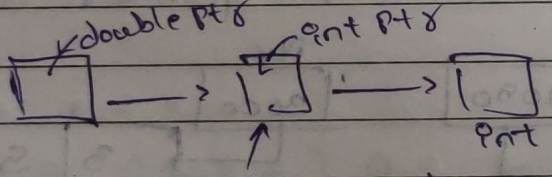
`&a[0][0]` = 1000

Date
Page 81

a ⟶ addr (of a row)
*a ⟶ addr (of single elem)

eg
```
int a[3][4];
a ← addr          a[2] ← addr
*a ← addr         a[2][3] ← int
*a[2] ← int       *(*a+3) ← int
**a ← int         (*(a+2)+3) ← addr
                  *(*(a+2)+3) ← int
```

eg.
```
int **t
int a[3][4];
t = a ;   addr of row (may throw error)
```
↳ suppose to have addr of some int ptr.



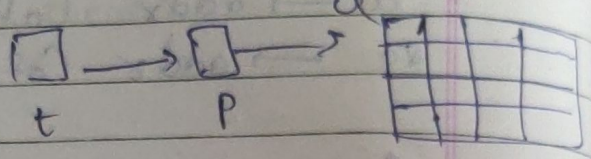
t is supposed to have addr of this type

t = *a ✗ *t = *a ✓
↳ bcz this is addr of int not int ptr.

eg
```
int a[3][4]
int *p;
p = *a ; // Valid
```

eg
```
int a[4][4]
int *p;
1) int *p = a ✗
2) int *p = *a ✓
3) int *p = &a[0][0] ✓
4) int *p = &a ✗
```

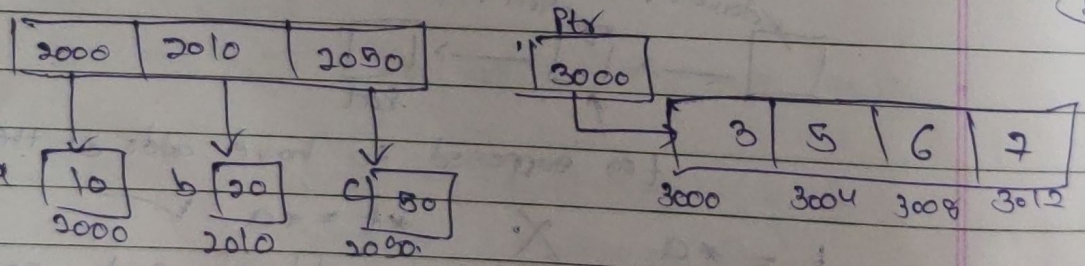
eg int a[4][4]
int **t, *p;
p = *a
t = &p
addr of some int ptr



Q T / F int arr[5][6]
arr[i] == &arr[i][0]
addr of int addr of int True

Lec-14

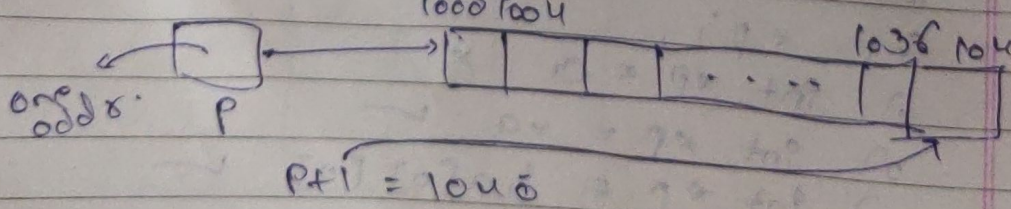
Array of pointers vs pointer to an array



Pointer to an array.
int (*p)[5].

Pointer to an int
int *p.

Pointer to an int array of size 10
int (*p)[10];



size of (p) ? 8 B
size of Ptr

Pointer to an array of 8 integers

int (*p)[8]; ✓

int (*p)[]; ✗ invalid / error.

