→ do
pf("Hai\n");
while(i++<3);

Session 31

Control statements
→ Switch case
→ break
→ Continue
→ return

Switch case:-

→ Switch is a conditional non-iterative control statement
→ It is also called as multi-directional conditional control statement
→ Mainly, the switch case is used in a Menu based programs.

Syntax:-

```
Switch (expression)
{
case constant1: statement 1
case constant2: statement2
default : statement
}
```

```
Switch (expression)
{
case label1:
    ...
    break;
case label2:
    ...
    break;
default:
}
```

→ Note:- These labels should be integer family constants. Variables are not allowed.
✗

'Break' is optional and 'default' is optional

Example:-

```
void main()
{
 int num;
 pf("Enter the number\n");
 sf("%d",&num);

switch(num) ← The result of the this exp is compared.
{
case 1: pf("1---\n");
        break;
case 2: pf("2---\n");
        break;
case 3: pf("3---\n");
        break;
default: pf("Not 1234\n");
   }
}
```

With the labels

o/p:- Enter 1
1---

→ Default could be written anywhere in the middle.

If break statement is not given, the remaining code is also executed::

eg:-
```
switch(num)
  {
  case 1: pf("1---\n");
  case 2: pf("2---\n");
  case 3: pf("3---\n");
  default: pf("Not 123\n");
  }
```

o/p
Enter a num
2
2
3
Not 123

Because break is not used so after 2 it will execute all as statements

Valid :-
Case 1: Case 'a'
Case 2+4: Case 'a'>'b'
Invalid :-
Case "first"

Vowels using switch case:-

```
{
 char ch;
 printf("Enter a char \n");
 scanf("%c", &ch);
 switch (ch)
 case 'a' :
 case 'e' :
 case 'i' :
 case 'o' :
 case 'u' :
 case 'A' :
 case 'E' :
 case 'I' :
 case 'O' :
 case 'U' :
      printf("Yes\n");
        break;
 default : printf("No \n");
    }
printf("Thank you \n");
    }
```

Fig: b1 books

Break :-

Break is an unconditional non-iterative.

Note:- Break should be within loop or switch case.

eg:-

```
Void main()
{
printf("Hello \n");
break; ← Treated as syntactical error
printf("Bye \n");
    }
```

Syntax:-

```
break;
```

return :-

→ return belongs to unconditional non-iterative control statement.

→ It is used to come out from the function.

There are two types of return.

1. Control return
2. Value return (more) Deals with function.

Statement below the keyword 'return' will not execute.

// Control return example

→
```
{
int i;
pf("Hello\n");
for(i=0; i<5; i++)
pf("Hai\n");
 return;
}
pf("Bye\n");
}
```

Continue :-

→ It is an unconditional non-iterative control statement.

eg :-

```
Void main()
{
   ≡
while (__)
{
   ≡
   continue;
}
≡
}
```

control goes from here to main()

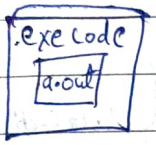
Session - 39

→ Secondary data types:

→ Memory map
→ Introduction to pointers

Memory Map:-

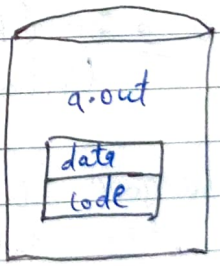
On compiling a program, e.g cc p1.c ↵ is given the executable file is created in the hard disk.



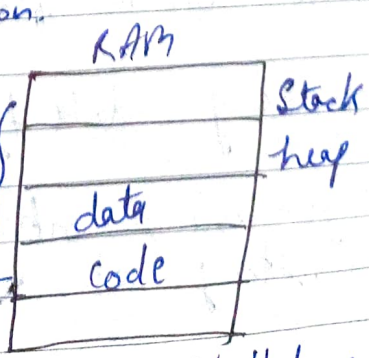
to execute the code type ./a.out
↑ represents present working directory

When ./a.out is typed, this code is loaded into RAM for executions. At that time, OS allocates some memory.

