C++

→ In the 1983 B. Stroustrup has invented at AT&T's laboratories which is located at USA
→ C++ is the object oriented pgming language
→ It is the extension of the C lang
→ object oriented is a pgming principles / the concepts like monolithic pgming, structured pgming procedural pgming, modular pgming, object oriented pgming, aspect oriented pgming.

list of C++ compilers

1) C++ OC
2) C++ OXC
3) Microsoft C++
4) TC C++
5) Borland C++
6) C++ linux
7) C++ unix
8) C++ solaris

object oriented concepts

1) **class**:
→ class is a collection of data members & member functions
→ class is a logical representation of data
→ class is a blue print of data
→ In C++ class is a datatype (user-defined datatype)

2) **object**:
→ object is a physical representation of data
→ It is an instance of a class
→ Instance means allocating memory for members of class
→ object holds the value
→ object is a class variable
→ SBI of an object :-
state :- It is the value given to each & every attributes of an object
Behaviour :- operations performed on the object
Identity :- Name given to the object

character set in c++

→ In c++ we have "128" characters which is called as standard character set

e.g. A-Z, a-z, 0-9, special symbols

Tokens

→ It is the smallest individual unit in the pgm

→ c++ support "6" tokens :-

1) keywords
2) Identifiers
3) Datatypes
4) Constant
5) operators
6) strings.

1) keywords

c++ support 49 keywords

→ from this 32 keywords has been taken from "c" lang

1) bool
2) class
3) public
4) private
5) protected
6) operator
7) friend
8) virtual
9) inline
10) try
11) catch
12) throw
13) namespace
14) using
15) this
16) new
17) delete

→ keywords are language related which have predefined meanings understood by compilers

→ keywords shouldn't be declared as identifiers

2) Identifiers

→ It is a name given to the variables, functions, & classes.

→ Identifier name should start with either alphabets/ underscore

→ It shouldn't start with digit

→ whitespaces aren't allowed b/w characters / words

→ maximum length of the identifier is "32" characters

3) Datatypes

→ Datatypes specifies "type & size" of data

→ By using datatype we specify how much memory should be allocated for the given variable

1) primitive / primary data types

1) int - 2 bytes
2) float - 4 bytes
3) double - 8 bytes
4) longint - 4 bytes
5) signedint - 2 bytes
6) unsignedint - 2 bytes
7) long double - 10 bytes
8) short - 2 bytes
9) char - 1 byte
10) Bool - 1 byte

→ It is directly stored inside the memory location
→ memory is allocated at the compiled time
→ It is stored in stack memory

2) Derived datatypes

Ex:- Arrays, pointers, objects, references
→ Data will not be stored directly rather address will be stored
→ memory is allocated at runtime
→ It is stored in heap memory.

3) User Defined datatypes

Ex:- structures, enum, class, union

4) Empty datatype

Ex: void

IO operations in C++:

Input:- info. given to the pgm
→ In C++ i/p & o/p operations are perform by using streams.
→ A stream is a class which contains set of functions to perform i/p & o/p operations
→ A stream contains 2 objects:-
1) cin & 2) cout
→ These two objects are declared in iostream.h / iostream header file

cout:- cout belongs to ostream class
→ It is used to perform o/p operations
→ cout represents console o/p

→ cout uses insertion operator (<<) to print any type of data
→ It is an overloaded operator
→ This function doesn't require any format specifiers

<u>syntax</u>:- cout<<value;
cout<<constant;
cout<<expression;

→ we can do the initialization in 2 ways
1) By using assignment operator
2) By using constructor

```
#include<iostream>
using namespace std;
int main()
{
int i=20; → o/p:- 20 30
int j(30); → constructor
cout<<i<<j;
return 0;
}
```

<u>endl</u>:- It is the manipulator available in std namespace.
→ we don't allocate any memory

<u>\n</u>:- It is a single character set which occupies 1 byte of space

<u>ex</u>:- cout<<i<<endl<<j;

<u>cin</u>:-
→ It is the object of istream class
→ cin represents standard input device
→ cin uses extraction operator (>>) to read the values

<u>syntax</u>:- cin>>value;

```
#include<iostream.h>
using namespace std;
int main()
{ int x,y,z;
cout<<"enter two values";
cin>>x>>y;
z=x+y;      o/p:- 2 3
                  5
cout<<z;
return 0;
}
```

