OOP Date:

Week 1 : Lecture : 1

> Pointers is a special data type used to store and manipulate addresses of main memory

> Pointers are used to control and manage dynamic memory

> Pointers

Content : Address

Name : Any name

Size : Most pointers are of size 4 bytes (to store address of any data type)

Address : Random Address

> While declaring a variable of pointer type, there is no mention of pointer as we previously used int for : integer, character.

> We can directly initialize a pointer to 0, NULL or nullptr.

> int *ptr = 34 or any other number, then this will cause a runtime error & this is illegal.

Eg:1

```
int x = 10;
int *ptr;
ptr = NULL/0/nullptr;
ptr = &x;
```

ptr could be any name here nullptr is preferrable

int * is type of ptr meaning that it can only store address of int variable

* Address operator

> Symbol : & (Ampersand)

> This is used to fetch address of any variable.

> This is used to assign the address of any variable to the pointer of the same type as the variable.

> Addresses of variable in hexadecimal type. and they always start with 0X

Eg:2

```
char r = 'A';
int *ptr2 = nullptr;
ptr2 = &r;
```

> Assigning address of a r is a char type variable variable of one type to ptr2 is int * type pointer the pointer of another int type pointer is assigned address of char variable type causes an error. The error will be compile time error.

> Compile time errors are pointed out before execution of the program. They dont let the program run

Date:

* Derefrence Operator

> Symbol: *

> This operator needs a valid operator

> It is used to print the content of the variable whose address is stored in a pointer

> From eg:1, cout << *ptr; will print 10

> From eg:2, cout << *ptr2; will print A if ptr2 is char type pointer

> We previously said that it needs valid addresses, following are examples of invalid addresses

- **Uninitialized Pointer:**

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

This won't cause an error message, rather this will cause erroneous results.

- **Null initialized Pointer:**

```
char *ptr2=nullptr;
cout<<*ptr2;
```

This will cause an error because there is no memory location labeled as/with the address (NULL) zero.

> `char *ptr2;` } This will print a random CHARACTER.
`cout<<*ptr2;` }

> `*ptr =50` } This is a valid statement.

If this is written after statements of eg:1 then 50 will be stored in

- **Pointers are used to both reading and writting purposes.**

```
1:  int x=50;
2:  int *ptr=&x;
3:  cout << *ptr;   } Reading
4:  *ptr=100;       } Writing
```

If ptr is uninitialized and there is a statement like 4th statement then there will be a error, as we may change value of some uninit memory location which might cause errors in future.

A variable can be largest possible value by
int a;
a = ~a;
~ inverts the number, 000 becomes !!!

Date:

Week 1 : Lecture : 02

& Address operator

* to declare a pointer / derefrence operator

Everything is stored in compiler is in bits let it be address characters, integers, floating values or anything.

Assignment Operation on Pointers

Compile-time error `q = *p` } This will cause an error ... we can't assign integer to pointer

Compile-time error `*x = *q` } *x is a constant. We can't assign a value to a constant

. You can't derefrence a regular variable
Derefrence operators are for pointer

```
int *p = 0;
int *q = 0;
int x = 50;
int y = 100;
```

This is used to assign a variable to a pointer { `p = &y;` `q = &x;`

Here 40 is in y { `*p = 40;`

Here y will have 50 { `y = *q;`

Copy contents of y into x { `*q = *p;`

Now p-q will point to same variable { `q = p;` (They must be of same data type)

Relational Operators

==, != Very useful (Used to compare if they have same addres

>, >=, <, <= Rare use

```
int x=10, y=20;
int *q = &x;
int *p = &y;
cout << q==p;   (0 if they point to different addresses
                 1 if they point to same addresses)
cout << q!=p;   (Opposite than the above)
```

Not V. useful `cout << q > p;`

`cout << *q == *p << *q + *p;` { After derefiencing, you can perform any operation which is valid on q or p (i.e, their data type)

Date:

Arithematic Operators:

★ Only two valid arithematic operations on pointers
- Addition (+) [This acts like a jump]
- Subtraction (-)

----- First 4 lines of last program -----

001	x	50
005	y	100
009		
002		

⇒ Addition:

- We can only add an integer to a pointer.
- We can not add two pointers

```
cout << p+1; } This won't display 006, this will display 009
```

030	p	005
035	q	001

Now, the jump is determined by the type of variable.
If it is int, the jump will be of 4 memory locations.
If the type is character then jump will be of 1 memeory location. Here each location is a byte.

