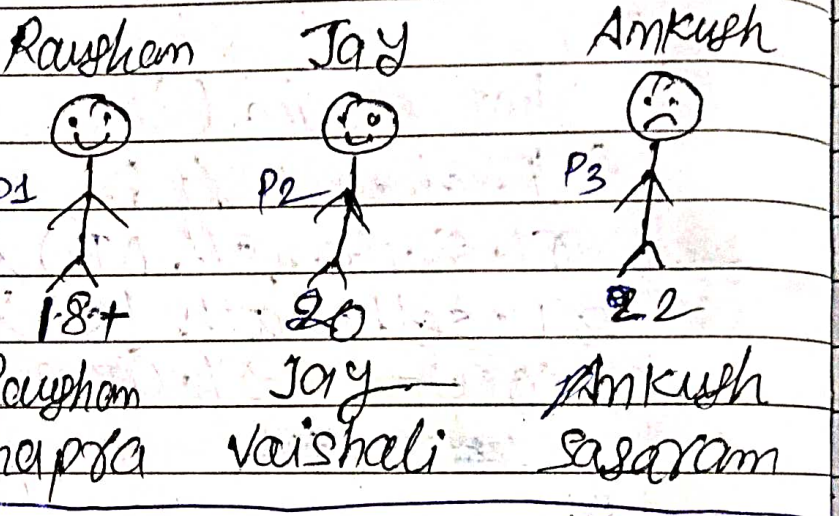
OOPS

* what is OOPS ?

→ Object oriented programming is a programming approach that are based on classes and objects which can contain data and code that manipulate that data.
(inst fun)

Ex ⇒
```
Class person
{
  int Age;
  string Name;
  string Address;
};
int main()
{
}
```



* what is class ?

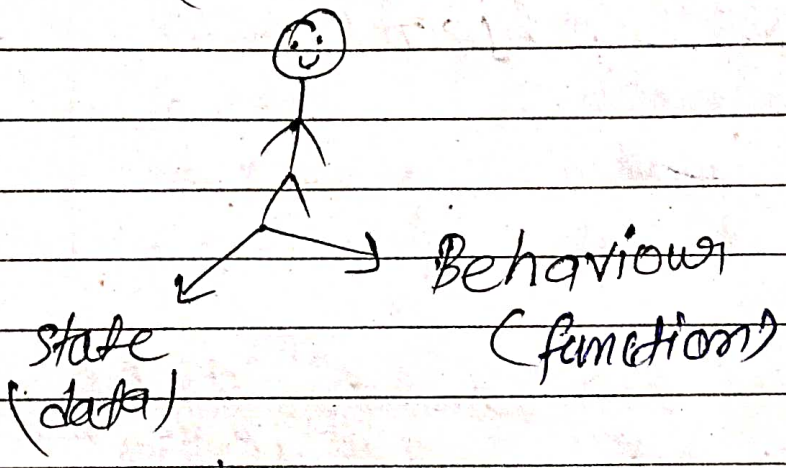
ans- class is a user-define datatype or blueprint that wrapped data and functions into a single entity.

Syntax :-
```
class class_name
{
  data
  +
  function / Methods
};
```

* What is object?

ans → Object is a concrete representation of the blueprint that is defined by the class.

(Ankush)




```
class person {
private:
int age;
string name;
string address;
public:
void input() {
cout << "Enter age: ";
cin >> age;
Cout << "Enter Name: ";
cin >> name;
Cout << "Enter address: ";
cin >> address;
}
void show() {
Cout << "Age:" << age << endl;
Cout << "Name:" << name << endl;
Cout << "Address:" << address << endl; }
```

```
void main() {
Person p1, p2, p3;
p1.input();
p1.show();
p2.input();
p2.show();
p3.input;
p3.show;
}
```

Enter age: 20
Enter name: Rayhan
Enter address: bihar
output is
age: 20
Name: Rayhan
address: bihar

* features of OOPS

1. Encapsulation.
2. Abstraction.
3. polymorphism.
4. inheritance.
5. class & object.

* Storage classes

Storage class	Keyword	Liftime	visibility
Automatic	auto	function Block	Local
External	extern	Whole program	Global
Static	static	whole program	Local
Register	register	function block	Local
Mutable	mutable	class	Local
Thread local	thread-local	whole thread	Local or global

initial value	→ auto
Garbage	→ extern
zero	→ static
zero	→ register
Garbage	→ mutable
Garbage	→ thread-local
Garbage	

* Storage class: A storage class defines the scope and lifetime and visibility of any variable.

Types: auto, static, extern → RAM
Register → Register

* Auto :- By default all variables are auto. It is use to declare automatic variable. It is a local visibility is local & lifetime only in function block and it's initial value is garbage.

Syntax :-

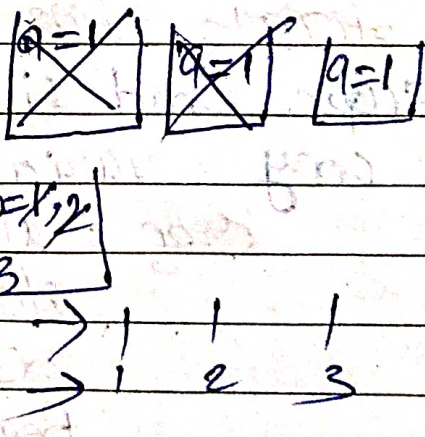
auto datatype var_name;
datatype var_name;

* Static Storge class :- It is use to declare static variable, It can be used as local or global variable, It retains their value between two function call. It's lifetime is in whole program, It's initial value is zero.

Ex → static datatype varname;
static int number;

Ex →

```
void main()
{ fun();
  fun();
  fun();
}
void fun()
{ int a=1;
  static int b=1;
  cout<<a<<endl<<b;
  a++;
  b++;
}
```



* Extern Storage class :- It is use to declare extern variable, It can do the linking between the global variable in two or more files. It's lifetime is in whole program, visibility global and initial value is zero.

Syntax: extern datatype var_name;

Ex⇒

file1.cpp

```
#include<iostream>
using namespace std;
int a;
extern int fun();
extern void fun();
void main()
{ a=15;
  fun();
}
```

file2.cpp

```
#include "file1.cpp"
int a;
void fun()
{ cout<<a;
}
//15
```

* Register Storage class :- It is use to declare register variable. It store variable in register of the microprocessor It can be accessed as fast as possible, lifetime only in function block and visibility local, initial value garbage.

syntax: register datatype var_name;

```
int main()
{ register int a=10;
  cout<<a;
} return 0;
```

* Mutable storage class: It is used to declare mutable variable, If there is requirement of modify one or more data member of a class then through mutable storage class we can do that even if variable is initialised by const, it's lifetime only in class visibility local & initial value garbage.

→
```
class demo
{ public :                          → 100
mutable const int x = 10;
      const int y = 20;

};
int main ()
{ demo d;

cout << d.x << endl;
cout << d.y << endl;
return 0;
}
```

before mutable
10
20
after mutable
100
20

* Thread local storage ~~Color~~ class: The thread-local variable can be combined with other storage specifiers like static or extern and the properties of the thread-local object changes accordingly. It's memory location in RAM & lifetime till the end of its thread.

Q.1 ⇒ What components constitute the basic structure of a program, and how do they ~~includes the~~ contributes to its organization

Ans ⇒ The basic structure of a program typically includes the following components which contribute to its organization.

(i) Includes: Depending on the programming language, you may need to include libraries and headerfiles modules to access pre-defined functions and classes that extend the program's capabilities.
Ex ⇒ #include<stdio.h>, #include<iostream>

(ii) functions / Methods: functions or methods are blocks of reusable code that perform specific tasks. They help in organizing code into manageable and reusable sections.

Ex ⇒ ~~main()~~
f(1) ⇒ p ( )
{
}

(iii) Main function: In many programming languages, a program starts executions from a designated main function. It serves as the entry point for the program.

(iv) Classes and objects : In object-oriented programming languages like java or python, classes and objects are fundamental for organization and modeling data and behavior. class file { }

(v) Conditional statement : These allow the program to make decisions based on certain conditions. Ex → if statements, switch statements. if( )

(vi) Loops : loops enable the program to repeat a set of instructions multiple times. Ex → for loop while loops, do-while loops.

→ for(int i=0 ; i<n ; i++)

→ while( ) , do { } while( );

{ }

(vii) variables : Variables are used to store data. They can be of different types. Example : Integers, strings, floats.

(viii) data types : data types define the type of data that variables can hold. for example int, float, string, boolean. This helps the program understand how to handle the stored data.

(ix) Input/output : Input mechanisms allow users to provide data to the program. While output mechanisms display results.

(x) Error Handling : Code should handle errors and exceptions gracefully to prevent crashes and provide meaningful.

(xi) Comments : Comment are used for provide human-readable explanations within the code. They don't affect the program's execution but are essential for documenting.

```
// a+c=c
```

These components collectively contribute to the organization of a program by providing structure, readability, and maintainability. Properly organizing code using these components makes it easier to understand, debug, and extend the program as needed.;

(Q5) Token, keywords, identifiers

Q2. What are tokens in programming, and how do they serve as the fundamental

ans ⇒ In programing, tokens are the smallest units of code in a programming language. They serve as the foundamental building blocks of code syntax, helping the compiler or interpreter understand the structure of the program. Token can be categorized into several types.

1. Keywords: These are reserved words in the programming language that have special meanings, Examples: "if", "else", "for", "while".

2. Identifiers: Identifiers are used to name variables, functions and other user-defined elements. They typically consist of letters,

```
int sum = a + b;
```

3. operators: Operators perform operation on variable and values → Example: include (+) addition, (-) subtraction (-) and assignment (=) are used

4. Literals: Literals represent fixed values in the code. Common type include integer literals (eg, 42), string literals (eg. "Hello world") and floating-point literals (eg, 3.14)

(5) Punctuation: Punctuation tokens include symbols like parentheses (), curly braces {}, commas (,), and semicolons (;) that are used to define the structure of code blocks.

(6) Comments: Comments are used for documentation and are ignored by the compiler or interpreter.
Eg ⇒ Single line comments → //
multi-line comments → /* */

Q3 Can you provide examples of keywords in programming languages, and describe their significance in controlling the flow of a program.

ans ⇒ Keywords in programming languages are reserved words that have special meanings and play a crucial role in controlling the flow of a program. Here are some examples in various programming language.