→ The executable file contains 2 sections.
(1.) data section
(2.) code section



→ When the exe file is loaded into RAM, 2 more logical sections are allocated by the OS apart from code & data section.
(i.) Stack section
(ii.) Heap section

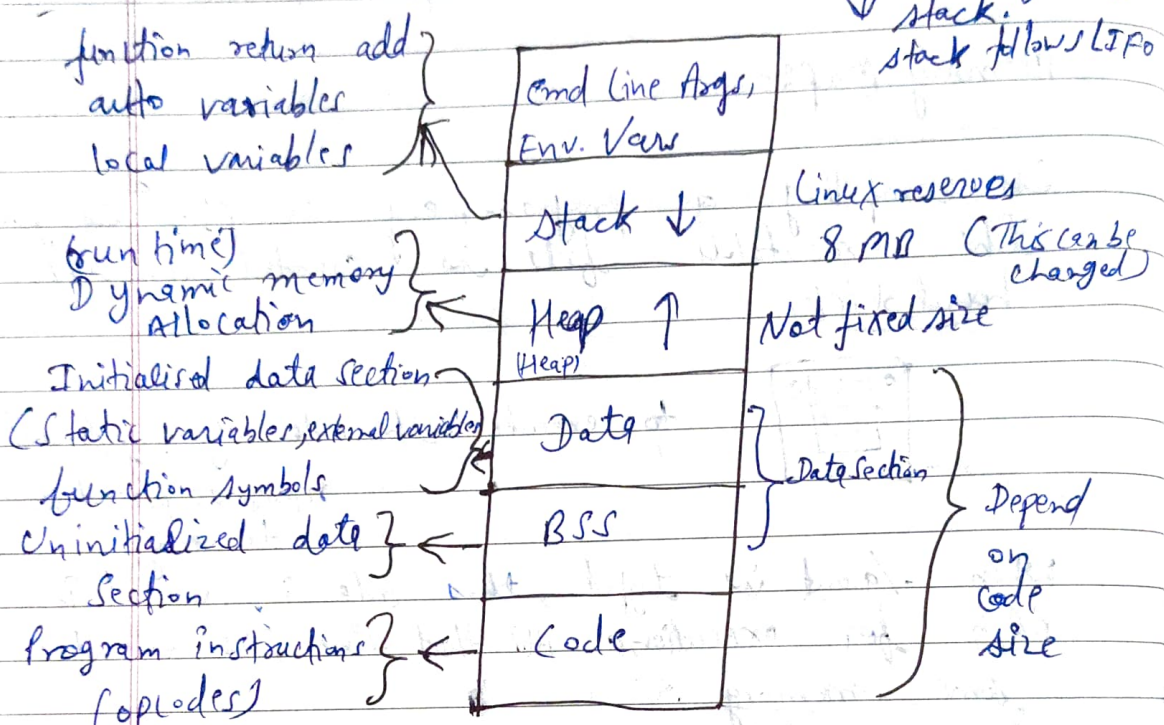


This memory is reserved until that program ~~file~~

gets completed. (i.e all instructions get executed)

→ In linux, stack section size is 8MB.

Memory section of one a.out file



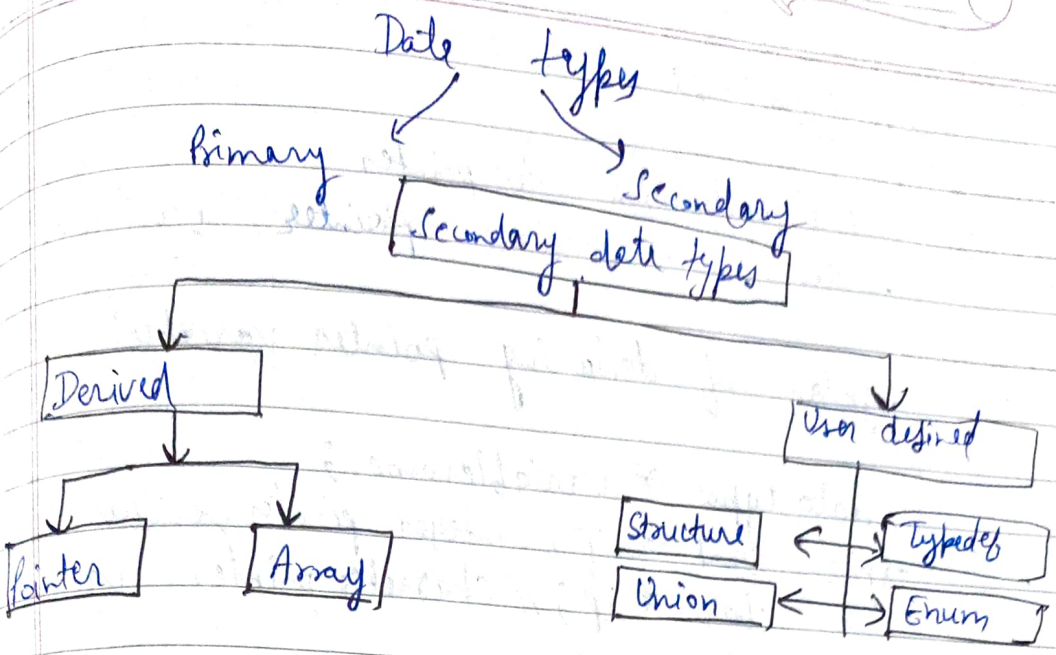
BSS :- (Block started by symbol)
Uninitialised data section.

