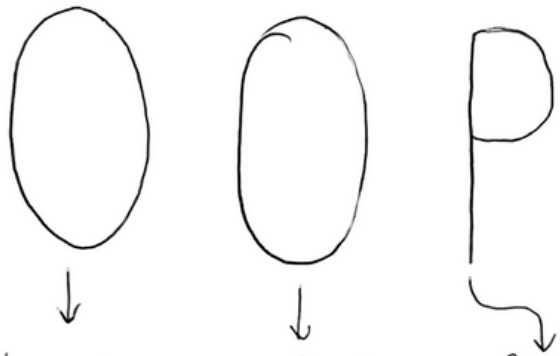


Object → Entity that exists in Real World

Oriented → Concerned / Interested

Programming

[C++, C#, Java, JavaScript, Python, .etc]

⇓

Data + **Code**

- Attributes / Properties
- Procedures / Methods

Procedure OP	Vs	Object OP
① Program is divided into "functions"		① Program is divided into "objects"
② Inheritence (x)		② Inheritence (✓)
③ Top-down Approach		③ Bottom up Approach
④ Virtual function (x)		④ virtual function (✓)
⑤ NO code Reusability		⑤ Existing code can be used.
⑥ Less secure (NO data hiding)		⑥ Encapsulation is used to hide data.
⑦ Easy to add new data (x)		⑦ Easy to add new data (✓)
⑧ Eg: C, FORTAN, PASCAL		⑧ Eg: C++, Java, Python.

* features of OOP

- Class & objects:

  Class ↓ (User defined Data type) [Data + functions] — can be used by creating an instance of class ↑ objects

```
Class student {
    char name[20];
    int Id;
    void fun1() { }
};
int main() {
    student s1;
    return 0;
}
```

- Encapsulation:

  (Data/variables, functions (Methods)) — single unit

  functions —manipulate→ Data

  [Data hiding]

Eg: T1 → Marks $[m_1, m_2 \cdots m_n]$ T2 → Marks $[m_1, m_2 \cdots m_n]$

- Abstraction: [BTS]

  → Hiding the details, & show only essential info.

  Eg: Driving a car.

  → can be acheived by classes.

- **Polymorphism**: [many forms]
  - ↳ f^n exhibit different behavior in different instances

Eg:-

```
int main() {
    R1 = sum(2, 3);
    R2 = sum(1, 2, 2);
}

int sum(int x, int y)
{ return (x+y); }

int sum(int x, int y, int z)
{ return (x+y+z); }
```

function Overloading.

Eg: (Operator Overloading)

```
void operator ++()
{ ++data; }
```

prefix

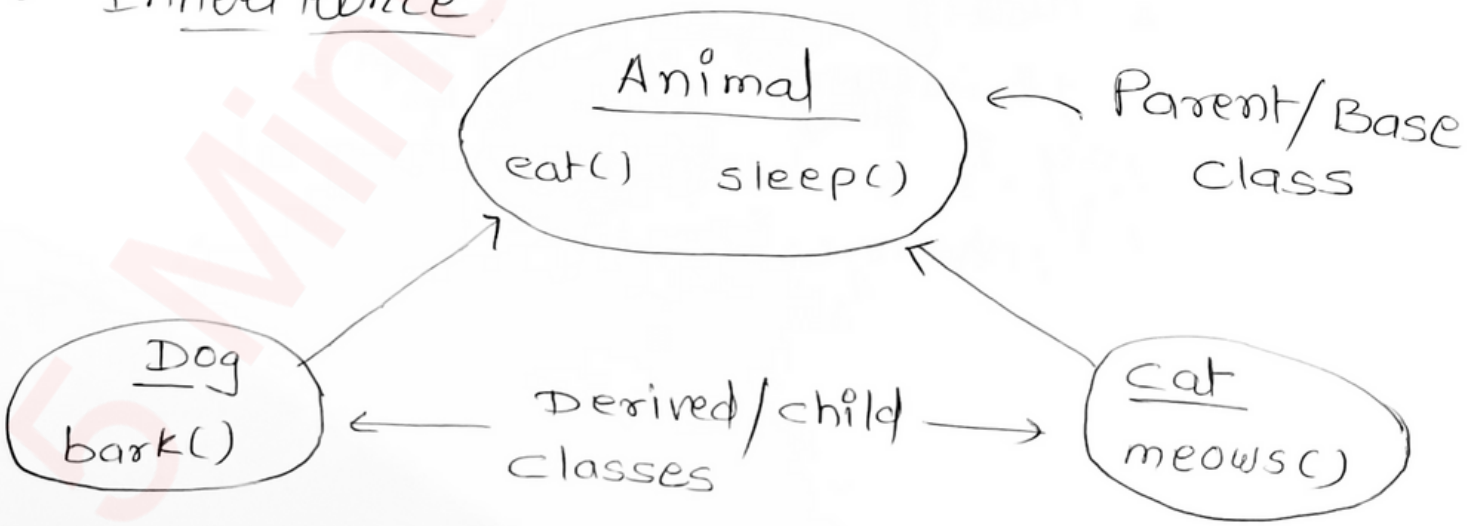
(++obj)

```
void operator ++()
{ data++; }
```