1. Python: If, else, elif used for conditional statements to control the flow based on conditions.
- for, while: Used for loops to iterate over data or execute code repeatedly.
- break, continue: Control the flow within loops allowing you to exit a loop prematurely or skip an iteration.

Q^ Java :

(A) Inheritance in C++ : Inheritance is a feature or a process in which the new class created is called "derived Class" or child class and existing class is known as "Base class" or "parent class". The derived class inherits all the properties of the base class without changing the properties of the base and may add new features to its own.

Implementing inheritance → syntax

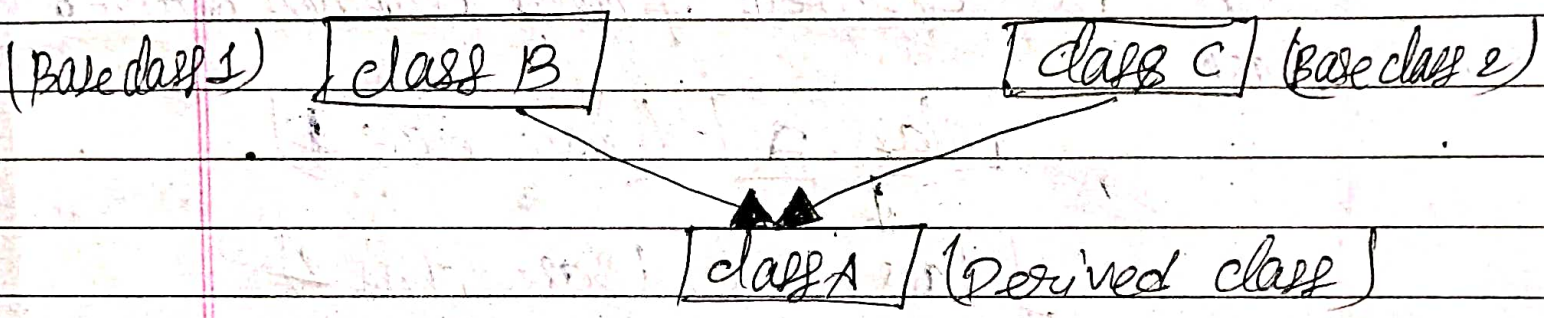
```
class derived class name : access specifire base class name
{
    // body
};
```

* Types of Inheritance :
1. single inheritance
2. Multilevel "
3. Multiple "
4. Hierarchical "
5. Hybrid "

(1.) single Inheritance : In single inheritance a class is Inherit from only one class.

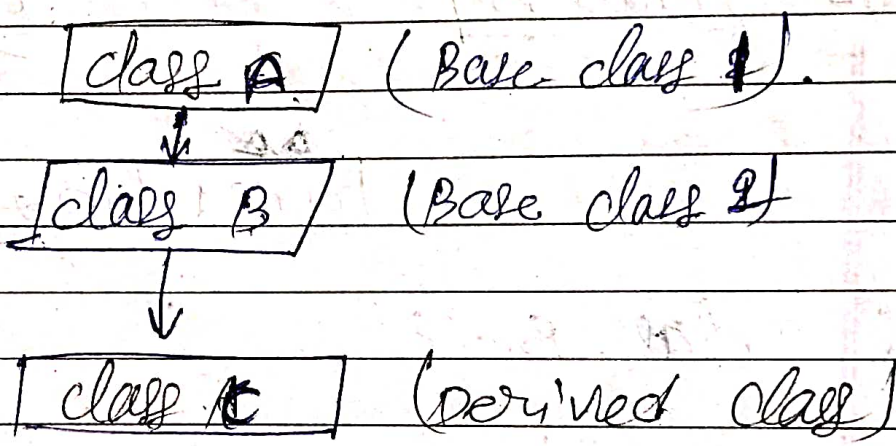
* Syntax: class classname(A)
{
body
};
class B : public A
{
};

(2) Multiple Inheritance: In Multiple Inheritance A class can inherit from more than one class.



Syntax:
class B. {
};
class C {
};
class A: public B, public C
{
};

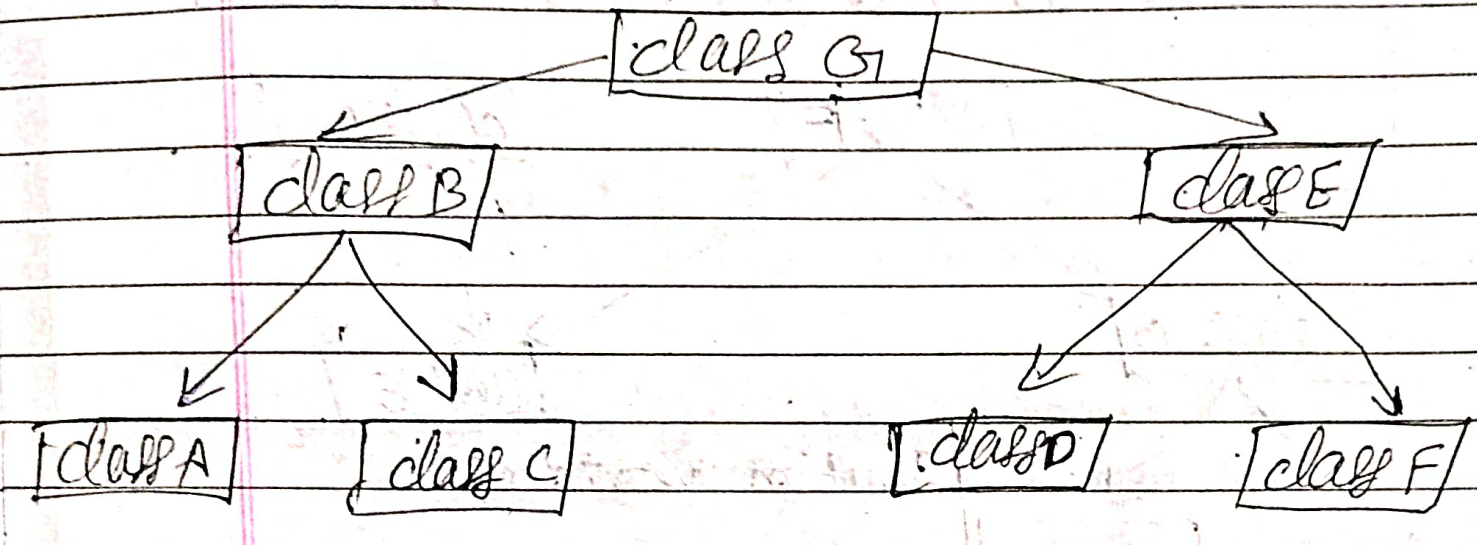
③ Multilevel Inheritance : In this A derived class is created from another derived class.



Syntax :

```
class A
{
};
class B : public A
{
};
class C : public B
{
- - -
};
```

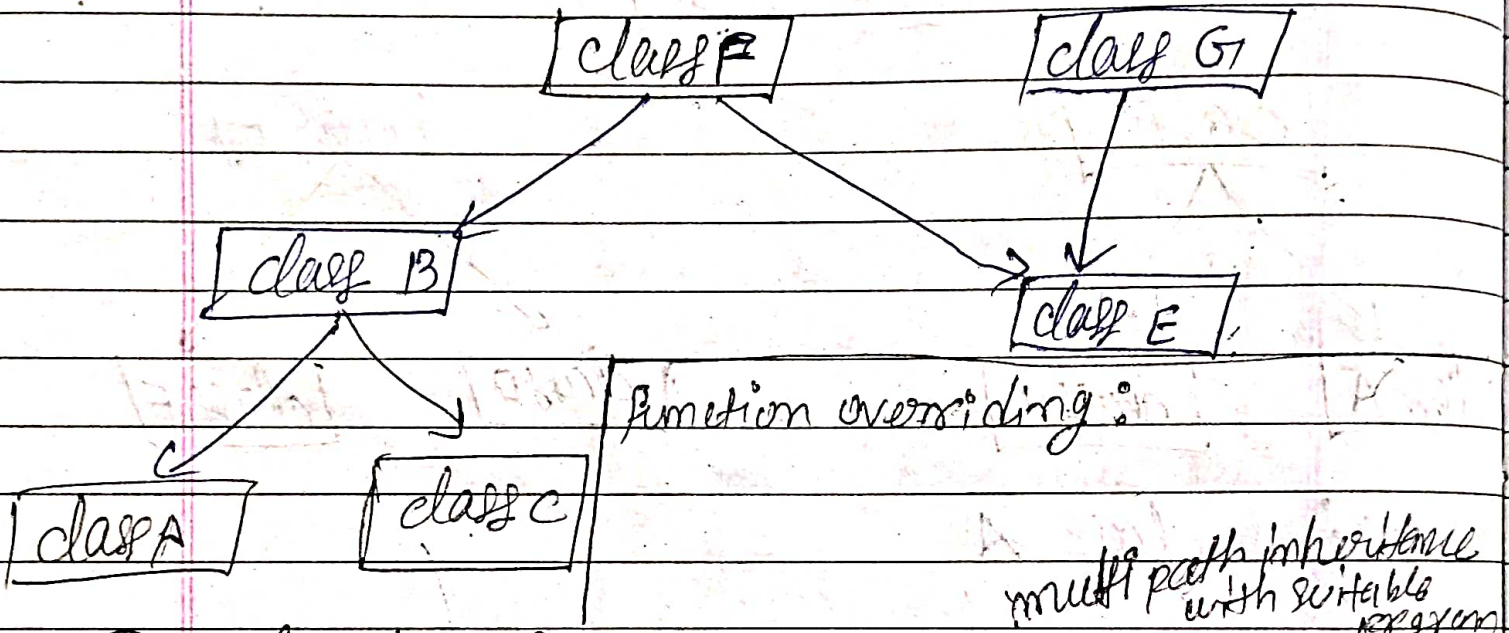
(4.) Hierarchical Inheritance: In this inheritence More than one derived class is created from a single base class.



syntax:

```
class A
{
};
class B : public A
{
};
class c : public A
{
};
class D : public A
{
};
```

