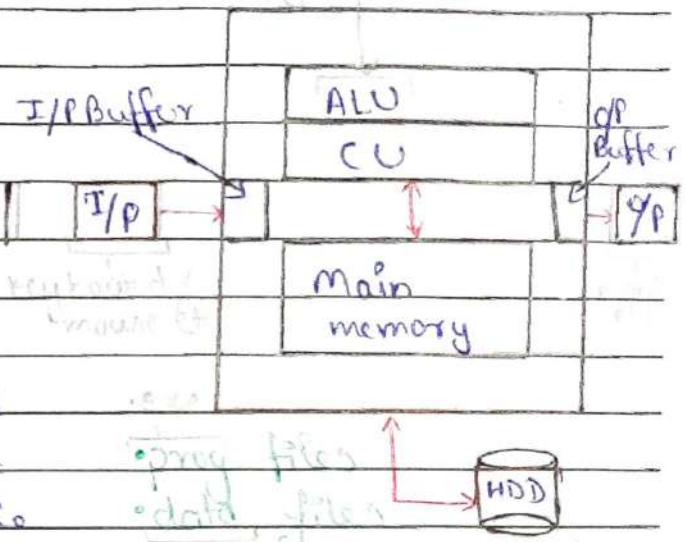
- **Essential Fundamentals:**
- How Computer Works
- Number System
- Low level vs High level language
- Compiler vs Interpreter
- Operating System
- **Program Development:**
- Programming Paradigm
- Algorithms
- Steps for program development
- **C++ Basics:**
- Skeleton of C++
- Why datatypes
- Increment Decrement Operators
- Overflow
- Bitwise Operators
- Enum & Typedef
- **Conditional Statements:**
- Logical Operators
- Short Circuit
- Dynamic Declaration
- Switch Case
- **Loops:**
- Loops-Iterative statements
- Types of loops
- **Arrays:**
- Introduction - Arrays
- Linear Search
- Binary Search
- Nested loop star patterns
- Multi dimensional array
- **Pointers:**
- Pointers - Introduction
- Why pointers
- Heap memory allocation
- Pointer arithmetic
- Problems with pointers
- Reference Variables
- **String:**
- Introduction - String
- String operations
- String class
- Functions of class string
- String class iterator
- **Functions:**
- Introduction - Function
- Function overloading
- Function template
- Default arguments
- Parameters passing
- Return methods
- Global & Local Variables
- Static Variables
- Recursive functions
- Switch Case
- **Oops:**
- Principles in OOP
- Constructors
- Types of functions in a class
- Scope resolution operator
- in-line functions
- this->pointer
- Struct vs Class
- Operator overloading
- Insertion operator overloading
- **Inheritance**
- Constructors in inheritance
- isA & hasA
- Access specifiers
- Types of inheritance
- Generalization vs Specialization
- **Base class pointer Derived class object**
- **Polymorphism**
- Function overriding
- Virtual functions
- Runtime polymorphism
- Abstract classes
- Friend & Static members
- Inner/Nested Classes
- **Exception Handling**
- Types of error
- Exception handling construct
- **Template Functions and Classes**
- **Const, Preprocessor, directives and Namespaces**
- Constant qualifiers
- Preprocessor
- Namespaces
- **Destructors**
- Introduction - Destructors
- Virtual destructors
- **Streams**
- Writing a file
- Reading a file
- **STL**
- What is STL
- STL - Main Components
- STL Classes
-
- **C++ 11**
- Auto keyword
- Final keyword
- Lambda Expressions
- Smart Pointers
- Ellipsis

Date 5/5/2023 Page No:

* How computer works

Language of the computer is Binary.



ALU: stands for Arithemetic and Logical Unit (+, -, ÷, x)

CU: stands for control unit, performs task of controlling funcs of the computer

Main memory: Random Access memory is the main memory of the computer. Applications are supposed to be present in main memory for them to run.

Note:

Input is first stored in input buffer, and same is done with the output

* Number system

There are 4 types of number system

Decimal	Binary	Octal	Hexadecimal
Base 10 (0-9)	Base 2 (0, 1)	Base 8 (0-7)	Base 16 (0-9, A-F)

ABHINAV

Date ___/___/___ Page No.: _____

Decimal	Binary	Octal	Hexa decimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

→ Conversion from decimal to binary

Decimal to Binary

2	25
2	12 – 1
2	6 – 0
2	3 – 0
2	1 – 1
	0 – 1

$25_{(10)} = 11001_{(2)}$

Binary to Decimal

$11001 \rightarrow$

1	1	0	0	1
2^4	2^3	2^2	2^1	2^0

$\Rightarrow 2^4 \times 1 + 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 0 + 2^0 \times 1$

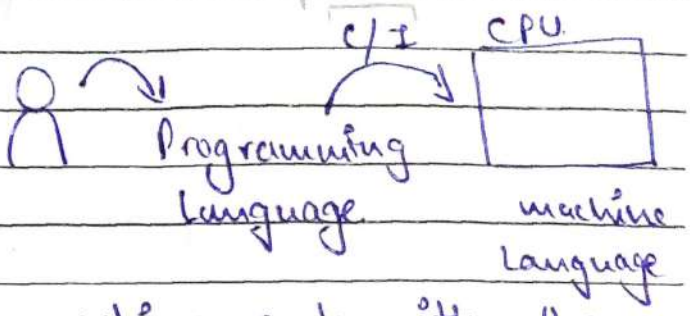
~~$\Rightarrow 32 + 8 +$~~

$\Rightarrow 16 + 8 + 1 \Rightarrow 25$

ABHINAV

Date ___/___/_____ Page No.:_____

What is a Program



We communicate with the machine using a programming language, which then converts our code into machine code with the help of a compiler.

Low-Level vs High-Level Language

	Low level language	High level language
→	Direct memory management	they are interpreted.
→	have open classes and message style methods	
→	Much fast than high level	Poor performance
→	Superb performance but hard to write	Flexible syntax and easy to read.
→	Few support and hard to learn	Object oriented and functional
		Large community
→	Statement corresponds directly to clock cycles	Codes are concise
→	examples	→ examples
	Machine Language	Python
	Assembly Language	C++
		Java

ABHINAV

Date __/__/____ Page No.:____

Compiler vs Interpreter

Compiler	Interpreter
→ Takes entire program as input	Takes single instruction as input
→ Intermediate object code is generated	No intermediate object is generated
→ Executes faster	Executes slower
→ more memory required	less memory required
→ Program need not to be compiled everytime	program is translated everytime
→ errors are displayed after entire program is checked	errors are displayed for every instruction interpreted.
→ C, C++	Python.

Operating Systems

→ a master ~~problem~~ program, after being initially loaded into the computer by a boot program, it manages all of the other applications.

→ Examples of operating system include Android OS, Windows OS, Linux, etc.

→ OS manages memory, processes, handles input and output and controls peripheral devices like disk drives and printers

ABHINAV

Date __/__/____ Page No.:____

pattern / model

Programming Paradigm

1. Monolithic Programming
2. Modular / Procedural
3. Object Oriented
4. Aspect - oriented / component assembly

1. Monolithic Programming:

- whole program is written in a single block
- uses all data as global data which leads to data insecurity
- no flow control statements like if, switch, for and while
- no concept of data types
- example - Assembly language, BASIC

2. Procedural Programming:

- uses modern concepts where program is divided into smaller modules
- it provides the concept of code reusability
- introduced with the concept of data types
- provides flow control statements that provide more control to user.
- functions may transform data from one form to another
- example - C, FORTRAN, Visual Basic

ABHINAV

Date __/__/____ Page No.:____

3. Object Oriented Programming:
- program is created on the concept of objects
- object may communicate with each other through function
- follows the bottom-up execution
- everything belongs to objects.
- example - C++, Python, Java.

4. Aspect-oriented / component assembly:
- enables modularization of cross-cutting concerns.
- example - C++, Smalltalk, C#

Algorithm → procedure
↳ step by step for solving a computional program

{ Algorithm were created even before programs even existed. but programs were developed afterwards }

algorithm → is something we write first before converting it into a program that our machines can understand.

Pseudo code: something which can be written in any language that is used for understanding algorithms.