postfix

(obj++)

- **Inheritance**



- Dynamic Binding:

  late / Runtime Linking / connecting

  ⇒ The code associated with a given procedure call is not known until the time of call (Runtime).

- Message Passing:

(Messages) → Request of a procedure execution

  → Name of object
  → Name of function
  → Information to be sent

- Benefits of OOP

  → Reusability
  → Data hiding.
  → Message passing.
  → Security ↑
  → Updates / changes easy

° Basic Structure of C++ Language

① **Documentation section:**

→ Objective, logic info/details.
→ It's the "comment" not compiled.
→ Its optional. (/*...*/)

② **Linking section:**

→ **Header files:** → preprocessor directive
[#include <iostream>]

standard I/P O/P facilities

→ **Namespaces:**
↳ Grouping various entities under a single name.
° using namespace std;

③ **Definition section:**
↳ Declare constants & assign them some values.
eg:- #define MAX 5

④ Global Declaration section:

Variables & Class definitions are declared.

accessible to all parts of program (throughout the Program)

⑤ C / function Declaration section:

↳ All sub programs / function / Class which our program needs.

```
int sum (int x, int y)
{ return (x+y); }
```

⑥ Main function section:

↳ It tells compiler where to start the execution.

→ All statements to be executed are written in main fn.

- Data types (Type of data in variable)
  - Primary
    - int
    - char
    - float
    - double
    - bool
    - void
  - Derived
    - Array
    - Pointer
  - User defined
    - structure
    - Union
    - Class
    - Enumeration

* Primary data types

① int
- → Integers
- → 4 Bytes
- → Range : -2147483648 to 2147483647
- → Types
  - Signed (MSB bit → 0 +ve, 1 -ve)
  - Unsigned (only magnitude)
- → format specifier: %i

② Char
- → characters
- → 1 Byte
- → Single quotes ' ' (In C++)
- → eg: char x = 's'
- → format specifier: %c

③ float & double
- → floating point Numbers (Decimals & exponentials)
- → float : 4 Bytes
  Double : 8 Bytes
- → Double has 2 times precision of float
- → eg:- float x = 25.25
  double y = 45E11 → 45×10^{11}
- → format specifier: %f, %lf

④ Boolean

- True or false
- Used in conditional statements & loops
- eg: bool x = True;
- 1 Byte