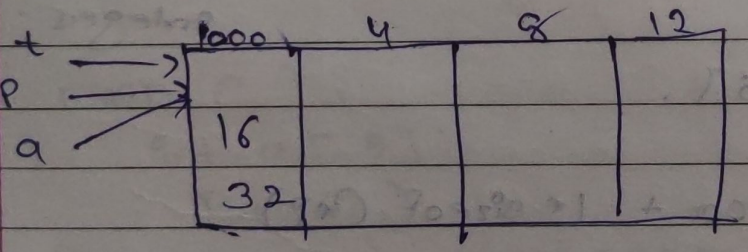
int (*a)[2]; a: a is a pointer to int array of size 2

int P[3][4]; P: P is a pointer to int array of size 4

a = p ✗ invalid.
↳ addr of array of size 4

a = (int(*)[2]) p ✓
Now this is valid.

int *t;
t = (int*) a ✓
P+1 → next row.



a → 1000 a+1 = ~~1006~~ 1008
P → 1000 P+1 = 1016
t → 1000 t+1 = 1004

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

✗ a) P = a ✗ c) P = *a
✓ b) P = &a ✓ d) P = (int(*)[5]) a

int a[10] // any size array 10, 20, ...

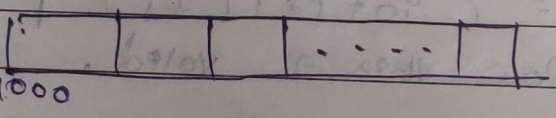
int *p;

p = a } p+1 : shift one int
p = &a } p+1 : "
(int *)

int (*p)[5]

p = a } p+1 : shift 5 integers
p = &a }

eg int a[5]
int (*p)[5] = &a;
sizeof (a) same as sizeof (*p)

eg 

char *c c → 1000 *c → give me one byte
int *t t → 1000 *t → give me 4 byte
int (*P)[5] P → 1000 *P → an addr of first integer.

eg int (*P)[5].
P = 1000
P+1 = 1000 + 1 * sizeof (*P)
= 1000 + 1 * 20
= 1020

For sizeof Pointer to an integer
*P : integer

Pointer to an array
*P : an array

Date
Page 89

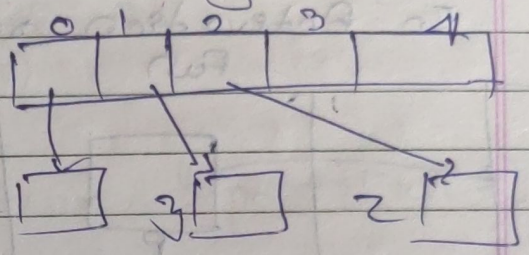
Array of Pointers

```
int *array[10]; // array of 10 int pointers
```

eg
```
int *a[5];
int x, y, z;

a[0] = &x
a[1] = &y
a[2] = &z
```

a : array of int pointers



Passing 2D array to function

```
Fun ( int (*a)[y] ) { }   ✓
Fun ( int a[] [y] ) { }   ✓
Fun ( int a[x] [y] ) { }  ✓

Fun (int **a) { }   ✗
Fun (int a[x] [] ) { }   ✗
```

eg
```
main ( ) {
    int a[5];
    Fun (a)
}
```
↳ addr of one int

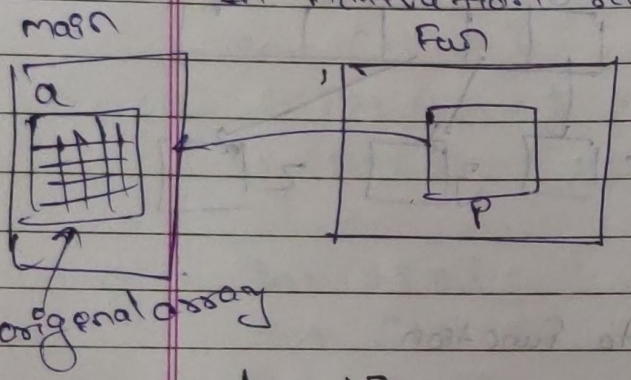
```
Fun ( int *p) {
}
```

eg
```
main ( ) {
    int a[5][6];
    Fun (a)
}
```
↳ addr of one row of size 6

```
Fun ( int (*p) [6] ) {
}
```

Fun (int a[5][6]) or int [][6] or int [50][6]
↳ internally
int (*a)[6]
will be ignored

In Activation record of Fun()



it won't have space for 6 element. It will have space just for one pointer var

lec-15

double points[3][4] = { // data };

0	1	2	3	4
32	5	6	7	8
64	9	10	11	12
96				

double (*p2)[4] = &points[1]; (or)
points + 1
both points to 2nd row

double (*p1)[3][4] = &points;
p1+1 = skip whole 2D array // 96

Now, (*p2)[0] = 1st element in 2nd row
same as points[1][0]

*p2 = addr to 1st element in 2nd row
same as &arr[1][0]

[1] size of (points) = whole 2D array
size of (*points) = size of 1-D array
size of (**points) = " " integers.
Things work differently in case of sizeof

Date
Page 87

double *p3 = &(*p2)[2];
= &*(*p2 + 2)

*p2 + 2

*p2 is pointing to an int not whole array

*p2 + 2 = addr of 7

~~&*~~ (*p2 + 2)

cancelled.

so p3 = & of 7 or & points[1][2].