- The integer here added means the number of variable's type size jump to be taken
- Increment operators can be used with pointers. Both pre and post increment operators are allowed.
- This is forward jump

⇒ Subtraction:

- We can subtract two pointers.
- We can subtract from a pointer any integer value

---- Considering above example (memory) -----

```
cout << p-1; } This will display 001
```

- Again, jump is determined by the (memory) type of the variable as previously.
- Now, cout << p-q ; (Refrencing to adj. fig)

p	007
q	030

∵ p & q are integers

p-q / q-p = 23

Now 23 bytes can possibly store 5 integers at max so ans will be '5'

- Pre & Post decrement is also valid.

Date:

Week 2 : Lecture 1

⇒ Pointers & Functions

- pass pointers to a function
  - > by value
  - > by refrence
- return from a function

Example.

```
int main( )
{  int x = 10;
   int *p = &x;
   fun (p)     By value
   fun (&x)    Illegal operation }
   fun2 (*p)   Will cause errors
   fun2 (&p)   By refrence
}
```

By Value

```
void fun (int *p)
{  cout << p ;
   cout << *p ;
   *p = 100 ;
}
```

By Refrence (* with data type & with variable)

```
void fun2 (int *&p)
{  cout << p ;
   cout << *p ;
   *p = 100 ;
}
```

Pointers by value or refrence, but they make the data to be passed by refren

⇒ Use of const with pointers

- **Name constants**
  - > These values do not change
  - > It is used to repeatedly use a complex value witho typing it again and again.
  - > It is also used with parameters while defining a function so that the original value doesn't change.

Date:

Use of const with pointers.

There are 4 variations of this type.

1) Non contant pointer to non-constant data

```
int x = 10;
int *p = &x;
p++;          } Non-constant pointer,
p = nullptr;  } we can change its value.
(*p)++;       } Non-constant data, the
*p = 500;     } content can be modified through pointer.
```

2) Non-constant pointer to constant data

```
const int x=10;
const int *p = &x;   } p is a pointer to const integers.
p++;          } This is valid, pointers value can be changed
p=NULL;       }
(*p)++;       } This is not valid, data is const so it can
*p=500;       } not be changed.
```

3) constant pointer to non-constant data

```
int x=10;
int * const p = &x;
p++;          } This is not valid as p is constant and it
p=NULL;       } will always point to x
(*p)++;       } this is valid, the content can be changed
*p=100;       } since it is not constant.
```

Date:

4) Constant pointer to constant data

```
const int x = 10;
const int *const p = &x;
p++;        } Not valid as pointer is constant
p = NULL;   }
(*p)++;     } Not valid as data is constant
*p = 100;   }
```

Example:

```
int main()
{
    int x = 10;   : Read Write
    fun(&x);
}
```

```
void fun(const int *p)   • Only Read
{ p++; } Valid
(*p)++; } Invalid, data is const
}
```

Example:

```
int y = 100;
int *p = &y;        ⇒ read + write
const int *p = &y;  ⇒ only read
```

Example:

```
const int y = 100;
int *p = &y; } Type mismatch error
```

- p is int point
- y is const int

```
void fun(int *& const p)
```

- Address can't modify
- Data can change.

Date:

Week 2 : Lecture 2

Dynamic Memory:

- Also called as heap
- (It is allocated through runtime).
- It is allocated during runtime and if it is no more required so it can be deallocated
- It has no name. It has size content and address.
- It is not compiler dependent.
- It is process & destroyed (deallocated) through use of pointers

Operator: new ⇒ reserved word

- 'new' is used for allocation of dynamic memory
- It will request system for allocation of memory (variable)
- on success, it will return address of allocated memory.

Eg: new int; } it will require to create a variable of 4 bytes in dynamic memory

↘ This is going to return address of the variable in heap.

