Array

* Array is a collection of similar data items or elements
* It is used to store group of data simultaneously.
* It can store data of same data type means an integer array can store only integer value, character array can store only character value and so on.
* Each location of an element in an array has a numerical index no., known as subscript, which is used to identify the element.
* Index is always starts with zero.

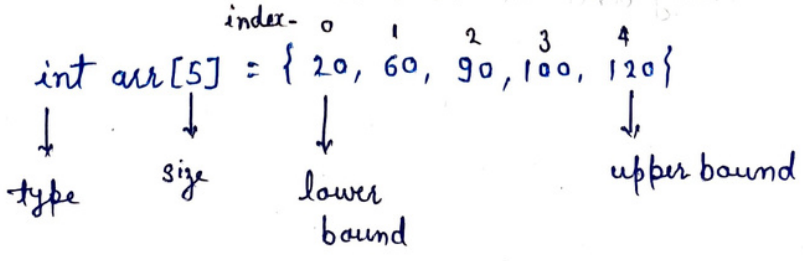
Syntax:

```
type  name[capacity];
```

Eg:- `int arr[100];` // declaration of an array

`int a[5] = {10, 20, 30, 40, 50};` // declaration & initialisation

* Array subscript (index) always start from zero which is known as lower bound and upper value is known as upper bound.



Q. Program to input values into an array and display them -

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5];
    for(int i = 0; i < 5; i++)
    {
        cout<< "Enter values for arr[i] :";
        cin >> arr[i];
    }
    cout<<"The array elements are:";
    for(int i= 0;  i<5; i++)
    {
        cout <<arr[i]<<" ";
    }
}
```

Note -

```
int a[3] = {9, 10, 11};
```

→ This array is a type of one-dimensional array and it is also known as vector.

* Two dimensional array (Matrix) -

Two dimensional array is popularly known as tables or matrix and can be easily visualized as having rows and columns.

* To create a two dimensional array, specifying both dimensions ie rows and columns in square brackets.

Syntax:

data type arrayname [row] [column];

* 2-D array is a collection of 1-D array placed one below the other.

* Total no. of elements in 2-D array is calculated as row * column.

Eg:- int arr [2] [3];

(It means, the matrix consist of 2 rows and 3 columns)

and No. of elements = 2×3 = 6

* Matrix initialization -

```
int mat[3][3];
```

or, `int mat[3][3] = {{1,2,3},{4,5,6},{7,8,9}};`

or, `int mat[3][3] = {1,2,3,4,5,6,7,8,9};`

→ While initializing, it is necessary to mention the 2nd ~~dimenti~~ dimension where 1st dimension is optional -

```
int mat[][3];
int mat[2][3];
```
} valid

```
int mat[][];
int mat[3][];
```
} Invalid

Q Program for taking input and printing the element of 2-D array -

```
#include <iostream>
using namespace std;
int main()
{
    int rows= 2;
    int cols = 3;
  int arr[rows][cols];
// Loop through each row and column to get user input
  for(int i=0; i< rows ; i++)
   {  for( int j = 0; j< cols ; j++)
      {
        cout<< "Enter a value for element [" << i << "][" << j << "]: " ;
        cin >> arr[i][j];
      }
   }
// Print the values of the array -
 cout << " The elements of the array :";
  for(int i=0; i< rows; i++)
   {  for( int j=0; j< cols; j++)
      {
        cout << arr[i][j] << " ";
      }
      cout << endl;
   }
```

Q. Program for taking input and printing the element of 2D array using "for each loop" -