(*p2) + 1 or *p2 + 1 is addr of 6

*(*p2 + 1) = 6 element.

p2 + 1 = addr of 9. and pointing to whole 3rd row

Q double points[3][4];

a) Points - An array of 3 arrays of 4 doubles

b) &points - The addr of (pointer to) an array of 3 arrays of 4 doubles.

c) &points[0][0] - The addr of (pointer to) a double (the first double in the first row)

eg int points[2][2];

sizeof (points) → 2D array 2 × 2 × 4 = 16.

sizeof (points + 1) → address → 1D = 2 × 4

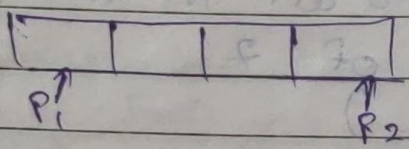
Points + 1 = 1000 + 1 * sizeof (*Points[1])

= address

*points to still 1-D array

Date
Page 88

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.

eg. int 

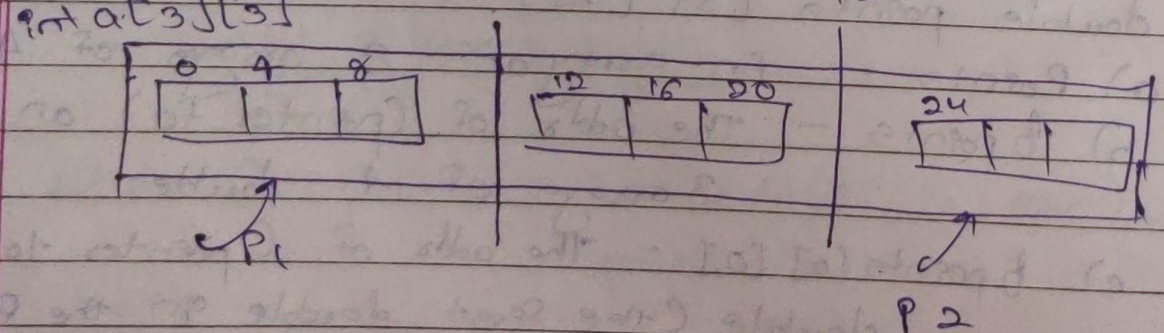
$P_2 - P_1 = 3$ size of $(P_1) = 8$

$P_1 - P_2 = -3$ size of $(*P_1) = 4$.

eg. int a[3][3].
int (*p1)[3] = a;
int (*p2)[3] = a+2.

$P_2 - P_1 = 2 = \frac{1024 - 1000}{\text{size of } (*P_1)}$

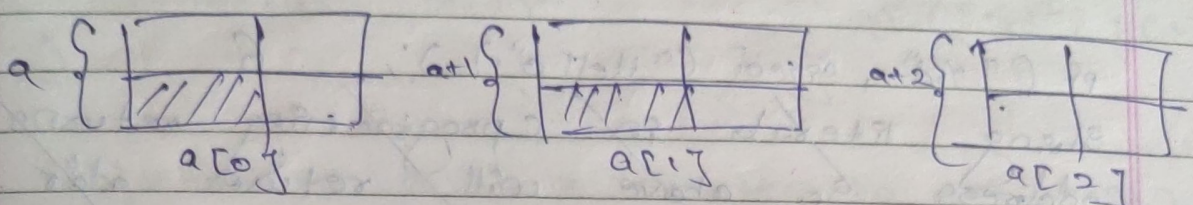
$= \frac{24}{12} = 2$



P_1 is pointing to an array of size 3
$*P_1$ is an array of size 3

P is pointer to an integer.
$*P_1$ ⟶ int.

eg a[3][2][2];



a[0][1] - a[1][1] = -4

To solve this better way is to find number of elements b/w two arrays.

