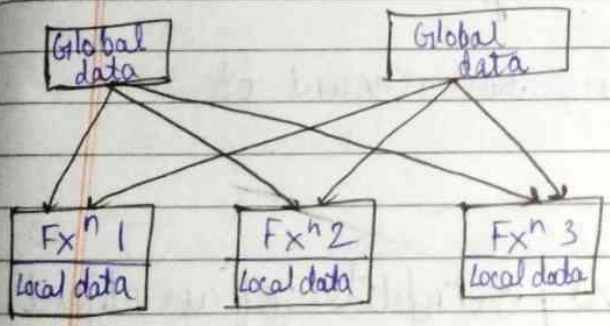
# OBJECT ORIENTED PROGRAMMING

## # PROCEDURE ORIENTED PROGRAMMING V/S OBJECT ORIENTED PROGRAMMING

### PROCEDURE ORIENTED

- Emphasis is on doing things (process/algorithm)
- Programs divided into functions
- Most functions share global data
- Data move openly around the system from $fx^n$ to $fx^n$
- $Fx^n$s transform data from one form to another
- Employs top-down approach



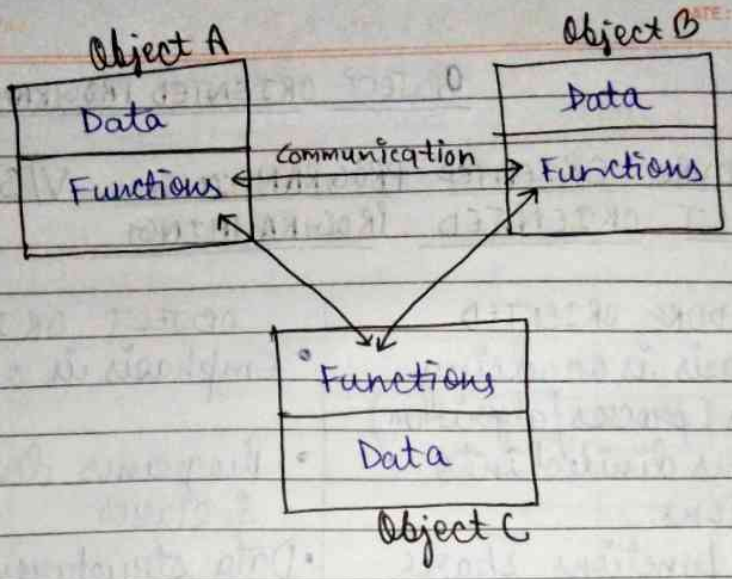

Ex- C, COBOL, FORTRAN

### OBJECT ORIENTED

- Emphasis is on data
- Programs divided into objects & classes
- Data structures are designed such that they characterize the object.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden & cannot be accessed by external functions
- Objects may communicate with each other through functions
- New data & functions can easily be added whenever necessary.
- Follows bottom-up approach

Ex- C++, JAVA, etc.

# C++ is C with classes




## # BASIC CONCEPTS

### ① Objects

↳ run-time entities in an object oriented system.
OR any real world entity.

Ex- any person, place, pen, bank account etc.