(5) Hybrid inheritance : Hybrid inheritance is implemented by combining more than one type of inheritance. for example : Combining Hierarchical inheritance and Multiple inheritance



Ambiguity problem in C++.

```
class A
{ public:
  void func()
  {
  cout<<"I am in class A"<<endl;
  };
};
  class B
{ public:
  void func()
{ cout<<" I am in class B"<<endl;
};
};
```

```
class C : public A, public B
{

};
int main()
{
  C obj;                    error
obj.func(); // 'func' is ambiguous
return 0;
  }
```

→ Ambiguous resolution: Same code as above but some change in main function.

```
   int main()
{ C obj;

  obj.A :: func();
  obj.B :: func();
  return 0;
}
```

* function overriding : when a function is define in both class having same name & signature then when you create object of derived class the function of derived class will be execute and function of derived class will override the function of base class. this concept is called function overriding. also known as late binding / runtime polymorphism

Ex →
```
class A
{ public :
  void fun()
{ cout<<"I am base class of class A:"<<endl;
}
};
class B : public A
{ public :
  void fun()
{
cout<<"I am derived class of class B:"<<endl;
}
};
int main()
{ B obj;
obj.fun(); //I am derived class of class B:
obj.A::fun(); //I am base class of class A:
return 0;
}
```

Page No.
Date 20

* This keyword : Whenever the name of instance variable and local variable both are same and if we initialize instance variable with the help of local variable then our compiler will gets confused that which one is local variable and which one is instance variable to overcome this problem we use "this" keyword with arrow operator (->)

```
A(int a, int b)
{this->a=a;
 this->b=b;
}
```

```
#include<iostream>
using namespace std;
class A
{int a;
 int b;
public:
A(int a, int b)
{this->a=a;
this->b=b;
}
void disp()
{cout<<a<<" "<<b;
}};
int main()
{A a(3,6);
a.display(); return 0;}
```

* Exception Handling :
(PP * 8 mar) Explain the exception handling mechanism write a program to catch all exceptions in a single catch block.

case ⇒ An exception is an unexpected problem that arises during the execution of a program.

Exception Handling : Exception handling mechanism provide a way to transfer control from one part of program to another. This makes it easy to separate the error handling code from the code written to handle the actual the actual functionality of the program.

other w →

Exception handling is a mechanism in c++ that allows you to deal with errors or exceptional situations in your code. it provides a structured way to handle unexpected program execution.

Eh built upon three keywords try, throw, catch.

- Try : Try statement allows you to define a block of code to be tested for errors while it is being executed.
- Throw : The throw keyword throws an exception when a problem is detected;

- catch : The catch statement allows you to to define a block of code to be executed if an error occurs in the try block. try and catch keywords come in pairs. Ex:

```
int main()
{
int numerator, dinominator, result;
Cout << "Enter two numbers:" << endl;
Cin >> numerator >> dinominator;
try
{ if (dinominator == 0)
  { throw dinominator;
  }
result = numerator / dinominator;
}
Catch (int ex)
{
Cout << "dinominator zero is not allowed:"
   << ex << endl;
}
cout << result << endl;
  return 0;
}
```

PQ: when do we need multiple catch blocks for a single try block ? give an example.

We need multiple catch blocks for a single try block when we need to handle different types of exceptions in different ways. Ex:-

```
int main()
{ try {
        throw 10;
      }
catch (double e)
{ cout << "Exception occurs" << e << endl;
}
catch (int e)
{ cout << Exception occurs" << e << endl;

catch ( [redacted] char *e)
{ cout << "Exception occurs" << e << endl;
}
return 0;
}
```

or

```
int main()
{ try { throw 10;
      }
catch (...)
{ cout << "Exception occurs" << endl;
}
return 0;
}
```

Q* Why do you think that exception handling is important in programming.

ans ⇒ Exception handling is crucial in programming for several reasons

(i) Error management.
(ii) Prevent crashes.
(iii) Robustness.
(iv) Debugging
(v) User - friendly
(vi) Predictability
(vii) Maintainability.

Q* [redacted] How would you explain the user-defined exceptions.?

ans: A user defined exception in c++ is a custom error message that a programmer can create to handle specific errors in their code. This is done by creating a class that inherits from the std::exception class. This new class can then be thrown and caught in a try/catch block.

```
#include <iostream>
#include <exception>
 using namespace std;
class MyException : public exception
{ public :
 char * what ()
 { return "C++ Exception occurred";
 }
};
   int main ()
{   try
   { throw MyException();
}}
   catch (MyException e)
   {
   cout << "..e.what() << endl;
   }
   catch (exception e)
{  other errors
   }
 return 0;

}
```

Q5

```
#include<iostream>
#include<exception>
using namespace std;
class overspeed: public exception
{ int speed;
public:
const char *what()
{
return "check your speed you are in the car
 not in an aeroplane"<<endl;
}
};
int main()
{ int carspeed=0;
try
{ while(1)
{ carspeed= carspeed+10;
 if(carspeed>100)
 { overspeed s;
  throws;
 }
cout<<"carspeed:"<<carspeed<<endl;
}
} catch(overspeed ex)
{ cout<<ex.what();
}
return 0;
}
```

```
carspeed: 10
        : 20
        : 30
        : 40
        : 50
        : 60
        : 70
        : 80
        : 90
        : 100
check your speed
you are in car
not in aeroplane
```

(Q9) Define file pointers in detail. Construct a program justifying their use.

Ans ⇒

```
#include<iostream> using namespace std;
int main()
{ int size;
Cout<<"Enter the size of the array:";
 cin>> size;
int * arr = new int[size];
for(int i=0; i< size; i++)
 { arr[i] = i*10;
  }
for(int i=0; i<size; i++)
{ Cout<<arr[i]<<" ";
}
delete[] arr;
 return 0;
}
```

Uncatch exception


```
#include<exception>
#include<iostream>
using namespace std;
void myfile()
{
cout<<"Inside file";
 abort();
}
int main()
{
set_terminate(myfile);
try{ cout<<"inside try\n";
   throw 100;
}
catch(char a)
{
cout<<"Inside catch block";
}
cout<<"outside";
   return 0;
}
```

* Difference between early binding and late binding.

	Early binding	late binding
(i)	It is also known as static binding.	It is also known as dynamic binding.
(ii)	faster execution than late binding	slower execution than early binding.
(iii)	It is a compile time polymorphism.	It is a runtime polymorphism
(iv)	Ex ⇒ inline function, function overloading	Ex ⇒ function overriding
(v)	Less flexible.	More flexible.

Page No.
Date 29

Object oriented paradigm:

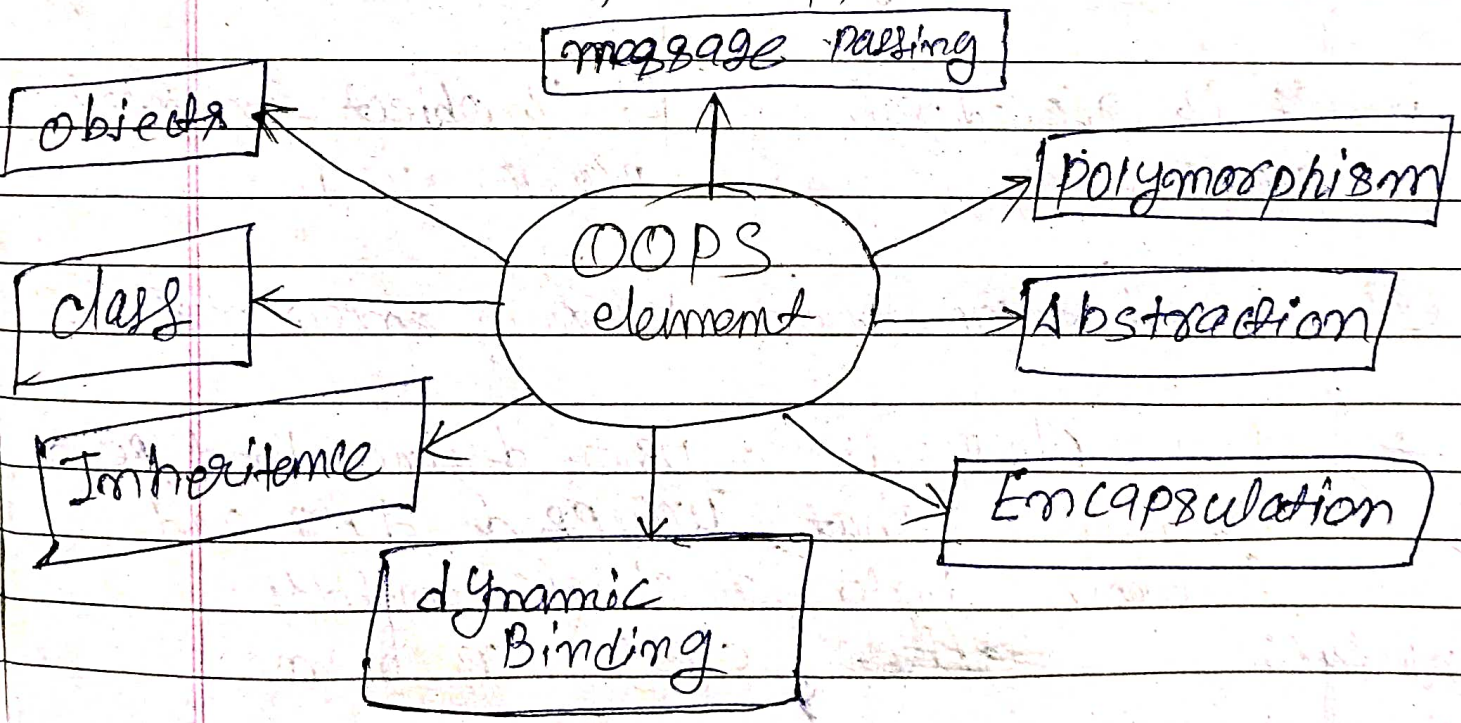
* Need of object oriented programming:

Ans: Object oriented programming (OOP) offers several advantages and fulfills various needs in software development.

