char Arrays

char ch[10];

'a'	'b'	'1'	'2'	'+'	'-'	'&'	'7'	'c'	
0	1	2	3	4	5	6	7	8	9

size → 1 byte

$-128 \to 127 \Rightarrow -2^7 \to 2^7 - 1 \Rightarrow 0 \to 2^8 - 1$

char ch[100];

cin>>ch

single character

cin >> ch[2];

int x;

cin>>x; → arr[0], arr[1], arr[2]

int x[10];

~~cout << arr[2]; x;~~

char name[100];

N	o	o	r	\0	- - - - -			
0	1	2	3	4	. . . . .	92	99	100

cin >> name;

"Noor"

↳ by default ↳ null character is placed at the end of input '\0' → ASCII value → 0.

↳ terminate string.

char ch[10];
char ch[100];
char ch[1000];

Space Complexity : O(1)

read characters only if not see

cin → space ' '
→ tab \t
→ Enter, newline \n

For char

char array[20];
cin.getline(array,20);
cout << array;

For string

string array;
getline(cin, array);
cout << array;

String

String:

→ datatype
→ represented as assays of bytes (or words) that store a sequence of characters.
→ Array of characters.
→ Terminated with a special character '\0'.
→ distinct data type.
→ Own set of operations & properties.
→ can be declared and manipulated using specific string-related functions & methods.

Below are some fundamental operations

- **Concatenation** (Append)
- **Length**
- **Access**
- **Substring**
- **Comparison** (compare)
- **Search** (find)
- **Replace**
- **Addition** (last character) push_back('a'); will be ↓ last character
- **Removal** (last character) pop_back();

Syntax:

```
string str;  // declaration
string name = "Blue Crystal";  /* Initialization at the time of declaration */
cout << name;  // print string "name"
```

Applications:

1. **Text Processing:** Text processing tasks such as searching, manipulating & analyzing textual data.
2. **Data Representation:** Fundamental for representing and manipulating data in formats like JSON, XM & CSV.
3. **Encryption & Hashing:** Commonly used in encryption & hashing algorithms to secure sensitive data & ensure data integrity.
4. **Database Operation:** Essential for working with databases, including storing and query text-based data.
5. **Web Development:** Utilized in web dev for constructing URLs, handling form data, processing input from web forms and generating dynamic content.

Function:

stoi( );

→ converts string to integer.

sort( ); T.C = O(n logn).

→ STL function used for sorting

* Pointer :

int a = 5

Data structure

Symbol-table
a → 104
mapping

storage location → 06X9F → 5 (a)

↓ address

& operator for address

int a = 5;
value 5 is stored in integer data type

int* ptr = &a;
ptr is a pointer to integer data.

pointer → address
xyz

pointer points to the address of the variable

syntax/ pointer creation — dereference operator — address of operator

data type = int* p = &a;

p → variable name

&a → variable to which pointer is pointing to.

char* ptr1 = &b;
pointer to character data

Pointer variable stores the address of a variable it is pointing to. int*

cout << p; → displays address e.g 4X7eff9 — it is same as &variablename.

cout << *ptr;
↓ Dereferencing
displaying value e.g: 5

sizeof(p)

It will remain same for all datatypes either 4 or 8 depending on system 32-bit or 64-bit or compiler.

Pointer only points to the address of the variable and address is same for every data type

Why Pointer?

→ Dynamically memory allocation.
→ ~~Dyn~~ to pass functions to other functions.
→ to iterate over elements in arrays or other data structures.
→ Memory management

Bad Practice

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

Null pointer

```
int* ptr = 0;
cout << *ptr;
```

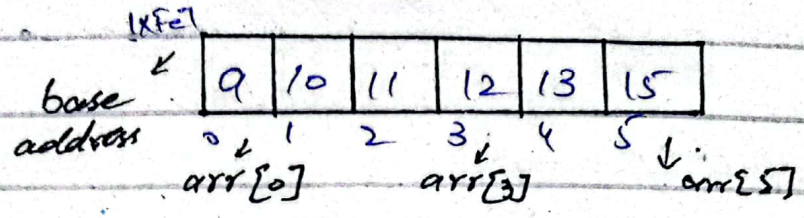
Copy pointer

```
int i = 18
int *ptr = &i
int* secondptr = ptr;
cout << *ptr;
cout << *secondptr;
```