Valid arithmetic on Pointers		Invalid
$P_1 + 3$	$P_1 \leq 2$	$P_1 + P_2$
$P_1 - 3$	$P_2 \geq P_2$	$P_1 * P_2$
$P_1 - P_2$	$P_1 == P_2$	$P_1 \% P_2$
Only when both are pointing to same array		P_1 / P_2

Void Pointer

```
int main() {
    int a = 7;
    void *p;        → It can hold an address
    p = &a;
    pf("%d", *(int*)p);
}
```

If we don't typecast then void pointer doesn't know how many bytes to read hence typecasting is necessary for void pointers.

Dereferencing a void pointer without typecasting is invalid.

pf("%d", strlen("Hello")); 5.

pf("%d", sizeof("Hello")); 6

String literals in C programming returns base address so above will return addr of H, so o/p will be dependent on size of pointer.

Lec-16

int (*p)[5];
p is a pointer to an array of 5 integers

int *f()
f is a function returning int pointers

int (*pf)();
pf is a pointer to function returning int.

int (*pf)(int);
pf is pointer to function taking int argument and returning int.

int *(*(*fp1)(int))[10];
fp1 is a pointer to function takes int as argument returns pointer to an array of 10 int pointers.

int (*ap[5])[10];
ap is an array of 5 pointers to array of 10 integers

Pointer to Function

```
int my-fun(int x) {
    return x+1;
}
```

```
main() {
    ?
    pf("%d", p(2));
}
```

my-fun is called a pointer to a function

In place of ?, `int (*p)(int);`

`p = my fun`

`p(2)` or `my-fun(2)`

Malloc Fun in C

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

(sizeof(int) * 10 → 40 B)

Malloc(10) will give you 10 B from heap.

malloc returns a void pointer. So in those 40B we can store 10 integers or 40 characters? So type casting is used.

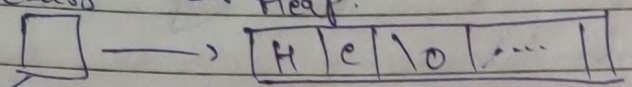
→ Malloc returns the void type pointer, we can use that without typecast.

→ Malloc returns the void because, malloc doesn't know for what purpose this memory is being used.

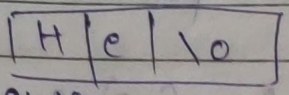
→ On Fail, malloc returns NULL.

c_arr = (char*) malloc (sizeof (char) * 10);
strcpy (c_arr, "He"); → static storing literal

c_arr Heap.



stack/static



static

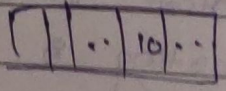
In heap section, uninitialized variables/cells will be garbage.

Releasing the used space: Free

Free(ptr)

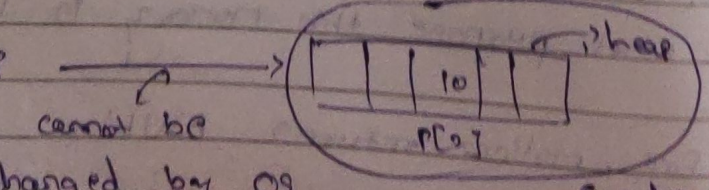
What does Free(ptr) do?

p = malloc(40)
p[2] = 10 p →



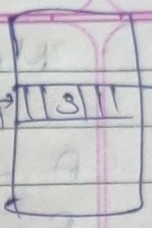
Free('p')
does this mean value '10' is deleted?

Free is handled by OS. So it does not matter what it does with Free.



Free(p) means the pointer do not own the memory now. It is upto the OS (reassignment, garbage value, etc) what it do with it.

eg P = malloc(40)
P[2] = 3
free(P) → P still points to 1000
PF("%d", P[2])
↳ dereferencing is not allowed



bcz after free, P does not own that memory.
runtime
So may lead to error.

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

```
char *fun() {
    char str[10] = {"hello"};
    return str;
}

main() {
    char *c;
    c = fun();
}
```

str is inside AR of fun(). After return its AR will be destroyed. So c is pointing to ~~an address~~ ~~which~~ something which has been destroyed.