(i) Modularity: (ii) Reusability (iii) Encapsulation (iv) Abstraction (v) Inheritance (vi) polymorphism (vii) classes & objects (viii) readability

OOPs helps in building robust and scalable software systems, promoting code reusability, maintainability, and collaboration among developers.

* Elements of OOPs



* difference b/w structured programming and objects oriented programming.

structured	OOP
(i) It break a problem into smaller parts.	It organize code around objects that represent real world entities.
(ii) Easy to understand and debug, efficient use of memory.	Reusable code, easier to maintain and extend better.
(iii) It focus on functions and procedures.	It focus on objects and classes.
(iv) It is procedural programming.	It is object-oriented programming.
(v) Limited encapsulation.	Strong encapsulation.

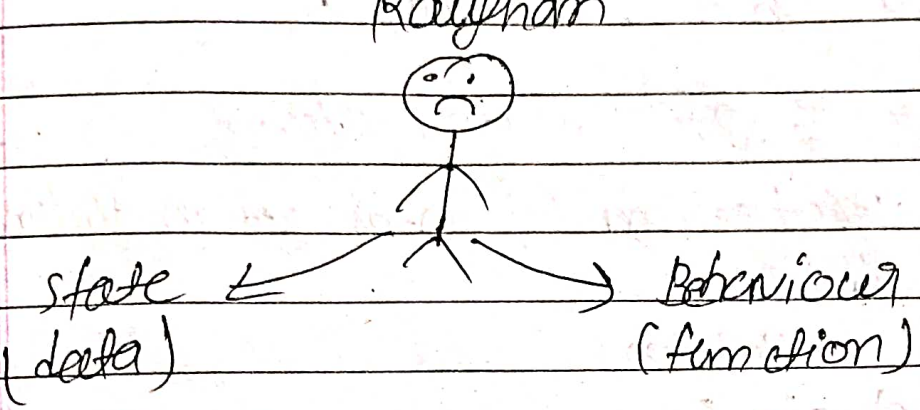
* Class :- Class is a user-define datatype or blue-print that wrapped data and functions into a single entity.

syntax:

```
class Class-name
{
    data
    +
    Methods
};
```

* How do you define classes and objects

* Objects: Objects is a concrete representation of the blue-print that is defined by the class.




```
class class_name
{ Access specifier: // private or public
  Data members ; // variables
  Member functions() // access data members
  {
  }
};                    // class ends with a semicolon
```

Declaring objects

Syntax:

```
ClassName objectName;
```

* Encapsulation in c++: Encapsulation defined as the wrapping up of data and information in a single unit or binding together the data and functions.

properties of Encapsulation

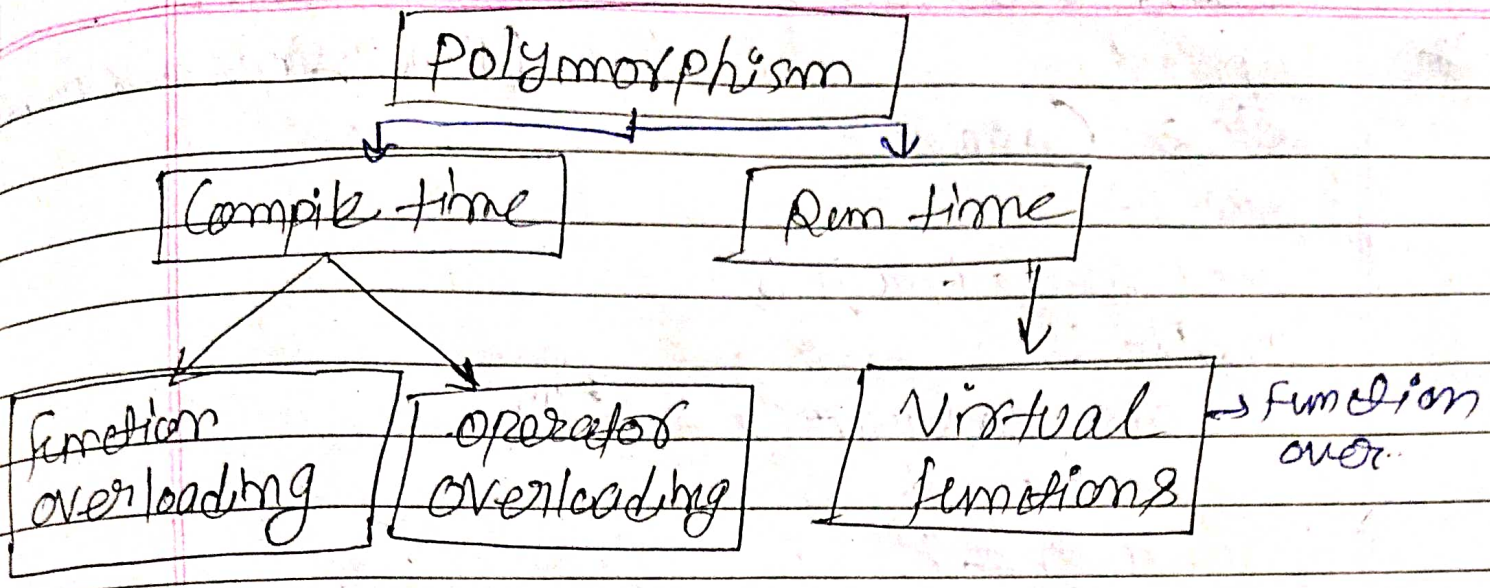
(i) Data protection (ii) Information Hiding
↓
private

Ex⇒
```
class person
{ private:
  data
  public:
  function ()
  {
  }
};
```

* polymorphism: Polymorphism is a concept in which an object can be treated in different ways. It means that objects of a class can be used as more than on form. Ex⇒

A man can be a husband, teacher, son, brother.

- polymorphism:
  - → static polymorphism
  - → Dynamic polymorphism



* function overloading:

```
class Rau
{ public:
  void fun()
{cout<<"no argument"<<endl;
}
 void fun(int x)
 {
 cout<<"argument"<<endl;
 cout<<x; }
void fun(string x)
{ cout<<"string"<<x<<endl;
};  int main()
{ Rau o;
  o.fun();
  o.fun('4');
  o.fun("Raghav");
}
```

* Binary operator overloading: A operator which contain two operands is called binary operator overloading.

* Operator overloading (Binary operator overloading)

```
class Complex
{ private:
  int real, imag;
  public:
    Complex(int r, int i)
  { real = r;
    imag = i;
  }
  Complex operator+(Complex obj)
  { Complex res;
    res.img = img + obj.img;
    res.real = real + obj.real;
    return res;
  } void display() { cout<<real<<"+"<<img<<"i"<<endl; }
int main()
{ Complex c1(12,7), c2(6,7);
  Complex c3 = c1+c2;
  c3.display();
  return 0;
}
```

// 18+i14.

~~function overriding~~

* ~~Binary~~ operator overloading: To assign more than one operation on an same operator knows as operator overloading.

* Syntax :-

```
return type operator op(arg list)
{ body;
}
```

* Unary operator overloading: A operator which contain only one operand is called unary operator overloading -

return type operator op ( )
{ body;
}

Ex ⇒

```
class hello
{ private:
  int kg;
  public:
  hello (int k=0)
  { kg = k;
  }
  hello operator ++ ()
  { hello h;
    ++kg;
    h.kg = h.kg;
    return h;
  }
  hello operator ++(int)
  { hello h;
    kg++;
    h.kg = h.kg;
    return h;
  hello operator --()
  { --kg;
    h.kg = h.kg;
    return h; }
  hello operator --(int)
  { hello h;
    kg--;
    h.kg = h.kg;
    return h;
  }
  void display ()
  { cout << "\n is" << kg << endl;
  }
};
int main ()
{ hello obj, obj1;
  obj.display ();
  ++obj;
  obj.display ();
  obj++;
  obj.display ();
  [illegible]
  obj1 = obj;
  obj1.display ();
  --obj;
  obj.display ();
  obj--; obj.display ();
```

* Inheritance: Inheritance allows a class to inherit the properties and behaviour from another class.

Example: class Base
{ data + functions add()
};
class Derive: public Base
{ add();
sub();
};
class father {
protected:
string surname = "Singh";
};
class son1: father {
string name = "chandan";
public:
void show() {
cout << name << " " << surname << endl;
}
};
class son2: father { string name = "Roushan";
public:
void disp() {
cout << name << " " << surname;
}
};

```
main()
{ son1.s1;
  son2.s2;
  s1.show();      // Chandan Singh
  s2.disp();      // Rajdhan Singh
}
```

Types: → Single inheritance.
→ Multiple "
→ Multilevel "
→ Hierarchical "
→ Hybrid "

* Abstraction: Abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Eg→

```
class Abstraction {
  Private:
    int a, b;
  public:
    void set(int x, int y)
    { a=x;
      b=y;
    }
    void display()
    { cout<<"a="<<a<<endl;
      cout<<"b="<<b<<endl;
    }
};
int main()
{ Abstraction. O
  O.set(10,20);
  O.display();
  return 0;
}
```

(*) function overloading: function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

An overloaded fun^{n} must have:

- different type of parameters.
- Different number of parameters.
- Different sequence of parameters.

```
1. void print();
2. void print(int a);
3. void print(float a);
4. void print(int a, int b);
5. void print(int a, double b);
6. void print(double a, int b);
```

Encapsulation	Abstraction
(i) Encapsulation binds the data and functions together in a single unit.	Abstraction hides the implementation details and shows only functionality to the user
(ii) It hides data for the purpose of data protection.	It hides the implementation details to reduce complexity
(iii) Encapsulation can achieve by making the data members private and access them through public methods.	Abstraction can achieve by Abstract classes.
(iv) Encapsulation solves the problem in implementation level.	(iv) Abstraction solves the problem in Design level.

* Defining member function

Member function can be defined in two ways:

(i) Inside the class definition.
(ii) Outside the class definition.

(i) Inside the Class definition

```
class add {
int x=5;
int y=6;
public:
void fun(int x, int y)
{
    int c = x+y;
    cout<<c;
}
};
```

// In this example fun is a member function of add class, member functions are typically declared within the class definition.

Ex ⇒ defining member function outside the class using the scope resolution operator ::

```
class Hero{
public:
void myHero();
};
void Hero::myhero()
{
// function implementation.
}
```

```
int main()
{
Hero h;
h.myhero();
}
```

In this example 'myhero' is declared within Hero class but its implementation is provided outside the class definition.

