Pointers

class-1

Date:
Page:

A Pointer is a variable that stores the memory address of other variable as its value.

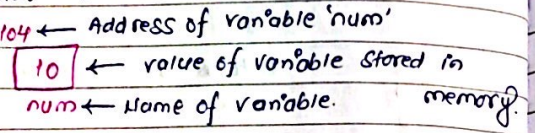
Pointer only store the address. Any value other than address can't be stored in pointer.

A Pointer variable Points to a data type (like int, char, double) of some type. & is created with the '*' operator.

Q What happened in behind, when we write int num = 10;

A A 4 bytes space is allocate to 'num' in memory, where value '10' is store. (int size is 4 byte so 4 byte memory space allocate)

We can access that memory location where '10' is stored is using only address of that location. In memory to access location only single way is address of that location.



Address of 'num' is 104

Address-of operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator.

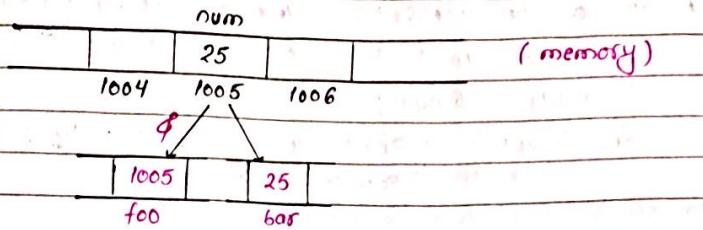
Eg:- int num = 10;
int add = #

This would assign the address of variable 'num' to 'add'. We're no longer assigning the content of the variable itself to 'add', but its address.

Date:
Page:

int num = 25;
int foo = #
int bar = num;

The values contained in each variable after the execution of this are shown in the following diagram.



→ first, we've assigned the value 25 to 'num' (a variable whose address in memory assumed to be '1005').

→ The 2nd statement assign 'foo' the address of 'num', which we've assumed to be '1005'.

Creation of Pointer

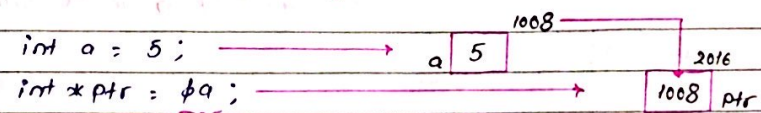
The Pointer variable is created by using '*' operator ('*' = Dereference operator)

Syntax:

Data-type * variable name;
(Pointer name)

Eg:- int * ptr; → Pointer name.
↳ Pointer to integer data.

} Here, we can say 'ptr' is a Pointer to integer data.



↳ memory address of variable 'a'.

- char * ch; → ch is a pointer to character data.
- bool * flag; → flag " " " boolean " .

'&' = address-of operator

'*' = dereference operator

Access Pointer

Accessing value stored at address stored in pointer variable. That value can be accessed by dereference operator (*).

Eg:- int num = 10;
int * ptr = #
cout << *ptr; // print value stored at address stored in pointer variable.

o/p => 10.

Code

```
int main () {
    int a = 10;
    cout << a << endl;  // value of a.    o/p => 10
    cout << &a << endl; // address of a.  o/p => 0x- - - -
    // create a pointer
     int * ptr = &a ;
    cout << ptr << endl; // Address of a.   o/p => 0x- - - -
    cout << &ptr << endl; //    "   "  ptr.  o/p => 0x- - - -

    // Accessing value stored at address stored in pointer variable
    cout << *ptr << endl; // print value of a, because 'ptr' stored
                             the address of 'a' which value is '10'

    return 0;
}
```

visualization of code :

int a = 10; int *ptr = &a;

1008 ← address of 'a' | 2016 ← address of pointer variable

10	a = 10	&a = 1008
a		

1008	ptr = 1008	&ptr = 2016
ptr		

cout << *ptr << endl; // o/p => 10

i) Go to the address stored in pointer variable 'ptr'. (1008)

ii) Access the value stored at address store in ptr. (10)

Difference betn reference variable & pointer.

Reference variable is the different name given to same memory location. It doesn't take extra space in memory, while pointer is a variable that stores the address of another variable & it takes additional space in memory.

Size of Pointer

i)
```
int a = 5;
int *ptr = &a;
sizeof (ptr) ; // 8 byte
```

ii)
```
char ch = 'S';
char * ptr = &ch;
sizeof (ptr) ; // 8 byte
```

iii)
```
long var = 15;
long *ptr = &var;
sizeof (ptr) ; // 8 byte
```

Q. Why pointer size is 8 byte ?

A. The pointer size is typically 8 bytes (64 bits) on modern 64-bits architectures because it matches the size of memory addresses used to access the large memory space available in these systems. This allows Pointers to efficiently reference memory locations within the entire addressable memory range.

Declaration of pointer

Bad Practice

```
int * ptr;
cout << *ptr << endl;
```

Declaring pointer like this is a bad practice as it is trying to access a random memory location which is not allowed. Generally it gives run-time error or segmentation fault.

Good Practice

```
int * ptr = 0;
OR, int * ptr = nullptr;
```

This is a good practice because it explicitly sets the pointer to a known state of pointing to nothing. It promotes code safety, readability & maintainability.

Some Practice questions of Pointer.