Processors will fetch the instructions from the code section.

The above diagram, shows the logical memory blocks created for an a.out file.

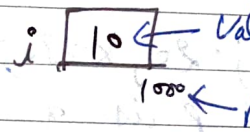
Secondary data types:

Primary:- int, short int, long int, char, double float, string etc.



Pointers :- (Derived Secondary data-types)

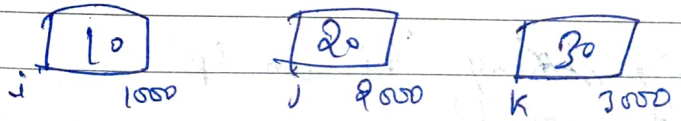
→ Pointer is a variable that represents the location, of a data item. Pointer stores the address of the variable through which the data can be access indirectly.



This address can be stored in pointer.

→ Any value stored in pointers, it is treated as an address.

eg: int i=10, j=20, k=30



→ Through i, j and k can't be accessed But there's a way for the above said access, using pointers.

eg:- p=&i → gives i's value
p=&j → gives j's value
p=&k → gives k's value

using the same ptr variable 'p' all the data can be accessed.

⇒ using pointers data can be accessed indirectly

Session - 33

→ Introduction to pointer
→ How to declare pointer variable

Syntax of declaring pointer variable:-

datatype *variablename ;

eg:- int *ip; // integer pointer variable
char *ip; // character pointer variables.

why we need to declare variable:-
To inform to the compiler what is what.

Wild pointers:- The pointers which are not initialised or uninitialised are called as wild pointer.

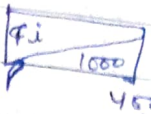
eg:- {
int i=10, j=20, k=30;
int *p; // p is a pointer variable

p → [ ] ← as it is not initialised, it holds garbage

Now, storing 'i' in 'p'

→ void main()
{
int i=10
int *p; // till here pointer is wild pointer
p= &i; // & is reference/starting address operator
printf("i=%d *p=%d\n", i, *p);
(*) for getoutput dereference operator use
p=&i; ← To store the address in pointer
*p ← to fetch the 'Data' from the addr
(dereference operator)

o/p :- i = 10 *p = 10



i = 10;
4i = 1000
p = (4i) = 1000
4p = 4000
*p = 10 // *p will fetch the value inside the address value it holds

→ To access the value stored in an address, a dereference operator is used.
Here p holds the address of i, *p will fetch the data present in address of i, which is the data stored in p.

[%p is used to print the address]

Example:
p = 4i;
printf("4i = %p p = %p\n", 4i, p);

→ eg. {
int i = 10;
int *p;
p = 4i
pf(" i = %d *p = %d\n", i, *p);
*p = 100 // Here the value of 'i' is indirectly changed.
pf(" i = %d *p = %d\n", i, *p);
}

→ Data could be read, written or modify using pointers;

→ { int i = 10; j = 20, k = 30;
int *p;
p = 4i;
*p = 100; // here pointer points to 'i'

p = &j; // Here pointer points to j
*p = 200; //
p = &k;
*p = 300; // pointer points to k
pf(" i = %d j = %d k = %d \n", i, j, k);
}

op = i = 100 j = 200 k = 300

→ {
int i = 10, j = 20
int *p;
int
p = &i;
*p = 100; // here value of i is changed
j = *p; // here the value of i (which is 100) is stored
pf(" i = %d j = %d \n", i, j); // into j
}
*p = j; // this is valid

pf(" i = %d j = %d \n", i, j);
o/p
i = 20 j = 20;

Size of pointer:-

{
char ch = 'a'; *cp;
int i = 10; *ip;
float f = 23.5, *fp;

* is present here to inform the compiler that it's a pointer variable

pf("%ld %ld \n", sizeof(ch), sizeof(cp));
pf("%ld %ld \n", sizeof(i), sizeof(ip));
pf("%ld %ld \n", sizeof(f), sizeof(fp));
}

o/p : 1 8 Same size (The addresses are same)
4 8 [In 64 bit OS, 8 bytes
8 8 In 32 " " 4 "]
Size of pointer depends upon OS.