Variable

→ Variable is the name of memory location where data is stored

→ C++ support 2 types of variables

1) local variable :- the variable which is declared within the function.

scope :- within the function

lifetime :- until function executes

default value :- garbage value

2) global variable :- the variable which is declared above all the functions

scope :- within the class

default value :- int, long = 0, float, double = 0.0

boolean = F(0), character = null / \0

→ C++ support 2 types of initialization :-

1) static initialization 2) dynamic initialization

ex :- int i = 10;
some o/p

ex :- int z = x + y;
diff o/p

→ we can have a global & local variable name same

→ the first preference will be given to local variable

```
int x = 10;
int main()
{
  int x = 30;
  int y = 20;
  cout << x;
  cout << ::x;
  return 0;
}
```

o/p : 30 10

```
int x = 10;
int main()
{ int y = 20;
  cout << x;
}
```

1) Block level variable (Anonymous block)

```
{
  =
}
```

2) conditional block

3) iteration block

→ In c++ main() function will return ~~an~~ integer value.

Ex:
```
#include<iostream>
using namespace std;
void main()
{
cout<<hi"; → error
}
```

→ Insertion operator is also called as put to/in operator which directs the info from right side to left side

→ Extraction operator is also called as get from operator it extracts the value from the keyboard & assign its value to the variable on the right

Block level Variables

```
#include<iostream>
using namespace std;
int main()
{
int x=10;
{
int y=20;
cout<<y; → 20
cout<<x; → 10
}
cout<<y; → syntax error
}
```

Constants

→ The values which can't be modified

there are 2 ways to create a constant:

1) By using const keyword
2) we can define set of integer constants by using enum keyword

→ By using "const" keyword we can define a constant in c++

Ex:
```
const float pi=3.14;
pi=11.23; → syntax error
cout<<pi;
```

```
const int size=10;
char name[size];
```

→ In "cff" always Constant should be initialized
→ In "cff" Constant values are local ~~variables~~;
→ In "c" Constant Values are global
const int x; → syntax error bcoz it is not initialized

Manipulators in cff

→ Manipulators are used to Customize the o/p in our own format

1) endl → It inspects new line & flush them
2) setw (int) → It is used to specify width of the o/p
3) setprecision (int) → It is used to set precision for floating point values.
4) setbase (int) → It is used to set base value of a numerical number
5) setfill (separator) : It is used with setw (int)

→ All the functions are available under <iomanip> header file

```
#include<iomanip>
using namespace std;
int main()
{
 float pi=3.14;
 int num=100;
 cout<<"enter new line"<<endl;
 cout<<setw(10)<<"output"<<endl;
 cout<<setprecision(10)<<pi<<endl;
 cout<<setbase(10)<<num<<endl;
 cout<<1<<setw(14)<<setfill('$')<<2;
 return 0;
}
```

o/p: enter new line
output
3.140000015
64
1$$$$$$$$$$$$$2

Namespaces

→ It is a collection of variables, functions & objects

→ Namespaces are used to avoid naming collisions

Syntax: using namespace namespacename;

Ex: using namespace std;

→ we can access the members of the namespace by using scope resolution operator.

Ex. Syntax: namespacename::membername;

Ex: n1::x;

→ there are 2 types of namespaces:

1) predefined namespaces 2) user defined namespaces

Ex:
```
#include<iostream>
using namespace std;
int main()
{
std::cout<<"welcome";
std::cout<<"iva"<<std::endl;
return 0;
}
```

→ we can declare variable as a void datatype but we can declare pointer variable as void datatype.

Ex: void i; void *i;

```
#include<iostream>
using namespace std;
namespace n1;
{
int x=10;
void f1()
{
cout<<"hello"<<endl;
}
}
namespace n2;
{
int x=10;
void f1()
{
cout<<"hi"<<endl;
} using namespace n1;
}
int main()
{
cout<<n1::x<<endl;
n1::f1();
cout<<n2::x<<endl;
::f1();
return 0;
}
```

operator:

→ It is a symbol which is used to perform operations on the operands.

→ C++ supports diff. types of operators.