CamScanner

Pointer & Array:

int arr[6];

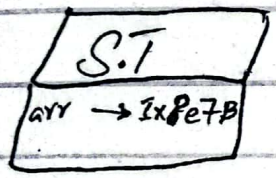


for address

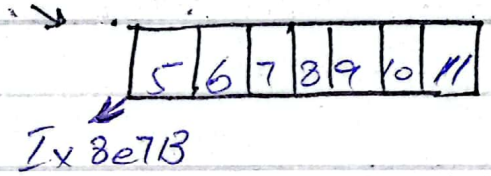
&arr[0]

arr[0] → 9

Base Address
- &arr[0] → 1xFe7
- arr → 1xFe7
- &arr → 1xFe7



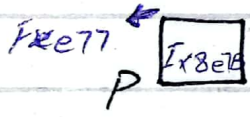
arr[7] →



int *p = arr;

p → Fxe77

arr → 1x 8e7B



B.A ← &arr → 1x8e7B

&p → 1x8e7B.

*p ⇒

*arr → 5

*arr + 2 → 8

*(arr + 1) → 6

arr is the pointer to the first location.

arr[i] = *(arr + i) = i[arr] = *(i+arr)

No error

Reference = Address

Dereference = Value of address.

int arr[10];

1 int takes 4 bytes.

10×4 = 40 bytes.

int *p = arr;

size = 4/8

Address

cout << arr+1; → 108 // let arr = 104

~~arr~~

int * p = arr+1 → 108

arr = arr+1; → Error: Can't be update

constant pointer

Pointer & char Array

char ch = "Aversion";

char* c = ch;

Output

cout << ch; // Aversion

cout << &ch; // 0x106

cout << ch[0]; // A

cout << c; // Aversion → C → Address = entire string

cout << &c; // 0x100

cout << *c; // A → *c = *(c+0) = c[0] = 0th index

cout << *c + 2; // 67 ∴ ASCII of A : 65, 65+2 = 67

Pointer with function:

Displaying size of array

in main	in function
{	solve(int arr[])
int arr[10] = {1,2,3,4} 4x10	{
sizeof(arr); → 40	sizeof(arr); → 4
solve(arr);	

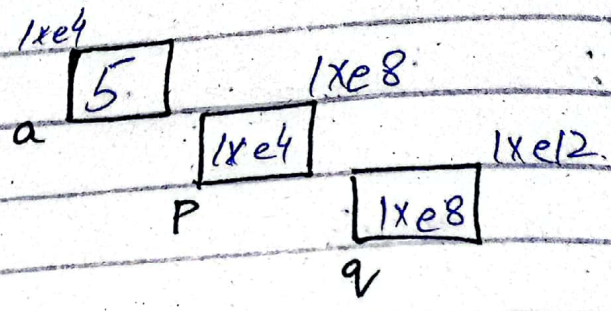
When you pass an array to a function it decays into a pointer to its first element. Function doesn't actually receives an array; it receives a pointer. So sizeof(arr) gives the size of pointer not array which is usually 4 or 8 bytes depending on the system architecture

Double Pointer:

pointer to integer data

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

is a pointer to int* data



a → 5	q → 1xe8
&a → 1xe4	&q → 1xe12
p → 1xe4	*q → 1xe4
&p → 1xe8	**q → 5
*p → 5	

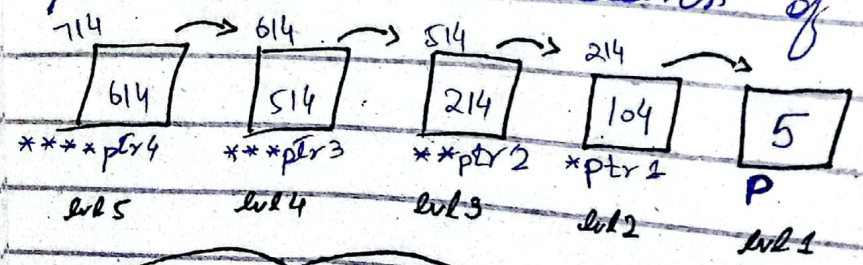
****ptr → value present at location stored in ****ptr.