Session 34

25-7-20

- Secondary datatypes (pointer)

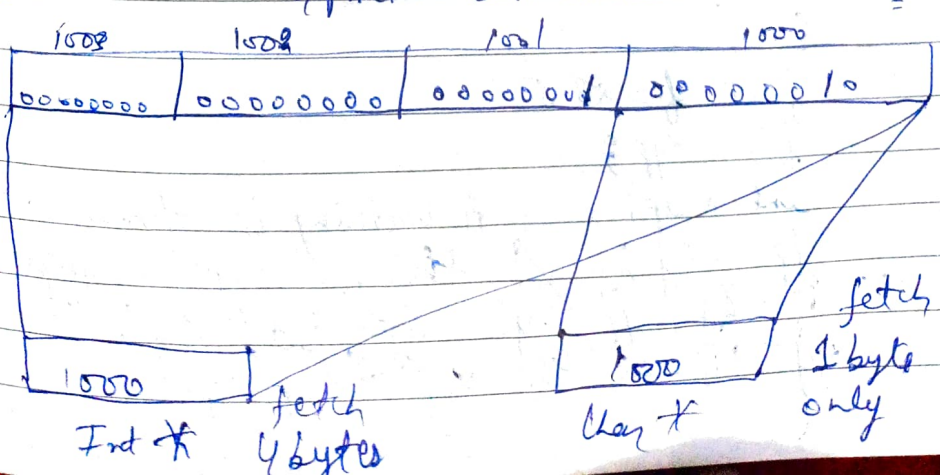
Difference b/w char * and int * (Pointer)

```
#include<stdio.h>
void main()
{
int i=258; // size is 4
int *ip; // size - 8
char *cp; // size 8
ip = &i; // address of i is assigned to the int *
cp= &i;                    and char *.
pf("%d %d",*ip, *cp);
}
```

'&' will give starting address

→ The compiler throwns a waning on the line 'cp=&i;' because the integer address is stored into char pointer. To avoid this warning, convert int to char explicitly
eg cp=(char *)&i;

int i=258 explicit conversion. Little endian

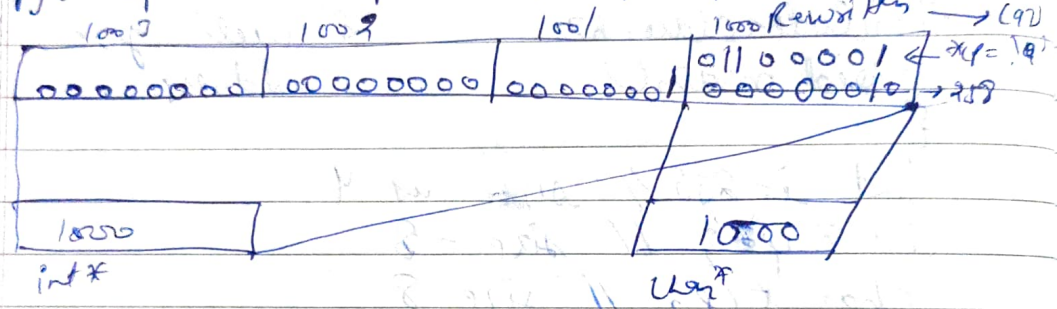


→ char pointer → when dereferenced, will fetch only 1 byte
→ int * → will fetch 4 bytes, when dereferenced.

o/p:- *i = 258 *cp = 2.