1) Arithmetic operators 2) Relational operators

3) logical operators

Precedence	operator	description	Associativity
1	::	Scope resolution operator	left to right
2	a++ a--	suffix/postfix inc & dec	
	type() type{}	functional cast	L to R
	a()	function call	
	a[]	subscript	
	→	member access	
3✓	++a --a	prefix inc & dec	
	+a -a	unary plus & minus	
	! ~	logical NOT & bitwise NOT	
	(type)	c-style cast	R to L
	*a	indirection (dereference)	
	&a	Address of	
	sizeof	sizeof()	
	new	new[] dynamic memory allocation	
	delete	delete[] " " deallocation	
4	.* ->*	pointer to member	L to R
5	a*b a/b a%b	mul, div, remainder	L to R
6	a+b a-b	Add, sub	
7	<< >>	bitwise left & right shift	
8	<=>	three way comparison operator	
9	< <= > >=	for relational operator < and ≤ & ≥	
10	== !=	equals to & not equals to	
11	a & b	Bitwise AND	
12	^	Bitwise EX-OR	

13	|	Bitwise OR
14	&&	logical AND
15	||	logical OR
16	a?b:c	Ternary conditional — R to L
	throw	throw operator
	=	Direct assignment
	+= -=	compound assignment by sum & diff
	*= /= %=	compound assignment by mul, quotient & remainder
	<<= >>=	Compound assignment by bitwise left shift & bit wise right shift
	&= ^= |=	Compound assignment by bitwise AND, X-OR & OR
17	,	Comma — left - Right

three way Comparison operator / space shift operator (<=>)

1) 0 ⇒ A==B 2) -1 ⇒ A<B 3) 1 ⇒ A>B

```
#includeciostream>
#include<compare>
using namespace std;
int main()
{
 double x=-0.0;
 double y=0.0;
 auto res = x <=>y;
 if (res <0)
 cout << "_0 is less ther 0";
 else if (res>0)
 cout << "_0 is greater ther 0";
 else
 cout << "both are equal";
 return 0;
}
```

Type casting

→ Converting one datatype to another datatype

1) Implicit → Converting lower DT to Higher DT. automatically done by the compiler

2. Explicit → Converting higher DT to lower DT manually done by programmer

Implicit → bool → char → int → unsigned int → long → long long → double → long double

Ex:
```
int main()
{
bool x=false;
int y=10;
y=y+x;
cout<<y;
return 0;
}
```
→ o/p: 10

and
```
char ch='a'
int z=100;
z=z+ch;
cout<<z;
```
o/p: 107

explicit:

c-style notation : (datatype) expression;

Ex:
```
char ch_var='a';
int int_var=(int) ch_var;
cout<<int_var;
```

function notation: datatype(expression);

Ex:
```
char ch_var='a';
int int_var=int(ch_var);
cout<<int_var;
```

Ex:
```
int int_var=17;
float float_var;
float_var=float(int_var)/2;
cout<<float_var;
return 0;
```

class:

→ class is a collection of data members & member functions

→ Encapsulation is possible in c++ by using "classes"

→ Difference b/w structure & class

structure	class
1) It contains only data members (variables)	1) It contains both data members & member functions.

2) By default the members of structure are global.	2) By default the class members are private
3) It can't support polymorphism & inheritance	4) It supports polymorphism & inheritance

syntax :

```
class classname {
private/public/protected:
data-members;
member_functions();
};
```

Data members :

→ the variables declared inside the class is called data members.

object :-

→ object is an instance of a class
→ It is a class variable
→ It is a physical representation of data

syntax :

```
classname objectname;
```

ex :-

```
#include<iostream.h>
using.
class Date {public:
    int dd, mm, yy;
};
int main(){
    Date obj;
    obj.dd=10;
    obj.mm=11;
    obj.yy=2024;
    return 0;
}
```

→ error bcoz class members are private

→ private members can't accessed in non-member function

Access specifiers

⇒ It says scope, visibility & accessibility to the members in class

⇒ C++ support 3 access specifiers :-
1) private 2) public 3) protected

private :-

→ private data members can access within the same class only.

⇒ It can't access in non-member function & in another class.

Ex: class A{ within class inside another class

```
class A{
    int x;
    void f1(){
        cout<<x; ✓
    }
}
```

```
class B{
    void f2(){ syntax
        cout<<x; ✗ error
    }
}
```

protected :-

→ protected data members can be accessible within the class & derived class.

public :-

→ public data member can be access within the class, derived class, in another class & in non-member function.