- The variable won't have any name.

Eg: int *p = new int; } Address of dynamic memory is stored in p.

cout << p << *p;
Address junk

→ We can initialize dynamic memory using following methods

1. int *p = new int;
   *p = 100;

2. int *p = new int(6)
   Address allocation | creation initialization.

Operator:

- delete ⇒ reserved word
- 'delete' is used for deallocation of dynamic memory
- it requires a valid address of allocated memory on heap
- it can only be used for heap memory

Eg: int *p = new int(6);
------ processing -----
delete p; The memory that we allocated previously now is free

- The pointer still has the address of that variable.

⇒ such a pointer is called dangling pointer and is dangerous.

Date:

- After deallocation, it is required to initialize pointer to 'nullptr'

Errors occur:

if ⇒ delete is used to deallocate same memory multiple times

if ⇒ delete is given address of stack memory

(1)
```
int x=10;
int *p=&x
delete p;
p= nullptr;
```
Runtime error stack memory can't be deallocated

(2)
```
int*p=new int(10);
delete p;
*p=50;
```
Runtime error can't write into deallocated memory.

(3)
```
int *p=new int(150);
delete p;
delete p;
```
Runtime error. can't delete a deallocated memory

Dangling Pointers:

- Contains address of unallocated or deallocated memory
- It corrupts data
- Program terminates with runtime error.

Eg: of dangling pointers:

(1)
```
int *p =new int (10);
p++;
delete p ;
```
] It has unallocated memory

(2)
```
int* p =new int (10);
delete p;
delete p;
```
] It has deallocated memory.

(3)
```
int *q, *p;
p=new int (30);
q=p;
delete p;
p=nullptr;
*q =500;
```
error

q is dangling pointer

Memory leaks

- A portion of memory which
  - ⇒ cannot be used (accessed)
  - ⇒ allocated for other process

Eg: (1)
```
int *p = new int (100);
```
Either of these
```
p=null ptr;
p=new int (30);
p=&x;
```

Date:

```
int *p =new int(50);
int *q =new int(100);
*q = *q + *p;        Simple data change
p++;                 move to unallocated memory
p= p + *q;           "
p++;                 "
*p = *q * *q + *p;   changes in unallocated memory
p = q;  or  q=p;     then p and q will point to same memory.
delete q;            q is deallocated
q =p;                q is dangling operator, now points to same aop
p =nullptr;          p is null
q=p;                 q & p points to same operator.
```

CODE:

```
int *fun( )        return type
{  int *p =new int(10);
   *p = 20;
   return p;
}
int main( )
{  int *q =fun();
}
```

Date:

Week 3: Lecture 1

Pointers and Arrays

1) An array name is a constant pointer
   ⇒ int arr [5];
   ⇒ cout << arr; This will print address of the first element in arr
   ⇒ cout << &arr[0]; This will do the same job as previous statement
2) Array data and processing is similar to the pointer
   ⇒ arr [5]; This makes a jump of 5 from the base address
3) int *ptr;
   ⇒ Store address of single variable (Static/Dynamic)
   ⇒ Store the base address of an array.
4) Subscript Operator:
   This has a natural/default derefrencing in it
   cout << arr [1];
5) Offset Notation:
   In this notation, we have to do derefrencing
   cout << *(arr+1)

Similarities
Eg: cout << arr; (base address of first element
= cout << ptr; Here ptr is a pointer to first element of array
cout << arr[1]; = cout << ptr[1]; } They have the same
cout << *(arr+1) = cout << *(ptr+1); } operations

Difference
6) Increment/Decrement of array name is illegal, however it is possible using pointer
   Eg: arr = arr+1; It is illegal
   ptr = ptr+1; It will make a jump and will point to next element of the array.

- After increment, cout << ptr[1] and cout << arr[1]; will give different values

7) A pointer can be used to point to any element of array
   Eg: int *ptr3 = &arr[5]
8) We can write data using pointer
   Eg: ptr[1] = 50;

Date

Dynamic Arrays:

⇒ They can be managed easily using pointers.