```
#include <iostream>
using namespace std;
int main()
{
    int A[2][3];
    for (auto &x : A)
    {
        for( auto &y : x)
        {
            cin >> y;
        }
        cout << endl;
    }
    for (auto &x : A)
    {
        for(auto &y: x)
        {
            cout << y << " ";
        }
        cout << endl;
    }
}
```

Pointer

* A pointer is a special variable that is used to store the address of some other variable.

* A pointer can be used to store the address of a single variable, array, structure, union or even a pointer.

* A pointer is a derived data type.

* Pointers provide a way to directly manipulate memory and can be used to create dynamic data structures.

* To declare a pointer, we use the asterisk (*) symbol before the variable name -

Eg:-
```
int *Ptr1; // Pointer to integer type
float *Ptr2; // Pointer to float type
char *Ptr3; // Pointer to char type
```

* When pointer declared, it contains garbage value ie it may point any value in the memory.

Note: `int *Ptr;`

→ Here the type int refers to the data type of the variable which is pointed by Ptr not the type of the value of the pointer.

→ It means pointer 'ptr' can hold only the memory address of an integer variable.

* Pointer declaration style -
  - int * ptr ;
  - int *Ptr;
  - int * ptr;

* Initialization of pointer variable -

→ int a ;
int * P ; // declaration
int P = &a; // initialization

or, int x, *P = &x ; // declaration with initializ

or, int *P = &x, x ; (Not valid, because target variable 'x' should be declared first.)

→ float a;
int * P;
P = &a ;
} Not valid, because we can not assign the address of a float variable to an integer type pointer.

→ float a;
float * P;
P = &a;
} It is valid

* Dereferencing-

Accessing the data present at the address where pointer is pointing.

```
int x = 5;
int *ptr = &x; // assign the address of x to ptr
cout << *ptr; // dereference ptr to get the value at that address
```

Eg:-

```
int main()
{
  int a = 10, b = 9;
  int *p, *q;
   p = &a;
   q = &b;
   c = *p;
cout << "Value of a = " << a ;
cout << "Value of a = " << *p;
cout << "Value of a = " << c;

cout << " Address of a = " << &a;
cout << "Address of a = " << p;
cout << " Address of p = " << &p;
}
```

o/p

a = 10

a = 10

a = 10

61fec4

61fec4

51abe4

* Printing all elements of an array using pointer-

```
#include <iostream>
using namespace std;
int main()
{
    int A[5] = {2, 4, 6, 8, 10};
    int *P = A;   (In array, no need to write '&' operator)

    for(int i=0; i<5; i++)
    {
        cout << A[i] << endl;
        or, cout << i[A] << endl;
        or, cout << *(A+i) << endl;

        cout << (P+i) << endl;
        or, cout << (A+i) << endl;

        cout << *(P+i) << endl;
        cout << P[i] << endl;
    }
}
```

All are used to printing the elements

Both will give address of the elements

These are also used to printing the elements.

* Pointer to pointer (Chain of pointers) -

As we know, pointers stores the address of a variable, similarly the address of a pointer can also be stored in some other pointer, it is called pointer to pointer variable.

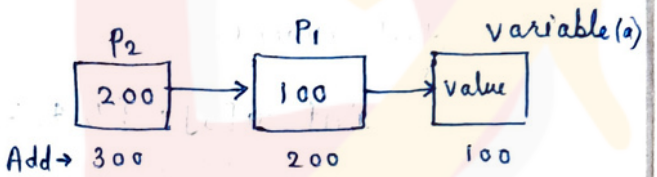
"Pointer within another pointer is called pointer to pointer".

Eg :-
```
int main()
{
  int a = 5;
  int *p1;
  int **p2;
  int ***p3;

  p1 = &a;
  p2 = &p1;
  p3 = &p2;

  cout << a << endl;          // o/p -> 5
  cout << *p1 << endl;        // 5
  cout << **p2 << endl;       // 5
  cout << ***p3 << endl;      // 5

  cout << *p2 << endl;        // address of p1 will be printed.
}
```



Note :

→ A two level pointer (**p2) always stores one level pointer (*p1) variable not normal variable (q)

→ Similarly, three level pointer (***p3) always stores two level (**) pointers.

* Pointer Arithmetic -

→ Pointer addition - (Performing on array)