ABHINAV

Date ___/___/_____ Page No.:_____

Example

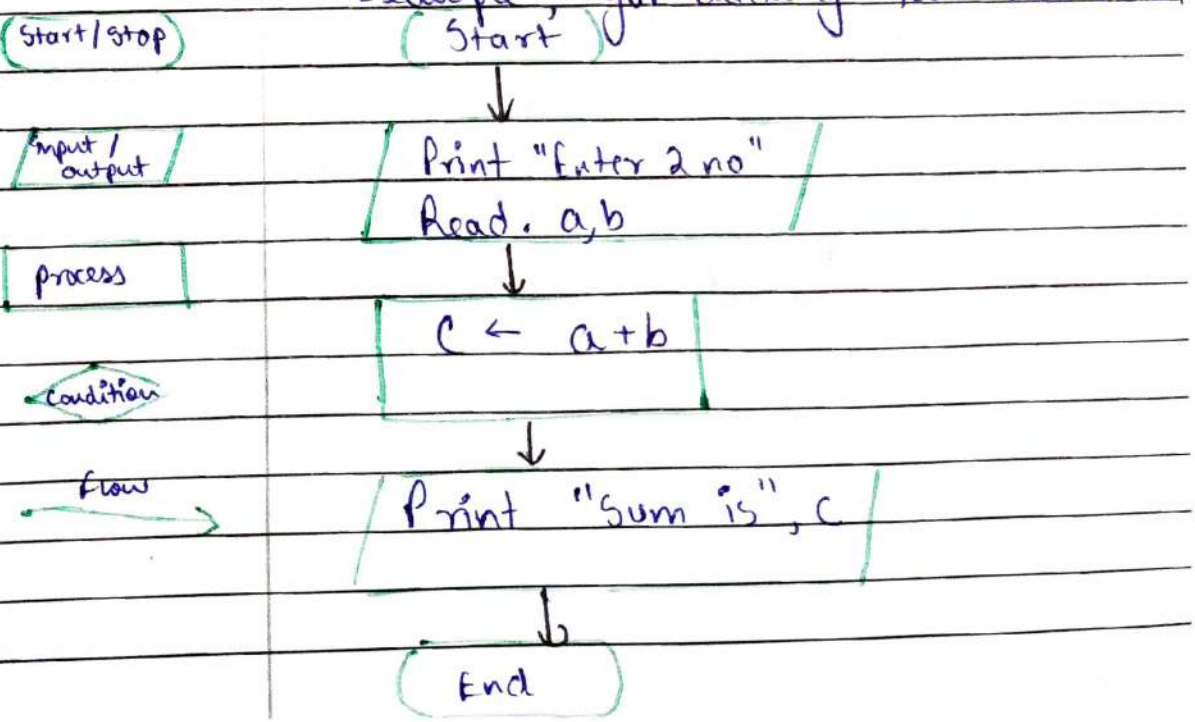
```
Algorithm average (List, n)
{
Sum <- 0;
for each element x in list
Begin
Sum <- Sum + x.
End;
Avg <- Sum/n;
Return avg;
}
```

What is flowchart

- used to show flow of control of a program
- used to understand program and its working
- take some input, process it and give output

Elements of flowchart are →

Example, for adding two numbers



ABHINAV

Date / /

Page No.:

Steps for Program Development and Execution

Main steps involved in the development and execution of a program are :

1. **Editing :**
   ↳ can be done on any text editor/IDE
2. **Compiling :**
   ↳ converts the source code into a machine code if there are no errors in the program
3. **Linking Libraries :**
   ↳ for various operations build in func./classes are available in C++ that are present in the header files supported by libraries where machine code is readily available to use
4. **Loading :**
   ↳ bringing the program into the main memory for its execution
5. **Execution :**
   ↳ once the code is in the main memory, OS will ask the CPU to execute the program thus the execution process starts.

ABHINAV

Date __/__/____ Page No.: _____

Skeleton of C++ program

```
#include <iostream>
int main()
{
std::cout << "Hello World";
   return 0;
}
```

scope resolution — insertion operator

Q- int main() vs void main()
some compilers support void main and some int main. Mostly int main() is used

Q- Why return 0;
↳ this means when a program is ending it should return 0
↳ It is like a standard in C++ programs, it must be written
↳ return 0; means program has terminated successfully.

Q- Why std
→ std is a namespace, all built in functions and object are included in namespace
↳ cout is also a namespace
and can be used by two ways →
1. std :: cout
2. using namespace std; then simply write cout

ABHINAV

Date ___/___/_____ Page No.:_____

Q- What is #include
↳ used as a preprocessor directive
↳ asks the compiler to include the header file

C++ Basics

Code to print user's name

```
#include <string>
#include <iostream>
using namespace std;

int main()
{
    string name
    cout << "Enter your name";
    getline (cin, name);

    cout << "Hello " << name << "!";
    return 0;
}
```

Note:
to read multiple words as input we use getline which can be used by including <string>.

Date ___/___/_____ Page No.: _____

Why data types

↳ on data we perform required computation -al operations

↳ Data is mainly of two types.

1. Numeric
2. Character / Alphabetic

1. Numeric:

↳ represented in binary form to the machine.

↳ they are of two types.

(A) Integer

↳ numbers without decimal

eg → 2, 1, 56

(B) Float

↳ numbers with decimal are called float numbers

eg → 12.6, 17.5, 2.1

2. Character / Alphabetic:

↳ collection of characters to form a single entity is called strings

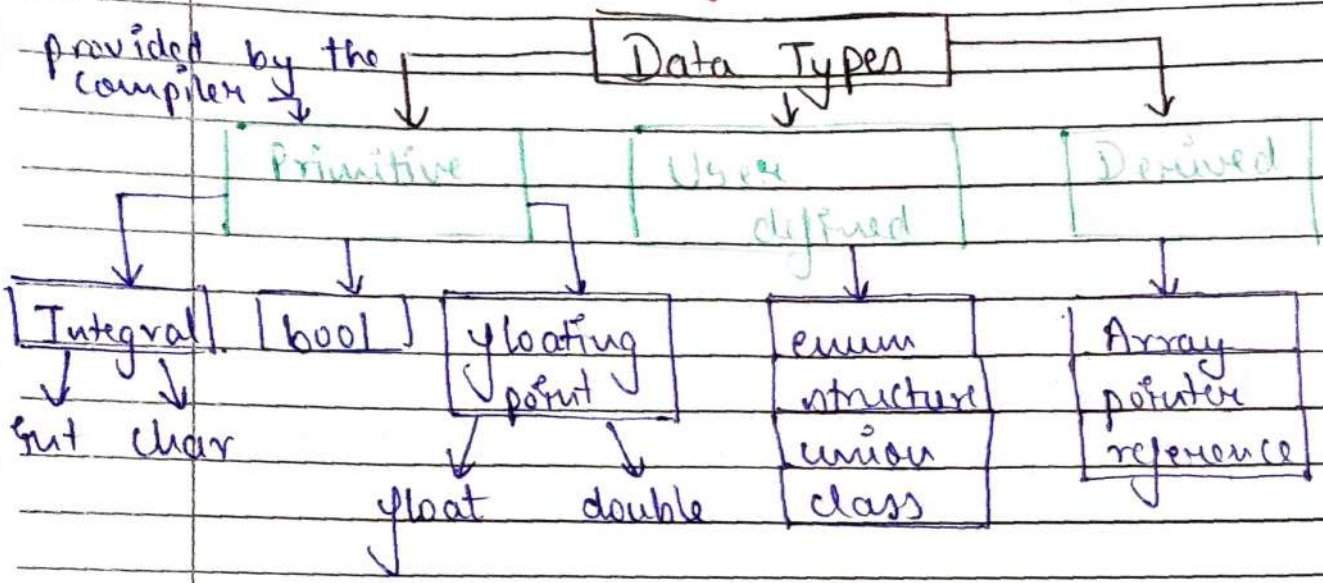
↳ can be a name, place or any word from dictionary

↳ C++ supports both single character or collections of characters

↳ represented in binary form through ASCII code.

Date __/__/____ Page No.:____

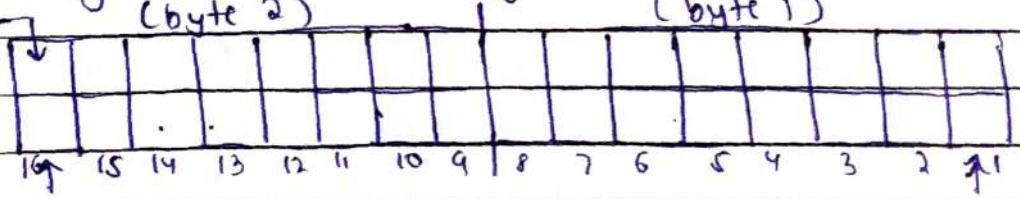
Primitive Data types



Data Type	Size	Range
int	2 or 4	-32768 to 32767
float	4	-3.4×10^{-38} to 3.4×10^{-38}
double	8	-1.7×10^{308} to 1.7×10^{308}
char	1	-128 to 127
bool	undefo.	true / false.

☆ Reason for integer data type range

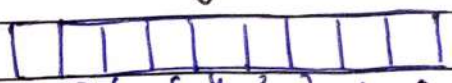
integer has 2 byte → 16 bits



ABHINAV

Date __/__/____ Page No.:____

✱ Reason for character range



$2^7 = 128$

∴ -127 to 128

→ ASCII

A - 65	a - 97	0 - 48
B - 66	b - 98	1 - 49
⋮	⋮	⋮
Z - 90	z - 122	9 - 57

✱ Modifiers → using modifiers we can modify the data types as per our reqs.

1. unsigned int
   range → 0 - 65535 (double of int)
   unsigned char
   range → 0 - 255 (double of char)
2. long int
   range int → 4 bytes (double)
   8 bytes
   long double
   range → 10 bytes

Variables

↳ names given to datatypes
↳ place where we store the data in our program
↳ variables are assigned depending on the dtype.

ABHINAV

Date / /

Page No.:

→ declaration
int rollno; // declaration
rollno = 10; // initialization

↳ variables will occupy the space in memory during runtime

↳ every character type variable should be enclosed in ' ' for instance
Char group = 'A';

↳ decimals numbers are stored in float datatype
float price = 12.75f;
if this is not written then the no. is considered as a double

~~Anthem~~ Arithmetic Operators & Expressions

Types of operators

	returns new value
Arithmetic	+, -, *, /, %
Relational	<, <=, >, >=, ==, !=
Logical	&&, ||, !
Bitwise	&, |, ~, ^
Increment / Decrement	++, --
Assignment	=

ABHINAV

Date __/__/____ Page No.:_____

Note

* Even if the result is in decimal, we will get an integer as in return if the declared variable is of type integer.

* To get the exact answer, we need to do type casting

```
int a=13, b=5;
float c;

c=(float)(a/b);
```

* Mod operation cannot be operated on floating point no.
  ↳ they can only be operated on integers and characters.