### ② Classes

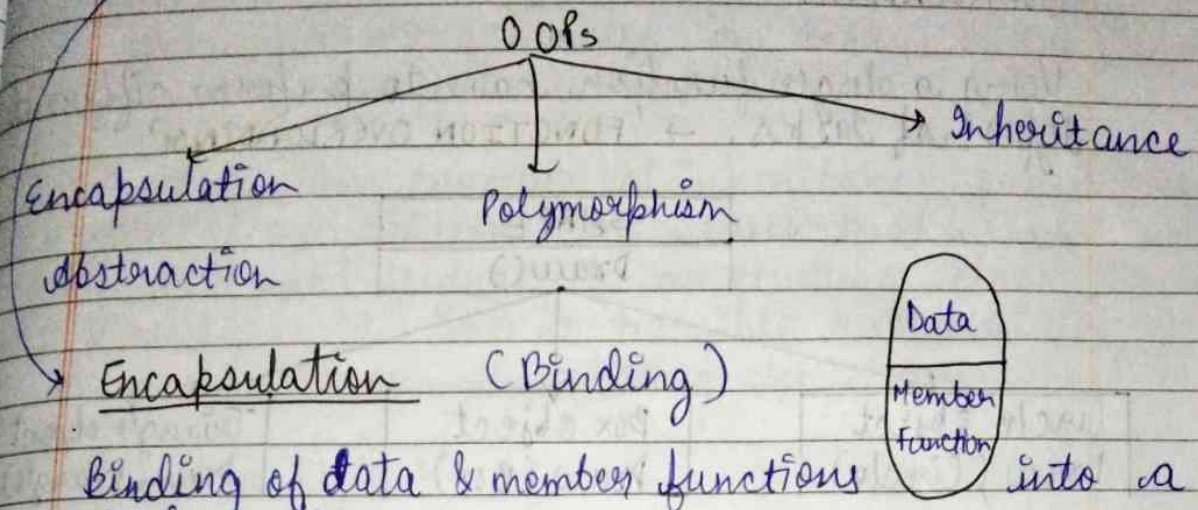
↳ The category / blueprint / template of an object.

↳ The entire set of data and code of an object can be made a user defined datatype using 'class'.

↳ Collection of objects of similar type.

ex- fruit mango;
class object

③ Data abstraction and Encapsulation




→ Encapsulation (Binding)




Binding of data & member functions into a single unit.

Attributes of an object – Data member
Functions that operate on data – Member functions

Abstraction (Hiding)

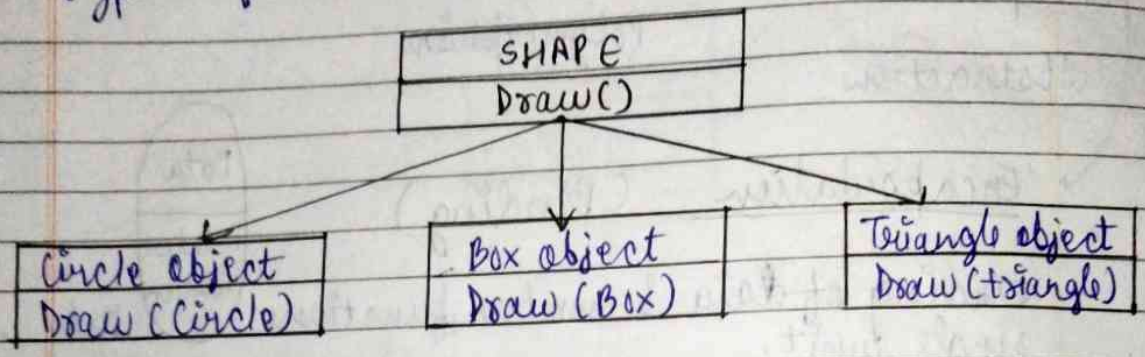
↳ Hiding data from program
↳ representing essential feature without including background details.
↳ "Class" use data abstraction so called Abstract Data Type (ADT)