```
int main()
{
    int a[5] = {1, 4, 2, -8, 0};
    int *p = a; (or int *p = &a[0])
    int *q = a;
    cout << "Value is : " << *p ;        → o/p 1
    p = p + 2  // Forward two position
    cout << "Value is : " << *p;         → 2
    cout << "Value is : " << (*p + *q) ; → 3
    cout << "Value is : " << p+q ;       → Not allowed (Invalid)
```

Note:

→ *p + *q → p+q

2 + 1 = 3

(In this case, we add the value which are present in pointer p & q)

(We can not add two pointers)

* Pointer Subtraction -

```
int main()
{
    int a[] = {2, 4, 3, 0, -7};
    int *p = &a[0];
    int *q = &a[3];                          o/p
    cout << "q-p = " << q-p << endl;  → 3
    cout << "p-q = " << p-q << endl;  → -3
    cout << *q << endl;              → 0 (a[3]=0)
    q = q-2; // Backward two position
    cout << *q << endl;              → 4
}
```

('q' points the 4th element and 'p' points the 1st element. So, 4-1 = 3)

→ q-p : (gives the no. of elements b/w q & p)

→ q = q-2 (Decrement the pointer means backward two position)

→ q = *q-2 (Decrement the value which are present in 'q' pointer means here subtraction occures)

* If 'p' and 'q' are two pointers then -

→ a = p+q; (x)

→ a = *p + *q; (✓)

→ a = *p + 2; (✓)

→ a = p-q; (✓)

→ a = *p - *q; (✓)

→ a = *p * *q; (✓)

→ a = p*q; (x)

→ a = ~~*~~*p/*q; (✓)

→ a = p/q; (x)

* Void pointers -

In void pointers, we can assign different types of data types using typecast

```
int main()
{
    void * vp;
    int a = 5;
    float b = 1.56
    char ch = 'a';
    vp = &a;
    cout << *(int*)vp;        → 5

    vp = &b;
    cout << *(float*)vp;      → 1.56

    vp = &ch;
    cout << *(char*)ch;       → a
```

* Dynamic memory in heap -

→ Dynamic memory is created in heap section using pointers.

→ We can use dynamic memory for creating array, linked list, trees, map etc.

→ Heap memory can be accessed from anywhere in the program, it is available.

→ Heap memory can be used for storing data of entire application.

* Allocate and deallocate of dynamic memory in the heap-

→ We use the 'new' operator to allocate dynamic memory in the heap.

→ And the 'delete' operator is used to deallocate the memory from heap.

```
int *ptr = new int ;  //allocate memory in heap
                         for an int
*ptr = 10;  // store the value

delete ptr;  // deallocate the memory
```

* Memory leak -

→ If a program requires memory at runtime, it will allocate in heap using pointer.

→ If heap memory is not in use, it should be deallocated.

→ And if a program is not deallocating then the memory reserved for heap may become full and it couse the memory leak.

* Dangling pointer -

If a pointer is having an address of a memory location which is already deallocated then it is known as dangling pointer.

```
int * P = new int[5];
delete [] P;   // deallocated
```

(Now 'P' is a dangling pointer)

NULL V/S nullptr -

* **NULL :**

→ It is a constant whose value is '0'.

→ NULL means, pointer is not pointing on any valid location.

→ But null(0) can be implicitly converted to pointers of any type, which can cause unintended behavior.

→ In place of 'NULL', 0 can be used.

* **nullptr :**

→ It is a keyword in c++.

→ nullptr means, pointer is not pointing on any valid location (same as NULL)

→ But nullptr can not be implicitly converted to other pointer types. It is safe.

→ '0' can not be used in its place.

* Reference :

A reference variable is an alias or alternative name for an existing variable.

→ When a reference variable is created, it is assigned to an existing variable and both the reference and the original variable refer to the same memory location.

→ Therefore, any changes made to one variable will also affect the other variable.

→ It must be initialized when declared.

→ It doesn't take any memory.

Eg:-

```
int main()
{
    int x = 10;
    int &y = x;
    x++;        // 11
    y++;        // 12
    cout << x;  // → 12
    cout << y;  // → 12
}
```

Reference