⇒ The size of the object is size of the data members

⇒ size of empty class is "1 byte"

Ex: class student {
    public:
        int sid;
        char name[20];
        float fee;
};

```
int main(){
    student s={21, "vishal", 1000};
    cout<<s.sid<<endl;
    cout<<s.name<<endl;
    cout<<s.fee;
} return 0;
```

o/p: 21
vishal
1000

ex: class student {
public:
int sid;
char name[20];
float fee;
void getdata() {
cout<<"enter sid";
cin>>sid;
cout<<"enter name";
cin>>name;
cout<<"enter fees";
cin>>fee;
}
void display() {
cout<<sid<<endl;
cout<<name<<endl;
cout<<fees;
};
int main() {
student s;
s.getdata();
s.display();
return 0;
}

o/p: enter sid 20
enter name ira
enter fee 5000
20
ira
5000

→ Every c++ pgm can also be extracted by using structures

Member functions

→ we can declared the function in two places
1) inside the class 2) outside the class

outside the class :-

→ Member function defining outside of the class

syntax

returntype classname::functionname(parameters)
{ stmts;
}

Ex: #include
using.
class student {
int sno;
int sub1, sub2;
public:
void setstd(int, int, int);
void find-result();
};
void student::setstd(int s, int s1, int s2){
sno = s;
sub1 = s1;
sub2 = s2;
}
void student::find-result(){
cout<<sno;
cout<<sub1;
cout<<sub2;
}
int main(){
student s;
s.setstd(10, 98, 99);
s.find-result(); → o/p: 10, 98, 99
return 0;
}

Array of objects

→ Allocating memory for more than one object of similar type.

Syntax:

classname objectname[size];

Ex:

class student {
char name[20];
int sns;
public:
void read();
void display();
};
void student::read(){
cout<<"enter name";
cin>>name.
cout<<"enter sns";

```
cin>>runs;
}
void student::display(){
    cout<<name<<endl;
    cout<<runs;
}
int main(){
    student s[2];
    for(int i=0;i<2;i++){
     s[i].read();
    }
    for (int i=0;i<2;i++){
     s[i].display();
    }
    return 0;
}
```

o/p: enter name VKC
enter runs 200
enter name PARA
enter runs 300
VKC
200
PARA
300

Dynamic object

⇒ The objects which are created at the run time are called dynamic objects
⇒ These are stored in Heap area
⇒ Dynamic objects are created by using 'new' operator

syntax:

```
<pointer variable> = new classname;
```

⇒ pointer variable holds the address of the object
⇒ It is called as pointer to object
⇒ we can access private members by using pointers
⇒ pointers are not secure

Ex:

```
class student{
        int sid;
        char name[20];
        public:
            void read();
            void display();
};
void student::read(){
    cout<<"enter sid";
    cin>>sid;
    cout<<"enter name";
    cin>>name;
}
```

```
void student::display(){
    cout<<sid<<endl;
    cout<<name;
}
int main(){
    student *s;
    s=new student;
    s->read() / (*s).read();
    s->display() / (*s).display();
    return 0;
}
```

o/p: enter sid 20
enter name ira
20 ira

Constructor

→ Once we create an object it is filled with garbage value

→ By using constructor we initialize values to the variables.

⇒ It is a special type of function

→ class name & Constructor name should be same

→ constructor should not have return type.

→ for one object once constructor is called

→ c++ support 3 types of constructors:

1) Default constructor
2) parameterized constructor
3) copy constructor

1) **Default constructor**:

→ The constructor which donot have any parameter

→ They are 2 types:

1) Compiler defined & 2) user defined

Eg:

```
class Triangle {
    float base, height;
    public:
        Triangle();
        void find_area();
};
Triangle::Triangle(){
    base=12.4;
    height=22.3;
}
void Triangle::find_area(){
    cout<<base*height;
}
```

```
int main(){
    Triangle T;
    T.find_area();
    return 0;
}
```

2) parameterized constructor :-

→ The constructor which have parameters are called parameterized.

→ The parameters can be values, address, reference variable etc.

Ex:-
```
class Data{
    int dd,mm,yy;
    public:
        Data(int,int,int);
        void display();
};
Data::Data(int d,int m,int y){
    dd=d;
    mm=m;
    yy=y;
}
void Data::display(){
    cout<<dd<<endl;
    cout<<mm<<endl;
    cout<<yy;
};
int main(){
    Data d(10,12,2024);
    d.display();
    return 0;
}
```

Constructor overloading :-

→ In one class we can have more than one constructor.