④ Polymorphism

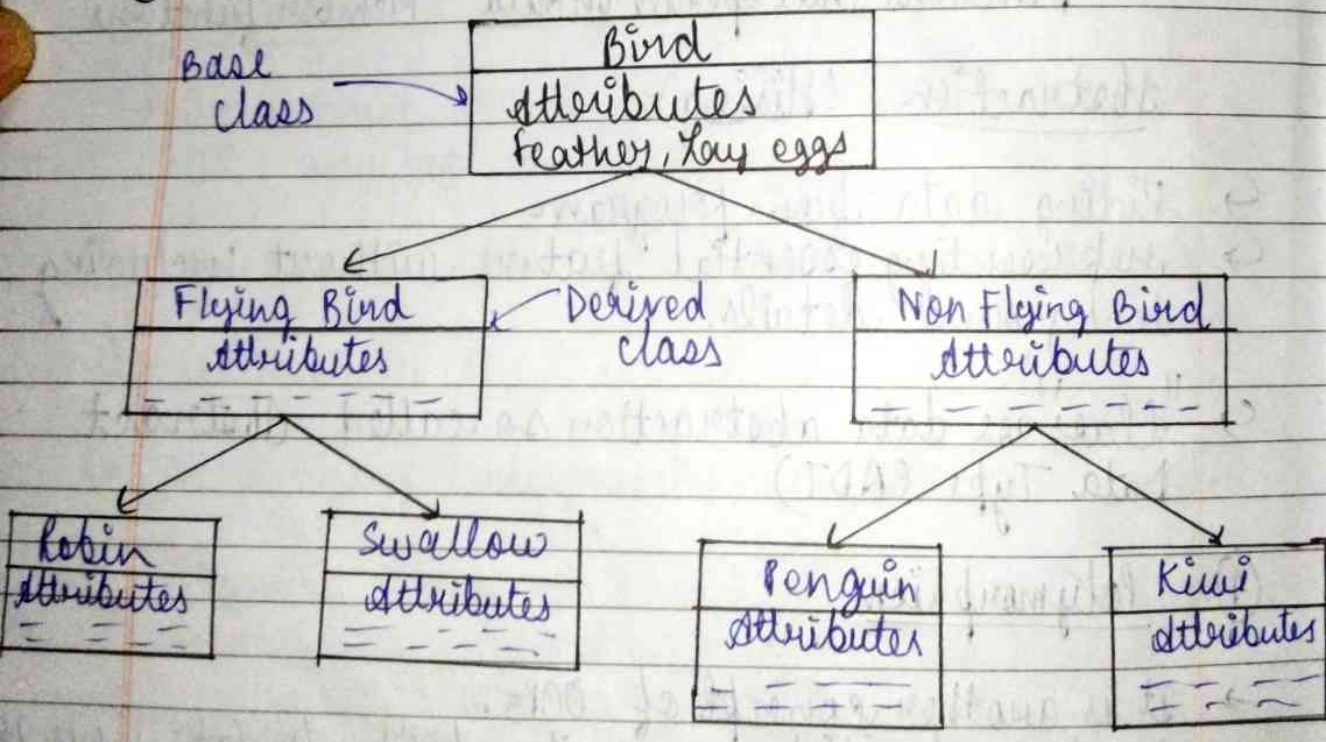
→ It is another concept of OOPs.
→ Greek word which means the ability to take more than one form.
→ The process of making an operator to exhibit different

behaviour in different instances is 'OPERATION OVERLOADING'

Using a single function name to perform different types of tasks. → 'FUNCTION OVERLOADING'




(5) Inheritance (Reusability)




The bird Robin is a part of a class 'flying bird' which is again a part of the class 'bird'. The

Freehand
DATE : / /

principle behind this sort of division is that each derived class shares common characteristics from which it is derived as shown in the figure and this property is inheritance.

In OOPs, the concepts of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one (base class)

Structure of 'C++'

| Include files |
|---|
| Class Declaration |
| Member functions Definition |
| Main function program |

Structure of 'C'

| Include File |
|---|
| Main function Program |

EXAMPLE CODE

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

class person
{
  char name [30];   } Data
  int age;
```

PAGE

```
public:
    void getdata (void);
    void displaydata (void);
};

void person :: getdata (void)
{
    cout << "Enter name : " << endl;
    cin >> name;

    cout << "Enter age : " << endl;
    cin >> age;
}

void person :: displaydata (void)
{
    cout << "name" << name << endl;
    cout << "age" << age << endl;
}