→ Modifying the pointer program a bit.

after cp = &i;
→ cp = 'a';
'a' is stored in ASCII value
ASCII value of 'a' is stored into 1 byte
printf("*ip = %d *cp = %d\n", *ip, *cp);



So after *cp = 'a';
*ip becomes (258 + 97) value of *ip = 353.
353

as the value of address 1000 is re-written by *cp = 'a' value

o/p
*ip = 353 *cp = 97

Difference b/w int * and float *

→ void main()
{
float f = 23.5
float *fp;
int *ip;
ip = &f; // A warning is thrown, To avoid this, do explicit typecast
fp = &f; // eg:- ip = (int *)&f;
printf("*ip = %d *fp = %f\n", *ip, *fp);
}

qf:-

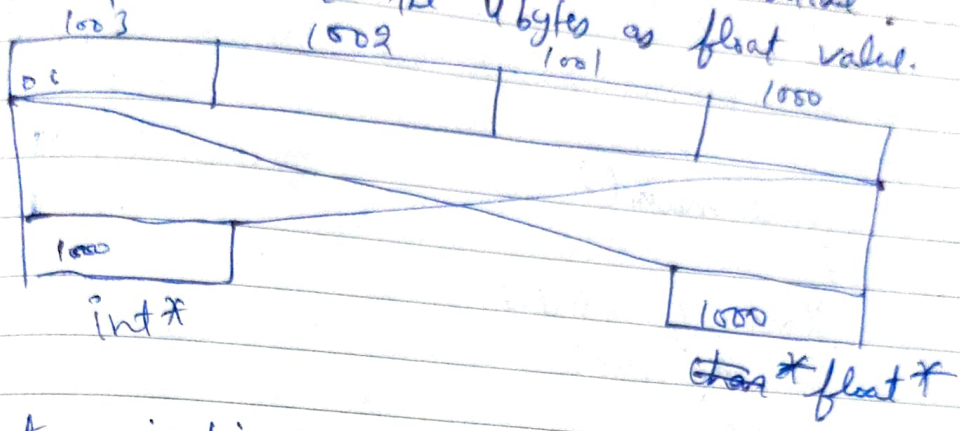
*ip = 1102839808

*ff = 23.500000

The binary value of both are same.

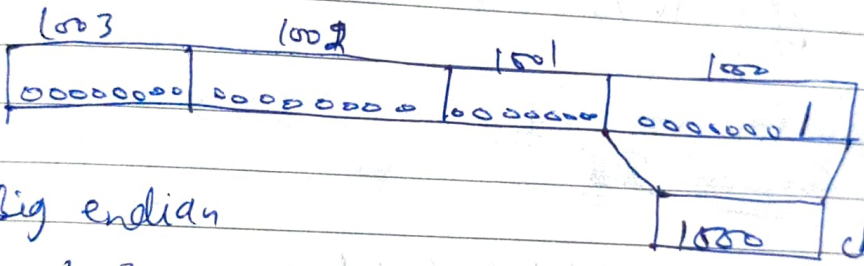
→ int pointer treat the 4 bytes as int value.

→ float pointer treat the 4 bytes as float value.

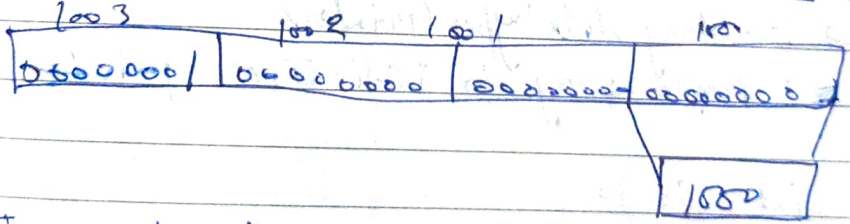


☆ Let i=1;

Little endian



Big endian



This char *, on a big endian gives up o/p = 0 as the address 1000 gives us the result '0'

char *, on little endian gives the o/p = '1'

→ ~~char~~ program :-

```
Void main ()
{
  int i = 1;
  char *cp;
```

```
    p = &i;
if (*p == 1)
    printf("Little \n");
else
printf("Big \n");
    }
```

#.

```
void main()
{
float f = 23.5;
int *p, pos;
p = &f;
for(pos=31; pos>=0; pos--)
    printf("%d", *p>>pos&1);
    printf("\n");
    }
```

Session 35

~~Session 35~~

27-7-20

→ Secondary datatypes (pointer)
→ Proper way to initialise pointer

```
int i = 10
int *p; // p is an integer pointer variable
p = &i; // the pointer here, holds garbage value, as
        it is not initialised with any value
        & thus, its a wild ptr.
        This is assigning a address to the
        pointer variable
```

# Syntax of initialise a pointer
datatypes * variablename = address;

```
main()
{
int i;
int *i;
```

These two can't be declared inside the same block with the same name. The compiler throws a syntactical error.

when we write `int *p = &i;` // the address of i stored into p

`int *p;` // garbage

✗ `*p = &i;` // This is an example of wrong dealing with the pointer

Here address of 'i' is stored in '*p' not in 'p'

But the approach → `int *p = &i` is correct.

```
int i=10; int *p;
p = i;   ✗ ← wrong dealing with pointer
```

* Address operator can't be written with a constant, the compiler thrown syntactical error.