1) By changing no. of parameters
2) By changing type of parameters
3) By changing order of parameters.

```
class Array{
    int size;
    public:
        Array();
        Array(int);
```

```
    void readelements();
    void sum();
};
Array::Array(){
    size=5;
    a=new int[size];
}
Array::Array(int s){
    size=5;
    a=new int[size];
}
void Array::readelements(){
    ~~int num=1;~~
    for(int i=0;i<size;i++){
        cin>>a[i];
        ~~num=num+1;~~
    }
void Array::sum(){
int sum=0;
    for(int i=0;i<size;i++){
        sum=sum+a[i];
        cout<<sum;
    }
int main(){
    Array arr1(3); (or) Array arr1();
    arr1.readelements();
    arr1.sum();
    return 0;
}
```

Destructor:

→ we are deallocating resources which are allocated by the Constructor.
→ Destructor is executed before destroying (or) deleting the object
→ classname & destructorname should be same
→ it is prefix with ~ (tilde) operator
→ it should not have any returntype

→ Destructor should not have any parameters
→ there is only destructor in the class

syntax :-

```
~classname()
  {stmts;
  }
```

→ this is a keyword in c++
→ It holds address of current object
→ If local variable names & data members names are same then we use this keyword.
→ this keyword is used for current class variables current class functions, current class constructor.

```
class Rectangle{
    float l, b;
    public:
        void set_rect(float, float);
        void rect_area();
};
Rectangle::
void set_rect(float l, float b){
  (*this).l = l;
  (*this).b = b;
}
void Rectangle::rect_area(){
    cout << l*b;
}
int main(){
    Rectangle r;
    r.set_rect(2,3);
    r.rect_area();
    return 0;
}
```

o/p :- 6

Nameless object

```
class nameless{
    int a;
    public:
        nameless(){
            a=10;
            print();
        }
    void print(){
        cout<<a;
    }
};
int main(){
    nameless();
    return 0;
}
```

o/p :- 10

Array of objects

```
class aob{
    int a;
    public:
        aob(int i){
            a=i;
        }
        void print(){
            cout<<a<<endl;
        }
};
int main(){
    aob obj[3]={obj(11), obj(12), obj(13)};
    for(int i=0; i<3; i++){
        obj[i].print();
    }
    return 0;
}
```

o/p :- 11
12
13

friend function

→ A non-member function can't have access private data of the class

→ friend function should be declared with friend keyword

→ It is not called by using object of the class like a normal function we can invoke

→ friend function can access outside of the class also

→ It need not be member of a class

syntax:

friend returntype functionname <parameters>;

Ex:

```
#include
using
class B;
class A {
  int x;
  public:
    A()
      x=100;
    }
    void print()
      cout<<x;
    }
  }; friend void sum(A,B);
class B {
  int y;
  public:
    B()
      y=200;
    }
    void print()
      cout<<y;
    }
    friend void sum(A,B);
};
void sum (A obj1, B obj2) {
  int s=obj1.x+obj2.y;
  cout<<s;
}
int main()
  A obj1;
  B obj2;
  sum(obj1,obj2);
  return 0;
}
```

→ friend can be declared in the either public / private
→ It has obj as arguments
→ For the friend function inheritance is also applicable

Friend class :-

→ friend function can access only the attributes
→ friend class can accept all the members of a class

```
class A {
  public:
    void fun1(){
      cout<<"f1";
    }
  protected:
    void fun2(){
      cout<<"f2";
    }
  private:
    void fun3(){
      cout<<"f3";
    }
  friend class B;
};
class B {public:
    void print(){
      A obj;
      obj.fun1();
      obj.fun2();
      obj.fun3();
    }
};
int main(){
    B obj;
    obj.print();
return 0;
}
```

Inheritance

→ Extracting the properties from one class (parent / base / super class) to another class (child / derived / sub class).