int main ()
{
    person obj;
    obj.getdata ();
    obj.displaydata ();
    return 0;
}
```

function

Member function 1

Member function 2

## ⑥ Dynamic Binding

→ Linking of a procedure call or function to the code to be executed in response to the call

→ It means that the code associated with a given procedure call is not known until the time of the call at run-time.

→ associated with polymorphism & inheritance.

ex- In draw() function. By inheritance, every object will have this function but algorithm is different for each object so draw() function will be redefined in each class that defines the object.

## ⑦ Message Binding

A message for an objects is a request for execution of a procedure (function) & therefore invoke a function in the receiving object that generates the desired result.

Message passing involves :
(i) specifying the name of the object
(ii) the name of the function (message).
(iii) the information to be sent

Ex - employee.salary(name);
object message information

Freehand
DATE: / /

## (8) Benefits of OOP

→ Inheritance can eliminate redundant code & extend the use of existing class.
→ We can build programs from standard working modules rather than writing whole code.
→ It helps to build secure programs through data hiding.
→ It is possible to have multiple instances of an object.
→ It is possible to map objects in problem domain to those in program.
→ It is easy to partition the work in a project based on objects.
→ Data-centered design approach enables us to capture more details of a model in implementable form
→ Message passing makes interface description with external systems simpler.
→ Software complexity can be managed.

## # WHAT IS C++

→ an object oriented language
→ Developed by Bjarne Stroustrup
→ C with classes.
→ bottom-up approach

## # Simple C++ program