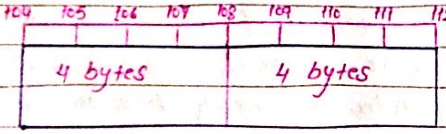
Guess the output of following question

1)
```
int a = 100;        ------>  a   | 100 |   104 ← address of 'a'
int *ptr = &a;      ------>  ptr | 104 |   200 ← address of 'ptr'
a = a+1;            ------>  a   | 101 |   104
ptr = ptr+1;        ------>  ptr | 108 |   200
```

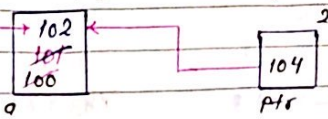
o/p ⇒ a = 101
ptr = 108

ptr = ptr + 1
= 104 + 1 = 108 (Address of next memory location).
= 104 + 1 × 4 ⇒ 108
↳ size of integer data type

(104+1)



2)
```
int a = 100;
int *ptr = &a;
a = a+1;      // 101
*ptr = *ptr+1;
```

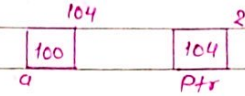


o/p ⇒ a = 101
*ptr = 102

*ptr = *ptr + 1;
value present at address stored in ptr.
⇒ *ptr = 101 + 1
= 102.

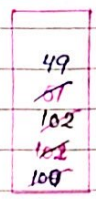
first value stored at '104' address in 'a' then a = a+1 it becomes '101' & when '*ptr' access that value & increment by (*ptr+1) it become 102. We can access the value of 'a' by '*ptr'

3)
```
int a = 100;
int *ptr = &a;
```



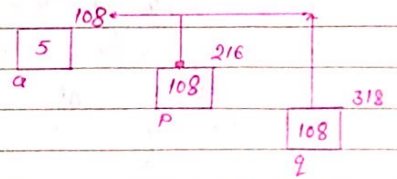
Find the value of following basis of above two lines.

i) a → 100
ii) &a → 104
iii) *a → Error
iv) ptr → 104
v) *ptr → 100
vi) &ptr → 208
vii) (*ptr)++ → *ptr = *ptr + 1 = 100 + 1 = 101
viii) ++(*ptr) → *ptr = *ptr + 1 = 101 + 1 = 102
ix) *ptr = *ptr/2 → *ptr = 102/2 = 51
x) *ptr = *ptr - 2 → *ptr = 51 - 2 = 49.



*a ⇒ Error, a is a integer variable, & we can't dereference integer variable.
That's why '*a' throw an error.

4)
```
int a = 5;
int *p = &a;
int *q = p; (Pointer copy)
```



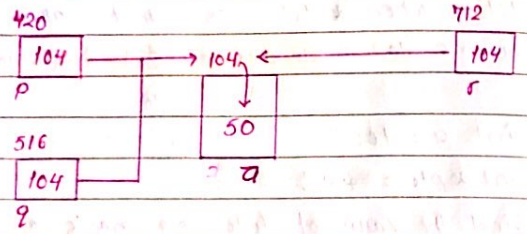
i) a = 5

iii) &a → 108 vi) *p → 5 (value present at address stored in p)
iiii) *a → Error vii) q → 108
iv) p → 108 viii) &q → 318
v) &p → 216 ix) *q → 5 (value present at address stored in q)

int *q = *p; → Error

*p = 5, which is an integer value & we're trying to store integer value in pointer variable. Pointer variable store only address of a variable not a value of that variable.

5) int a = 50;
int *p = &a;
int *q = p;
int **r = q;



i) a → 50 vi) *p → 50
ii) &a → 104 viii) q → 104
iii) *a → Error viii) &q → 516
iv) p → 104 ix) r → 104
v) &p → 420 x) *r → 50
xi) &r → 712 xii) *q → 50

*p = *p/2
*q = *q/2
*r = *r/2
} a = a/2

*p = *q/2
*r = *p/2
*q = *r/2
} a = a/2

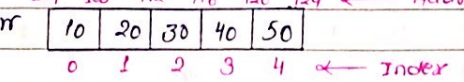
We can access 'a' by : *p, *q & **r

Pointer size depends on size of computer system RAM.
4 GB RAM ⇒ Pointer size is 4 byte
8 GB RAM ⇒ " " " 8 byte
16 " " ⇒ " " " 8 "

```
int main () {
    int a = 10;
    int *ptr = &a;          // Store address of 'a'
    cout << ptr << endl;    // Print address of 'a'
    cout << *ptr << endl;   //   "     value of 'a' => 10
    int b = 20;
    ptr = &b;               // Store address of 'b'
    cout << ptr << endl;    // Print   "    "  'b'
    cout << *ptr << endl;   //   "     value "  'b' => 20
    b = 30;
    cout << *ptr << endl;   // Print value of 'b' => 30
}
```

Pointer With Array.

int arr[5] = {10, 20, 30, 40, 50};



i) arr → Represent the base address of array (starting address of array). When we print 'arr' it gives '104' as a output.

ii) arr[0] → Print value present at 0th index → 10

iii) &arr[0] → Print address of '0th' index / base address of arr → 104

iv) &arr → Base address of arr → 104.

In the case of array → integer (Imp++)

i) arr
ii) &arr
iii) &arr[0]
} All statement indicates the Base address of array.

arr = Name of array.

500	504	508	512	516	520 ← Address
11	7	8	12	14	
0	1	2	3	4 ← Index	

arr :

int *ptr = &arr[0] → 500 (Base address)
int *ptr1 = &arr[1] → 504 ('1st' index address)

arr → 500 arr+3 → 512
arr+1 → 504 arr+4 → 516
arr+2 → 508

Q: If we print 'arr+2', How computer generate the value of 'arr+2'?
If we want to know address of 'ith' index, computer work as follow:

ith index address = Base address + i x datatype_size

100	104	108	112	116	120 ← Address
20	25	30	45	90	
0	1	2	3	4 ← Index	

Eg:- arr :

cout << arr+2 << endl;

Procedure :

Base address + i x datatype_size
⇒ 100 + 2 x 4
⇒ 108

Integer size = 4 byte
Base address : 104
i = 2

∴ (arr+2) → 108 (Address of 2nd index)
∴ *(arr+2) → 30 (value present at 2nd index).

