Advantages →

(1) It increases the execution speed of C++ program.
(2) It reduces the length & complexity of program.
(3) These are more efficient in handling the data table i.e two dimensional arrays.
(4) Pointer have direct link with structure & union.
(5) It saves the memory space. ie by using dynamic memory space.
(6) These are very useful to handles files.
(7) Pointer are very useful in data structure manipulation.

declaration →

```
int *p, q;        → dereferencing operator
p = &q;           → "address of" operator / reference
cout << p;
cout << *p;
```

Notes →

(i) &(x+y)
(ii) int &x(10)
(iii) *p, p=1000;
(iv) [illegible];
(v) int y = &x;

⇒ invalid

pointer in different ways

(i) &*p
(ii) *&*p
(iii) *&q

Pointer expression

```
int *p1, *p2;
int x, y, c;
p1 = &x;
p2 = &y;
c = (*p1) + (*p2);
c = (*p1) * (*p2);
c = (*p1) / (*p2)   //   *p1 / *p2;   → invalid
```

Note ⇒ (i) address can be incremented or decremented by using pointer.

i.e

```
p1++;
p1--;
p1 = p1 + 1;
```

(ii) to increase the value in pointer variable i.e

```
*p1 = *p1 + 10;
*p1 = *p1 * 2;
```

Scale factor → `p1++;` `p2--;` It is measurment of length of cell address of different data type :ie

Data type	
char	1 byte
int	2 byte
float	4 byte

Foreg

```
int x[] = {10, 20, 30};
int *p;
p = &x;
p = p + 1;
p = p + 1 * 2 = p + 2
= 1000 p2 = 1002
```

x [10] 1000

p [1000] 2000

Pointer to void ⇒

①
```
int x;
int *p;
p=&x
```

②
```
float x;
float *p;
p=&x;
```

③
```
float
int x;
int float *p;
p=&x;
```
invalid ⇒ address of a float variable can't be assigned to a pointer to the int

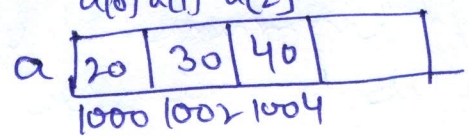
```
void main()
{
int a[] = {20,30,40};
void *p;
p=&a;
for(int i=0; i<=2; i++)
  {
  cout<< *(p++);
  }
```

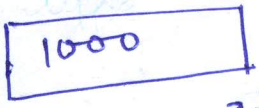
```
void *p;
int x; float y;
p=&x;
p=&y;
```

void pointer is called generic pointer because it can point to variable of any data type.

Pointer and Array → pointer to an array.

① Pointer to an array → One dimensional array





*(p=1000)

i=0 → 20

i=1 → 30 → *(p++) ⇒ *(p=p+1) ← scalar factor

i=2 → 40

*(p=1000+1X2)

*(p=1002)

*(p++) = *(p=1002+1*2)

*(p=1004)

Note ⇒

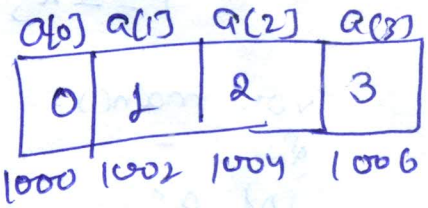
(a) Base addres of Array can't be changed.

```
int a[4] = { 0, 1, 2, 3 };
a = a + 8;   // error
```

Array name indicate the Base address.

(b) `int *p = &a[0];`

`p = p + 2;` // ✓



⇓

$= 1000 + 2 \times 2 \Rightarrow 1004$

⇓

$a[2] = 2$ & $*(p+2) = 2$

(c)