Operator Precedence and Expressions

Operator	Precedence
()	3
*, /, %	2
+, -	1

Basic formulas in C++

1. Area of triangle:
   A = 0.5 * (b * h)
2. Perimeter of rectangle:
   P = 2 * (l + b)
3. Sum of n terms:
   S = n*(n+1)/2
4. nth terms of AP series:
   t = a + (n-1) * d

ABHINAV

Date ___/___/_____ Page No.:_____

Note

To use square root fn C++, we need to #include <cmath> which has the function sqrt().

Compound Assignment

- related to arithmetic and as well as other operators as well.
- some of the compound assignment are listed below.

+=	&=
-=	|=
*=	<<=
/=	>>=
%=	

Example:

sum = sum + a;

can also be written as →

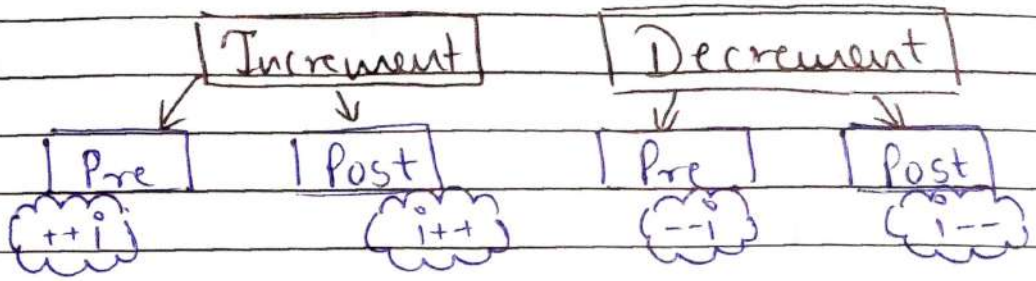
sum += a;

benefits →

- readable
- faster
- gives more meaning

Date __/__/____ Page No.:____

Increment and Decrement operators



→ these variables are useful for counting as they are very common so C++ has introduced this concept.

→ mostly used in loops

→ ~~prem~~.

~~pre → first increment then the value~~

pre → first increment the value, then assign the result to the variable

post → first assign value and then increment.

Overflow

- When the value is more than the capacity, it will take the values again from the ~~begg~~ beginning, this is called overflow.
- Suppose there is a byte and we are already having values in it, if 1 is added to this max byte value it becomes a signed bit.
  ∴ it becomes -128 magically, this happens when we try going to the next value.

ABHINAV

Date / /

Page No.:

Bitwise Operators

↳ these operators are performed on the bits of the data and not as the whole data unit

&	|	^	~	<<	>>
and	or	xor	not	left shift	right shift

AND truth table

bit 1	bit 2	bit1 & bit2
0	0	0
1	0	0
0	1	0
1	1	1

{ these tables are used when developing, device drivers, system programs, core system programs & in general if a person is working more close towards electronic }

OR truth table

bit 1	bit 2	bit1 or bit2
0	0	0
1	0	1
0	1	1
1	1	1

XOR truth table

bit 1	bit 2	bit1 ^ bit2
0	0	0
1	0	1
0	1	1
1	1	0

ABHINAV

Date ___/___/_____ Page No. _____

Left and Right Shift

```
int x = 5 , y ;
```

x = 0 0 0 0 0 1 0 1

0 0 0 0 0 1 0 1 ←

↳ when you shift all the bits on the left hand side by one space then 5 will get multiplied by 2, if you move them by 2 space, it will be multiplied by 4

↳ Here signed bit is not included. if the number is positive then it will be positive and vice versa.

Enum and Typedef

Enum

↳ aka enumerated datatype is used to define user defined datatype

↳ with the help of existing datatype we can define our own data type

↳ To make the work faster, codes are given to the words, it is a common practise among us all as well

↳ If there a lot of words, then they can be grouped together under one name that is enum.

ABHINAV

Date __/__/____ Page No.:____

↳ method of defining enum →

enum days { mon, tue, wed, thu, fri, sat, sun };
0 1 2 3 4 5 6

Typedef
↳ used to give meaningful names to the variables typedef
↳ also used to define user define datatype
↳ used for better readability of a program

eg →
typedef int marks;
int main()
{
marks m1, m2, m3;
}

Conditional Statement

Syntax

if (<condition>)
{
}
else {
}

→ statement is false if value is 0, any other value is True
→ relational operators used (<, >, <=, >=, ==, !=)

ABHINAV

Date ___/___/_____ Page No. _____

Example

```
#include <iostream>
using namespace std;

int main()
{
    int x, y;
    cout << "Enter two numbers";
    cin >> x >> y;
    if (x < y) {
     cout << y << "is greater"; }
    else {
     cout << x << " is greater"; }

    return 0;
}
```

Logical Operators

↳ if we have more than one conditional statements and we want to join them, we can do it using compound conditional statement

↳ Such operations can be done using logical operators

Logical Operators →

&&	and	used to make compound conditional statement
||	or	
!	not	used for negation, i.e, makes true statement false.

ABHINAV

Date __/__/____ Page No.:____

// Truth tables are given in our previous page

Short Circuit

→ Short circuiting occurs while evaluating '&&' (AND) and '||' (OR) logical operators

→ while evaluating '&&' operator if the left-hand side of '&&' gives false, then the expression will always yield false, because checking right hand side won't make any sense

→ Similarly, in the case of OR '||', operation when the left hand side gives 'true', the result of the expression will always be true irrespective of the value on the right hand side.

Dynamic Declaration

↳ allows one to create variables/objects at runtime and allocate memory for them as needed. This

↳ allows one to manage memory more efficiently

eg →

```
int a=10, b=5;
if (true){
    int c = a+b;
    cout << c << endl; }
return 0;
```

c variable is limited to the block in which they're declared

ABHINAV

Date ___/___/_____ Page No.: _____

Switch Case

- ↳ branch and control statement
- ↳ switch can have 0 or more cases
- ↳ each case is defined with a label
- ↳ if case block is not available, then the default block is executed
- ↳ default block is optional
- ↳ every case block must terminate with break, or else the program would execute the next case
- ↳ eg → switch is used in menu-driven programs.

Syntax

```
int main()
{
    int day;
    cin >> day;
    switch (day) {
        case 1 : cout << "mon";
        break;
        case 2 : cout << "tue";
        break;
        case 3 : cout << "wed";
        break;
        case 4 : cout << "thu";
        break;
        case 5 : cout << "fri";
        break; }

    return 0;
}
```

ABHINAV

Date __/__/____ Page No.:____

Loops - Iterative Statements

↳ used to execute a block as long as specified condition is true

Loops are of 4 types in C++

1. while
2. do while
3. for
4. for each

1. while loop

↳ loop loops through a block of code as long as specified condition is true.

Syntax

```
while (<condition>)
  {
    //code
    i++
  }
```

2. do-while loop

↳ variant of while loop

↳ loop will execute the code once, before checking if the condition is true or not, then it will repeat the loop

Syntax

```
do {
    // code to be executed.
   }
while (condition);
```

ABHINAV

Date __/__/____ Page No ____

3. for loop

↳ generally used when we know how many times the loop has want to to iterate

Syntax initialization condition updation

```
for (statement 1, statement 2, statement 3) {
    // code of block to be executed.
}
```

Rev algorithm:

```
while (num % 10 > 0)
{
    cout << num % 10;
    num /=10;
}
```

this algorithm returns the reverse of a digit by simply dividing the number by 10 and the remainders together

Arrays

- collection of similar data elements under one name, each element is accessing using its index
- memory of an array is allocated contiguously
- for loop is used for accessing array
- n-dimension arrays are supported by C++
- 2D Arrays are used for matrices
- can be created on stack or heap section of memory

ABHINAV

Date ___/___/_____ Page No.:_____

Syntax

```
int A[5] = {1,2,3,4,5}
```

can be char, float (single quote are imp)

```
char C[5] = {'A','B','P','Q','S'}
```

→ create array without specifying size

```
int B[] = {2,4,5,6}
```

2	4	5	6

→ accessing each value in an array

```
int my_array[5] = {1,2,3,4,5}
cout <<
for (int i=0; i<5; i++)
{
    cout << my_array[i] << endl;
}
```

Note: 0 value is added if the size of an array is less than specified

```
int C[5] = {1,2,3}
```

this array will have

1	2	3	0	0	value
0	1	2	3	4	index

=>

ABHINAV

Date __/__/____ Page No: _____

we can also iterate through each value using a for - each loop

```
int main()
{
    float A[] = {2.5f, 5.6f, 9, 8, 7};
    for (int x : A)
        { cout << x << endl; }

    return 0;
}
```

use auto for automatic type cast declaration.

→ output

```
2
5
9
8
7
```

→ 5.6 not used given in loop

Note to change the value of an array

```
int main ()
{
    float A[] = {2.5f, 5.6f, 9, 8, 7};
    for (int x : A)
        { cout << ++x; }
    return 0;
}
```

if 8 wouldn't have been there, then simply a copy of the element would have been edited and printed

ABHINAV

Date / /

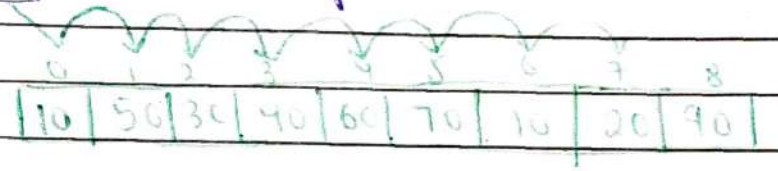
Page No.:

Linear Search

↳ process
↳ ~~method~~ to find a location of a element

Linear Search is defined as sequential search algorithm, starts at one end and goes through each element of a list until the desired element is found., otherwise the search continues till the end of the data set.