Consider,

① `int i=10;` ✓

② `int *p;` ✓

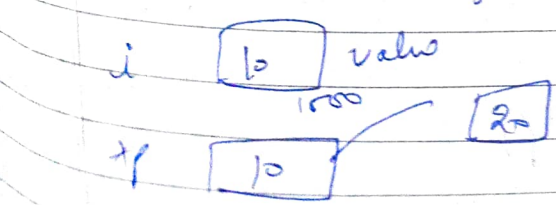
✗ ③ `*p = &i;` ✗ // pointer is dealing in wrong way

Here,

① `int i=10;` ✓

② `int *p = i;` ✗ (wrong) (4)

③ `*p = 20;` // ✓ (If warning is ignored, segmentation fault)



In RAM, lower addresses are allocated to OS

Initially when a pointer has garbage value, it is permitted by the compiler.
But when it is trying to be accessed with a pointer, it generates error/segmentation fault

→
```
{
int i=10;
int *p;
int *q;
p=q=&i;
p=q=&i;
*p=20;
pf("*p=%d *q=%d. i=%d", *p, *q, i);
```
will give the latest value

*p=20; → will go the address 'p' holds & changes the data there as '20'.

Consider,
```
*p=20;
*q=100;
pf("*p=%d %d %d\n", *p, *q, i);
```
100 100 100

pointer, the result is the modified data

Applying constant term to the pointers:-

① Const int *p; } *p is constant (i.e) read only
② int * const *p;

③ int * const p;
④ const int * const p;

① const int *p;

{
int i=10;
int *p;
p=&i; → No error
*p=20; → No error

Now,
int i=10
const int *p ← Read only pointer
p=&i;
*p=20; ← Error

Read only

The data can't be modified through the pointer because (p) has only READ permission on the data.

→ int i=10, j=20 k=30;
int *p;
p=&i;
*p=100 ← No error will occure but when pointer is declared as const like below

① const int *p; ② p=&i;
↳ *p=100; → Error

④ p=&j; ← This is allowed.
Here 'p' can be changed, but '*p' can't be 'p' can hold different address values but the address value can't be changed using the pointer.

② int const *p; This is similar to the ① type. In both the cases, p is a read only pointer.

e.g :- { int i=10; int const *p;
p=&i;
*p=20; ← Error
}

③ int * const p;

This is a constant pointer

→ {
int i = 10; → here the * points to a which it is trying to point to the diff location, compiler generate error
int * const p; //
p = &i; // error
*p = 20;
}

→ This pointer when initialised to one location, can't be pointed to other locations.

⇒ These type of pointers must be initialised. A wild pointer of this type is no use.

→ eg:- int * const p = &i;
*p = 20; ← This statement is permitted.

Here, the * once initialised can't be made to point to any other locations, but the data can be changed using this pointer.

④ const int * const p;

This should be initialised

Session 36 28-7-20

→ Secondary datatypes (pointer)
→ Inc and Dec pointer

Inc and Dec pointer :-

→ {
int i = 258;
int *p;
char *q;

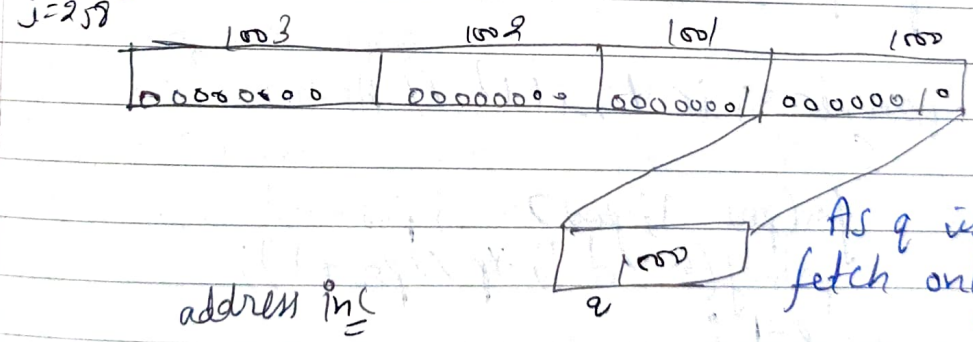
```
p = &i;
q = &i; // warning → int addr. assign to char *
printf("%p\n", &i);
printf("&p = %p\n p+1 = %p\n", p, p+1);
printf("&q = %p  q+1 = %p\n", q, q+1);   // %u can also be used
```

qp :- 0x ... (24
p = 0x2 ... (24 → p+1 = 0x2 ... (28 (4 byte inc)
q = 0x2 ... (24 → q+1 = 0x2 ... (25 (1 byte inc)