```
#include <iostream>  // include header file
using namespace std;
int main()
{
    cout << "C++";  // C++ statement
    return 0;
}
```

* Every program must have main() function

# Comments

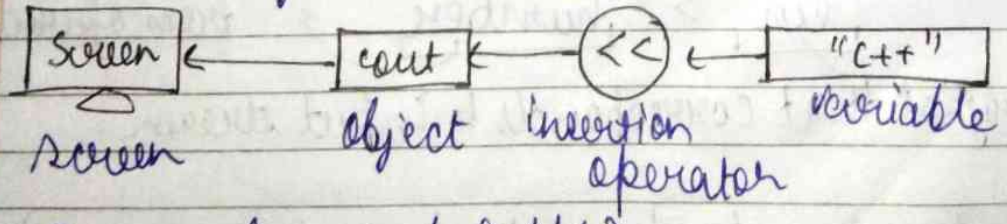
- Single line comment → '//' ex- // C++ program
- Multiline comment → '/*, */' ex- /* C++ is a OOP language */

# Output operator

cout <<  → insertion or put to operator

predefined object that represents standard output stream (screen)

<< → inserts the content of variable on its right to the object on its left.




* We can also use printf().
↳ like printf in C.

Freehand
DATE: / /

# <iostream> file

# include <iostream>
preprocessor directive → specific header file

→ This line adds contents of iostream file to program
→ It contains declarations of functions.

# namespace

Defines a scope for the identifiers that are used in a program.

using namespace std;
namespace

↳ brings all identifiers in std to current global scope

# Return type of main()
function

↳ every C++ ~~program~~ has a default return type "int"
↳ void main() is invalid

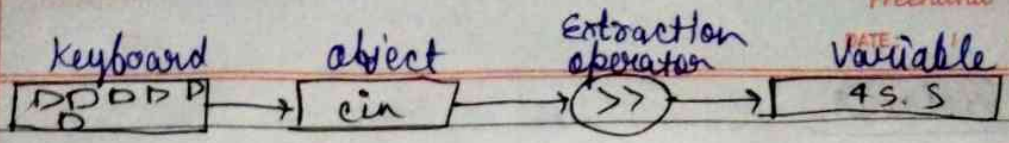
# Input operator

cin >> number
extraction or get from operator

predefined object that corresponds to input stream

↳ extracts value from keyboard & assigns it to variable on right.
↳ like scanf in C.

PAGE:



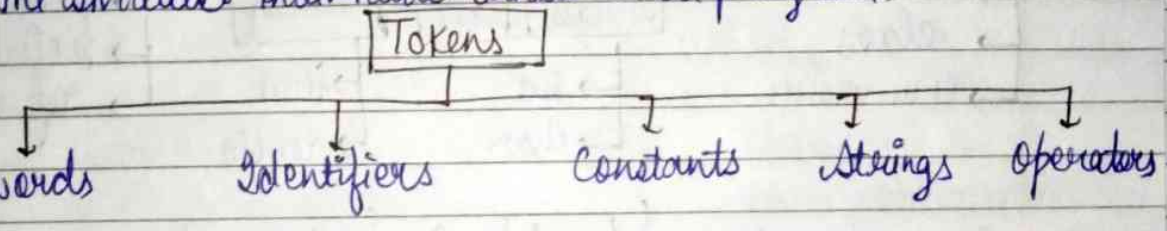

# Cascading of I/O operators

→ multiple use of << or >> in one statement.

ex-
```
cout << "sum = " << sum << "\n";
cin >> num1 >> num2;
```

# TOKENS

→ The smallest individual units in a program.




## ① Keywords

→ Reserved identifiers in C++

ex- auto, break, int, class, etc.

## ② Identifiers & Constants

→ Identifiers are the names of variables, functions, arrays, classes, etc.

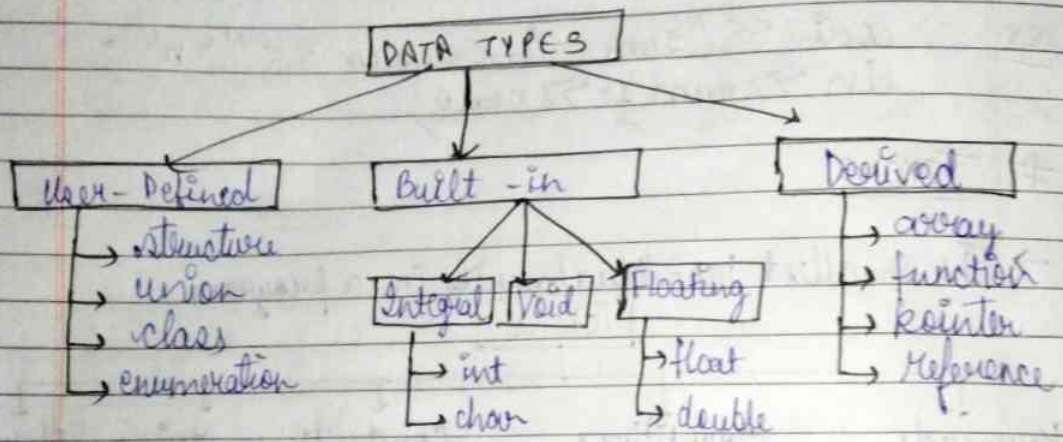
→ Rules:

(1) Only alphabetic characters, digits and underscores are permitted.
(2) The name cannot start with a digit.
(3) Uppercase and lowercase letters are distinct.
(4) Keywords can not be used

→ **Constants**

fixed values that do not change during execution

# DATA TYPES




→ C++ also has 'bool' & 'wchar_t'

→ Use of void :

1) to specify return type of function
2) to indicate empty argument list of function

ex -

```
void func (void);
```

3) Declaration of generic pointers

ex -

```
void * ptr; // generic pointer
```

it can be assigned a pointer value of any basic data type

ex - `int * ip; // integer pointer`

`ptr = ip; // assign int pointer to void ptr.`

But it can not be dereferenced

ex - `* ip = * ptr;` ✗

But,

```
void * ptr1;
char * ptr2;
⇒ ptr2 = ptr1; // Assigning non void pointer a void pointer
```

The above statement is allowed in C but not in C++
In C++, we need to cast:

```
⇒ ptr2 = (char*) ptr1;
```

→ User-Defined Data types

(1) Structures

ex-
```
struct book
{
    char title[25];
    char author[25];
    int pages;
    float price;
};
```
→ struct book b1, b2;

↳ Size is sum of all individual sizes of member objects.

↳ Here, size = 25 + 4 + 4

↳ same as class

↳ all members can be accessed.

(2) Union

ex-
```
union book
{
    char title[25];
    char author[25];
    int pages;
    float price;
};
```
→ union book b1, b2;

↳ Size is the size of largest member object.

↳ Here, size = 25 (max)

↳ not same as class

↳ only one member can be accessed at a time.

Freehand
DATE: / /

(3) Classes

(4) Enumeration
- attaching names to numbers (0, 1, 2 ...)
- using keyword 'enum'
- declaration same as struct

```
enum shape { circle, square, triangle };
shape ellipse ; // ellipse is of type shape

shape ellipse = square; ✓
shape ellipse = 7;      ✗
```

anonymous enums :

```
enum { off, on };
int switch_1 = off; // 0
int switch_2 = on;  // 1
```

# STORAGE CLASSES

| | auto | extern | static | register |
|---|---|---|---|---|
| Lifetime | Function block | Entire program | Entire program | Function block |
| Visibility | Local | Global | Local | Local |
| Initial value | Garbage | 0 | 0 | Garbage |
| Storage | Stack segment | Data segment | Data segment | CPU registers |
| Keyword | auto | extern | static | register |

PAGE:

# Derived Data Types

(1) Arrays
→ same as C
→ Character arrays → size should be one larger than number of characters in string.

but valid in C

```
char string[3] = "xyz";   // Not in C++
char string[4] = "xyz";   // Valid in C++
```

(2) Functions
↳ block of code

(3) Pointers
→ stores address of another variable.

```
int x; int *ip;
ip = &x;   // address of x given to ip
*ip = 10;  // 10 assigned to x
```

→ In C++, we have constant pointer & pointer to a constant.

```
char * const ptr1 = "GOOD";   // constant pointer
```
↳ address can't be changed

```
int const * ptr2 = &m;   // pointer to constant
```
↳ content can't be changed

const → qualifier

# Reference Variables

↳ provides an alias (alternate name) for a previously defined variable.

Freehand

DATE: / /

SYNTAX: `data_type & reference_name = variable_name;`

ex- `int total = 100;`
`int &sum = total;`

total → 100
sum

sum is a reference to the variable total then they can be used interchangeably to represent the variable. They both refer to the same data object in memory.

```
→ cout << total;   // 100
→ cout << sum;     // 100
→ total = total + 10;
  ↳ cout << total;   // 110
  ↳ cout << sum;     // 110
```