Q: Guess the output of the following:-
int arr[5] = { 10, 20, 30, 40, 50 };

104	108	112	116	120	124
10	20	30	40	50	
0	1	2	3	4	

arr :

i) arr → 104 (Base address of array)
ii) &arr → 104 (" " " ")
iii) &arr[0] → 104 (0th index address / Base address)
iv) arr[0] → 10 (value at 0th index)
v) *arr → 10 (value at base address → (*104) → 10)
vi) *arr+1 → 11 (*arr = 10 ; *arr+1 = 10+1 ⇒ 11)
vii) *(arr+1) → 20 (*(arr+1) ⇒ *(104+1x4) ⇒ *(108) ⇒ 20)
viii) *(arr)+1 → 11 (*(arr) = 10 ⇒ *(arr)+1 ⇒ 10+1 ⇒ 11)
ix) *(arr+2) → 30 (*(arr+2) ⇒ *(104+2x4) ⇒ *(112) ⇒ 30)
x) *(arr+3) → 40 (*(104+3x4) ⇒ *(116) ⇒ 40)

104	108	112	116	120	124
10	20	30	40	50	
0	1	2	3	4	

arr :

*arr ←OR→ *arr+0 ←OR→ *(arr+0) → 10
They all give the same output (value of base address).

arr[0] equal to *(arr+0)
arr[1] equal to *(arr+1)
arr[2] equal to *(arr+2)
arr[i] equal to *(arr+i)
OR i[arr] equal to *(i+arr)
(Imp++)

arr[i] → *(arr+i) → *(Base address + i) } Behind the scene

500	504	508	512	516	520
10	20	30	40	50	

arr = &arr[0]
= &*(arr+0)
= &10
arr = 500
500 = 500
∴ LHS = RHS

arr → 500
&arr[0] → 500
&arr → 500

```
int main () {
    int arr [5] = {1, 2, 3, 4, 5};
    int *ptr = arr;
    // Print the address of first element or 0th index
     cout << arr << endl;
     cout << arr+0 << endl;
     cout << &arr[0] << endl;
     cout << ptr << endl;
    // Print the address of second element or 1st index
     cout << arr+1 << endl;
     cout << &arr[1] << endl;
    // Print the value of 0th index
     cout << arr[0] << endl;
     cout << *arr << endl;
     cout << *(arr+0) << endl;
     cout << *ptr << endl;
    // Print address of all index
     for (int i=0; i<5; i++) {
         cout << arr+i << endl; OR. cout << ptr+i << endl;
     }
```

```
    // Print all the value
     for (int i=0; i<5; i++) {
         cout << *(arr+i) << endl; OR. cout << *(ptr+i) << endl;
     }                             OR cout << ptr[i] << endl;
}
```

Pointer Arithmetics.

i) Increment of a pointer (++)

```
int arr[5] = {2, 4, 6, 8, 10};
int *ptr = arr;
```

500	504	508	512	516	520
2	4	6	8	10	
0 (ptr)	1	2	3	4	

```
// Print all value of array using increment operator
for (int i=0; i<5; i++) {
    cout << *ptr << endl;
    ptr++;
}
```

500 (ptr)	504 (ptr)	508 (ptr)	512 (ptr)	516 (ptr)	520 (ptr)
2	4	6	8	10	

i=0 → i=1 → i=2 → i=3 → i=4 → i=5

int *ptr = arr; → ptr = 500 & *ptr = 2;

i) i = 0;
cout << *ptr; → (2)
ptr++; ptr = ptr+1
ptr = 500+1x4 = 504

ii) i = 1.
cout << *ptr; → (4)
ptr++; 504+1x4 = 508

iii) i = 2
cout << *ptr; → (6)
ptr++; 508+1x4 = 512

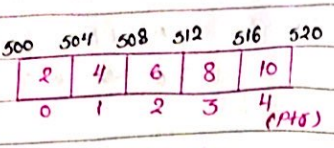
iv) i = 3;
cout << *ptr; → (8)
ptr++; 512+1x4 = 516

v) i = 4;
cout << *ptr; → (10)
ptr++; 516+1x4 = 520

vi) i = 5;
out of bound. (STOP)

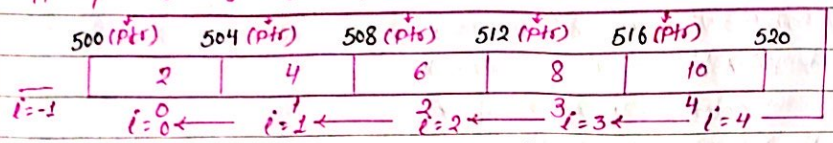
ii) Decrement of a Pointer (--).

```
int arr[5] = { 2, 4, 6, 8, 10 };
int *ptr = arr+4;
```




```
// print all value of array using decrement operator
for (int i = 4; i >= 0; i--) {
    cout << *ptr << "  ";
    ptr--;
}
```

O/P → 10 8 6 4 2



int *ptr = arr+4; ptr = 516 & *ptr = 10;

i) i = 4;
cout << *ptr; → (10)
ptr--; ptr = ptr-1
ptr = 516-1x4
ptr = 512

ii) i = 3;
cout << *ptr; → (8)
ptr--; 512-1x4 = 508

iii) i = 2;
cout << *ptr; → (6)
ptr--; 508-1x4 = 504

iv) i = 1;
cout << *ptr; → (4)
ptr--; 504-1x4 = 500

v) i = 0;
cout << *ptr; → (2)
ptr--; 500-1x4 = 496

vi) i = -1
out of bound (STOP)

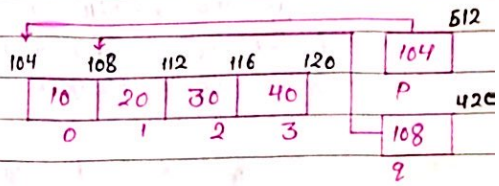
```
int arr[5] = { 1, 2, 3, 4, 5 };
int *ptr = arr;
ptr++;  → This is a valid.
arr++;  → "  "  "  invalid.   } Why ?
```

Q) Why we can't perform arithmetic operation on array pointer?