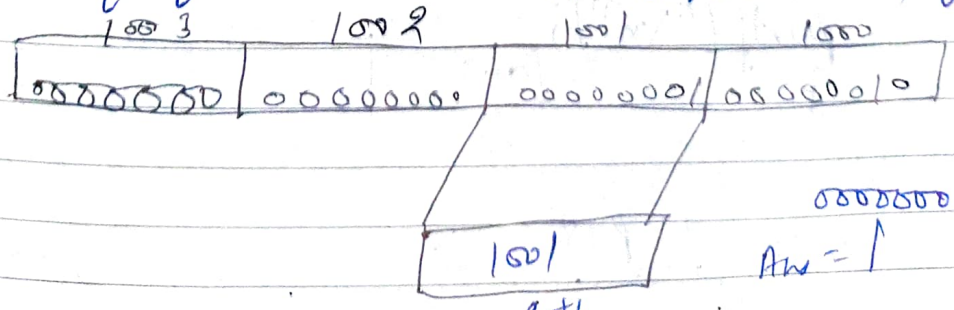
→ Integer pointer focus 4 bytes
→ char pointer focus 1 bytes

eg:-
```
{
int i = 258;
char *q;
q = &i;
printf("%d\n", *q);
q = q+1; // the address is incremented
printf("%d\n", *q);
}
```

o/p :-
2
1



When q = q+1 = 1000+1 = 1001 // addr incremented by 1 byte



Instead of q = q+1 → *q = *q+1 = 25 9 Ans
value inc

q=q+1; // address is incremented
*q = *q+1 // value is incremented.

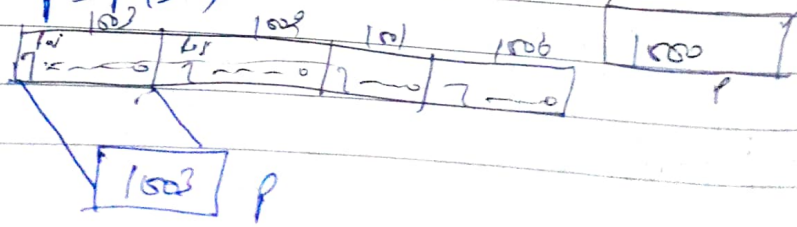
After this statement executes, the address still remains the same as 1500.

q+1; // q is not changed
q=q+1; // 4 bytes incremented
*q = *q+1; // value is changed.

write a program to display binary format of a float using character pointer.

→
```
#include<stdio.h>
void main()
{ float f=23.5;
  char *p;
  int i, pos;
  p= &f;
  p=p+3;
for(i=0; i<4; i++) // As there is 4 byte we rotate outer loop 4 times
  {
    for(pos=7; pos>=0; pos--) // 8 times 8 bits
     printf("%d", *p>>pos&1);
    p=p-1;
  }
  printf("\n");
}
```

After p=p+3;



p=p-1 ---- → moves until false.

Now, here
the outer loop is written as
starting address
for(p=4f; p=p+3; p7=4f; p=p-1)
{
}

→ if the char * is declared as char *f const p;
the p=p+3 // will throw an error

Example :-

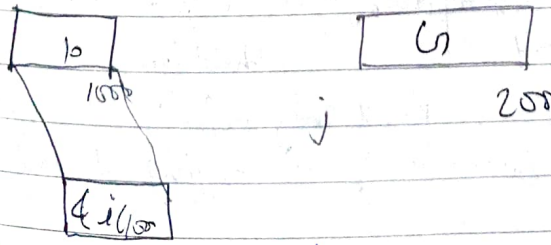
j=++*p;

{
int i=10, j;
int *p=4i;
pf("*p=%p\n", p);

j=++*p; // //
← Associativity - right to left)
pf("p=%p\n", p);

pf("i=%d j=%d\n", i, j); /// // //
}

p=.ox----bc8
p=6x---bc8
i=11 j=11



j=++*p; there are 2 operators (1 unary, 1 binary)
unary has the higher priority [Ass-R to L]
* ++

$j = ++*p$; (so dereference is first)

$(*p)$ is the value of (i)

$j = ++(*p)$; means ++ of 10 is 11

{ value of i assigned to j

→ eg:- $j = *p++$;

address

inc first p ↑

*, ++ same priority (R to L)

$j = *1000$ after that is becomes 1004 in case of int

eg:-

```
int i=10, j;
int *p, *q;
p=&i
```

suppose, &i = 1000, so p's value is 1000.