→ Advantage is code Reusability

syntax :-

```
class Base_class{
    members;
}
class Derived class : Access specifier Base class name
{
}
```

Types of inheritances

1) single level inheritance
2) Multiple inheritance
3) Multi-level inheritance
4) Hierarchical inheritance
5) Hybrid inheritance

single level inheritance :-

```
class B{
    public:
        B(){
            cout<<"bclass Constructor";
        }
        ~B(){
            cout<<"b class destructor";
        }
};
class D: public B{
    public:
        D(){
            cout<<"dclass Constructor";
        }
        ~D(){
            cout<<"dclass destructor";
        }
};
int main(){
    D obj;
    return 0;
}
```

o/p :- dclass destructor
bclass destructor
dclass Constructor
bclass Constructor

Multiple Inheritance

```
class Base1 {
    public:
        void f1(){
            cout<<"b1class f1";
        }
};
class Base2 {
    public:
        void f1(){
            cout<<"b2class f1";
        }
};
class c:public Base1, public Base2{
    public:
        c(){
            cout<<"cclass Constructor";
        }
};
int main(){
    D obj;
    obj.Base1::f1();
    obj.Base2::f1();
return 0;
}
```

→ we are creating the derived class with baseclass in public mode

→ public, protected, private inheritance in C++

→ public members of baseclass to public in derived class

→ protected members of base class to protected in derived class

protected inheritance makes

→ protected inheritance ~~means~~ public & protected members of base class to protected in derived class

private inheritance

→ public & protected members in the base class will become private in derived class.

→ private members in base class can't be accessible in derived class

Ex:

```
class B{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
}
class derived : public B{
 x is public
 y is protected
 z can't access
}
class derived : protected B{
 x is protected
 y is protected
 z can't access
}
class derived : private B{
 x & y is private
 z can't access
}
```

polymorphism :-

polymorphism

- compile time polymorphism → achieved by overloading
  - 1) fn overloading
  - 2) operator overloading
  - 3) Constructor overloading
- Runtime polymorphism → achieved by overriding
  - 1) virtual functions

Virtual functions

→ It is achieved by using runtime polymorphism
→ It is a member function in the base class & it should be redefined in the derived class
→ pointer of the base class refers to derived class object
→ virtual function can't be static. It must be member of some class
→ constructor can't be declared as virtual.
→ Destructor can be declared as virtual.

Ex:
```
class Base{
  public:
    virtual void f1(){
        cout<<"virtual fn";
    }
};
class Derived:public Base{
 public:
    void f1(){
        cout<<"derived f1();
    }
};
int main(){
  Base *ptr;
  Derived d;
  ptr=&d;
  ptr->f1();  -> derived f1()
  return 0;
}
```

pure virtual function

→ It is a function in the base class with no definition
→ It should be defined in the derived class
→ It is also called as nothing function
→ pure virtual function should be assigned "zero" in declaration

syntax

virtual return_type fun_name(parameters)=0;

```
class Base{
  public:
     virtual void f1() =0; (declaration)
};
class Derived : public Base{
 public:
    void f1(){
        cout<<"Hi"; } (definition)
  }
};
int main(){
   Derived d;
   d.f1(); → Hi
return 0;
}
```

Abstract Base class

→ the base class containing pure virtual function is called Abstract Base class.

→ for Abstract base class we can't create object but it can be inheritated & we can create base pointer.

ex:

```
class shape{
    public: int b,h;
    cout<<"enter d1";
    cin>>b;
    cout<<"enter d2";
    cin>>h;
    virtual void area() =0;
};
class Derived : public shape{
    public:
        void area(float b, float h){
            area = 0.5 x b x h;
            cout<<area;
        }
};
int main(){
  Deri
```

```
class shape{
    protected:
        float d1,d2;
    public:
        void readin(){
            cout<<"enter dims";
            cin>>d1>>d2;
        }
        virtual float findarea()=0;
};
class Derived: public shape{
    public:
        float find area(){
            return=0.5*d1*d2;
        }
};
int main(){
    shape *ptr;          Derived d;
    Derived d;     (or) d.readin();
    ptr=&d;              a=d.findarea();
    ptr->readin();       cout<<a;
    a=ptr->findarea();
    cout<<a;
    return 0;
}
```

C++ Templates

→ Templates are used for writing generic classes & functions.

→ there are 2 ways to implement templates:

1) function template
2) class template

Function templates/ generic templates

→ It is used to create single function to work with diff. datatype values. by using templates

Defining a function template

→ function template starts with template keyword followed by template parameters

ex: template <typename T>

syntax: T fn_name<T para, T para2>
{
stmts;
}

ex: template<typename T>
T add (T n1, T n2){
return n1+n2;
}

→ 'T' is the template argument which can handle any datatype values

→ Calling the function template

fn_name<datatype>(parameter);