[Find '20']

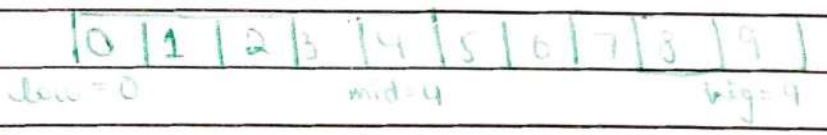


Binary Search

Binary search is used in a sorted array by repeatedly ~~divided~~ dividing the search interval in half

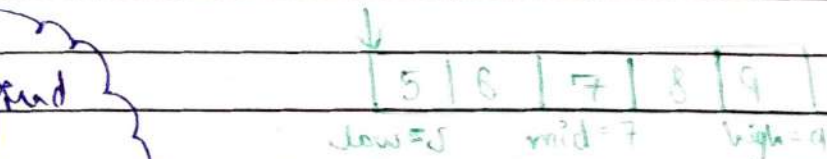
→ data set should be sorted

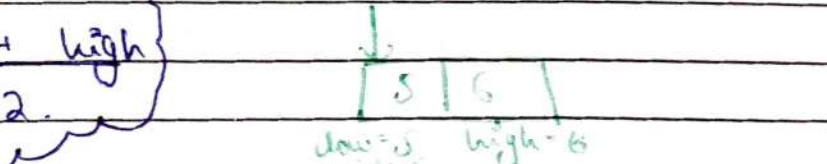
→ [Find 5]

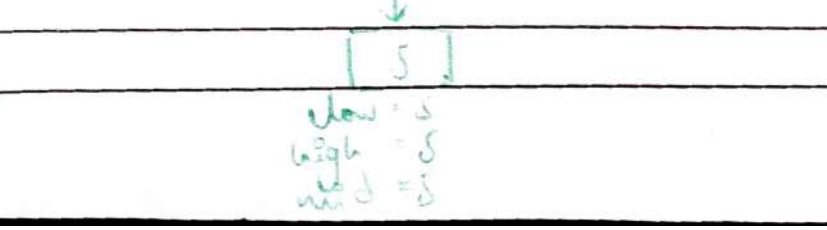


formula to find mid →

$$mid = \frac{low + high}{2}$$







ABHINAV

Date ___/___/_____ Page No.: _____

Example →

```
while (l <= h)
{
    mid = (l+h) / 2;
    if (num == A[mid])
    {
        cout << "found at : " << mid;
        return 0;
    }
    else if (num < A[mid])
        h = mid-1;
    else
        l = mid+1;
    }
cout << "Element not found";
```

ABHINAV

Date ___/___/___ Page No.: _____

Nested Loop Star patterns

1.
```
*
* *
* * *
* * * *
```

```
for (int i=0; i<4; i++)
{
    for (int j=0; j<4; j++)
    {
        if (i>=j)
            cout << "*";
    }
    cout << endl;
}
```

2.
```
* * * *
* * *
* *
*
```

```
for (int i=0; i<4; i++)
{
    for (int j=0; j<4; j++)
    {
        if (i>=j)
            cout << " ";
        else
            cout << "*";
    }
    cout << endl;
}
```

3.
```
   *
  * *
 * * *
* * * *
```

```
for (int i=0; i<4; i++)
{
    for (int j=0; j<4; j++)
    {
        if (i+j > 4-1)
        {
            cout << "*";
        }
        else cout << " ";
    }
    cout << endl; }
```

ABHINAV

Date ___/___/_____ Page No.:_____

4.
```
* * * *
* * *
* *
*
```

```
for (int i=0; i<4 ; i++)
{
    for (int j=0 ; j<4 ; j++)
    {
        if (i+j > 4) cout " = " ;
        else cout " * " ;
    } cout << endl ;
}
```

* Multidimensional Arrays

↳ to declare a multidimensional array, we define the variable type, specify the name of the ~~arr~~ array followed by square brackets which specify row and columns.

Syntax

```
string letters [2][4];
letters = { { "A", "B", "C", "D" },
            { "E", "F", "G", "H" } };
```

	0	1	2	3
0	A	B	C	D
1	E	F	G	H

Accessing every element is

```
for (int i=0; i<2, i++)
    for (int j=0 ; j<3 ; j++)
    { cout << A[i][j] ; }
    cout << endl ;
```

ABHINAV

Date / /

Page No.

int a - & x
and - address

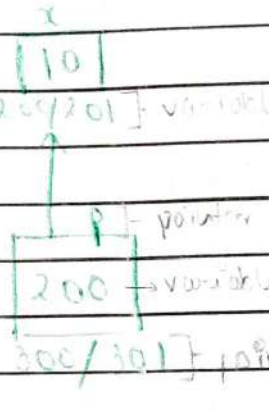
Pointers

↳ a variable that stores the memory address as its value

↳ pointer variable points to a data type. (int / string) of a same type and is created with the * operator

```
int x = 10;
int * p ; //declaration
p = &x ; //initialization

cout << x ;    - 10
cout << &x ;   - 200
cout << p ;    - 200
cout << &p ;   - 300
cout << * p ;  - 10
```



declaration initialization dereferencing	int * p ; p = &x ; cout << * p ;

Why pointers

Program can access heap with the help of only pointers.

∴ Pointers are used for accessing the heap memory

We use file pointers for accessing files

ABHINAV

Date ___/___/_____ Page No.: _____

Network, printer, mouse, monitor, etc., etc. can be accessed by our program ~~not~~ indirectly with the help of a pointer.

concluding point

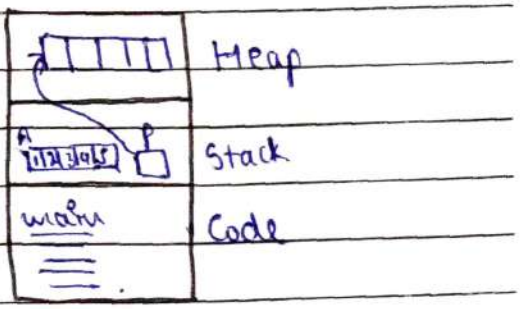
a program can directly access the contents from only two places, code section and the contents inside stack. To access elements ~~in~~ or from other places we need to use pointer.

Java, C# does not have pointers hence, they cannot access the printers, networks, etc. etc. directly with the program.

* Heap memory allocation

↳ Heap memory is accessed dynamically i.e, size of the memory required is decided at runtime and not at compiletime.

```
int A[5] = {1,2,3,4,5}
int *p;
p = new int[5];
```



Heap memory allocation in Cpp refers to the process of dynamically allocating memory from the heap which is a region of the computer's memory used for dynamic memory allocation

ABHINAV

Date __/__/____ Page No.:____

Unlike stack memory, which is automatically managed and has a limited size, heap memory allocation allows you to allocate memory at runtime and manage it manually

In C++ 'new' operator is used to allocate memory from the heap.

```
// Allocating memory for an integer on the heap
int* ptr = new int;
// use the allocated memory
*ptr = 42;
// deallocate the memory
delete ptr;
```

It's important to deallocate it using the 'delete' operator to prevent memory leaks. The delete operator frees the memory and makes it available for reuse

↳ useful when we need to allocate the memory dynamically, when we don't know the required size at the compile time

↳ also useful when we need to allocate memory with a longer lifetime than the scope of a func or a block.

* It's imp to manage heap memory carefully to avoid memory leaks or accessing deallocated memory.

ABHINAV

(tip: pointers can be used to access each element of an array)

Date / / Page No.

* Pointer Arithmetic

↳ refers to the manipulation of pointers in a programming language, typically used in low level languages like C, C++

↳ Pointer arithmetic allows us to perform arithmetic operations on pointers, to navigate through memory, accessing elements of arrays, and dynamically allocating memory

Basic pointers operations →

1. Addition :
   ↳ adding an integer value to a pointer, results in pointer being incremented by a certain number of memory locations based on the size of the data type the pointer is pointing to.

```
p++;
```

2. Subtraction :
   ↳ subtracting an integer value, results in the pointer being decremented by a certain number of memory locations based on the size of the data type.

```
p--;
```

3. `p+2;`
4. `p-2;`
5. `p-q;`

these are the operations we can use on operators

ABHINAV

Date ___/___/_____ Page No.:_____

* Problems using Pointers

- uninitialized pointers
  ↳ the pointer was never initialized
- pointer may cause memory leak.
- dangling pointers
  ↳ pointer was initialized but memory was deallocated and still trying to access it

* Reference Variables

↳ an alias or alternative name for an existing object
↳ provides a way to access the same memory location as another variable

```
int x = 10;
int & ref;
ref = x;
```

* ↳ address of main variable and the reference variable is the same

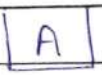
Date ___/___/_____ Page No.: _____

* String Introduction

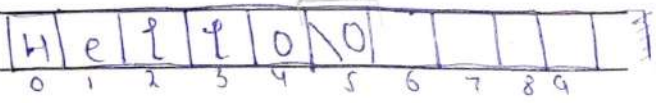
Declaring and Initialising String:

(A)

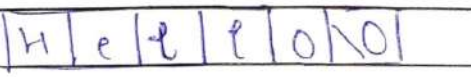
(A) char x = 'A';



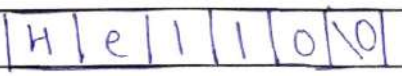
(B) char S[10] = "Hello";



(C) char S[] = "Hello";

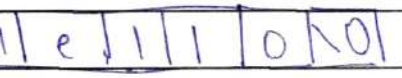


(D) char S[] = {'H', 'e', 'l', 'l', 'o', '\0'};



(E) char S[] = {65, 66, 67, 68, '\0'};
char array
pointer char ASCII code

(F) char *S = "Hello";
not supported by latest compiler



Note

cin can take only one word as the input, but we can use .get(s, 50) to take multiple word input.

var name
var size

cin.get() and cin.getline() have the same functions

cin.ignore() → to ignore values in the input buffer

ABHINAV

Date / /

Page No.