***ptr → value at loc sto in ***ptr.

**ptr → value at l.s in *ptr.

*ptr → value stored at l.s in ptr.

ptr → stores the address of a value of variab



Reference Variable

Same Memory Location → but → different name

Reference Variable

```
int a = 5;
int& b = a;
```

b is a reference variable & pointing to the same memory location

104 [5]

S.T

a → 104
a → 104

Pass by Value

```
main()
{
    int a = 5;
    solve(a);
    cout << a;
}
```

[5] a

5

```
solve(int b)
{
    b++;
}
```

copy create [5] 6 b

Pass by Reference

```
main()
{
    int a = 5;
    solve(a);
    cout << a;
}
```

b [5] 6 a

6

```
solve(int& b)
{
    b++;
}
```

Valid pointer

Memory allocated (using new or malloc)

Addr of var (int* ptr = &var).

Mem within an array or other DS.

Invalid pointer

NULL or 0, doesn't point to any mem

Points to deallocated mem using delete

Points outside the bounds of an array.

* Recursion *

→ process of solving a problem by breaking down it into smaller, simpler sub-problems.

Recursion

← when bigger problem depends small & same type of problem.

→ when a function calls itself until a given condition is satisfied.

```
void Solve()
{
    ....
    ....
    solve();
}
```

Key Concepts:

1. Base Case: condition that stops the recursion.
2. Recursive case: Part of function calls itself with a smaller input. (Recursive relation)
3. Processing: optional

Example 1:

Factorial of n ('n!'): The product of all positive integers upto 'n'.

Recursive Definition:

- factorial(1) = 1 → Base case
- factorial(n) = n* factorial(n-1) → Recursive case.

Code:

```
int factorial(int n){
    if (n<=1) return 1;        // Base Case
    return n* factorial(n-1);  // Recursive Case
}
```

Example 2:

$2^5 \rightarrow 32$

B.P → 2^5

$2 \times \boxed{2^4}$

$2^5 = 2 \times 2^4$

$f(n) = 2^n$

$f(n-1) = 2^{n-1}$

$2^n = 2 \times 2^{n-1}$

$f(n) = 2 \times f(n-1)$

Why use Recursion?

- Simplifies code for problems that naturally fit a recursive solution.
- Less code compared to iterative approaches for certain problems.
- Ideal for recursive data structures like trees.

Drawbacks:

- Performance Overhead: Multiple function calls increase memory usage.
- Risk of Stack Overflow: Too many recursive calls can exhaust the stack.

Recursion vs Iteration:

- Recursion: More intuitive for problems like tree traversal, sorting (quicksort, mergesort) and problems that can be defined in terms of smaller subproblems.
- Iteration: Preferred when performance and memory usage are critical.

Diagram: Recursive flow for 'factorial(3)'

Formula: $f(n) = n \times f(n-1)$.

factorial(3) 18

|

3 * factorial(2) 6

|

2 * factorial(1) 2

|

1 (Base class)

Tail Recursion

```
void fun() {
    // B.C  ---> Constant
    // Processing
    // R.R  Recursive call made after any other processing
}
```

Head Recursion

```
void fun() {
    // B.C  Recursive call
    // R.R  is made before any other processing
    // Processing
}
```

Time & Space Complexity of Recursive Solutions.

```
fun(n) {
    if(n==0) return;
    // Processing
    int a,b,c;
    fun(n-1)
}
```