Exp:

```
template<typename T>
T add(T n1, T n2){
  return (n1+n2);
}
int main(){
  int result=add<int>(11,22);
  cout<<result<<endl;
  float result2=add<float>(11.2f, 22.1f);
  cout<<result2;
  return 0;
}
```

class Templates

→ It is declared with template keyword followed by template parameters, followed by class declaration

```
template<class T>
class classname{
private:
    T variable;
public fn_name(T args){
};
```

} → creation of class template

Calling the class template:

T add (T x, T y)

classname <datatypes object name>;

ex pgm

```
template <class T>
class Number {
  T num;
  public:
    Number(T n){
      num=n;
    }
    getnum(){
      return num;
    }
};
int main(){
    Number<int> numobj(11);
    cout<<numobj.getnum();→11
  Number<double> dobobj(11.23);→11.23
    cout<<dobobj.getnum();
return 0;
}
```

Compile time polymorphism

→ It is achieved by using overloading concept.

→ C++ support 3 types of overloading

1) Constructor overloading 3) operator overloading

2) function overloading

function overloading

→ In a class we can have more than one function with same name but parameter list means datatype, no. of parameters, sequence of parameters can be different.

Two functions with diff. parameter type

sum(int num1, int num2)
sum(double num1, double num2)

Two functions with diff no. of args

sum(int n1)
sum(int n1, int n2)

Two functions with diff order/signature

sum(int n1, float n2)
sum(float n2, int n1)

Pgm

```
class Addition {
 int public:
  void sum (int a, int b){
      return a+b;
  }
int void sum (int x, int y, int c){
      return x+y+c;
  }
};
int main(){
    Addition add;
int i = add.sum(10, 20);
   cout<<i;
   cout<<add.sum(20,30,40);
  return 0;
}
```

operator overloading

→ In C++ operator overloading is compiled time polymorphism.

→ It gives special meaning to existing operator.

→ By default operator overloading works on only standard datatypes int, float, char, void.

→ operators can't be overloaded

1) scope resolution operator (::)
2) ternary operator (?:)

3) sizeof operator (sizeof())
4) member operator (.)
5) member pointer selector operator (*)

→ operators which can be overloaded
1) Unary operator
2) Binary operator
3) Special operators like [ ], ( )

Rules :

4) Overloading
1) Only existing operators can be overloaded
2) operator overloading should have atleast one operand
3) All operators can't be overloaded.
4) when overloading unary operator by member function we don't require any argument
5) when we are overloading binary operator by member function we need one argument
6) if it is a friend function unary operator needs one argument & binary operator needs two args

Syntax for operator overloading

```
classname operator op(argslist){
  stmts;
}
```

→ op means operator which we are overloading

```
class Complex{
int real, img;
public:
  complex(int r=0, int i=0)
  {
  real=r;
  img=i;
  }

complex operator +(complex obj)
{
 complex res;
   res.real=real+obj.real;
   res.img=img+obj.img;
 } return res;
```

```
void print(){
cout<<real<<"+i"<<img<<endl;
}
};
int main(){
complex c1(1.0,12), c2(11,13);
complex c3=c1+c2;
c3.print();
return 0;
}
```

iostream of files

input: the data/info. given to the pgm

output: the data/info. given by the pgm

→ files are used to store data permanently in secondary storage device.

~~string~~ : stream : with the help of stream we can perform i/p & o/p operations

Types of streams :

1) iostream < istream, ostream 2) fstream < ifstream, ofstream

ostream (output stream class)

→ writing the data to the files, to the console, to the socket.

functions :

1) put() : It is used to display the line of characters

syntax: put(ch).

2) write() :

Syntax: write(line, size)

ex: write("we (come", 3)

line → ~~no. of characters to~~ name of the string to display

size → no. of characters to display

ex:

```
int main(){
  cout.put('A');→ A
  cout.write("object", 3);→ obj
  cout.write("object", 6);→ object
```

3) width() :

```
int a=10;
cout.width(5);
cout<<a;
```

4) precision() :

→ By default floating point value will display six digits after the point ex: 12.000000

→ we can specify no. of digits to be display

Syntax : precision(p, s)

p → total no. of digits to display s → to display digits after point

Ex pgm : int main()
{
int a=1.2356;
cout.precision(2);
cout<<a; → 1.2

5) fill():

→ unused spaces are filled with white spaces by default.