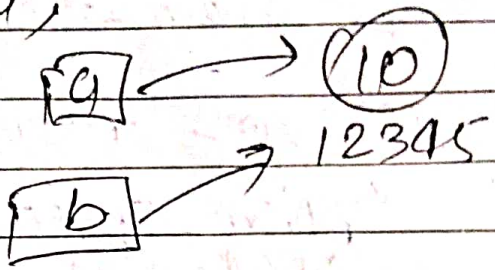
* class specification: class specification usually refers to the declaration of a class. It defines the structure and members of the class without providing their implementation.

```
class point {
private:
    int x;
    int y;
public:
    point();
    point(int x, int y);
    void setx(int x);
    int getx();
    void sety(int y);
    int gety();
};
```

- class point is the class declaration.
- public, private and protected are access specifiers;
- point(); is the constructor declaration
- void setx() & void sety() are members function declaration.

* Referemce variables : A variable which provides alternate name for the existing variable is called reference variable.

Ex ⇒ int a = 10;
int & b = a;



* Instance variable : Instance variable also known as a member variable. Instance variable is a variable that is declared within a class but outside of any method, changes made on instance variable in one object will not affect the other objects of the class.

```
class car {
public:
int id;
string color;
};
int main()
{ Car car1;
car1.color = "red";
cout << car1.color << endl;
}
```

(11) Scope resolution operator: The scope resolution operator is represented by the double colon (::), it is used to access data members within class or namespace. It is used to distinguish b/w class members and other names.

(1) accessing a variable.

```
MyNamespace::myvar;
MYclass::myvar=10;
```

(2) Calling a function

```
MyNamespace::myfemction();
myclass::myfunction();
```

(12) Object as arguments and returning object:

```
#include<iostream>
using namespace std;
class sample
{ public:
  int value;
  void getvalue()
{ cout<<"enter a number:";
  cin>>value;
}
  void display()
{ cout<<value;
}
sample sum(sample s)
{ s.value=s.value+10;
  return s;}};
```

```
int main()
{ Sample t,s,p;
 t.getvalue();
cout<<"e before giving address";
 t.display();
cout<<endl;
 s=p.sum(t);
 s.display();
cout<<endl;
 t.display();
  return 0;
}
```

Two most common short-circuit operators (&&) and & (||) or.

99

Q Evaluate the following statement with the help of code snippet(s) :"" Compiler knows the value of a Boolean expression before it has evaluated all of its operands.

ans> Yes, the compiler can know the value of a Boolean expression before it has evaluated all of its operands. This is of a programming technique called short-circuit evaluation.

```
#include<iostream>
using namespace std;
bool firstoperand()
{ cout<<"" first operand ""<<endl;
    return false
}
bool secondoperand () {
cout<<"" second operand ""<<endl;
  return true;
  }
  int main()
{ if (firstoperand()&&secondoperand())
{ cout<<"" Both operand are true :""
  return 0;
  } // In this example we have two functions
firstoperand() and secondoperand(), each returning
a boolean value.
in the main() function we use the logical
And operator according to short-circuit
```

evaluation in if statement, if the first operand of && operator false, then the second operand is not evaluated because the overall result of the expression will always be false.

and short-circuit evaluation is done using short-circuit operator the (&&) and (||) operator are called short circuit operator (&&)

```
if(a==b && c==d)
{ // do something
}
```

if the above expression a==b is false, then the compiler will not evaluate the next expression of if (c==d) because the result of the expression is already known (false).

(||)

```
if(a==b || c==d)
{ // do something
}
```

if the above expression a==b is true then the compiler will not evaluate the next expression of if (c==d) because the result of the expression is already known (true).

* Access specifiers: Access specifiers, also known as access modifiers are keywords in programming languages that determine the visibility and accessibility of classes, methods, variables or other members within a program.

Common access specifiers
(i) public (ii) private (iii) protected
(iv) Default.

```
class
{
  private : (i) member, function
            (ii) friend
  int a;
  protected : (inheritance
  int a;
  public:
    int a;
};
```

* ~~Constructure :~~

Constructor: Constructor is an special member function of a class used to ~~allocate the memory of ob~~ initialize objects of a class, constructor is automatically called when object is created.

features:

Constructor has same name as class
" " no return type not even void
" can initialize the member data of class
" called automatically when object is created.

Example of default Constructor

Ex ⇒

```
Class ATS
{ int a,b,c;
  public:
  ATS()
  {a=0;b=0;c=0;}
  void input()
{ cout<<"enter three no";
  cin>>a>>b>>c;
}
void output()
{ cout<<"A:"<<a;
  cout<<"B:"<<b;
  cout<<"In C:"<<c;
}
};
```

```
void main()
{ ATS p;
  p.input();
  p.output();
}
```

default c → Default Constructor is the constructor which doesn't take any argument, it has no parameters.

* Types of Constructors
  (i) Default constructor
  (ii) parameterized Constructor
  (iii) Copy constructor

(ii) parameterized Constructor: Parameterized Constructor is the constructor which takes argument, it has parameter, these arguments help in initialize an object when it is created.

Example Code

```
Class ATS                    | void main()
{ int a,b,c;                 | { ATS p;
  public:                    |   p.output();
  ATS()                      | ATS.S(10,20,30)
  { a=0;                     |   S.output();
    b=0;                     | }  // 0 0 0
    c=0;                     |    // 10 20 30
  }
  ATS(int x, int y, int z)
  { a=x;
    b=y;
    c=z;
  }
  void output()
  { cout<<a<<b<<c;
  };
```

* Copy Constructor : A copy constructor is a member function which initializes an object using another object of the same class.

OR

(Ans) write a program to create two object using copy constructor

OR

object as argument

Ex →

```
Class ATS
{ int a, b, c;
public:
ATS()
{ a=0;
  b=0;
  c=0;
}
ATS(int x, int x, int z)
{ a=x;
  b=y;
  c=z;
}
ATS(ATS& T)
{ a=T.a;
  b=T.b;
  c=T.c;
Void output()
{ cout<<a<<b<<c;
}
};
```

```
Void main()
{ ATS P;
  P.output();
ATS S(10,20,30)
  S.output();
  ATS t(S);
  t.output();
}
```

```
0 0 0
10 20 30
10 20 30
```

* Need of constructors and destructors.

→ Constructor is used to initialize an object of the class and assign values to data members corresponding to the class. while destructor is used to deallocate the memory of an object of a class.

* Destructor: Destructor in Computer science, destructor is a special member function in a class that is used to clean up resources and perform necessary cleanup operations when an object of the class is no longer needed. It is called when an object of that class is destroyed. They are used to deallocate the memory that was allocated to the object.

```
~Myclass()
{ // destructor
  // cleanup code
}
```

* Static data members: Static data members are class members that are declared using static keywords. It retain its previous value at the end of program.

* Static member function: Static member function is class method that are declared using static keyword, it retains the previous value at the end of the program.

* static data and member functions:

(1) static data member: Static data member is shared by all objects. All static data is initialized to zero when the first object is created. They are declared using static keyword within class definition. It can be initialized outside the class using the scope resolution operator (::).

```
class cube
{
public:
static int objectCount;
cube()
{ objectCount++;
}
};
int cube::objectCount=0;
int main()
{ cube c1;
cout<<"object"<<cube::objectCount<<endl;
cube c2;
cout<<"object"<<cube::objectcount<<endl;
cube c3;
cout<<"object"<<cube::objectcount<<endl;
return 0;
}
```

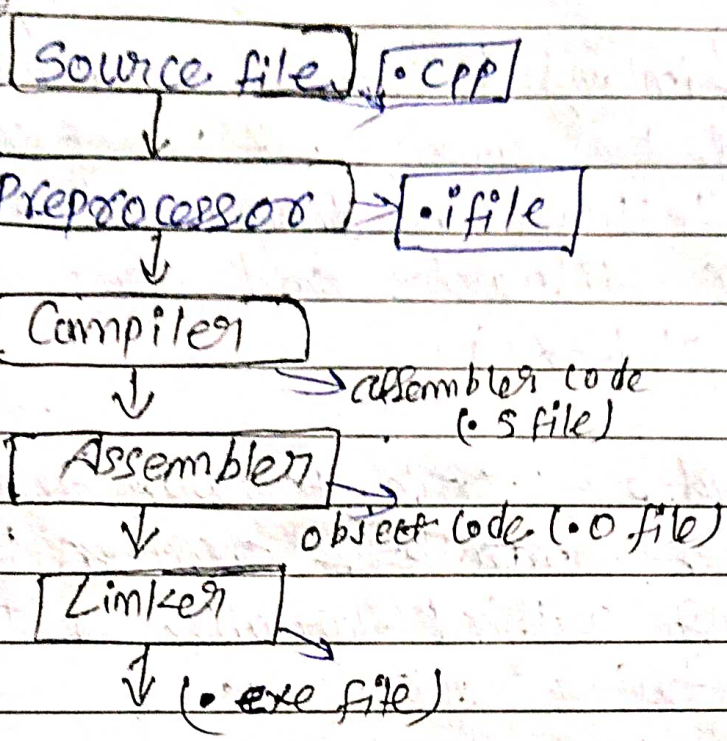
1, 2, 3

* static member function : A static member function is independent of any object of the class, A static member function can be called even if no objects of the class exist. A static member function can also be accessed using the class name through the scope resolution operator.

Ex→
```
class cube {
  public:
    static int objectCount;
    cube()
  { objectCount++;
  }
  static int getCount()
  { return objectCount;
  }
};
int cube::objectCount=0;
int main()
{ cube c1;
  cout<<"object:"<<cube::getCount()<<endl;
  cube c2;
cout<<"object:"<<cube::getCount()<<endl;
  return 0;
}
```
// 1, 2

Q What goes behind the scene when you attempt to get an output from a source code? Elaborate the steps with the help of diagrams.

ans →



Step-1 : We first create a c++ program using an editor and save the file as (.c++)

Step-2 : We compilling our filename.c++

Step-3 : After compilation we run the file.