Session – 37 29-7-2020

→ Secondary datatypes (pointer)
- → Inc & Dec pointer
- → Void pointer
- → Null pointer

initial addr

j=10 1000

eg:- $q = p++$;

①	②	③	④	⑤	⑥
	$j = *p++$;	$j = ++*p$	$j = *++p$	$j = ++(*p)$	$j = (*p)++$
	R to L post inc.	Data inc (*p)	p is inc	It is	Here dp
	So j gets 10	p is still	p becomes	Similar	is Inc
	& pointer goes	1000, j = 11	1004, may	to ③	Post inc
	to 1004		hold Garbage		

const int *p; // Read only pointer.

(1) ++*p; ✗ (because *p is trying to inc.

(2) *++p; ✓

(3) *p++; ✓

(4) (*p)++; ✗ (because *p is trying to inc).

int * const p = &i; // Const pointer
(i) ++*p; ✓
(ii) *++p; ✗
(iii) *p++; ✗ [Address can't be modified const pointer]
(iv) ++(*p); ✓
(v) (*p)++; ✓

Note:- If const int * const p = &i; all are error.

Void pointer or generic pointer:-

{
char ch = 'a';
int i = 10;
float f = 23.5;
void *p; // Point any datatype.
p = &ch; // Typecasting not req. when storing address.
printf("%c\n", *(char *)p); // Type cast necessary here when dereferencing the ptr.
p = &i;
printf("%d\n", *(int *)p); // After; once statement is over it will changes into its original behaviour i.e void
p = &f;
printf("%f\n", *(float *)p);
}

(int *)*p; ✗ wrong way of typecast.

Null pointer:- (Null refers to zero)

If a ptr is holding '0' - zero as an address, that ptr is called as null ptr.

{
int i = 10;
int *p = 0; // This is a null pointer

∴ To avoid wild pointer make it zero / null

```
pf("%d",*q);
}
```

* Note:- when the work is finished with pointer, don't leave the pointer as it is. Make the pointer as Null pointer. To avoid weild / dangling pointer we use null pointer.

Null pointer means resulting to zero

Arithmatic operations on the pointer:-

p+q;	✓	Value are operated	p+q;	✗
p-q;	✓		q-p / p-q;	✓
*p*q*;	✓		p*q;	✗
p/q;	✓		p/q;	✗

Address are operated

When we subtract two pointer, we will get how many such type of data we can store in b/w those addresses.

→
```
{
int *p=(int*)1016;
int *q=(int*)1000;
pf("%d\n",p-q);
}
```
o/p:- 4 (1016 byte - 1000 byte gives result in bytes format)

4 int data type can be store in b/w p and q.

→
```
{
char *p=(char*)1016;
char *q=(char*)1000;
pf("%ld\n",p-q);
}
```
o/p = 16

16 bytes can be store in b/w p and q.

Session → 38

Secondary data types (Array)

→ int array
→ array declaration

Array :- Array is a collection of similar types of data. or similar type of data can be put together in Array.
In an array, similar elements can be put-together, so that the processing becomes easy.

Syntax for array declaration :-

datatype variablename [elements];

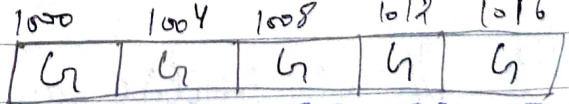
Index operator

Ex :-
int a[5]; → a is an array of 5 int elements
char ch[10]; → ch is an array of 10 char elements
float f[5]; → f is an array of 5 float elements

→ If we want to access the elements of an array, we need to use the index operator [].

→ Array index starts with zero.

int a[5]



Declaration :-
→ When an array is declared, it is filled with garbage values.
But we can initialise an array to avoid this. Eg :- int a[5] = {10, 20, 30, 40, 50}