⇒ You have to do
- allocation
- deallocation

⇒ Dynamic Arrays have no name

Allocation:

```
int *ptr = new int [5];
```

int value / named constant / int variable ***

- A dynamic array is treated in the same way like a static array.
- Data can be searched /sorted/ elements can be removed or any operation could be performed

Deallocation:

- delete operator

It requires
- valid address
- allocated memory
- heap memory

- You can delete the dynamic array element by element using a loop

Eg:
```
for (int i=0; i<5; i++)
    delete (ptr+i);
```

- You can use a single statement for deallocation of array.

Eg: `delete [] ptr;`

Date:

Week 3 : Lecture 2

Misconceptions abt arrays:

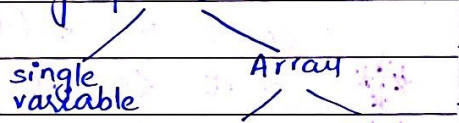
1) If we change size variable, array's size will also change
2) deallocating any element from array will also remove it from array

To resize an array:

1) Create new array of smaller or larger accent
2) Copy data from old array to new one element by element.
3) deallocate original array
4) Update the pointer variable

function and dynamic 1D Array

⇒ pass to function [pointer → by refrence / by value]
⇒ can return base address of array from function
⇒ data is always passed by refrence through pointers.



Shallow vs Deep Copy

```
int *p1, *p2;
p1 = new int [10];
p2 = p1;          ⟹ shallow copy created
delete [] p1;
p1 = nullptr;
```

Shallow copy

- Two pointers pointing to the same memory location. When memory is deleted using one pointer, both pointers become dangling and must be initialized.

Deep copy

- Two pointers pointing to 2 diff locations with the same content/elements

Date:

Week 4: Lecture 1

2D dynamic arrays:

To make a ~~do~~ 2D dynamic array, you should know how to create an array of pointers.

Eg:

```
int *ptr;                    a simple pointer
int *arrptr[5];              an array of pointers is made
int x=10;                    two simple variables
int y=30;
arrptr[0]=&x;                pointers point to simple variables
arr ptr[1]=&y;
arr ptr[2]=new int[100];     pointer points to an array in dynamic memory
arr ptr[3]=new int(5);       pointer points to a dynamic variable
int arr[3]={1,5,9};
arrptr[9]=arr;
cout << *(arrptr[0]);        dereferencing an element in array of
cout << *(arrptr[2]);        pointers
cout << arrptr[2][1];        This will print 5
cout << *(arrptr[2]+1);      This acts the same way as abv statement
```

Now we can access the elements of array declared in arrptr[2] as follows

```
arrptr[2][0]=1
arrptr[2][3]=2
```

for a 2D dynamic array:

1. An array of pointers
   - On stack: Partially dynamic 2D array
   - On heap: Completely dynamic array
2. Store 1D array at each index of arrays of pointers.

Date:

Eg:

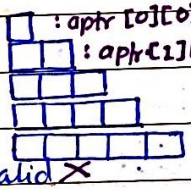
```
const int s=10;
int *aptr[5] = {0};        All elements are initialized to Nullptr
aptr[0] = new int [5];
aptr[1] = new int [5];
aptr[2] = new int [5];
aptr[3] = new int [5];
aptr[4] = new int [5];
```

This can be done using a loop

```
for (int i=0; i<s; i++)
    aptr[i] = new int [5];
```

We can also vary sizes of each array element i.e., row

```
for (int i=0; i<s; i++)
    aptr[i] = new int [i+1];
```



* In this allocation, aptr[0][1] is invalid ✗

```
aptr[0][0] = *(aptr[0] + 0);  = *(*(aptr + 0) + 0);
```

deallocation:

```
delete [] aptr[0];
delete [] aptr[1];
```

```
for (int i=0; i<s; i++)
    delete[] aptr[i]
```

array of pointers:

- created on compile time
- we can't change or resize array of pointers (no. of rows can't be changed)

Multiple indierection:

```
int x = 10;
int *ptr = &x;
cout << x << *p;
        direct  indirect

int **dp = &p;
   double aestrick

cout << dp << *dp << **dp;
        &p    &x    x=10
```