Step 4 : Pre-processing : Removal of comments & space
Expansion of Macros
Expansion of included files
Conditional compilation
We get (.ifile)

Step-5 : Compiling : Compile the .ifile and produce an intermediate code (filename.s)

Step-6 : Assembling : in this process (filename.s) is converted into (filename.o) by the assembler.

Step-7 : Linking : This is the final phase in which all the linking of function calls with their definitions is done. Linker links object files or libraries and generates the executable file(.exe).

(*) Develop a code to swap two accepted positive integer numbers through keyword without using a third variable with the help of bitwise operators.

Ans →

```
#include<iostream>
using namespace std;
int main(){
int num1, num2;
cout<<"Enter num1:";
cin>>num1;
cout<<"Enter num2:";
cin>>num2;
num1 = num1 ^ num2;
num2 = num1 ^ num2;
num1 = num1 ^ num2;
cout<<"After swapping num1"<<num1<<
"num2"<<num2;
return 0;
}
```

ST

* How do you declare and initialize an array in C++?

ans ⇒

int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

datatype arrayname[size] = { data }

* How are keywords different from identifiers in C++? Give examples short answer?

ans ⇒ Keywords in C++ are reserved words that have predifined meanings and cannot be used as names for variables, functions or other user-defined entities. example: of key work: 'int' 'if' 'while' and 'class'.

Identifiers, on the other hand, are user defined names for other entities. They must follow specific naming rules that but are not reserved.
Ex ⇒ "myfunction" "calculateArea" "MyClass".

** How does the continue statement is different from the break statement.

ans ⇒

Continue : Continue skips the current iteration of the loop and continues with the next iteration.

```
ex⇒ for(i=0; i<=5; i++)
    {  if(i==2)
       { continue;
       }
       cout<<i<<endl;
    }
```

```
0
1
3
4
5
```

break : break immediately terminates the loop or switch statement if used in loop.

```
for(i=0; i<=10; i++)
{
    if(i==5)
    { break;
    }
    cout<<i<<endl;
}
```

* Justify advantage of member functions
  - code readability
  - Data encapsulation: Nested member functions can access private members of the class, which can help to encapsulate data and prevent unauthorized access.
  - Efficiency:
  - Access to private data member.
  - Improved readability.
  - Enhanced security.

and Some Code:

* Friend function: A friend function of a class is defined outside of that class scope but it has the right to access all private and protected members of the class.

```
class Distance
{ private:
    int meters;
  public:
    Distance()
    { meters = 0;
    }
    void displayData()
    { cout << "Meters value" << meters << endl;
    }
    friend void addvalue(Distance &d);
};
void addvalue(Distance &d)
{ d.meters = d.meters + 6;
}
int main()
{ Distance d1;
  d1.displaydata();
  addvalue(d1);
  d1.displaydata(); return 0; }
```

Q) C++ program to add two complex no using binary operator overloading

* Operator Overloading:

```
class Complex
{   private:
    int real;
    int img;
    public:
    Complex (int r=0, int i=0)
    {  real = r;
       img = i;
    void display ()
    { cout << real << "+i" << img;
    }
    friend Complex operator+(Complex c1, Complex c2);
};
Complex operator+(Complex c1, Complex c2)
{   Complex temp;
temp.real = c1.real + c2.real;
temp.img = c1.img + c2.img;
  return temp;
}
int main ()
{ complex c1(5,3), c2(10,5), c3;
  c3 = c1+c2;
  c3.display();      //15+i8
  return 0;
}
```

* Insertion operator overloading.

```
class Complex
{ private:
    int real;
    int img;
  public:
    Complex (int r=0, int i=0)
    { real=r;
      img=i;
    }
  friend ostream& operator<<(ostream &out,
                             Complex &c);
};
ostream &operator<<(ostream &out, Complex &c)
{ out<<c.real<<"+i"<<c.img;
  return out;
}
int main()
{ Complex C(10,5);
  cout<<C; //operator<<(cout,C); // 10+i5
  return 0;
}
```

friend function: A friend function is a function that isn't a member of a class but has access to the class's private and protected members.

friend function:

```
class Test
{ private: int a;
  protected: int b;
  public: int c; friend void fun();
};
Void fun() { Test t;
  t.a = 10;
  t.b = 15;
  t.c = 9;
}
```

friend class: A friend class is a class in which A friend class can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. for example, a LinkedList class may be allowed to access private members of Node.

```
class your;
class MY
{ private : int a;
protected : int b;
Public : int c;
friend your;
};
class your
{ public :
My m;
void fun()
{ m.a = 10;
m.b = 10;
m.c = 10;
}
};
int main()
{

}
```

* Nesting member function:

```
class Outer
{ class Inner
  { public:
    void display()
    { cout<<"display of Inner"<<endl;
    }
  }; public:
  void fun()
  {
    i.display();
  }
  Inner i;
};
int main()
{
  Outer::Inner i;
}
```

Vector

arr[] ⇒ *arr

* Advantage of Vector:
  - ⇒ Dynamic size.
  - ⇒ Rich library functions → find, erase, insert, etc.
  - ⇒ Easy to know size ⇒ int n = v.size();
  - ⇒ No need pass size.
  - ⇒ Can be returned from a fun.
  - ⇒ By default initialized with default values.
  - ⇒ Copy a vector to other

Ex)

```
int fun (vector<int> v)
{
}
```

```
int fun (int arr[], int n)
{
}
```

Ex)

```
vector<int> fun ()
{ vector<int> v;
  return v;
}
```

⇒ V1 = V2;

* Virtual function: A virtual function is a member function that is declared within a base class and it can be overridden by derived class. if you call derived class using base class pointer than you will get output of base class not of derived class for accessing derived class's properties you have to use a keyword that is virtual before the member function in the base class after that you can call derived class using base class pointer and you will get derived class's output.

Ex⇒
```
class base
{ public:
  virtual void function()
{ cout<<"I am in base class:"<<endl;
}
}; class derived1 : public base
  { public:
    void function()
  { cout<<"I am in derived class 1:"<<endl;
  }
};
class derived2 : public derived1
{ public:
  void function()
  { cout<<"I am in derived class 2:"<<endl;
};
```

```
int main()
{
    base *ptr;
    base b;
    ptr = &b;
    ptr->function(); // 3 am in base class
    derived d1;
    ptr = &d1;                    after virtual
    ptr->function(); // 3 am in derived class
    derived d2;
    ptr = &d2;
    ptr->function(); // 3 am in derived class
    return 0;
}
```

* pure virtual function: A pure virtual function ~~is~~ is function which has no definition. They start with virtual keyword and ends with equal to zero. If we don't override the pure virtual function in derived class, then derived class also becomes abstract class.

Ex of pure virtual function and example of abstract class will be shown latter both with only one example.

* Abstract Class: A class which contain at least one pure virtual function, we can't declare the object of abstract class.

Also known as late binding

```
class A
{
public:
    virtual void show() = 0;
};
```

Ex ⇒

```
class shape
{ virtual void get() = 0;
};
class area_Of_circle : public shape
{ public :
  void get()
  {
    int r;
  cout << "enter radius of circle" << endl;
   cin >> r;
cout << "the area of circle is :" << (3.14 * r * r)
   << endl;
  }
}; class rectangle : public shape
{ public :
   void get()
{ int length, breadth;
 cout << "enter length and breadth :" << endl;
  cin >> length >> breadth;
cout << "the area of rectangle is :"
  << (length * breadth) << endl;
}
};
class perimeter : public shape
{ public
   virtual void get() = 0;
};
int main()
{ area_of_circle c;
  c.get();
  rectangle r;
  r.get();
  return 0;
}
```

String Handling

(Q) Elaborate string handling function in C++.

Ans ⇒ String handling is performed using the "std::string" class which is part of the C++ standard library. This class provides a wide range of functions for creating, manipulating and working with strings.

(1) String creation and initialization
String str = "Hello";

(2) String concatenation: Concatenating two strings
String str1 = "Hello";
String str2 = "world";
string result = str1 + str2;

(3) String length: int length = str.length();
finding length of a string

(4) Accessing characters: Access characters at a specific position.
Char ch = str[0];

(5) String comparision: Comparing string
if (str1 == str2)
{ }

(6) substring Extraction: Extracting a substring.
string sub = str.substr(7,5);

(7) String search: finding the position of a substring within string

```
size_t pos = str.find("world");
```

(8) String Modification: Replacing a substring with another.

```
str.replace(0, 5, "Hi");  // Hi, World!
```

(9) String conversion: Converting a string to other data types.

```
string num = "12345";
int num = Stoi(num);
```

(10) String iteration: Iterating through each character in a string.

```
for (char c : str)
{
}
```

(Q4@) How strings are used in C++? write a program to extract a substring from given string.

Ans → String in C++ are typically represented of an array of characters and are part of the standard Template library (STL) you can use the 'string' class from the C++ standard library:

Extracting substring from a given string.

```
#include<iostream>
#include<string>
using namespace std;
int main()
{ string original = "Hello world";
string substring = original.substr(5,5);
cout<<"original string:"<<original<<endl;
cout<<"substring:"<<substring<<endl;
return 0;
}
// Helloworld
// world.
```

Creating string objects:

```
#include<iostream> using namespace std;
#include<string>
int main()
{ string emptystring; // create an empty string
string original = "Hello"; // create a string with
// an initial value
// concatenate string
string fname = "John";
string lastname = "Doe";
string full name = fname + " " + lastname;
return 0;
}
```

(Q48) Discuss the role of access specifiers in inheritance and their visibility in inheritance and their visibility when they are inherited as public, private and protected.

ans: Access specifiers in inheritance are used to control which members of a base class are accessible to derived classes. There are three access specifiers: public, protected, private

- public: Public members of a base class are accessible to derived class.
- protected: protected — accessible to derived class & base class itself.
- private: private members of a base class are accessible only to the base class itself, and not to derived classes.

visibility of inherited member.

- public members of the base class remain public in the derived class.
- protected members of the base class remain protected in the derived class
- Private members of the base class become inaccessible to the derived class.

(Q8) What is the importance of dynamic binding in the programming.

ans⇒ Dynamic binding allows you to ignore the type differences by providing us with the flexibility in choosing which type of function we need at that instant of runtime.