String operations

1. strlen()
(string length)
gives the length of the string
strlen(s);
string name

2. strcat()
(string concat)
combines two strings
strcat(destination, source)
→ output (destination source)

3. strncat()
combines n no. of string's characters
strncat(dest, source, num)
→ output → (dest source)
num = 4

4. strcpy()
(string copy)
~~str~~ copies source to dest.
strcpy(dest, source)

5. strncpy()
copies n charac. to dest.
strncpy(dest, source, num)
example →
strcpy("", "Neha")
this becomes Neha

6. strstr()
find the first occurrence of string 2 in string 1
strstr(main, sub);
main | A | A | I | T |
sub | A |
output → | A | I | T |

ABHINAV

Date __/__/____ Page No. _____

7. strchr( ) — used to find the occurrence of a string 2 in string 1

strchr(
strchr(main, char);
main [c|h|a|g|r|a|r|2]
char [a]
output → agrar2

8. strrchr( ) — used to find the occurrence of string 2 from right side →

strrchr(main, char);
main [c|h|a|g|r|a|r|2]
char [r]
output → [r|a|r|2]

9. strcmp( ) (string compare) — compares string 1 to string 2
- if string 1 > string 2 → output +ve
- if string 1 < string 2 → output -ve
- if strings are equal → output 0

strcmp(string 2, string 2);
string 1 [H|E|L|L|O]
string 2 [h|E|L|L|O]
output → -32 (negative because diff. in ASCII code string 1 < string 2)

10. strtol(str1) (string to long int) — converts string to long integer

strtol(str1); — need to pass for decimal nos. system
strtol(str1, NULL, 10)

11. strtof(str1) (string to float) — converts string to float

strtof(str1, NULL);

ABHINAV

Date __/__/__ Page No.:

12. strtok();
string tokenizing

used to tokenize a string based on delimitter.
(Kinda like split() in python)

Example code

```
int main()
{
    char s1[20] = "x=10; y=20; z=35";
    char *token = strtok(s1, "=;");
    while (token != NULL)
    {
        cout << token << endl;
        token = strtok(NULL, "=;");
    }
    return 0;
}
```

Output

```
x
10
y
20
z
35
```

ABHINAV

Date __/__/____ Page No.:

* Class String

```
#include <string>
string str; // declaration
getline(cin, str); // initialization
```

* Functions of string class

declared string name

1. s.length();
returns length of the string

2. s.size();
same as ~~size~~ length

3. s.capacity();
returns capacity of a string

4. s.resize();
capacity can be resize of string
→ integer value

5. s.max_size();
possible maximum size of a string you can have on this compiler

6. s.clear();
clears the contents of a string

7. s.empty();
finds out if string is empty or not.

ABHINAV

Date __/__/____ Page No.:____

8. s.append( ); → string value
adds the string at the end of s.

9. s.insert(num, string); → location, where to add
inserts string at custom location, size of second string can also be declared.
s.insert(num, string, 2);
// first two letters of the string would be added at num in 's' string

10. s.replace(num1, num2, string1);
replaces something in 's' string by takes starting value (num1) and length to change (num2), also what to put is mentioned (string1)
string str = "programming";
str.replace(3, 6, "ti");
cout << str << endl;

→ proting

11. ~~s.erase~~ s.erase();
will erase a string just like the clear function.

12. s.push_back(string);
Inserts a single characters at the end of a string.

13. s.popback();
removes a single character from the back of string s - (kinda like ~~takes~~ backspace)

ABHINAV

Date ___/___/_____ Page No.: _____

14. s.swap(string1);
swaps the values of s and string 1.

15. s.copy(char des[]);
used to ~~convert~~ copy a string into a character array

16. s.find(str/char);
returns index of matching character., searches for left side
In order to search from right side we can use →
s.rfind(str/char)

17. s.find_first_of();
find the first occurrence of a given letter, we can also specify a number to start from specific index

18. s.find_last_of();
similar to find_first_off, but starts from last.

{ group of letters can be given to find_first_of & find_last_of }

19. s.substr(start, number);
used to extract a portion of string, returns a string object
string str = "abcdefgh"
cout << str.substr(3,2);

ABHINAV

→ de

Date __/__/____ Page No.:____

20. s.compare (str);
similar as strcmp(), compares two strings and returns negative, positive or zero.

21. s.at (num);
used to ~~locate~~ return the character stored at given num index.

```
string str = "Hello"
cout << str.at(3);
or
cout << str[3];

output → l
```

22. two strings can be added using a operator

23. s.front();
to access the element of a string
back() returns the last element.

24.
* String - Class Iterator
using to iterate through every string in a string
here's an example code:

Date ___/___/_____ Page No. _____

```
int main()
{
    string str = "today";
    string :: iterator it;
    for(it = str.begin(); it != str.end(); it++)
    {
        cout << *it;
    }
    cout << endl;
    return 0;
}
```

→ reverse iterator

iterates through a string in reverse order

```
int main()
{
    string str = "today";
    string :: reverse_iterator it;
    for(it = str.rbegin(); it != str.rend(); it++)
    {
        cout << *it;
    }
    cout << endl;
    return 0;
}
```

```
for(int i=0; str[i] != '\0'; i++)
    { cout << str[i]; }
```

↳ iterating in from left to right without iterator

ABHINAV

Date __/__/____ Page No.: _____

* Functions

↳ block of code that perform specific code tasks and can be reused throughout the programs

↳ provide a way to modularize code, and make it more organized

Important notes:

(i) function can only return atmost one value

(ii) if func. doesn't return any value then func. type should be **void**

(iii) avoid using cin, cout in func. except main()

Defining a function:

return-type function-name (parameter list)

{

}

Important note

Once the function terminates, it's memory inside the stack will be removed. It's activation record is deleted

* Function overloading

Function overloading is a feature in CPP that allows you to create multiple functions with the same name but different parameter lists.

ABHINAV

Date ___/___/_____ Page No.:_____

Overloading refers to the idea of adding more than one meaning or behaviours to a single concept or name.

* Function template
  - ↳ way to define a generic function that with different data types
  - ↳ allows you to write a single function definition that can be used for multiple types without having to write seperate versions of the function for each type

Syntax →

```
template <class T>
T max(T x, T y)
{   if (x > y) return x;
    else return y;
}
int main()
{ cout << max(3, 5); }
```

* Default arguments
  - ↳ allows us to specify default values for a function parameters

A default argument is a value that is automatically used by the function if no corresponding argument is provided by the caller.

ABHINAV

Date __/__/____ Page No.:____

* Parameters passing method

1. Call Pass by value:
→ makes a copy of the argument and passes it to the function
→ changes made inside the function do not affect the original argument
→ default way of passing parameters if no other way is specified.

2. Pass by address:
→ involved passing a pointer variable that holds the memory address of the argument
→ func. # can use the pointer to indirectly access and modify the original variable.

3. Pass by reference:
→ method passes a reference to the argument allowing the function to directly modify the original variable

* Return by methods

1. Return by value
→ most common way to return value from a function
↳ func. calcs. or generates a value and returns it directly

ABHINAV

Date ___/___/_____ Page No _____

2. Return by reference
↳ Instead of returning a copy of the value, we can return a reference to a variable/object.
↳ allows the caller to directly modify the original value
↳ should be used with caution to ensure the referenced value remains valid

3. Return by pointer return
↳ allows us to dynamically allocated memory or a memory address
↳ similar to returning but provides more flexibility
↳ proper memory management is req. to avoid memory leaks

* Global and local variables

Global variables
→ declared outside of any function, typically at the top of the file or in a namespace
→ have global scope, can be accessed throughout the entire program, including all functions
→ have a lifespan that spans the entire execution of the program

ABHINAV

Date ___/___/_____ Page No.:_____

Local variables

→ declared inside a block or function,
→ have a local scope
→ only accessible within the block or function in which they are created
→ not automatically initialized to a default value.
→ initial value is indeterminate

* Static variables

↳ those variables that remains in the memory, just like global variables
↳ diff. b/w global and static variables is that global can be accessed in any func. but static cannot be.
↳ visibility is limited to a function

* Recursive function

↳ a function that calls itself is called a recursive function
↳ kinda similar to loops but recursion is a better alternative

```
fun(int x){
  if(x>0){
    cout << x;
    fun(x-1);
  }
}
int main(){
  fun(5);
}
```

← Example code

ABHINAV
5 4 3 2 1

Date ___/___/_____ Page No. _____

* Function Pointer

↳ used to point functions, same as being other pointers ~~are used~~ being used to point variables

↳ utilized to save a function's disease address.

↳ either used to call the function, or can be sent as an argument to another function

* Object Oriented Programming

↳ a programming paradigm that emphasizes the use of objects and classes.