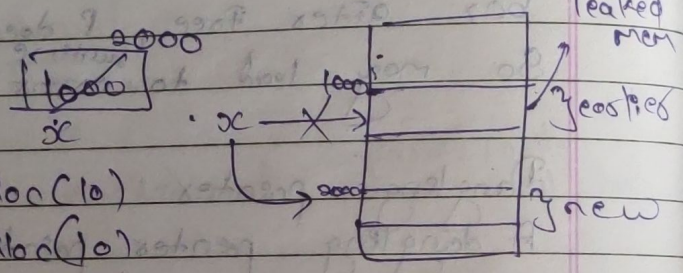
So c is a dangling pointer here

eg
```
int *c = malloc(sizeof(int));
free(c);
*c = 3; // writing to freed location
```
runtime error

Memory Leak

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

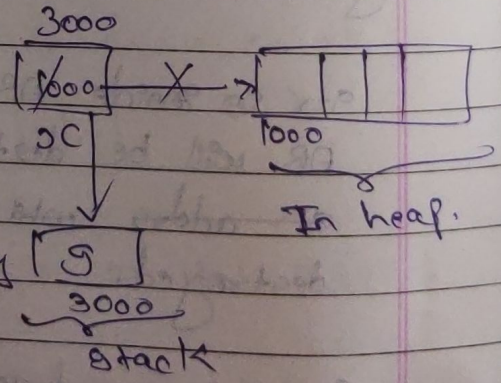
```
main() {
    int *x = malloc(10)
    x = malloc(10)
}
```



No way to access old memory. Old mem is not freed and there is no way to access it.

Lec-17

eg.
```
main() {
    int *x, *z, y;
    y = 9;
    x = malloc(4);
    free(x);
    x = &y
}
```



eg.
```
main() {
    x = malloc(4);
    x = malloc(8);
}
```

Memory leak problem

Same ques

```
x = malloc(4)
z = x
x = malloc(8)
```

Not a memory leak.

Date
Page 99

Memory Leak:
You do not have the address of the memory and memory is not freed.

Dangling Pointer:
Points to a memory which is invalid.
You have addr of memory but memory is freed.

getchar() : Reading a character

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

putchar(): writes a character to screen

putchar(c) is equivalent to pf("%c", c);

Macros
All lines that start with # are processed by preprocessor

eg. #define plusone(x) x+1
i = 3 * plusone(2) ;
becomes
i = 3 * 2 + 1
= 7

$

Date
Page 96

Structures

Used for handling a group of logically related data items

datatype may differ

eg
```
struct Box {
    int width, length, height;
};
```
easy to forget.

Declaring struct variables

```
struct Box b1, b2;
```
or
```
struct Box {
} b1, b2;
```

Typedef

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

```
typedef type newname;
typedef int dollars;

dollars b;   same as  int b;
```

We can typef a struct.

```
typeplef struct Box Boxes;

Boxes b1, b2;
```

typedef

Simplex: `struct {`

```
    int w, l, h;
} Box;
```

Date
Page 97

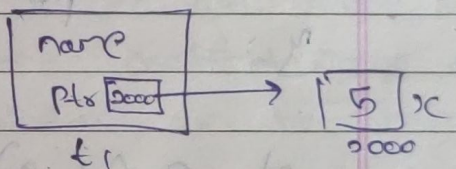
Structure member can be accessed using '.' operator

```
Box b1;
b1.length = 10;
```

Pointer as member of struct

```
struct test {
    char name[20];
    int *ptr_mem;
};
struct test t1;
```

```
x = 5
t1.ptr_mem = &x;
```



*(t1.ptr_name) ≡ *t1.ptr_name Valid

bcz '.' has higher precedence than *

*(t1).ptr_name invalid.

Then

*(t1.ptr_name) = 5

2000

t1.ptr_mem is an integer pointer (able to store address)

*t1.ptr_mem is an integer (able to store integer num).

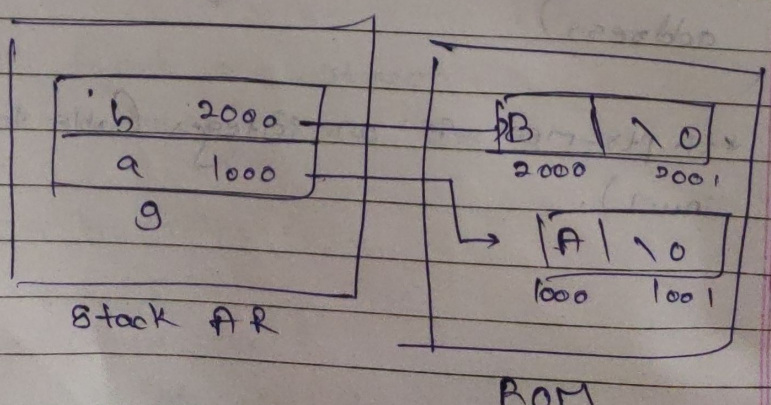
```
typedef struct {
    char *a;
    char *b;
} t;

main() {
    t g = {"A", "B"};
    pf("%g %g\n", g.a, g.b);
    F1(g); // local copy
    pf("%g %g\n", g.a, g.b);
    pf("%c %c\n", *g.a, *g.b);
}

void F1(t g) {
    g.a = "U";
    g.b = "V";
    pf("%g %g\n", g.a, g.b);
}
```