Now $\boxed{a[i] = *(a+i)}$ for 1-D Array

$a[2] = *(a+2) = *(a+2\times2) = *(a+4)$

$*(a+0) \Rightarrow *(a) \Rightarrow *(1000)$ skip 0th zero element

$*(a+1) \Rightarrow *(a+1) \Rightarrow *(1002)$ skip one elem

$*(a+2) \Rightarrow *(1004) \Rightarrow$ skip two elem

(11) Pointer with Two dimensional Array

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

	a[0]	a[1]	a[2]
a[0]	1	2	3
a[1]	4	5	6

a[0][2] ⇒ 3

a[1][1] ⇒ 5

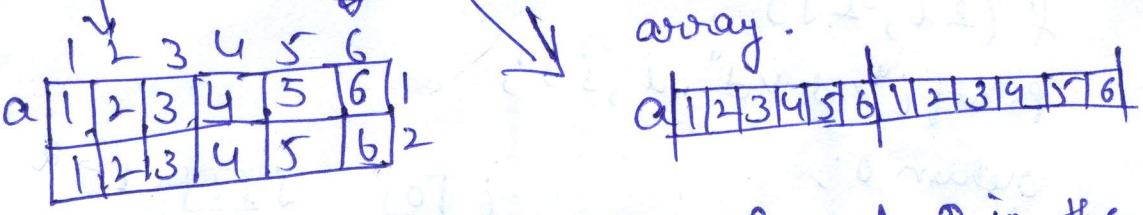
2-D Array can be represented in two;

(1) Pointer to a group of continuous one-dimension Array

(2) Array of pointers

(1) `int a[2][6];`

`int (*a)[6]; // 2d, a is a pointer to group of one dimension 6 elements of integer array.`



⇒ *a is pointer to the element 0 in the 0th row

⇒ *(a+1) is 0 in the 1th row

⇒ a is pointer to the row 0 of 6 element array.

⇒ a+1 is pointer to the row 1 of 6 element array.

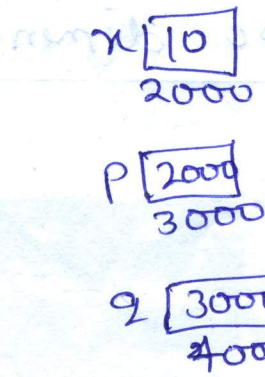
(2) Array of pointer → every element can hold addres of any variable & every element of this array is a pointer variable. It contains the collection of addres.

`int *a[2][5];`

`int *a[5];` is an array of three pointer

Pointer to pointer

```
int x=10;
int *p=&x;
int **q=&p;
```



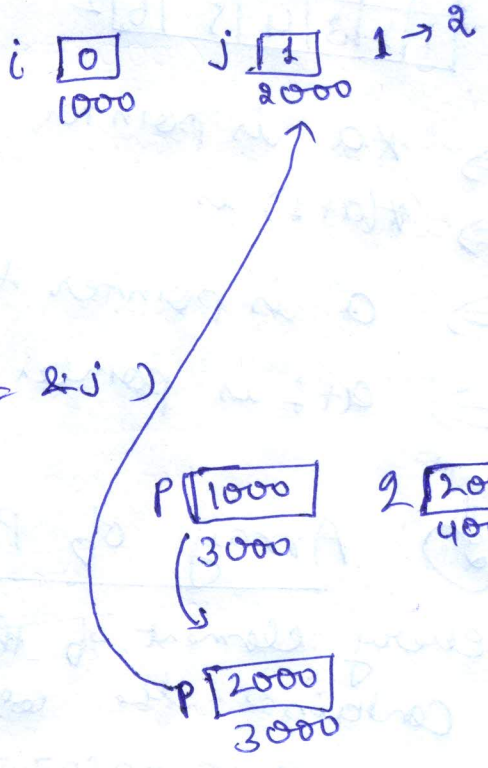
immediate addr Mode

Direct Addr Mode
One Memory

Indirect Addr Mode
Two Memory Access

for eg →

```
void f(int *p, int *q)
{
  p = q;
  *p = 2;
}
int i=0, j=1;        // global variable
int main()
{
  f(&i, &j);
  printf("%d %d", i, j);
  return 0;
}
```



Ans i = 0, j = 2;

→ f(1000, 2000)

→ *p = &i, *q = &j)

→ p = 2000;

→ *p = 2;

⇓

*(2000) = 2;

a[0] points to 1st element in 0th row

a[1] ——— 1st elem in 1st row

(a[2]+3) → 3rd elem in 2nd row

*(a[2]+3) → point to value of 3rd elem in 2nd row

↓

((a+2)+3)