↳ provides to a way to structure and organize the code by grouping related data and functions into objects

→ Principles of OOPs →

1. Abstraction → allows you to create data types and hide unnecessary implementation details
2. Encapsulation → bundling data and the methods that operate on the data into a single unit called a class
3. Inheritance → allows you to create new class based on existing class
4. Polymorphism. → allows objects of diff. classes to be treated as objects of a common class.

ABHINAV

Date __/__/____ Page No.:____

// Declaration

```
class Rectangle
{ public:
    int length, breadth;
    int area()
      { return length * breadth; }
    int perimeter()
      { return 2*(length + breadth); }
};
```
- this is important

// Initialization

```
void main()
{
  Rectangle r1, r2;
  r1.length = 10;
  r1.breadth = 5;
  cout << r1.area();
  r2.length = 20;
  r2.breadth = 10;
  cout << r2.area();
}
```

→ any no. of objects can be created

→ dot operator to access the members of an object

// using pointer to an object

```
void main(){
  Rectangle r1;
  Rectangle *ptr;
  ptr = &r1;
  ptr->length = 10;
  ptr->breadth = 5;
  cout << ptr->area() << endl;
  cout << ptr->perimeter() << endl;
}
```

→ accessing object using arrow operator and pointer

ABHINAV

Date __/__/____ Page No.:____

It is a good practice to set data members as private.
we can do that by creating set and get methods for our class.

Example →

```
class Rectangle
{ private:
  int length, breadth;
  public:
  void setLength (int l)
  { length = l; }
  void setBreadth (int b)
  { breadth = b; }
  int getLength ()
    { return length; }
  int getBreadth ()
    { return breadth; }
};
  int area ()
  { return length * breadth; }
  int perimeter ()
  { return 2*(length + breadth); }
```

Even better approach would be to use constructor instead of get() functions

Date ___/___/_____ Page No.:_____

* Constructors

↳ compiler provided constructor

1. Default constructor:
   - ↳ a construction with no parameters is called a default constructor
   - ↳ automatically generated

2. Non-parameterized:
   - ↳ takes no parameters
   - ↳ needs to be defined explicitly

Code:

```
Rectangle ()
{ length = 0;
  breadth = 0;
}
```

3. Parameterized:
   - ↳ take parameters, one or more
   - ↳ allows us to initialize the object with specific values provided during object creation.
   - ↳ need to be defined explicitly

Code:

```
Rectangle (int l, int b)
{
  setLength (l);
  setBreadth (b);
}
```

ABHINAV

Date __/__/____ Page No.:____

4. Copy constructors:
   - used to create a new object as a copy of an existing object
   - takes reference to an object of the same class as a parameter

Code

```
Rectangle (Rectangle (& rect))
{
    length = rect.length;
    breadth = rect.breadth;
```

5. Deep copy constructors:
   - creates new objects and performs a deep copy of the data members
   - involves creating a new object copies of dynamically allocated memory or any other resources

Code:

```
Test ( Test &t )
{ a = t.a;
  ~~p = t.p;~~ instead of this
  p = new int[a];
}
```

* Types of functions in a class

A class is said to be perfect if it is written in the following format:

ABHINAV

Date __/__/____ Page No.:____

```
class Rectangle
{ private:
        int length, breadth;        // private data members
  public:
    Rectangle ();                   // constructors
    Rectangle (int l, int b);
    Rectangle (Rectangle &r);
    void setLength (int l);         // mutators
    void setBreadth (int b);
    int getLength ();               // accessors
    int getBreadth ();
    int area ();                    // facilitators
    int perimeter ();
    int isSquare ();                // enquiry
    ~Rectangle ();                  // destructors
};
```

* **Scope Resolution operator**

There are two ways to define functions within a class.

1.) define the function within the class

2.) use scope resolution and define it outside the class.

Date ___/___/_____ Page No.: _____

This involves elaborating the function outside within the class, and only the prototype is written in the class along. This is done using scope resolution

Example:

```
class Rectangle
{
    private:
        int length, breadth;
    public:
        int area ();
        int perimeter ();
};
int Rectangle :: perimeter ()
{
    return (length + breadth) * 2;
}
```

func. def. outside

What is the difference?

In line functions are preferred not to have a complex logic, such as loops and nested loops, always use a scope resolution function and create a func. outside.

* In-line functions
  is functions that expand in the same line

ABHINAV

Date __/__/____ Page No.: _____

NOTE

A function defined outside can be made inline function, simply by writing inline in front of it. This would cause the function to be present inside the class itself when the machine code is generated

Code:

```
class Test
{ public :
        void fun1()
        {
            cout << " Inline " ;
        }
  inline void fun2() ;
};

void Test :: fun2()
    {
        cout << " non-Inline " ;
    }
```

* this → pointer

↳ special pointer that is implicitly available within the member functions of a class.

↳ Points to the object for which the member function is called.

↳ Allows us to access the members of the class using the arrow operator

ABHINAV

Date __/__/____ Page No.:____

Code:

```
class Rectangle
{
private:
    int length, breadth;
public:
    Rectangle (int length, int b
    {
        this -> length = length;
        this -> breadth = breadth;
    }
    int area()
    { return length * breadth; }
```

using this pointer we refer to the current object

* Struct vs Class

Both are same, but in structure by default everything is private and in a class everything is public

In C language, struct could only have data members but in C++ they can also have functions, making it similar to a class

ABHINAV

Date ___/___/_____ Page No.: _____

* Operator Overloading

↳ used to redefine the behaviour of existing operators for user-defined types or classes

Example code

```
class Complex
{ private:
      int real, img;
  public:
      Complex (int x=0, int i=0)
      {
        real = x;
        img = i;
      }
      Complex operator + (Complex x)
      {
        Complex temp;
        temp.real = real + x.real;
        temp.img = img + x.img;
        return temp;
      }
};
int main()
{ Complex c1(3,5);
  Complex c2(5,7);
  Complex c3;
  c3 = c1 + c2;
}
```

signature function → `Complex operator + (Complex x)`

defines + operator function

Date ___/___/_____ Page No.:_____

→ **Friend operator overloading**

A friend operator is a function that is allowed access to the private and protected members of the class

To overload a friend operator, we need to declare the function as a friend inside the class definition

Example code

```
Class Complex
{ public:
  int real, img;
  Complex ( int r=0 , int i=0 )
  { real = r ; img = i ; }
  friend Complex operator+ (Complex c1,
                            Complex c2);
};
```

→ operator declared outside without scope resolution

```
Complex operator+ (Complex c1, Complex c2)
{ Complex temp;
  temp.real = c1.real + c2.real;
  temp.img = c1.img + c2.img;
  return temp;
}
```

→ **Insertion operator overloading**

↳ Insertion operator is represented by << symbol

↳ commonly used for output operations

ABHINAV

Date __/__/____ Page No.: ____

Example code

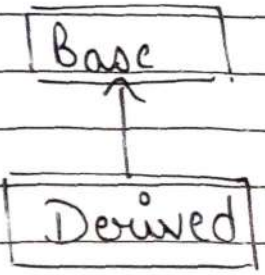
```
class Complex
{ private:
    int real, img;
  public:
  friend ostream & operator << (ostream &O,
                                complex &C1);
};

ostream & operator<< (ostream &O, complex &C1)
{
  O<< c1.real << "+i" << c1.img;
  return O; ← a letter
}
```

* Inheritance
  - allows us to create new classes based on existing classes. It enables the creation of a ~~hierarchy~~ hierarchy of classes, where derived classes inherit characteristics from their base class.
  - → benefits of inheritance
    Polymorphism, Modularity, Code reusability

Code:

Date ___/___/_____ Page No.: _____




```
class Base
{
    public :
    int x ;
    void show ()
      {
        cout << x ;
      }
};

class Derived : public Base
{
    public :
      int y ;
      void display ()
      { cout << x << y ; }
};

int main () {
Base b ;
b.x = 25 ;
b.show () ;  → 25

Derived d ;
d.x = 10 ;
d.y = 5 ;
d.show () ;  → 10
d.display () ;  → 105
```

ABHINAV

Date __/__/____ Page No.:____

→ Constructors in Inheritance

- Constructors of base class is executed first then the constructor of derived class is executed.
- By default non-parameterised constructor of base class is executed
- Parameterised constructor of base class must be called from derived class constructor.

Code

```
class Base
{ public:
    Base() { cout << "Non-param Base" << endl; }
    Base(int x) { cout << " param Base" << endl; }
};
class Derived : public Base
{ public:
    Derived() { cout << " Non param Derived"
                << endl; }
    Derived(int y) { cout << " Param Derived"
                << endl; }
    Derived(int x, int y) : Base(x)
    { cout << " Param of Derived" << y << endl;
    }
};
int main() {
    Derived d(5,10); }
```

Date ___/___/_____ Page No.:_____

* isA and hasA

If a class is subscribing from some other class then it is having a isA relationship.