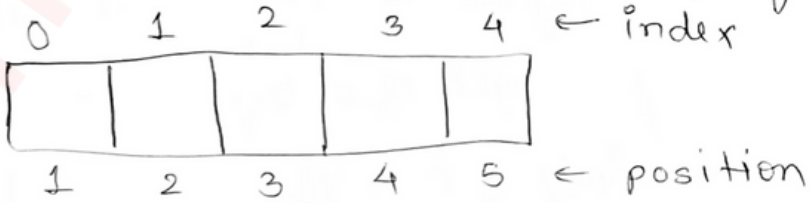
⑤ void

- Empty / Nothing / NO value
- 0 Byte
- void fn() { }

* Derived Data type

① Array

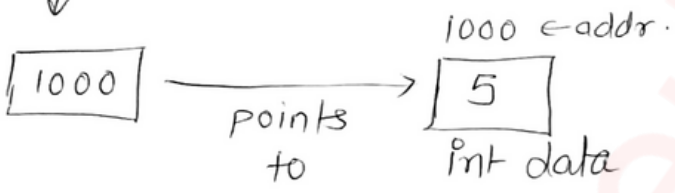
- Collection of items stored at continuous memory locations
- eg: Datatype Arrayname [size of array]



- Arr[i] ← (Move) a

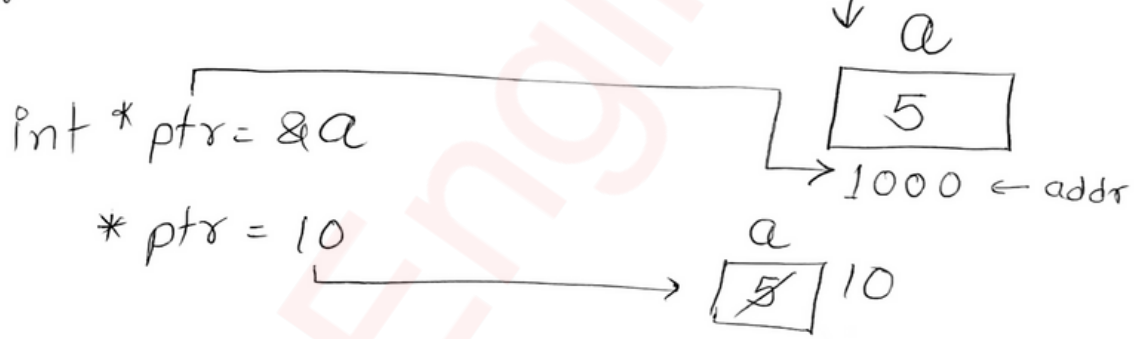
② Pointers

→ int *ptr



→ datatype *ptr_name.

→ eg:- int a = 5



int **ptr = &ptr

**ptr = 20 → ~~10~~ 20

③ Reference:

→ datatype & ref = variable

eg: int a = 5

int &b = a

↓

'b' is a reference to a

b = 10

→ 'a' value will be changed.

User Defined Data types

① structure

```
struct structName {
    int a;
    int b;
};
```

members of structure

eg:- `struct structName x`

`x.a = 5`

`x.b = 10`

`or`

`struct structName x = {5, 10}`

② Union

- All members shared the same memory location.

```
union abc {
    int x
    int y
};
```

eg:- `union abc z`

`z.x = 5` → also update the value for 'y'

`cout << z.x << z.y`

(z.x → 5, z.y → 5)

③ Enumeration (enum)

→ used to assign names to constants

→ enum EnumName {
Constant 1,
Constant 2 };

eg:- enum week { Mon, Tue, wed, Thu, Fri };

enum week w1;

w1 = fri

→ w1 [4]

* C++ Tokens

→ Identifiers → [int shridhar = 5]
→ keywords → [int, char, if, for .. 95]
→ Constants → [const / #define]
→ Strings → [steing var = "steing"]
→ special symbol → [; . = " ' [] {}]
→ Operators