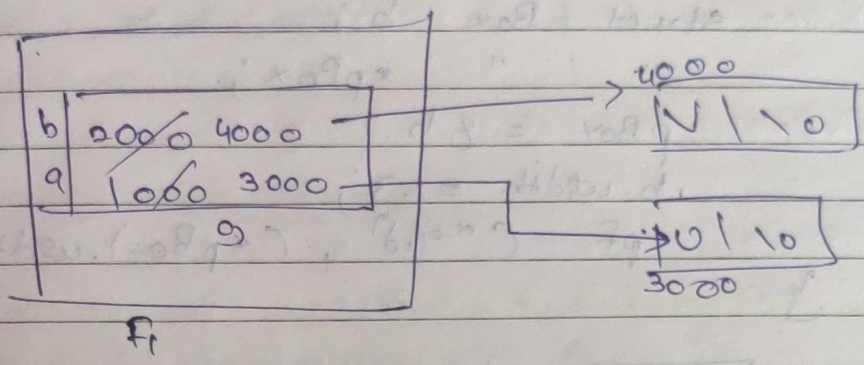
t g = {"A", "B"}

"A" & "B" are string literals they are in static area and that too in ROM portion.



f1 (t g)

Here, ~~local copy~~ value of g passed. So changes done in f won't be modified on actual f.



The contents of g cannot be changed since in 'f', g is a local copy. but the ~~content~~ value of value pointed by a pointer can be changed.

eg. *g.b = 'a'

but since these are string literals and not array, the value of pointers too cannot be changed.

string literals are in static ROM and cannot be changed.

Note: struct names are not pointers to anything

```
struct example {
    int foo;
    int bar;
};
struct example e;
```

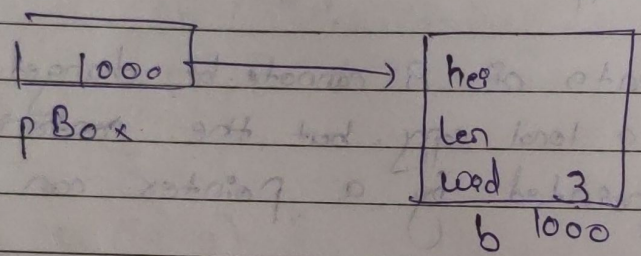
e is not a pointer to anything
e is not a mnemonic like array.

Date
Page 100

Pointer to struct

Box struct.

```
int main() {
    struct Box b;
    "      "  *pBox;
    pBox = &b.
    .b.width = 3;
    pf ("%d", (*pBox).width);
}
```



pBox = &b
*pBox = b.

(*pBox).width. ✓
pBox.*width ✗

b.width.
(*pBox).width
b.length
(*pBox).length

*(pBox.width) same ✗
*pBox.width ✗

Once pBox points to a structure variable, the members can be accessed as:

pBox → width // same as (*pBox).width
pBox → length // " (*pBox).length
pBox → height " height.

Date
Page 101

Copying one struct to another

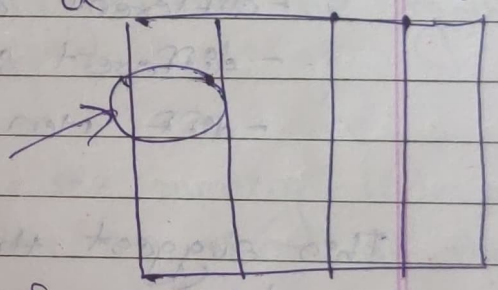
```
new = old;
```

Rest were solved eg's. If needed check structures slides from pg 41.

Pointers and 2D array:

There is no relation between 2D arrays and double pointers.

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



p is expecting address of a int pointer. There is no way p can directly point to a.

A workaround is

```
int *t.
t = &a[1][0]
p = &t.
```