If a class is having an object of another, then it is having a hasA relationship.

```
class Cuboid : public Rectangle
```

```
class Table
{
    Rectangle top;
};
```

* Access Specifiers

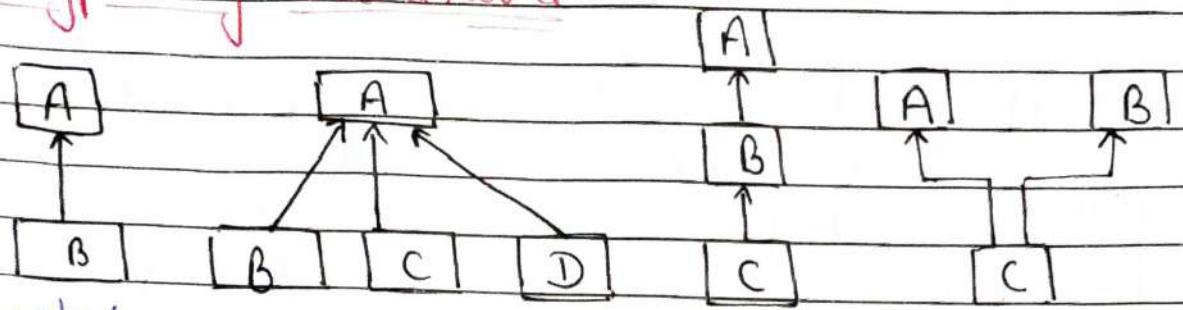
	private	protected	public
within the class	✓	✓	✓
in a derived class	X	✓	✓
on an object	X	X	✓

✓ → accessible

ABHINAV

Date __/__/____ Page No.:____

* Types of Inheritance



Simple/ Single | Hierarchical | Multi -level | Multiple

1. Simple Inheritance:
Here, a derived class inherits from a single base class.

2. Hierarchical Inheritance:
This involves multiple derived classes inheriting from a single base class. There's a single base class and multiple derived classes are created from it.

3. Multilevel Inheritance:
Here, a derived class is created from another derived class.
Involves chain of inheritance

4. Multiple Inheritance:
allows a derived class to inherit from multiple base classes
means derived class can have features from more than one Base class

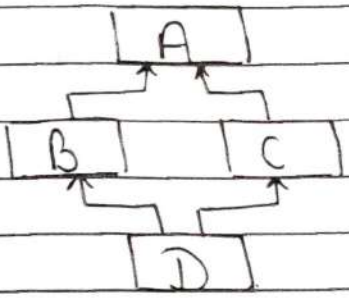
ABHINAV

Date __/__/____ Page No.:____

5. Hybrid Inheritance.

↳ combination of multiple inheritance and multilevel inheritance

↳ Involves mix of two or more types of inheritance




```
class A
{ == };
class B : virtual public A
{ == };
class C : virtual public A
{ == };
class D : public B, public C
{ == };
```

Note: We can inherit a class publicly, privately and protected.

* Generalization vs Specialization

Two purpose of inheritance:

1. Share its features to child classes (Specialization).
2. To achieve polymorphism (Generalization)

ABHINAV

Date __/__/____ Page No. _____

1. Generalization

This refers to the process of creating a more general class (base class) from two or more specific classes (derived classes)
It captures the common attributes and behaviours shared by the derived classes and abstracts them into a higher-level, generalized class.

2. Specialization

The process of creating a more specific class from a general class
Involves extending or refining the attributes and behaviours inherited from the base class to cater to a more specialized class.

* Base class pointer derived class object:

```
int main() {
Base *p;
p = new Derived();
p->fun1();
p->fun2();
p->fun3();
```

```
class Base
{ public:
void fun1();
void fun2();
};
class Derived : public Base {
public:
void fun3(); };
```

Date ___/___/_____ Page No. _____

we can have a base class pointer and a derived class pointer object and we can call only those functions which are present in base class. (because the pointer is base class) We cannot call the functions that are derived defined in derived class object.

Note

We cannot have an a pointer of the derived class and an object of base class.

* **Function Overriding**

↳ Function overriding is a feature that allows a derived class to provide a different implementation for a function that is already present in the base class

Example Code:

```
class Parent
{ public:
      void display() { cout << "Parent class"; }
};
class child
{ public:
      void display() { cout << " child class"; }
};
int main ()
{
    child c;
    c.display();
}
```

→ output → Child class

ABHINAV

Date __/__/____ Page No. _____

* Virtual functions

- Such functions are used for achieving polymorphism
- Base class can have virtual functions
- Virtual functions can be overrided in virtual derived class
- Pure virtual functions must be overrided by derived class.

Example Code:

```
class Basic_Car
{ public:
      virtual void start()
          { cout << "Basic Car started"; }
};
class Advanced_Car
{ public:
      void start()
        { cout << "Advanced Car started"; }
};

int main()
{
   Basic_Car *p = new Advance_Car();
   p -> start();
}
```

output → Advance Car started

ABHINAV if virtual is not written, then output would have been "Base Car started".

Date ___/___/_____ Page No. _____

* Runtime Polymorphism

To achieve runtime polymorphism, we need:
- Virtual functions
- overriding
- base class pointer & derived class object

* Polymorphism

↳ ability of an object to take on many forms.

↳ In cpp, polymorphism allows you to write code that can work with objects of different types, treating them as if they were objects of a common base type.

Example code:

```
class Car
{ public:
      virtual void start() = 0;
      virtual void stop() = 0;
};
class Innova : public Car
{ public:
      void start() { cout << "start Innova"; }
      void stop() { cout << "stop Innova"; }
};
int main()
{
  Car *p = new Innova();
  p->start();
}
```

ABHINAV

Date __/__/____ Page No.:____

```
class Swift : public Car
{
  public:
    void start() { cout << "Swift"; }
    void stop() { cout << "end swift"; }
};

int main() {
  Car *p = new Innova();
  p->start(); // start Innova

  p = new Swift();
  p->start(); // swift
```

* **Abstract Classes**

an abstract class is a class that cannot be instantiated. It is designed to serve as a base class for other classes and provides a common interface or blueprint for its derived classes.

Note

Overriding helps in achieving polymorphism

Date __/__/____ Page No. _____

* Friend functions and classes

Friend functions and classes provide a way to grant special access privileges to functions or classes that are not part of the regular class heirarchy.

A friend function is a function that is allowed to access private and protected members of a class as if it were a member of that class.
It is declared as a friend inside the class declaration, usually preceded by the 'friend' keyword.

Code:

```
class MyClass {
private: int secretvalue;
public:
    MyClass(int value) : secretvalue(value) { }
    friend void printSecretValue
                (const MyClass & obj);
};
void printSecretValue (const MyClass & obj) {
    cout << "Secret is : " << obj.secretValue ; }
```

ABHINAV

Date __/__/____ Page No.:____

A friend class is a class that is granted access to the private and protected members of another class.
To declare a class as a friend, you can use the 'friend' keyword in the class declaration.
This allows the friend class to access the private and protected members of the class it is friends with.

Code:

```
class My
{ private:
      int a = 10;
  friend class Your;
};
class Your
{ public:
      My m;
      void fun()
      { cout << m.a; }
};
```

* Static Members
  - are variables or functions that belong to the class itself, not to individual objects
  - shared among all instances of the class and can be accessed without creating an object of the class

ABHINAV

Date / / Page No

* Important point : Static member functions can access only static data members of a class.

- static members can be used as a counter
- static members can also be used as shade memory for all the objects

* Inner/Nested Classes

↳ A nested class is a class that is defined inside another class.
↳ also known as inner class.
↳ main advantage of using nested class is that it allows for better organization and encapsulation of code
↳ ✓ nested class can be hidden from the outside world and only accessed through the outer class.

Code:

```
class Outer {
public:
    void fun()
    { i.display(); }
    class Inner ← inner class
    { public:
        void display()
        { cout << "Display of Inner"; }
    };
    Inner i;
};
```

ABHINAV

Date ___/___/_____ Page No.:_____

Exception Handling

There are three types of errors mainly →

1.) Syntax Error
a mistake in the structure of code or formatting of the code that violates the language's rules.

Can be solved with the help of a compiler

2.) Logical Error
occurs when the program doesn't perform the way it was intended to

Can be solved with the help of a debugger

3.) Runtime Error
caused due to bad input or problem with resources

Can be solved through exception handling construct

Date ___/___/_____ Page No _____

Exception Handling code Example

```
int main() {
    int a=10, b=0, c;

    try
    {
        if (b==0)
            throw 1;
        c = a/b;
        cout <<c
    }
    catch (int e)
    { cout << "Division by zero"; }

    cout << "Bye";
}
```

Output → Division by zero

(since b=0)

Exception handling is used to handle exceptions b/w functions.
The following example explains it →

Example on next page →

ABHINAV

Date __/__/____ Page No.:____

Catching exceptions b/w functions

```
int main()
{
    int a=10, b=0, c;
    try
    {
        c = division(a, b);
        cout << c;
    }
    catch (int e)
    { cout << "Division by zero"; }

    cout << "Bye";
};

int division(int x, int y)
{ if (y==0) {th
    { throw 1; }
    return x/y;
}
```

Note

- functions are communicating with each other using the throw - catch
- a function can throw any data type, even a class object

ABHINAV

Date __/__/____ Page No.:____

About catch

- We can have multiple catch blocks for different datatypes.
- Ellipsis catch blocks : These blocks are the ones which can catch any type of exception
  → They must be kept in the last after of other catch blocks

Code →

```
try
{ throw string ("error"); }
catch (int e)
{ cout << "int catch"; }
catch (double e)
{ cout << "double catch"; }

// ellipsis catch block
catch (...)
{ cout << "all catch"; }
```

Template functions and Classes

template is a feature that allows you to define generic functions or classes. It allows you to write code that can work with different datatypes without having to duplicate the code for each specific type.

ABHINAV

Date ___/___/_____ Page No.: _____

Example code

```
template <class T>
class Stack
{
  private :
    T s[10];
    int top;
  public :
    void push (T x);
    T pop ();
};

template <class T>
void stack<T> :: push(T x)
{
  ========
}
template <class T>
T stack<T> :: pop()
{
  ========
}

int main()
{
  stack<int> s1;
  stack <float> s2;
```

Date __/__/____ Page No.:____

* Constant Qualifiers

const qualifier is used to declare a variable or function as constant, meaning its value or behaviour cannot be modified

1. For variables

We can declare a variable constant, meaning it's value can't be changed in the future

Code:

```
const int myNumber = 10;
```

2. For functions

declaring a function 'const', means the function will not modify the state of the object it is called on.