Now there are 2 indierections

Also `int *** tp = &dp` ← 3 indierections

Date:

Eg:

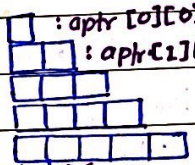
```
const int s=10;
int *aptr[5] = {0};        All elements are initialized to Nullptr
aptr[0] = new int[5];
aptr[1] = new int[5];
aptr[2] = new int[5];
aptr[3] = new int[5];
aptr[4] = new int[5];
```

This can be done using a loop

```
for (int i=0; i<s; i++)
    aptr[i] = new int[5];
```

We can also vary sizes of each array element i.e., row

```
for (int i=0; i<s; i++)
    aptr[i] = new int[i+1];
```



* In this allocation, aptr[0][1] is invalid ✗

aptr[0][0] = *(aptr[0] + 0); = *(*(aptr + 0) + 0);

deallocation:

```
delete[] aptr[0];
delete[] aptr[1];
```

```
for (int i=0; i<s; i++)
    delete[] aptr[i]
```

array of pointers:

- created on compile time
- we can't change or resize array of pointers (no. of rows can't be changed)

Multiple indierections:

```
int x = 10;
int *ptr = &x;
cout << x << *p;
        direct  indirect

int **dp = &p;
   double asterisk
cout << dp << *dp << **dp;
        &p    &x     x=10
```

Now there are 2 indierections

Also `int ***tp = &dp` → 3 indierections

Date:

Week 4: Lecture 2

2D dynamic Array (Not partially dynamic)

- To move the 1D array of pointers to heap and then using it to get a 2D dynamic array, we will get to use multiple indirection.

```
int ** dp = new int *;
```
↳ This could point to a single variable or array.

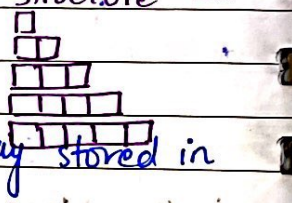
It can point single pointer or can point to a array of pointers.

```
int ** dp. = new int *[constant]
```
↳ This could be value, named const or integer variable.

Example:

```
int size = 5;
int ** arr = new int * [size];
for (int i=0 ; i < size ; i++)
    arr [i] = new int [i+1];
```

Structure will be



- Derefrencing arr once, will give us base address of array stored in an element in array of pointers.
- Derefrencing arr twice, can help us to edit the data of the dynamic array. Eg: **arr=10; or *(*arr+1) = 30;

Operator's Precedence

Highest ↑	subscript operator	[]
	derefrence operator	*
	addition	+
Lowest		

```
arr [3][1] = *(*(arr+3)+1)
```
They point to the same address

- To deal with these arrays, it is better to store a sentinal value at the end of each row.
- U can increase/decrease no. of rows.

Date:_____

To resize array (either the rows or columns) (either to extend / shrink)

- Make new array of new size
- Copy the data
- Delete the old pointer
- Update pointer

We have made a fun for that

void extend(int*& arr, int &size);
↳ This is only for columns.
void extend(int** arr, int &size)
↳ This is for rows.

Now we can resize all rows.

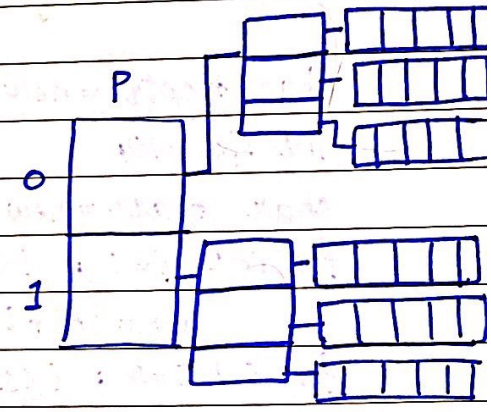
As

```
for(int i=0 ;i<size ;i++)
    extend(arr[i], s[i]);
```

s is an array to store size of each row

Week 5 : Lecture 1

Multiple Indirection can be used to create multidimensional arrays.