A) In C++ & C, the name of an array like 'arr' represents the memory address of its first element. We can't directly perform arithmetic operation on array, because it is a constant pointer that points to the first element of the array, & its value can't be changed. If you try to perform arithmetic operation on array you get an error.

Q) Guess the output

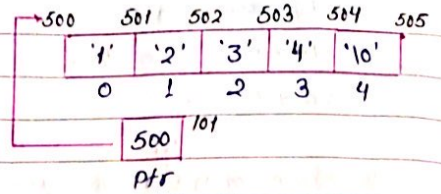
```
int arr[4] = { 10, 20, 30, 40 };
int *p = arr;
int *q = arr+1;
```



i) arr → 104 (Base address)
ii) & arr → 104 (" ")
iii) arr[0] → 10 (value at 0th index)
iv) & arr[0] → 104 (address of 0th ")
v) p → 104 (value of p) → Base address
vi) &p → 512 (address of p)
vii) *p → 10 (p = 104, *p = *104 => 10) → value of 0th index
viii) q → 108 (address of 1st index)
ix) &q → 420 (" " q)
x) *q → 20 (q = 108, *q = *108 => 20) → value of '1st' index
xi) *p+1 → 11 (*p = 10, *p+1 = 10+1 => 11)
xii) *(p)+2 → 12 (*p = 10, *(p)+2 = 10+2 => 12)
xiii) *(q)+2 → 22 (*q = 20, *(q)+2 = 20+2 => 22)

Pointer with char array

```
char ch[5] = "1234";
char *ptr = ch;
cout << ch;   // o/p => 1234
cout << ptr;  // o/p => 1234
```

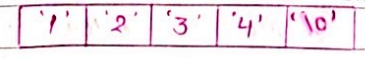


NOTE:- When we make a pointer for char-type data & tries to print that pointer, it will print the array character / element until it found the null character (\0), rather than its address like in integer array.

character array in c/c++ are often treated as null-terminated strings. When we dereference a char pointer pointing to the start of such an array, the compiler assumes we're interested in the string data itself, not the memory address of the array. Therefore, it returns the value stored at that memory location.

If you want to print address of array instead of value in char array, then you have to use void pointer (void*).

```
char arr[5] = "1234";
char *ptr = arr;
cout << (void*) arr;  // o/p => 500
cout << (void*) ptr;  // op => 500
```




```
char ch = 'a';
char *ptr = &ch;
cout << ptr;  // o/p => a@?@. (Random character with 'a')
```

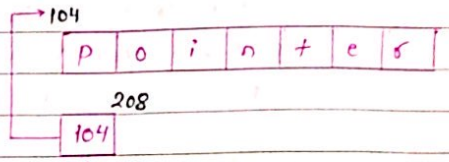
It print some random character with 'a' until it found a null character ('\0').


```
char ch = 'a';
char *ptr = &ch;
cout << (void*) ptr;
cout << (void*) &ch;
```

} In this time both statement point the address of 'ch' variable

Q. Guess the output

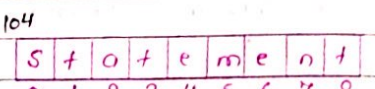
```
char ch[10] = "Pointer";
char * ptr = ch;
```

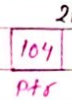


i) ch → Pointer
ii) &ch → 104
iii) ch[0] → P
iv) &ptr → 208
v) *ptr → P (*(ptr+0) => ptr[0] => P)
vi) ptr → Pointer.

Q.

```
char ch[50] = "Statement";
char * ptr = &ch[0];
```





i) ch → Statement
ii) &ch → 104
iii) *(ch+3) → t
iv) ptr → Statement
v) &ptr → 216
vi) ptr+2 → atement
vii) *ptr → S (ptr[0] => *ptr => S)
viii) ptr+8 → t
ix) *(ptr+8) → t (ptr[3] => *(ptr+3))


```
char *ptr = "Pointer";  → Bad Practice
```

It is a bad practice, because it creates a pointer that points to a string literal, which is typically stored in 'RAM'. Which can be delete or remove any time from 'RAM', & can lead to undefined behaviour, including program crashes or unexpected results.

Pointer

class - 2

Date :
Page :

Pointer to an array.

Pointer to an array is a pointer that points to first element address of an array.

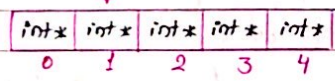
Eg:- `int arr [] = { 1, 2, 3, 4 };`
`int * ptr = arr ;` // Pointer to an array which is pointing to the first element address of array.
`cout << *(ptr+2) ;`

o/p :- 3

Array of Pointer

An array of pointer is an array, where each elements is a Pointer to a memory location of itself.

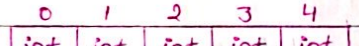
Eg:- `int *arr [5] ;` // Array of Pointer



→ Stored integer pointer data.

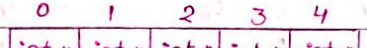
Every element is a pointer to itself.

`int arr [5] ;` → arr =



Stored integer data (integer array)

`int * arr [5] ;` → arr =



Stored integer pointer data.
(Array of pointers)

#
`int arr [5] ;`
`int * ptr = arr ;` pointer to an array

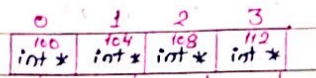
`int *arr [5] ;` Array of pointer, where every element is a pointer

`int (*ptr) [5] ;` // Pointer to an array of 5 size.

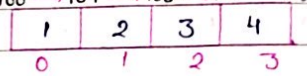
The difference betn, `int *arr [5]` & `int (*ptr) [5]` is Parentheses, so don't confuse with these two statements.