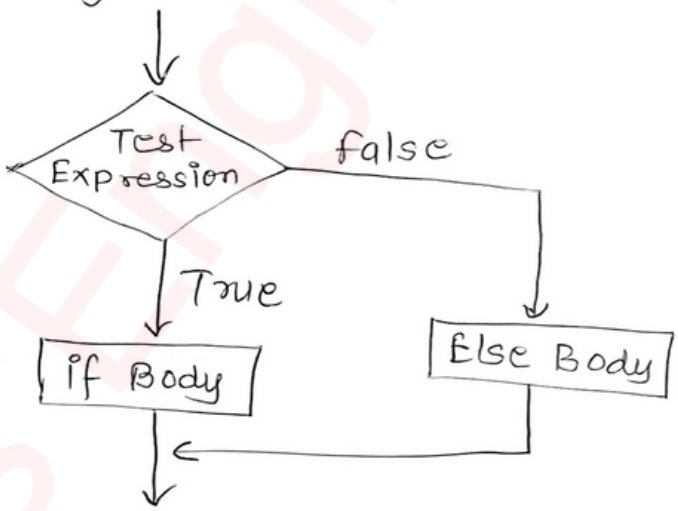
* Operators

- Arithmetic
  - +
  - -
  - *
  - /
  - %
  - ++
  - --
- Assignment
  - =, +=, -=, *=, /=, %=, &=, |=
- Comparison / Relational
  - ==
  - !=
  - >
  - <
  - >=
  - <=
- Logical
  - &&
  - ||
  - !
- Bitwise
  - &, |, ^, <<, >>, ~
- Conditional / Ternary ?: Exp1 ? Exp2 : Exp3

- Control Statements
  - sequential
  - Conditional
    - Decision making
    - switch case
    - Loop