```
→ int &x = 50;   x invalid
→ int x;
  int *p = &x;
  int &m = *p;   // m refer to x
```

# OPERATORS

↳ Some additional operators than C:

| | | |
|---|---|---|
| :: | → | Scope resolution operator |
| ::* | → | pointer-to-member declarator |
| →* | → | pointer-to-member operator |
| .* | → | pointer-to-member operator |
| delete | → | Memory release operator |
| endl | → | Line feed operator |
| new | → | Memory allocation operator |
| setw | → | Field width operator |

PAGE:

Freehand
DATE: / /

↳ Scope Resolution operator

:: variable_name → allows access to global version of variable

# Memory management operator

new → allocation of memory like malloc(), calloc()
delete → deallocation of memory like free()

SYNTAX: datatype * pointer_variable;
pointer_variable = new data-type;

ex- int *p;
p = new int;

↳ p is a integer pointer given a size of integer in memory.
*p = 75; // assign 25 to created int object.

↳ new operator allocates sufficient memory to hold the data object & return the address of object.
↳ if memory not available ⇒ it returns bad_alloc exception
↳ We can specify the initial value as:
pointer_variable = new data_type (value);
int *p = new int (25);

↳ Creating memory space for array:
pointer_variable = new datatype [size];
int *p = new int [10];
↳ creates memory to store 10 integers.

→ delete ⇒ delete pointer_variable;
↳ allocated by new;
ex- delete p;

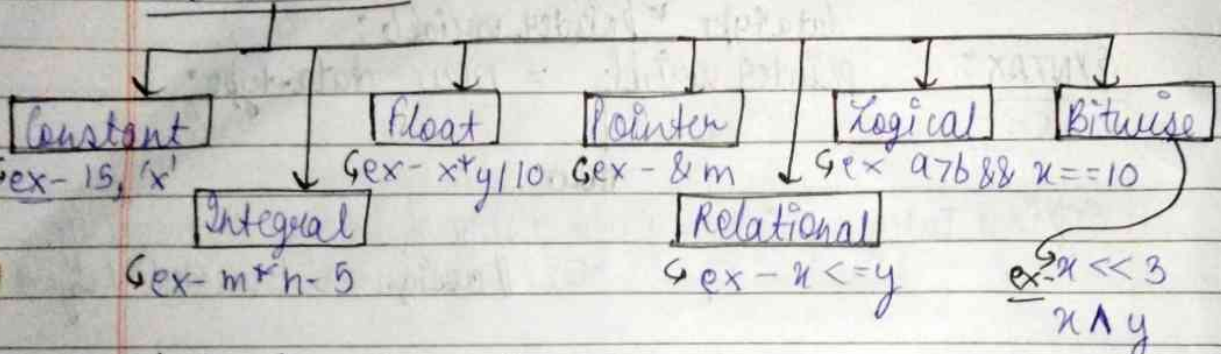
Freehand
DATE : / /

# TYPE CAST OPERATOR

(type-name) expression // C
type-name (expression) // C++

ex- average = sum / (float) i; // C
average = sum / float(i); // C++

# EXPRESSIONS




# chained assignment

x = (y = 10);
OR
x = y = 10;

# Embedded assignment

x = (y = 50) + 10; // x = 50 + 10 = 60
y = 50

# Compound assignment

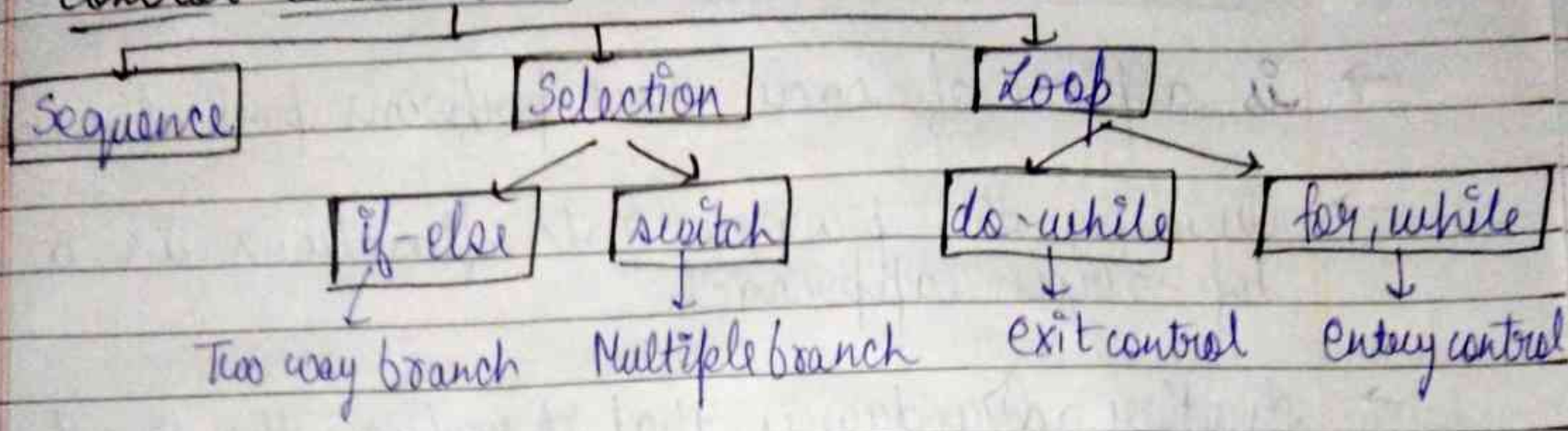
x = x+10; OR x += 10;

PAGE:

Freehand

DATE: / /

# Control structures




→ SYNTAX same as C

Freehand

DATE: / /

# Functions in C++

→ is a piece of code that performs particular task.

→ Dividing the program into functions is a top-down approach

→ Another advantage is that it reduces the size of the program by calling and using them at different places in the program.

example
```
void show();  // function declaration
              Or function prototyping
int main()
{
    show();  // function calling
}
void show() {
    // function definition
}
```

→ In C++, void main() does not work only int main() does.

# Function Prototyping

is a declaration statement in the calling program and it is of the following form:

```
type func_name (arguments);
```

PAGE:

The 'arguments' contains the types & names of arguments that must be passed to the function.

Function to add two numbers

```
int sum ( int a, int b);
```

* Note that each argument variable must be declared independently inside the parenthesis "()" i.e. a combined declaration like :

```
float sum (int x, float y z);  ✗
```

is illegal.

* In a function declaration the names of the arguments are dummy variables therefore they are optional. So, we can also write :

```
int sum (int , int );  ✓
```

# Call by value & Call by Reference

① Call by value

→ copy of the variable is passed.
→ changes made in the called function are not shown in caller fx$^n$.

```
void swap (int x, int y);
{
  int t;
  t = x;
  x = y;
  y = t;
```

```
cout << "Value in swap func" << x << y;
        // "Value in swap func 20 10
}
int main() {
    int x = 10, y = 20;
    swap(x, y);
    cout << "Value in main fun" << x << y;
}   // Value in main func 10 20
```

## ② Call by reference

→ reference / address of the variable is passed
→ changes made in called function are visible in calling program.
→ can be done using pointers or reference variables

**Reference variables**

```
void swap (int &a, int &b)
{
    int t = a;
    a = b;
    b = t;    // 20 10
}
int main()
{
    int m = 10, n = 20;
    swap(m, n); // 20 10
}
```

**Pointer**

```
void swap (int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;    // 20 10
}
int main()
{
    int m = 10, n = 20;
    swap(&a, &b); // 20 10
}
```

m n

Freehand
DATE: / /

# INLINE FUNCTIONS

- Every time a function is called, it takes a lot of time in executing a series of instructions for tasks such as jumping to the functions, saving register, putting arguments into the stack and returning to the calling function.
- When a function is small, a substantial percentage of execution time may be spent in such overheads.
- C++ has a different solution to this problem:
- To eliminate the cost of calls to small functions, C++ proposes a new feature called **inline function.**

SYNTAX: inline return-type func-name (arguments)
{
---
}

```
inline double cube (double a)
{
    return (a*a*a);
}
```

- Some of the situations where inline expansion may not work:

(1) For functions returning value, if a loop, switch, go to exist.
(2) For functions not returning values if a return statement exist

PAGE:

(3) For function containing static variables

(4) for recursive functions.

# Default arguments

```
int mul (int i, int j, int k=5)
{
}
```
↳ legal

```
int mul (int i, int j = 10, int k)
{
}
```
↳ not legal

```
int mul (int i=2, int j=10, int k=5)
{
}
```
↳ legal

↳ useful when some arguments always have same value
→ It always starts from right

# Recursion

→ When a function calls itself.

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

long fact (int n) {
    if (n == 0)        // base case
        return 1;
    return (n * fact (n-1));
}                      // recursive function call
```

Freehand

DATE : / /

```
int main() {
    int num;
    cout << "Enter positive integer";
    cin >> num;
    cout << "Factorial of : " << num << " is : "
        << fact (num);
    return 0;
}
```