`int *arr [5] ;` → Array of Pointer
`int (*ptr) [5] ;` → Point to an array of 5 size

#
`int *arr [4] ;` → arr =



`int num [] = {1,2,3,4} ;` → num =



Assigning address of array 'num' to each pointer element of array 'arr', because 'arr' is a array of pointer, which store the address of element in each memory block.

```
arr[0] = & num[0] ;
arr[1] = & num[1] ;
arr[2] = & num[2] ;
arr[3] = & num[3] ;
```
} Each elements of array 'arr' store the address of each elements of 'num' array.

arr[0] = 100, arr[1] = 104, arr[2] = 108, arr[3] = 112.

Accessing the element of 'num' through 'arr' array.

```
*arr[0] → *(100) = 1
*arr[1] → *(104) = 2   → Element of 'num' array.
*arr[2] → *(108) = 3
*arr[3] → *(112) = 4.
```

```
int main () {
    int num [5] = { 1, 2, 3, 4, 5 };
    int * arr [5];   // Array of Pointer.
    // Print the value of 'num' using array of Pointer 'arr'.
    for (int i = 0; i < 5; i++) {
        arr [i] = & nums [i] ;  // store the address of each element
        cout << * arr [i] << " " ;   of array 'num'.
    }
}
```

↳ Dereferencing the address stored in each element of array 'arr'.

o/p => 1 2 3 4 5

NOTE: While making Pointer to an array.

```
int arr [3] = { 1, 2, 3 };
int (*ptr) [3] = & arr;  // Pointer to an array of 3 size.
```

The Parentheses around '*ptr' is necessary because the de-reference operator '*' has lower precedence than the array subscript operator [].

Pointer With Functions

Q. What happen When an array is passed to a function ?

A. When we pass an array to a function, we're passing a pointer to the first element of the array, not the whole array. That Pointer allow us to access & manipulate the array's elements directly. This can be efficient for working with large array as it avoids unnecessary coping of data.

Eg:-

```
void solve (int arr [], int size) {        -> int * arr
    cout << " size of array inside function : << sizeof (arr) << endl;
}
```

```
int main () {
    int arr [] = { 10, 20, 80, 40, 50 };
    cout << " size of array in main function :" << sizeof (arr) ;
    solve (arr, 5) ;
}
```

o/p => size of array in main function : 20 (5x4)
size of array inside function : 8 (Size of Pointer)

In 'solve' function array passed as a pointer. Pointer size is always '8 bytes' we learn in previous class. Pointer size doesn't depend on data type, it is depends on machine architectures.

We can write int arr[] in solve function as int *arr.

#

```
int main () {
    int arr [] = { 1, 2, 3 };
    solve (arr, 3) ;
    // Print
    cout << arr << endl;
    cout << & arr << endl;
}
```

o/p => 104 (Base address)
104 (" ")

```
int arr [] = {1, 2, 3};
```

104	108	112	116
1	2	3	
0	1	2	

arr = 104, & arr = 104.

int * arr

```
void solve (int arr [], int size) {
    // Print
    cout << arr << endl;
    cout << & arr << endl;
}
```

o/p => 104 (value in 'arr' Pointer)
216 (Address of 'arr' Pointer)

```
int arr [] (int *arr), int size
```

216	300
104	3
arr	size

arr = 104, & arr = 216

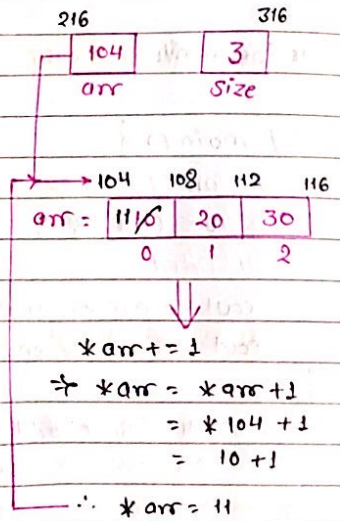
```
void Solve (int *arr, int size) {
    cout << " inside Solve -> arr :" << arr << endl;
    cout << " inside Solve -> &arr :" << &arr << endl;
}

int main () {
    int arr [] = { 1, 2, 3, 4, 5 };
    Solve (arr, 5);
    cout << " inside main -> arr :" << arr << endl;
    cout << " inside main -> &arr :" << &arr << endl;
}
```

only this value is differ, rest of the 3 print the same value.

```
void Solve ( int *arr, int size ) {
    *arr += 1;
}
int main () {
    int arr [] = { 10, 20, 30 };
    Solve ( arr, 3 );
    for (int i=0; i<3; i++) {
        cout << arr [i] << "  " ;
    }
}
```

O/P => 11 20 30



Function to double the value of array using Pointer

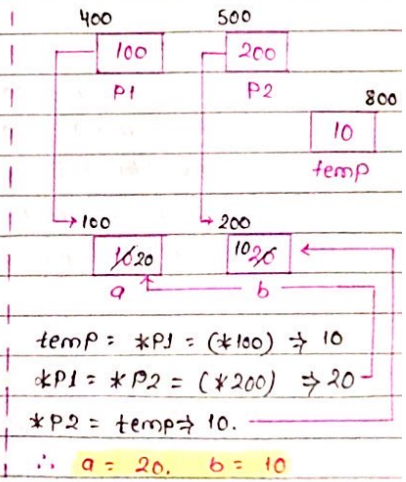
```
void doubleValue ( int *arr, int size ) {
    for (int i = 0; i< size; i++ ) {
        arr[i] = arr[i] x 2 ;
    }
}
int main () {
    int num [] = { 1, 2, 3, 4, 5 };
    doubleValue (num, 5) ;
    for (int i = 0; i<5; i++) {
        cout << num[i] << "  " ;
    }
}
```