Pointer and function →

(1) Pointer to function →

```
int mul(int, int);
int (*p)(int, int);        → pointer to function
p = &mul;
```

eg multiply two No by using pointer:

```
void main()
{
  int mul(int, int);      // prototype
  int (*p1)(int, int);    → pointer to function
  int x, y, z;
  cin>>x>>y;
  z = (*p1)(x,y);         // calling fn
  cout<<z;
  getch();
}
int mul(int a, int b)     // called fn
{
  int m;
  m = a*b;
  return(m);
}
```

(11) pointer as function argument! (call by reference)

```
int mul (int *p1, int *p2)
          ↓↓
     argument as pointer
```

Pointer and string →

```
char name[] = "CSE";
for(int i=0; name[i] != '\0'; i++)
  { cout << nam[i];
  }
```

(1) pointers with strings perform many string operation like → to find string length
→ to compare two string
→ to copy one string to another
→ to concatenate the string

```
char *name = {"CSE"};   // pointer to char
char *name[] = {"CSE", "ECE", "ME"};
```

↳ array of pointer to char

name[0] = | C | S | E | \0 |

name[1] = | E | C | E | \0 |

```
while(name != '\0')
 {
   cout << *name;
   name++;
 }
```

```
Char name[] = "CSE";
Char *name1 = "CSE";
name = name+1   X
name1 = name1+1
```

array address can't be change

pointer can be manipulate

Que: to copy string into another string

Dynamic memory allocation or Dynamic Structure

int a[20] ⇒ 40 Byte

(1) New operator → allocate memory at run time.

(a) pt_var = new data_type

↓

type of data for which you want to allocate memory dynamically

(b)
```
int *ptr;
ptr = new int;
```

(c)
```
float *p;
p = new float;
```

(d)
```
struct Student
{
int roll;
char nam[a];
};
student *stu;
stu = new student;
```

(e)
```
int *ptr;
ptr = new int[5];
```

② Delete → delete ptr_var;

delete [] ptr_var;

ⓐ
```
int *p;
p = new int;
delete p;
```

ⓑ
```
char *ptr;
ptr = new char[20];
delete [] ptr;
delete ptr;    ///invalid
```

Pointer (Practice)

①
```
int x = 10;
int *p = &x;
*p = 20;
cout << *p;
cout << *&x;
```
O/P ⇒ 20 20

④
```
int x = 10;
int *p;
*p = 20;
p = &x;
cout << *p;
```
O/P ⇒ 10

③
```
int x = 10;
int *p = 30;
p = &x;
cout << *p;
```
gives error
int to int *

②
```
int x = 10;
int *p = &x;
p = 20;
cout << *p;
```
O/P ⇒ error
int to int *

⑤
```
int x = 10;
int *p = &x;
*p = 20;
p = &x;
cout << *p;
```
O/P ⇒ 20

⑥
```
int x = 2;
int *x;
```

⑦
```
int *p;
cin >> p;
```
can never be read through keyboard

*p = 30;

int *p

p | NULL

Char P[] = {"Computer Science"};
cout << P;

↳ String Constant

O/P ⇒ Computer Science

② Char P[20];
cin >> P; Computer Science // IP
Cout << P; Computer // O/P

③ Char P[20];
gets(P); computer science // IP
Cout << P; computer Science //op

↳ Stdio.h //header file

Practice (Pointer)

(1) char *p;

*p = 100; ⇒ *p = 'd';

cout << *p;

O/P ⇒ d character

---

(2) char *p;

cout << *p;

O/P ⇒ point nothing

---

(3) ~~char *p;~~

int x = 10;

int *p;

p = 30;

O/P ⇒ error

int x = 10;

int *p;

*p = 30;

cout << *p;

O/P ⇒ 30

---

(4) int *p;

*p = 30;

cout << *p;

O/P ⇒ 30

int * p = 30;

cout << *p;

O/P ⇒ error

---

(5) void *ptr;

int x;

ptr = &x;

compile ⇒ error (need to typecast)

void
Null → Pointer
wild
dangling

Wild pointer → Pointer that are not specifically initialized may point to unpredictable address in memory.

```
char *p2;
*p2 = 'b';
```

p2 may ~~here~~ point to anywhere in memory, so performing the assigment *p2='b' will corrupt an unknown area of memory that may contain sensitive data.