(Q9) what is the need of passing objects as arguments? Discuss different ways to pass objects as arguments to a function.

ans⇒ passing objects as arguments to function can be useful for a number of reasons.

- To modify the object: This ensures that the changes made to the object in the function are reflected in the main function.

- To pass a large amount of data passing obj
- To decouple the code: help to decouple the code of the function from the code of the caller. this makes the code more reusable and maintainable.

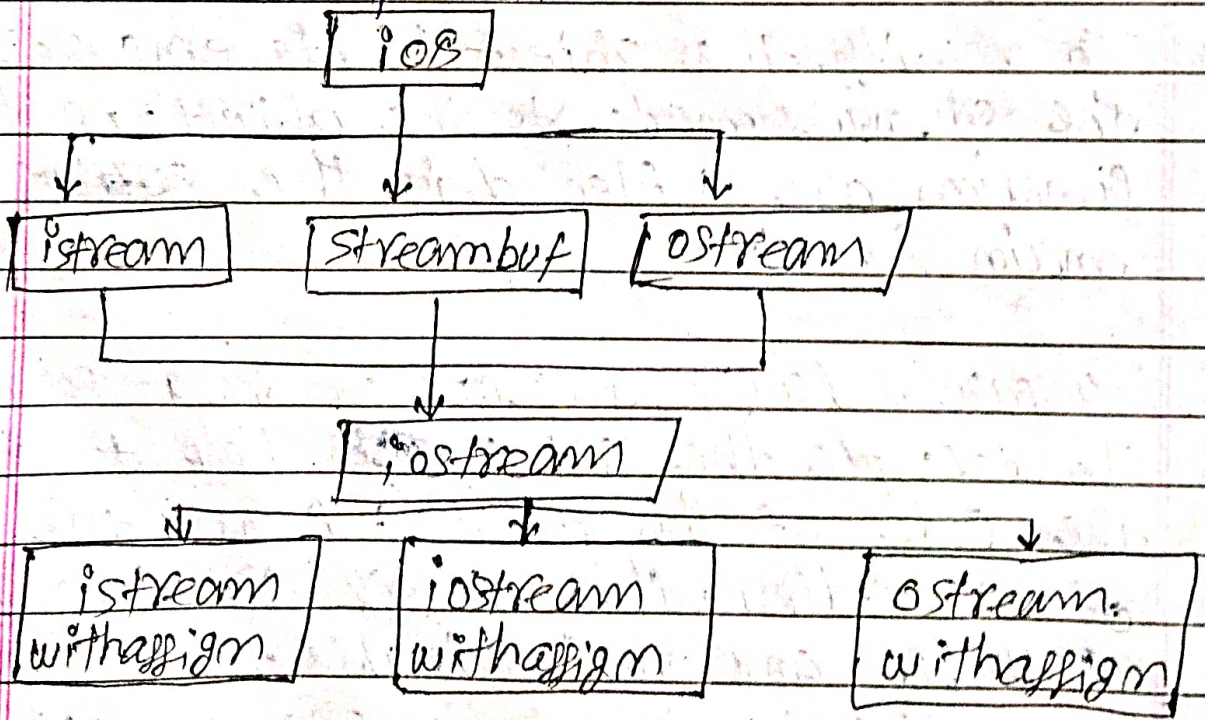
There are two main ways to pass objects as arguments to functions

1. pass by value.
2. pass by reference.

* pass by value: When an object is passed by value, a copy of the object is created and passed to the function, And any changes made to the object in the function will not be reflected in the ~~calling~~ main function.

- pass by reference: when an object is passed by reference, a reference to the object is passed to the function. Any changes made to the object in function will be reflected in the ~~calling fn~~ main function.

*



Q18) Differentiate b/w overloading and overriding with the help of example:

overloading	overriding
(i) function name must be same	function name must be same.
(ii) function signature must be different.	function signature must be same.
(iii) function overloading done with in the class.	function overriding done in another class or derived class.
(iv) No inheritance required.	Inheritance is required.
(v) It is compile-time polymorphism.	Run-time polymorphism

These both code is in page no 80 (Q2).

Q19) what are the different modes in which c++ file is opened?

ans ⇒ in - open for reading
out - open for writing
[redacted] -
app - Append mode
trunc - Truncate file if already exists
binary - opens file as binary
nocreate - open fails if file doesn't exists
noreplace - open fails if file already exists

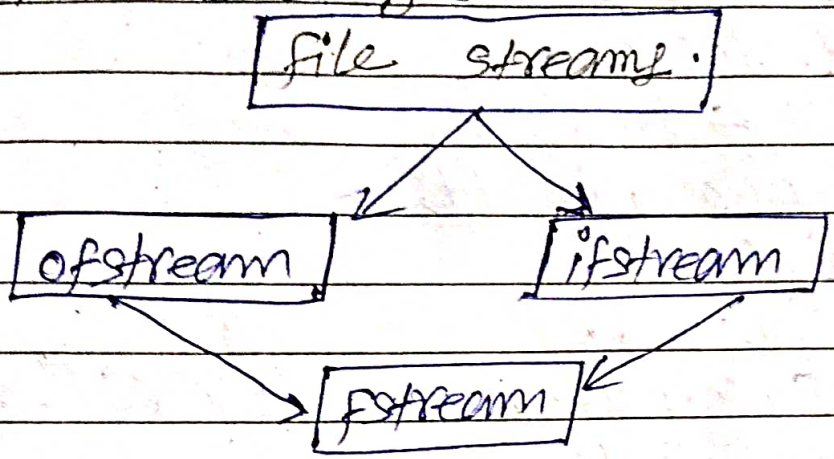
(pg7) If the body of a for loop is executed 'm' times, how many times is the counter updated and how many times is the condition checked?

ans ⇒ the counter is updated m times and the condition is checked m+1 times.

(pg8) "Inheritance supports the concept of reusability". Comment on this statement.

ans ⇒ The statement is accurate. It enable programmers to reuse the existing code to create new classes and functionalities. This promotes code efficiency, reduce time, enhance maintainability and readability.

File handling:



① ofstream: This data type represents the output file stream and it is used to create files and to write information to files.

② ifstream: This data type represents the input file stream and it is used to read information from files.

③ fstream: This data type represents the file stream and it have the capabilities of both ofstream and ifstream means it can create files and read information from files.

(*) Opening / writing a file / closing file.

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{ char ch[100];
cout<<"Enter your name & age:"<<endl;
cin.getline(ch,100);
ofstream fout;
fout.open("notepad.txt",ios::app);
fout<<ch;
fout.close();
return 0;
}
```

(*) reading a file :

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{ char ch[100];
  ifstream inf;
  inf.open("notepad.txt");
  inf.getline(ch,100);
cout<<"file read operation:"<<ch<<endl;
  inf.close();
  return 0;
}
```

(Q8) How can you improve the functionality a C++ program by using error handling during file operations?

C++ programming language provides several buil-in functions to handle error during file operations. The following are the built in functions to handle file errors:

(1) int good(): It returns a non-zero (true) value when no error has occurred; otherwise returns ~~zero~~ (false).

(2) int bad(): It returns a non-zero (true) value if an invalid operation is attempted or an unrecoverable error has occurred. otherwise it returns zero if no any error has occurred.

(3) int fail(): It returns a non-zero (true) value when an input or output operation has failed. otherwise it returns zero value.

(4) int eof(): It returns a non-zero (true) value when end-of-file is encountered while reading. otherwise returns zero (false).

Ex ⇒ int good().

Fig. 2

Ex ⇒ int bad

```
#include<iostream>
#include<fstream
using namespace std;
int main()
{ fstream file;
file.open("file.txt", ios::out);
char ch[100];
cout<<"Enter your name:";
cin.getline(ch,100);
file>>ch;
if(file.good())
{
cout<<"operation is
successfull"<<endl;
}
else{
cout<<"operation is
unsuccessfull"<<endl;
}
return 0;
}
```

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{ fstream file;
file.open("file.txt", ios::
char ch[50]; file.
if(!file.bad())
{
cout<<"operation
not successful"<<endl;
}
else{
cout<<"operation is
successfull"<<endl;
}
return 0;
}
```

// operation not successful.

operation is
// unsuccessfull

Ex: int fail()

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
fstream file;
file.open("file.txt",ios::out);
char ch[100]; file.getline(ch,100);
if(file.fail())
{
cout<<"operation Not successful"<<endl;
}
else{
cout<<"operation is successful";<<endl;
}
return 0;
}
```

Ex: int eof()

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{ fstream file;
file.open("file.txt",ios::in);
char ch[100];
while(!file.eof())
{ file>>ch;
}
cout<<"data is:"<<ch<<endl;
return 0;
}
```

(Q8) create a Class called Time that has separate int member data of hours, minutes, and seconds. one constructor should initialize variables to 0. and another should initialize it to fixed values. A member function should display it, in 11:59:59 formate. write program to add time of two objects by overloading '+' operator.

```
#include<iostream>
ing namespace std;
ass Time
rivate:
int hours;
nt minutes;
nt seconds;
ublic:
ime()
hours=0;
minutes=0;
conds=0;

me(int h, int m, int s)
hours=h;
minutes=m;
seconds=s;

id display()

t<<hours<<":"
<minutes<<":"<<
conds<<endl;
```

```
Time operator+(Time time2)
{ Time result;
result.seconds=seconds+time2.seconds;
result.minutes=minutes+time2.minutes
+(result.seconds)/60);
result.hours=hours+time2.hours+
(result.minutes)/60);
result.seconds=result.seconds%60;
result.minutes=result.minutes%60;
return result;
};
int main()
{ Time t1(1,30,45);
Time t2(2,15,20);
Time t3=t1+t2;
Cout<<"Time1:";
t1.display();
Cout<<"Time2:";
t2.display();
Cout<<"time3";
t3.display();
return 0;
}
```

* Templates: Templates in C++ are a powerful feature that allows you to write generic code to work with different data types.

→ Generic swap functions:

```
#include<iostream>
template<typename T>
void swap using namespace std;
void swap (T &a, T &b)
{  T temp = a;
   a = b
   b = temp;
}
int main () int x = 5, y = 10;
{ swap(x, y);//
Cout<<"x:"<<x<<", y:"<<y<<endl;
double a = 3.14;
       b = 2.71;
swap(a, b);
cout<<"a:"<<a<<" "<<"b:"<<b<<endl;
   return 0;
}
```

```
#include<iostream>
template<typename T>
class Box
{ public:
  T length;
  T width;
  T height;
  Box(T l, T w, T h)
  { length = l;
    width = w;
    height = h;
  }
  T volume()
  { return length*width*height;
  }
  int main()
  { Box<int> intBox(2,3,4);
  cout<<"volume of the integer box:"<<intBox.volume()
  <<endl;

  Box<double> doubleBox(1.5, 2.5, 3.5);

cout<<"volume of the double box is:"<<
    doubleBox.volume()<<endl;
    return 0;
    }
```

Build c++ program for dynamic memory management using pointer.