① Decision making

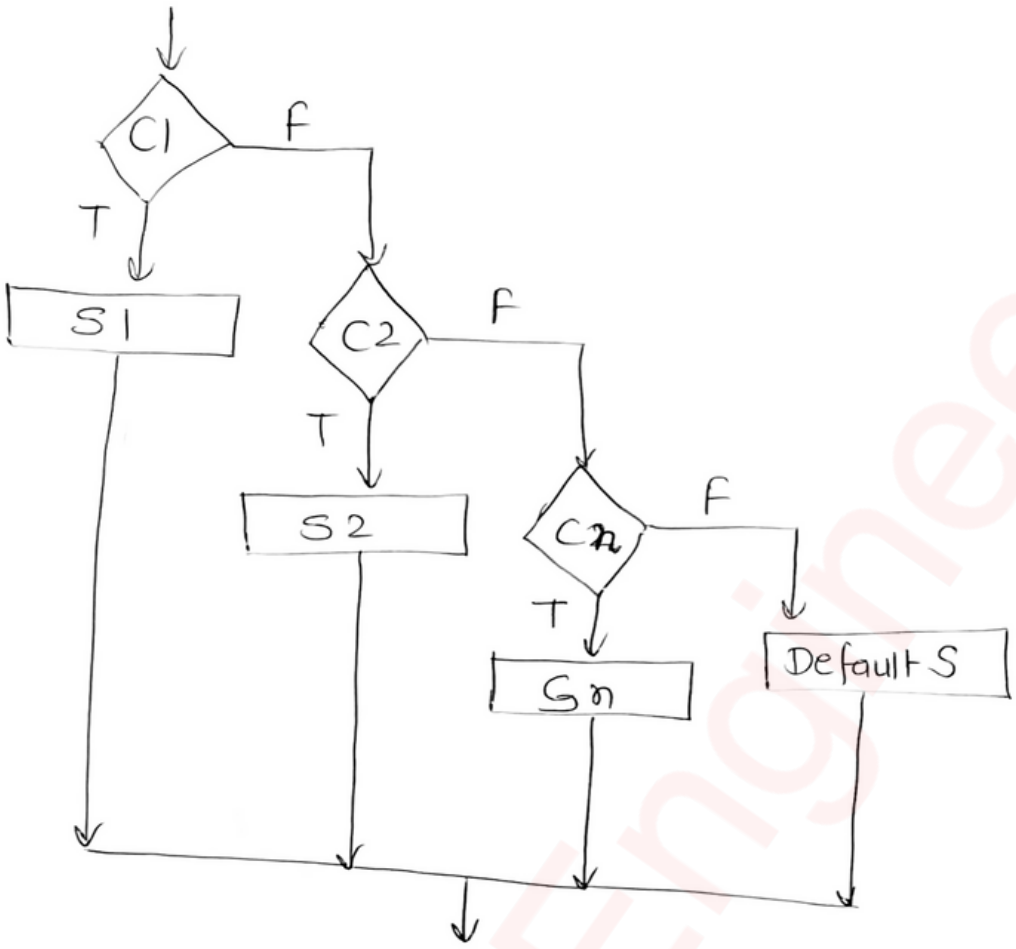
→ if-else



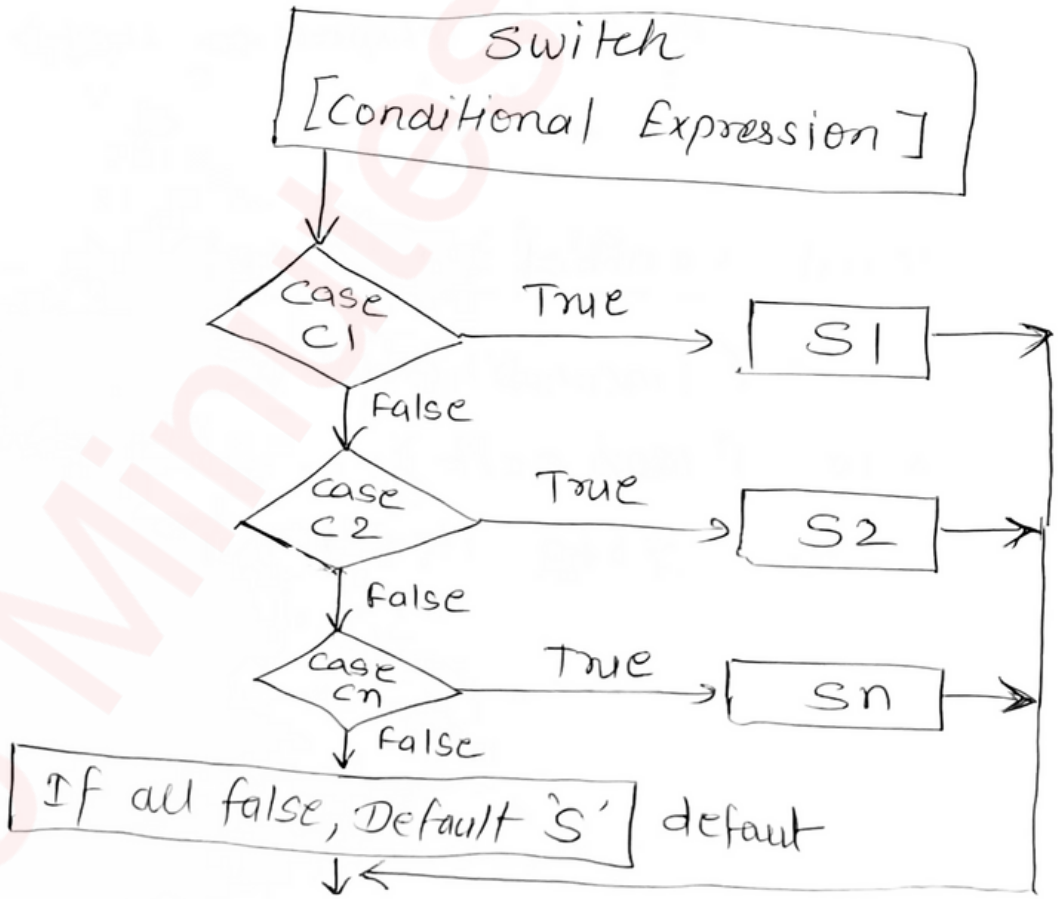
→ Nested If-else

```
If (TE)
  { If (TE)
      {     }
    else
      {     }
  }
else {
   If (TE) { }
   else { }     }
```

→ If else ladder



② Switch Case



③ Loops

- while

eg:-
```
while (x < 5)
{
    body of loop
    (x++)
}
```

- do while

eg:
```
do {
    body
} while (x < 5);
```

- for

eg: `for (Initialization; TestExpression; inc/dec)`
`{ statements };`

* Unconditional control statements
  - Goto (Unconditional branching)
  - break (loop exit)
  - continue (skip iteration)