Code:

```
class Test {
public:
    int getValue() const {
        return value;
    }
private:
    int value = 5;
};
```

* 'const' qualifier ensures that variables and functions are treated as read-only, preventing modifications to their values or behaviour

ABHINAV

Date __/__/____ Page No.:_____

* Preprocessors / Macros

using a preprocessor is a tool that performs some preliminary tasks before the actual compilation of the code takes place

By using #define we can define symbolic constants

#ifndef → if not defined, then define.

code:

```
# define max(x, y) (x > y ? x : y)
# define PI 3.14
# define msg(x) # x

# ifdef PI
    # define PI 3
# endif
```

* Namespaces

namespaces are used to organize and group related code elements such as variables, functions and classes into separate named scopes

Date ___/___/_____ Page No.: _____

namespaces are used for removing name conflicts b/w resources like functions, classes and objects

Example code

```
namespace first
{ void fun()
    { cout << "first" << endl; }
}

namespace second
{ void fun()
    { cout << "second" << endl; }
};

int main()
{ first :: fun();
  second :: fun();
}
```

* **Destructor**

these are special member functions that are automatically called when an object is about to be destroyed or deallocated.

Note: unlike constructors, a function can only have one destructor

ABHINAV

Date __/__/____ Page No.: ____

Code :

```
class Demo
{
  public :
     Demo ( )
       {
         cout << " Constructor " ;
       }
    ~Demo ( )
       { cout << " Destructor " ; }
};

Void fun ( )
  { Demo d ; }

int main ( )
     { fun ( ) ; }
```

Output →

Constructor Destructor

Note → both constructor and destructor were called

Date __/__/____ Page No. _____

But, when we create an object dynamically in a heap, then only the constructor will be called.

Code

```
Demo *p = new Demo();
(Here only the constructor will be called)
```

for calling the destructor, we must say →

```
delete p;
(now the destructor will be called)
```

* Virtual Destructor

→ a virtual destructor is a special type of destructor that is declared in the base class, and ensures that the correct destructor is called when deleting an object through a pointer to a ~~base class~~ base class

→ when you delete an object through a base class pointer, and the base class destructor is not declared as virtual, only the base class destructor will be called which can lead to memory leaks and undefined behaviour in derived class

ABHINAV

Date __/__/____ Page No.:____

Code:

```
class Base {
public:
    Base()
    { cout << "Base constructor" << endl; }
    virtual ~Base()
    { cout << "Base destructor" << endl; }
};

class Derived : public Base {
public:
    Derived()
    { cout << "Derived constructor" << endl; }
    ~Derived()
    { cout << "Derived destructor" << endl; }
};

int main()
{ Base * p = new Derived();
  delete p; }
```

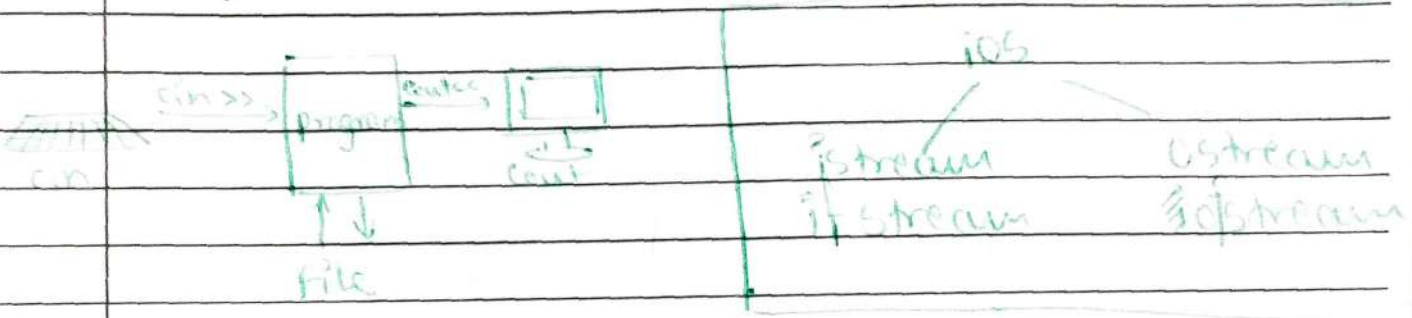
Output →

```
Base constructor
Derived constructor
Derived destructor
Base destructor
```

Date ___/___/_____ Page No. _____

* Streams

- Streams in Cpp are a way to handle input and output operations.
- They provide a convenient and consistent way to read data from and write data to various sources, such as the console, files, or even network connections.



→ Writing in a file

File handling →

```
#include <fstream>
int main()
{   ofstream outfile("my.txt");
    outfile << "Hello" << endl;
    outfile << 25 << endl;
    outfile.close();
}
```

output → .txt file named my.txt will be created with (Hello and 25) as values in it

ABHINAV

Date ___/___/_____ Page No.:_____

→ Reading from a file

```
#include <fstream>
int main()
{
    ifstream infile;
    infile.open("my.txt");
    string str;
    int x;
    infile >> str;
    infile >> x;
    cout << "str << "  " << x;
    if (infile.eof())  will work if we have reached the end of file
        {. cout << "end of file";}
}
```

* STL (Standard template library)

It is a powerful library, that provides a collection of template classes and functions for common data structures and algorithms

STL consists of three main components.

1. Containers → These are the classes that hold objects collection of objects
   → provides various operations for manipulating and accessing the elements they hold.

ABHINAV

Date __/__/____ Page No.:____

2. Algorithms → These include sorting, searching, merging and more
They are implemented as template functions and can be used with different containers or user-defined types

3. Iterators → these are used to traverse the elements of a container.
→ allow algorithms to operate on containers independently of their underlying implementation

Using STL can significantly simplify the process of writing C++ code by providing efficient and standardized implementations of common DSA

Algorithms	Containers
search()	vector] array, but can dynamically change by copying elements into bigger element
sort()	list
binary_search()	forward_list] singly linked list (not bidirectional)
reverse()	deque] elements can be inserted from left or right
concat()	priority_queue] removes max value when asked
copy()	stack] LIFO
union()	set] unique elements
intersection()	multiset] same as set allows duplicate
merge()	map] data stored in key-value pair
heap()	multimap] same as map, but keys can be duplicated, but same key-value pair not allowed

ABHINAV

Date ___/___/_____ Page No.:_____

Iterator example code →

```
void main()
{
    vector <int> v = {10, 20, 30, 40, 50};
    v.push_back(25);
    v.push_back(70);
    vector <int> :: iterator itr;
    for (itr = v.begin(); itr != v.end(); itr++)
        cout << *itr;
}
```

* C++ 11 features

(1.) Auto

→ used to specify automatic type deduction for variables

→ allows compiler to automatically determine the type of a variable based on its initialized expression

Code:

```
auto x = 10;   // x is deduced as an int
auto name = "John";
          // is deduced as a const char
```

(2.) final

This keyword is used to indicate that a class, virtual function, or override cannot be further derived or ~~overridden~~ overridden by derived classes

ABHINAV

Date __/__/____ Page No.:____

→ useful for enforcing the immutability of an within class hierarchies and preventing further modifications to virtual functions

Code:
```
class Base final {
    //....
};
```

In short →
it prevents inheritance and function overriding

(30) Lambda expressions (an unnamed function)

Syntax

[capture_list](parameter_list) → returntype {body};

a lambda expression is an anonymous function that can be defined inline and used as a convenient way to create function objects

Code:
```
int main()
{ int a = 5, b = 10;
  [a,b](){ cout << a << " " << b; }();
}
```

a, b put in capture list, in order to make them accessible

ABHINAV

Date __/__/____ Page No.:____

(4.) Smart Pointers

→ Smart pointers are objects that manage the lifetime of dynamically allocated memory
→ are safer and more convenient alternative to raw pointers.
→ helps in preventing memory leaks and other memory-related issues.
→ automatically release the memory they own when it is no longer needed.

There are three types of smart pointers →

(1.) **unique_ptr** :
→ represents exclusive ownership of a dynamically allocated object
→ ensures that only one unique_ptr instance can own a particular object at a time.
→ when it goes out of scope, or is explicitly reset, it automatically deletes the owned object.

(2.) **shared_ptr** :
→ allows multiple instances to share ownership of the same object
→ uses reference count internally to track how many pointers point to the object
→ object is only deleted when the last 'shared_ptr' that owns it goes out of scope or is reset.

ABHINAV

Date ___/___/_____ Page No _____

(3.) weak_ptr :
→ provides non-owning non-owning, weak reference to an object managed by shared_ptr.
→ does not affect the reference count
→ if the object is destroyed, the 'weak_ptr' will automatically be set to nullptr.

Advantages of smart pointers →
- automatic memory management
- exception safety
- improved code clarity

(5.) Ellipsis ⟶ (...)
→ ellipsis is used to define a variadic function or to access variadic arguments within a function
→ Variadic functions are functions that can accept a variable number of arguments.

Code :
```
int sum (int n, ...)
{ va_list list;
  va_start (list, n);
  int x;
  int s=0;
  for (int i=0; i<n; i++)
  { x = va_arg (list, int);
    s+=x;
  }
  return s;
}
```

ABHINAV

— END —