Eg:

```
int *** p = new int **[2];
for (int i =0 ;i < 2 ;i++)
{
    p[i] = new int *[3];
    for (int j =0; j<3; j++)
    {
        p[i][j] = new int [5];
    }
}
```

deallocation:

```
for(int i=0;i<2;i++)
{
    for(int j=0; j<3;j++)
        delete[] p[i][j];
    delete[] p[i];
}
delete[] p;
p=nullptr;
```

Date:

Week 5 : Lecture 1

- **C strings**

> Anything in double quotes
> It is always null terminated

Relational Operators can be performed well with char pointers as well.
eg: if(*cptr1 > *cptr2)

> char ch = 'A';
cout << &ch; ⇒ A.... junk value becuz program considers it as an array and finds null.
char *cptr = &A;
cout << cptr; → A..... junk
cout << &cptr; → correct address

- **Character arrays & C-strings**

```
char *cptr = new char('#');
int size = 10;
char *carr = new char[size];
for(int i=0 ; i<size ; i++)
    carr = 'A' + i ;
for(int i=0 ; i<size ; i++)
    cout << carr[i] << " ";
delete []carr;
delete cptr;
cout << carr << endl;
```

This doesn't add null at the end.

Output:
A B C D E F G H I J

Output: ABCDEFGHIJ.... till null is encountered
↳ To avoid this, add null at the end.

↳ Remedy:
carr[10] = '\0'; (if size is also 11)

* Aggregate only works for character array input and output
↓
This gives data and not addresses.

Date: _____

`cin >> carr;` → Taking input from user adds null character at the end.
This take input till space /tab/enter is encountered

To resolve this issue
`cin.getline (carr, 12)` ⟹ This automatically adds null at the end.

- size of ( ) → gives no of bytes
  strlen ( ) → function for cstrings

- carr++;
  `delete [] carr[1];` → delete needs valid base address of allocated memory.
  ⇒ Trigger exception
  Incase of arrays give valid base addresses of alloced heap memory.

- `cout << (carr+1);` Output: BCDEFGHIJ..... null encountered
- `cout << *(carr+1);` Output: B (carr[1])

Cstrings:

Character Arrays: Container to Store Characters.
Cstrings are immutable (cannot be modified).
⇒ Cstrings are constant. Cstrings are only like `carr = "Abeeda"`

character arrays → single char
→ char array
→ cstring

According to miss, all processing that we have done above of carr is a character array and not cstrings

`carr = "Abeeda";`

* Cannot deallocate cstrings: They act as static memory
* Cannot input in cstrings: `cin>>carr;` ⟶ Breakpoint
* Cannot modify cstring

Here miss said that Cstring is only when we define sth like carr = "Abeeda" or cout << "Lala"

* char *carr = "Fatimah	
carr[3] = 's';	Invalid
cin >> carr;	Invalid
delete []carr;	Invalid
carr = "beautiful";	Valid

Date:

⇒ Can copy in dynamic array also (Just like we were initializing cars initially)

cstring is a constant message → but not dynamic

- const char * cptr = new char('#')
  - ↳ cannot be changed.
  - ↳ ~~cptr will always point to the same location~~
  - ↳ cptr can alter locations as well

⇒ 2D Arrays: ⇒ however content (i.e., #) can't be changed.

```
int size = 5;
char ** dp = new char* [size];
for (int i=0 ; i<size ; i++)
 {
   dp[i] = new char [size+1];
   for (int j=0 ; j<size ; j++)
      dp[i][j] = '0' + j;
   dp[i][size] = '\0';
 }
```

DEALLOCATION:

```
for (int i=0 ; i<size ; i++)
      delete [] dp[i];
delete [] dp;
dp = null ptr;
```

Date:

Mutators:

- read data members
- write data members

⇒ By default, all functions are mutators

Accessor:

- read data members
- can't write data members.

⇒ This requires adding const after function name

⇒ It is better to make getters as accessors

eg:
```
int getX() const
{
    return x;
}
```

☆ Whenever an object of a class is created, it is not initiallized.

Constructors:

- used to initiallize data at the time of creation
- name is same as class name
- they have no return type
- we don't call them, system does automatically.

⇒ Copy constructor is a const constructor who's passed by refrence.