O/P => 2 4 6 8 10

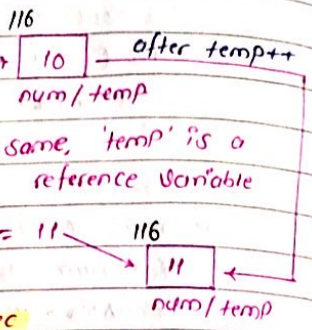
Function to swap the value of two variables.

```
void swapping ( int *p1, int *p2 ) {   -> Pass By Pointer
    int temp = *p1 ;
    *p1 = *p2 ;
    *p2 = temp ;
}
int main () {
    int a = 10, b = 20;
    swapping (&a, &b) ;
    cout << a << "  " << b ;
    return 0 ;
}
```

O/P => 20 10



Reference variable




```
int main () {
    int num = 10;
    int & temp = num;   // num & temp are same, 'temp' is a reference variable
    cout << temp << endl;   // 10
    temp++;   // temp = temp+1 => 10+1 = 11
    cout << num << endl;   // 11
    cout << & temp << endl;   // 0x16b487lec
    cout << & num << endl;   // 0x16b487lec
}
```

Swap two value using reference variable

```
void swapping (int &a, int &b) {   → Pass by reference
    int temp = a;
    a = b;
    b = temp;
}

int main () {
    int num1 = 10;
    int num2 = 20;
    swapping (num1, num2);
    cout << num1 << endl;   // 20
    cout << num2 << endl;   // 10
}
```

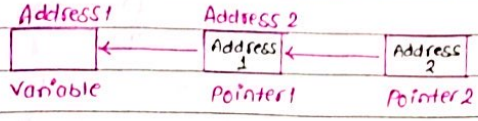
Solve vector & string using reference
Solve integer array, character array using pointer

} while pass in a function

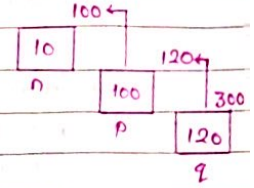
Pointer To Pointer

1) Double Pointer

A double pointer is a pointer that holds the address of another pointer. When we define a pointer to pointer, the first pointer is used to store the address of the variables & the second pointer stores the address of the first pointer.



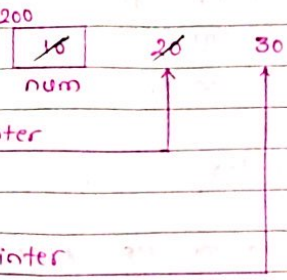
Syntax: data_type ** name_of_pointer = & normal_pointer_variable.



Eg:-
```
int n = 10;
int *p = &n;   // Pointer to a integer.
int **q = &p;   // Pointer to a Pointer
```

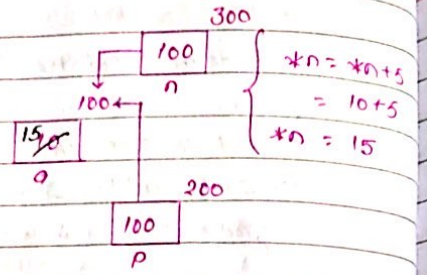
#
```
int num = 10;
int *ptr = &num;   } Single pointer
*ptr = 20;

int **ptr1 = &ptr;   } Double pointer
**ptr1 = 30;
```



∴ final value of num = 30

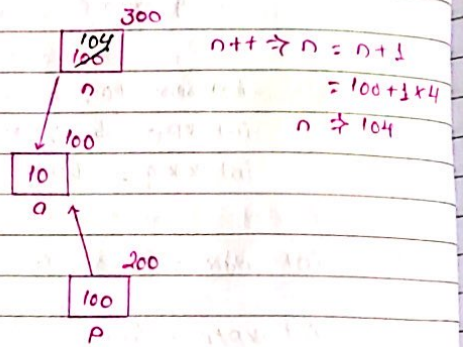

```
void addvalue (int *n) {
    *n = *n + 5;
}
int main () {
    int a = 10;
    int *p = &a;
    addvalue(p);
    cout << n;
}
```



Pointer 'n' is also store the address of variable 'a'.

o/p ⇒ 15

```
void fun (int *n) {
    n++;
}
int main () {
    int a = 10;
    int *p = &a;
    fun(p);
    cout << p;
}
```



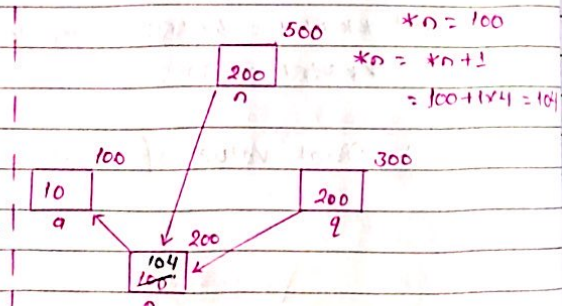
o/p ⇒ 100

'n' doesn't effect the value present in pointer 'p'.


```
void changevalue (int**n) {
    *n = *n+1;
}
int main () {
    int a = 10;
    int *p = &a;
    int **q = &p;
    changevalue(q);
```

Single Pointer

Double Pointer

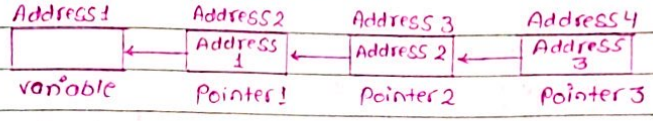



```
    cout << p;
}
```

o/p ⇒ 104

2) Triple Pointer

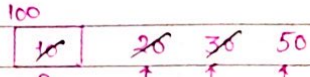
A triple pointer is a pointer that holds the address of a double pointer. In other words, it is a pointer to a pointer to a pointer.