In System

fun(0)
⋮
fun(n-1)
fun(n)
main()

```
void printArr(int a[], int n) {
    if (n==0) return;
    cout << *a << " ";
    printArr(a+1, n-1);
}
```

m bytes allocated

Stack : RAM

Recursive f^n

$T(n) = k + T(n-1)$

$T(n-1) = k + T(n-2)$

$\vdots$

$T(1) = k + T(0)$

$T(0) = k$

$T(n) = nk + k_1$

$T(n) = O(n)$

$PA(n) \to k$

$PA(n-1) - k$

$n-2 - k$

$n-3 - k$

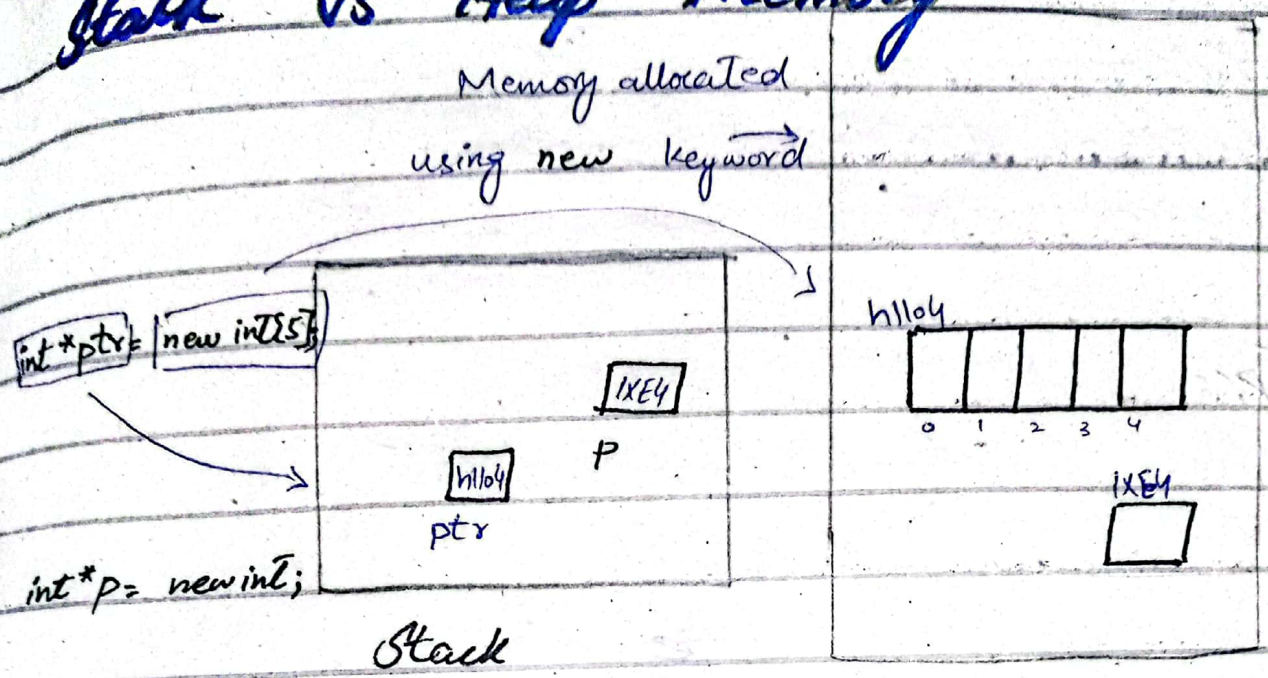
$n=0 - k_1$

nk

$T(n) = nk + k_1$

$T(n) = O(n)$

Stack vs Heap Memory



Stack	Heap
→ Static memory allocation	→ Dynamic memory allocation
→ Allocate size and location to a fixed variable	→ Suitable for allocating larger DS like arr, LL.
→ Smaller than heap	→ Larger than stack
→ Memory allocation at compile time	→ Memory allocation at run time
→ Direct access to memory address	→ Indirect access using pointer
→ Example: • int num; • float weight;	→ Example: • p = malloc(sizeof(int)); • int arr = new int[SIZE]; • int *p = new int[a];
→ Can be clean automatically	→ de-allocated using delete