```
#include<iostream>
using namespace std;
int main()
{ int *p = new int;
  if (p == nullptr)
{ cout<< "Error: Memory allocated failed"<<endl;
    return 1;
  }
  *p = 42;
cout<< "value stored in allocated memory:"<<*p;
   delete p;
   return 0;
}
```

(P) How templates are useful in generic programming

- Templates allow you to write a single piece of code that can work with multiple data types means that you don't have to write the same code over and over again for different data types.
- Improve readability.
- reduce complexity
- reduce time Complexity
- More efficient
- less lines of code

Q9 Compare and contrast virtual functions and pure virtual functions with the help of an example:

	virtual function	pure virtual function
(i)	It is not a abstract class.	It is abstract class.

(ii)	Syntax: virtual void show()
	{ //
	}
		Syntax: virtual void show()=0;

(iii)	Definition of virtual function is provided in base class.	Definition of pure virtual function is not provided in the base class
(iv)	'virtual' keyword is used to make virtual function.	'virtual' keyword is used to make pure virtual function
(v)	It is not necessary for all derived classes to override the virtual functions of the base class.	It is necessary for all Derived class should override the pure virtual function of the base class
(vi)	virtual function can have body.	pure virtual function cannot have body.
(vii)	"=0" notation is not required.	"=0" notation is required.

example of virtual function and pure virtual function in page no. 64 & 99

1 (Pyq) Explain various control statements available in C++ through proper examples.

ans ⇒ Control Statements: if, if-else, nested if-else, for, while, do-while, break, switch, continue and goto Statement.

Open document file with question name. you will get its pdf file.

(Pyq) (2) write a program to differentiate function overloading and function overriding.

ans ⇒ function overloading: functions with the same name can have different parameters
function overriding: Derived classes can redefine functions from their base class.
for answer open file manager search with question name you will get it's pdf

(Pyq) How would you differentiate between parameterize and copy constructors?

Parameterize	copy constructor
(i) It initializes an object with specific values	It creates a copy of an existing object.
(ii) It takes one or more parameters.	It takes a reference to an object of the same class.
(iii) It is used to create new objects with different values.	It is used to create copies of existing objects

Q9) What explanation do you have for the need of pointers?

ans⇒ Use of pointers

(1) To pass arguments by reference.
(2) for accessing array elements
(3) To return multiple values
(4) Dynamic memory allocation
(5) pointers save memory space.
(6) pointers holds the address of an integer variable or holds the address of memory whose value can be accessed of integer value.
(7) pointer is also a variable which point the other variable.

Q10) Illustrate the concept of explicit and implicit type conversion with the help of programs.

(1) Implicit type conversion/casting: In implicit or automatically casting, compiler will automatically change one type of data into another.

Ex⇒ char → int → long → float → double.

(2) Explicit Type conversion/casting: When the user manually changes data type from one type to another. This is called as explicit conversion.

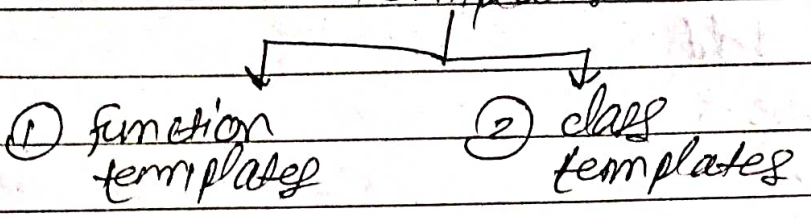
Ex⇒ long → int → char → float ⇒ Syn ⇒ (datatype) expression
Type

① explicit program:
```
#include<iostream>
using namespace std;
int main()
{ char a='A';
int b=(int)a;
double c=(double)b;
cout<<"the value of a is:"<<a<<endl; //A
cout<<"the value of b is:"<<b<<endl; //65
cout<<"the value of c is:"<<c<<endl; //65.0
return 0;
}
```

② Implicit program:
```
#include<iostream>
using namespace std;
int main()
{ int x=3;
cout<<"the value of x is:"<<x<<endl; //3
float y=x;
cout<<"the value of y is:"<<y<<endl; //3.0
int z=65;
char t=z;
cout<<"the value of z is:"<<z<<endl; //65
cout<<"the value of t is:"<<t<<endl;
return 0;
}
```
//A

(Pyq) Explain the concept of class and function templates with the help of programs.

ans⇒ Templates: Templates allows you to write a generic code that can work with different data types.



(1) function templates: function templates allow you to create generic functions that can work with different data types.

Ex⇒
```
#include<iostream>
using namespace std;
template<typename T>
T add(T a, T b)          // 3
{ return(a+b);           // 7.8
}                        // 7.976
int main()
{ cout<<"The sum is:"<<add<int>(3,4)<<endl;
cout<<"The sum is:"<<add<float>(3.5f,4.3f)<<endl;
cout<<"The sum is:"<<add<double>(3.033,4.543)<<endl;
 return 0;
}
```

(2) class template: class template allows you to define a generic class that can work with different data types.

Ex→

```
#include<iostream>
using namespace std;
template<typename T>
[illegible]
class Add
{ private:
  T a;
  T b;
  public:
Void setdata(T x, T y)
{ a=x;
   b=y;
}
T add()
{ return(a+b);
}
};
int main()
{ Add<int> a;
a.setdata(5,5);
cout<<"sum is:"<<a.add()<<endl;
Add<double>b;
b.setdata(5.348, 4.35345);
cout<<"sum is:"<<b.add()<<endl;
Add<float> c;
c.setdata(4.3f, 2.3f);
cout<<"sum is:"<<c.add()<<endl;
return 0;
}
```

Q.9 write a program in C++ to copy the content of one file to another.

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{ ifstream fin;
  ofstream fout;
  string str;
char ch[50], ch2[50];
 cout<<"Enter source file with extension"<<endl;
  gets(ch);
fin.open(ch);
if(fin.fail())
{ cout<<"error in opening file"<<endl;
  fin.close();
  return 1;
}
cout<<"Enter Destination file with extension"<<endl;
gets(ch2);
 fout.open(ch2);
  if(fout.fail())
{cout<<"Error in opening destination"<<endl;
      fout.close();
      return 2;
} while(getline(fin, str))
  { fout<<str<<endl;
  } cout<<"source file successfully copied";
fin.close();
fout.close();
return 0;
```

visiability of Specifiers

Base class Specifier	Public Inheritance	protected Inheritance	private Inheritance
Public	public	protected	private
protected	protected	protected	private
private	Not accessible	Not accessible	Not accessible

* goto statement: C++ goto statement is also known as jump statement. It is used to jump from anywhere to anywhere within a program.

```
#include<iostream>
using namspace std;
int main()
{
 cout<<"hello"<<endl;
  goto ab;
 cout<<"will be skipped"<<endl;
 ab:
 cout<<"will be printed";
 return 0;
}
```

Q50 Why the use of goto statement is not good for quality programming.

ans=> The use of goto statemant is not good for qixality programming because.

(i) The goto statement is a low-level Control flow statement.

(ii) It makes the program very complex.

(iii) bad readability.

Q51 why do we need data types in C++?

ans=> we need data types to inform the operating system that what is the type of data, we are handling based on the type of data operating system will allocate the memory in Bytes in the main memory.

Q52 Ternary operator: The ternary operator also known as the Conditional operator It is a single-line Conditional statemant that takes three operands (i) condition statement (ii) expression if true (iii) expression if false.

```
int y = 5;
int x = y>10 ? 1 : 0;
```

condition ? expression if true : expression if false;

* The rules for naming variables are:

- variable names can only contain letters, digits and underscores (_)
- variable must begin with a letter or an underscore(_)
  variable
- Names are case sensitive (myVar and myvar are different variables)
- Variable name cannot contain whitespaces or special characters like (!, #, %, etc.)
- Reserved words like int cannot be used as name.

* Call by value & Call by reference

Call by value	Call by reference
(i) It passes copy of the original value	It passes address of the original value.
(ii) changes made inside the function will not reflect in the main function	changes made inside the function will reflect in the main function.
(iii) It does not requires pointers.	It requires pointer
(iv) It does not used & operator.	It uses & operator

(v) It requires more memory.	It requires less memory.
(vi) It is less efficient.	It is more efficient.
(vii) It takes More time to execute.	It takes less time to execute.
(viii)	

```
void swap(int a, int b)
{ int temp = a;
  a = b;
  b = temp;
}
int main()
{ int a = 5;
  int b = 7;
  swap(a, b);
```

```
void swap(int *a, int *b)
{ int temp = *a;
  *a = *b;
  *b = temp;
}
int main()
{ int a = 5;
  int b = 7;
swap(&a, &b);
```

* what is buffer? How does it affect programming? Give example.

ans→ A reserved segment of memory (RAM) within a program that is used to hold the data being processed called buffer

for example: In a video streaming application the program uses buffers to store an advance supply of video data.

pure virtual function

```
#include <iostream>
using namespace std;
class shape
{ public:
  int length = 14;
  int breadth = 14;
  int radius = 6;
virtual void area() = 0;
};
class areaofcircle : public shape
{ public:
  void area()
{
cout << " area of circle is: " << (3.14 * radius * radius) << endl;
}};
class areaofrectangle : public shape
  public:
   void area()
{ cout << " area of rectangle is: " << (length * breadth) << endl;
}};
  int main()
{ areaofcircle c;
  c.area();
  areaofrectangle r;
  r.area();
  return 0;
}
```

// area of circle is: 113.04

// area of rectangle is: 56