Syntax: datatype *** name of = & double-Pointer
of Pointer a variable variable

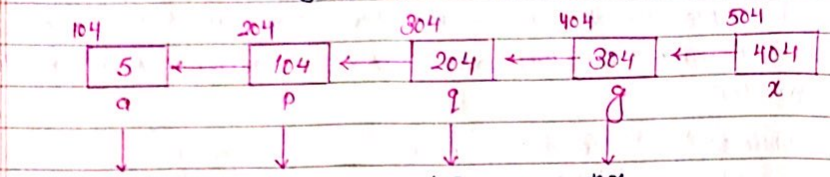
Eg:-
```
int n = 10;      // Integer variable
int *a = &n;     // Single Pointer
int **b = &a;    // Double   "
int ***c = &b;   // Triple   "
```


```
int n = 10;
int *a = &n;
  *a = 20;
int **b = &a;
  **b = 30;
int ***c = &b;
  ***c = 50;
```



∴ final value of n = 50

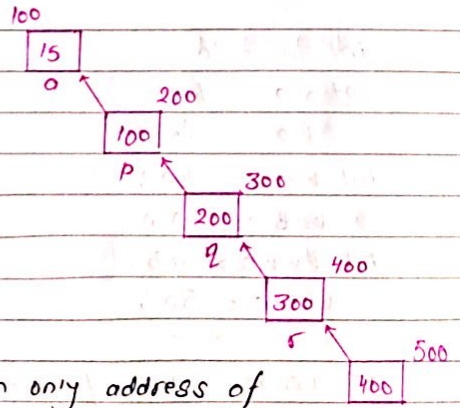
```
int a = 5;
int *p = &a;      // single Pointer
int **q = &p;     // Double  "
int ***g = &q;    // Triple  "
int ****x = &g;   // Multi   "
```



*p	*q	*g	*x	
**q	**g	**x	can access	
***g	***x	can access	pointer 'g'	
****x	can access	Pointer 'q'	using these	
can access	Pointer 'p'	using these	pointer.	
'a' by using	using these	pointer		
these Pointer	Pointer			

Remember: In case of pointer, 'n' Pointer always contains 'n-1' Pointer address.

```
int a = 15;
int *p = &a;
int **q = &p;
int ***r = &q;
int ****z = &r;
int *****n = &(n-1)
```



In this eg: 'z' Pointer contain only address of 'r' pointer. It can't contains the address of 'q' Pointer. similarly, 'r' Pointer can contains address of 'q' Point. can't contains address of 'p' Pointer & so on.

Code:

```
int main () {
    int n = 10;
    int *p = &n;    // Single Pointer contain address of variable 'n'
    int **p2 = &p;  // Double  "      "      "      "  Pointer 'p'

    cout << p << endl;  // o/p -> print address of variable 'n'
    cout << p2 << endl; // o/p -> "      "      "      Pointer 'p'.
    cout << &p << endl; // o/p -> "      "      "      "      'p'.

    int ***p3 = &p2;  // Triple Pointer contain address of double
    cout << p3 << endl;   } Pointer 'p2'.
    cout << &p2 << endl;  } // Both print the address of 'p2'.

    // Modify the value of 'n' using Pointer
    *p = *p + 5 ;  // 10+5 -> 15
    cout << n << endl;  // o/p -> 15

    **p2 = **p2 + 10; // 15+10 -> 25
    cout << n << endl;  // o/p -> 25

    ***p3 = ***p3 + 25 ;  // 25 + 25 = 50
    cout << n << endl;  // o/p -> 50

}
```

∴ final value of 'n' will be '50'

```
int main () {
    char c[] = "Gate2024";
    char *p = c;
    cout << p + p[3] - p[1];
}
```

o/p ⇒ 2024

	100	101	102	103	104	105	106	107	108
Address					↓				
c =	'G'	'a'	't'	'e'	'2'	'0'	'2'	'4'	'\0'
Index	0	1	2	3	4	5	6	7	8

*p = c ⇒ p = 100. (p contain the base address array 'c')

p + p[3] - p[1]

100 + 'e' - '0' { ASCII value of 'a' = 97, " " " 'e' = 101 }

100 + 101 - 97

100 + 4

104 (100 + 4 × 1) ↳ Size of 'character' data-type

cout << p + p[3] - p[1]

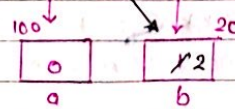
↳ cout << 104 ; print from address '104' until found the null character in array.

∴ cout << p + p[3] - p[1] ⇒ 2024.

```
void second (int *p1, int *p2) {
    p1 = p2 ;
    *p1 = 2 ;
}
int main () {
    int a = 0, b = 1 ;
    second (&a, &b) ;
    cout << a << " " << b ;
}
```

o/p ⇒ 0 2

p1 = p2 (p1 = 200).

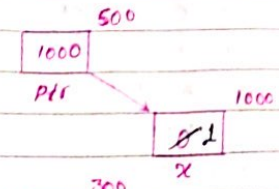


(*p1 = 2)

∴ a = 0 & b = 2

```
int main () {
    int *ptr ;
    int x = 0 ;
    ptr = &x ;
    int y = *ptr ;
    *ptr = 1 ;
    cout << x << y ;
}
```

o/p ⇒ 1 0



(y = *ptr ⇒ 0).

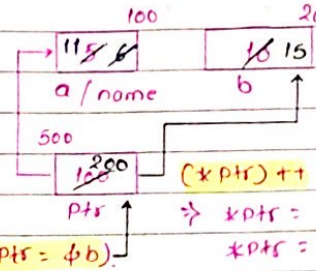
(*1000 ⇒ 0, value of x)

(*ptr = 1 ⇒ *1000 = 1)

↳ change the value of x to '1'

```
int main () {
    int a = 5 ;
    int b = 10 ;
    int &name = a ;
    int *ptr = &a ;
    (*ptr)++ ;
    ptr = &b ;
    *ptr = *ptr + 5 ;
    name += 5 ;
    cout << a << endl ;
    cout << b << endl ;
}
```

o/p ⇒ 11 (value of a)
15 (" " b)



(*ptr)++ ⇒ *ptr + 1

⇒ *ptr = *(100) + 1

*ptr = 5 + 1 = 6.

(ptr = &b).

*ptr = *ptr + 5

= (*200) + 5 ⇒ 10 + 5 = 15.

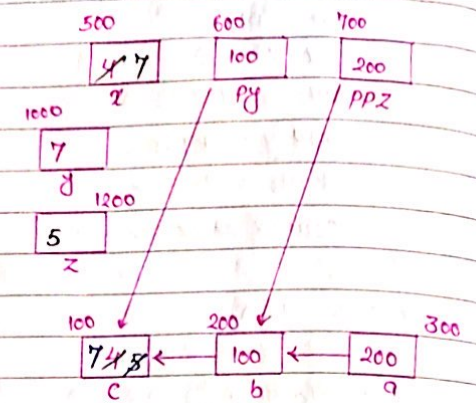
*ptr = 15 (value of b)

name += 5

name = name + 5 ⇒ 6 + 5 = 11

name = 11

```
int four (int x, int *py, int **ppz) {
    int y, z;
    **ppz += 1;
    z = **ppz;
    *py += 2;
    y = *py;   x+=3;
    return x+y+z;
}
int main () {
    int c, *b, **a;
    c=4, b=&c, a=&b;
    cout << four (c, b, a);
}
```



o/p → 19.

DRY RUN.

Inside four() function

i) `**ppz += 1;` (can access 'c' using 'ppz' Pointer)

`**ppz = **ppz + 1`

`= 4 + 1 = 5`

`**ppz = 5;` (new value of c)

ii) `z = **ppz;`

`z = 5;` (value of z)

iii) `*py += 2;` (can access 'c' also using 'py' Pointer)

`*py = *py + 2`

`= 5 + 2`

`*py = 7` (new value of c)

iv) `y = *py`

`y = 7` (value of y)

v) `x += 3`

`x = x + 3`

`= 4 + 3`

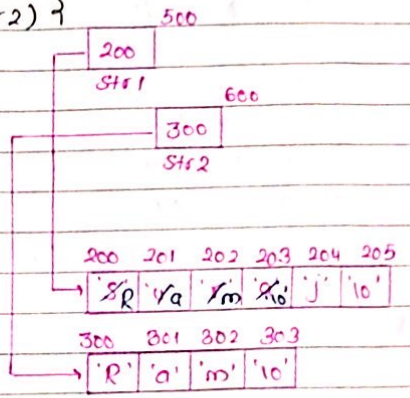
`x = 7`

vi) return (x+y+z)

`= (7 + 7 + 5)`

`= 19.`

```
void five (char *str1, char *str2) {
    while ((*str1 = *str2)) {
        str1++;
        str2++;
    }
}
int main () {
    char first [] = "Suraj";
    char second [] = "Ram";
    five (first, second);
    cout << first;
}
```



o/p → Ramaj

DRY RUN.

Inside five() function

str1 = 200, str2 = 300

while (*str1 = *str2) {

str1++, str2++;

}

i) `*str1 = 'R'` (*str2 = R) str1++; str2++;

ii) str1 = 201, str2 = 301

`*str1 = 'a'` (*str2 = a)

str1++, str2++;

iii) str1 = 202, str2 = 302

`*str1 = 'm'` (*str2 = m)

str1++; str2++;

iv) str1 = 303, str2 = 303.

`*str1 = '\0'`

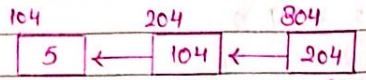
while loop become, while ('\0') {

means break the loop.

Inside while loop, when not null value pass, it run the loop & when null value is pass, it break the loop (loop terminate).
→ Previous page code is a code of string copy. While we've to copy string value, we use this code.

Guess the output

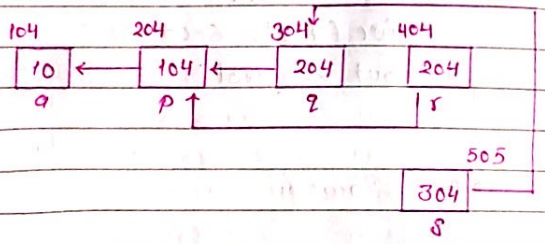
```
int a = 5;
int *p = &a;
int **q = &p;
```



i) a → 5
ii) &a → 104
iii) p → 104
iv) &p → 204
v) *p → 5
vi) q → 204
vii) &q → 804
viii) *q → 104
ix) **q → 5

Guess the output

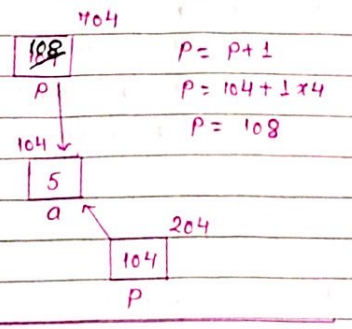
```
int a = 10;
int *p = &a;
int **q = &p;
int **r = &p;
int ***s = &q;
```



i) *s → 204
ii) **r → 10
iii) ***s → 10
iv) **q → 10
v) **s → 104
vi) *q → 104
vii) *p → 10
viii) p → 104
ix) &s → 505
x) &r → 404
xi) &q → 304
xii) **s + 1 → 108

↓ ↳ size of integer data type
104 + 1 x 4 = 108


```
void solve (int *p) {
    p = p+1;
}
int main () {
    int a = 5;
    int *p = &a;
    cout << p;  // 104
    cout << &p; // 204
    cout << *p; // 5
    solve (p);
    cout << p;  // 104
    cout << &p; // 204
    cout << *p; // 5
}
```



change happen only in solve() function, because we pass the pointer by value in function. copy of pointer created in function & value change only in solve() function.


```
void solve (int *&q) {
    q = q+1;
}
int main () {
    int a = 5;
    int *p = &a;
    cout << p;  // 104
    cout << &p; // 204
    cout << *p; // 5
    solve (p);
    cout << p;  // 108
    cout << &p; // 204
    cout << *p; // Garbage value. 'p' store the address of unknown location
}
```