→ we can fill with specific character

Ex: int a=10;
cout.width(5);
cout.fill('*');
cout<<a; → ***10.

syntax : fill('char');

6) setf():

→ ios::showpoint → it is used to display ending zeroes

→ ios::showpos → it, display '+ve' sign before positive no

Ex: float a=10;
cout setf(ios::showpoint); → total 6 digits varttayi
cout<<a; → 10.0000

7) istream class :-

1) get() : it reads single character including white spaces, tab, newline,

2) getline() :-

syntax : getline(line, size)

→ ~~it can read~~ Reading is terminated as soon as '\n' is encountered (or) size character is read

Ex pgm : int main()
{
char name[20];
cout<<"enter name";
cin.getline(name, 20);
cout<<name;

→ In c++ there are three file streams :
1) fstream, ifstream, ofstream

ofstream :

→ It will create a file & writes info. to the file.

ifstream :

→ It will reads info. from the file

fstream :

→ It will performs both read & write operations on the file.

→ these three stream classes are available in <fstream> header file.

→ Before we perform any operations on the file, first we have to open the file.

there are 2 ways to open the file :

1) ofstream my-file("example.txt");

2) By using open function:

```
std::ofstream my-file.open("example.txt");
```

→ once operations are completed close the file by using close function. my-file.close("filename");

→ there are 3 ways to check the file for errors

1) By checking file object

```
if(!my-file){
cout<<"file not avail";
return 1;
}
```

2) By using is_open() function :

```
if(!my-file.is_open()){
cout<<"file not avail";
return 1;
}
```

3) By using fail() function :

```
if(my-file.fail()){
cout<<"file not avail";
return 1;
}
```

Reading the data from file

```
#include<fstream.h>
#include<iostream.h>
using namespace std;
int main(){
ifstream my_file("example.txt");
if(!my_file){
    cout<<"file not avail";
    return 1;
}
string line;
while(my_file.eof()){
    getline(my_file, line);
    cout<<line; → welcome to isa
}
my_file.close();
return 0;
}
```

writing the data into the file ofstream class

```
#include<fstream.h>
#include<iostream.h>
using namespace std;
int main(){
ofstream myfile("example123.txt");
myfile<<"welcome"<<endl;
myfile<<"isa";
myfile.close();
return 0;
}
```

File Handling by using fstream class

→ By using fstream we can perform both read & write operations

→ fstream class constructor takes two arguments:
1) file name 2) mode of file

Modes:-

1) ios::in :- opening the file for read operation
2) ios::out :- opening the file for write operation
3) ios::app :- opens the file & appends the data

4) ios::binary :- opens the file in binary mode
5) ios::ate :- It moves the control to the end
6) ios::trunc :- Removes the data from existing file.

```
#include<fstream>
int main(){
fstream my-file("example.txt", ios::out);
if(my-file){
 my-file<<"welcome to file handling";
 my-file.close();
}
else{
cout<<"file can't be created"<<endl;
 return 1;
}
string line;
if(my-file){
    while(!my-file.eof()){
        getline(my-file, line);
        cout<<line;
    }
    my-file.close();
}
else{
 cout<<"erable to open";
 return 1;
}
my-file.open("example.txt", ios::app);
if(my-file){
   my-file<<" data appending";
    my-file.close();
}
return 0;
}
```

Inline functions :-

→ In c++ we use inline functions for saving the execution time such as jumping to calling function.

→ Inline function is a function that expanded line which is invoke by saving the time

→ the compiler replaces function call with function code.
→ If a function contain loop's
→ If a function is recursive function
→ If a ~~uses~~ function uses switch stmt & goto
→ If a function doesn't return value
→ For inline function declaration & definition should be together
→ It is declared with inline keyword

<u>Syntax</u>:

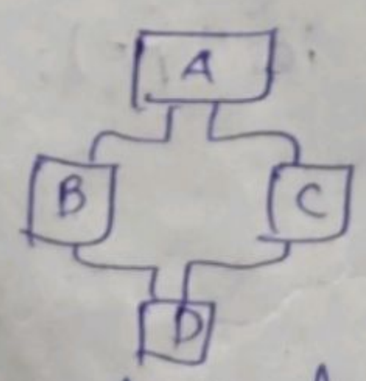
```
inline return-type fn_name(){
  stmts;
}
```

Ex:- ~~class cube{~~

```
inline int cube(int x){
    return x*x*x;
}
int main(){
cout<<cube(3);
return 0;
}
```

Multipath Inheritance

→ It is a combination of multilevel, hierarchical, & multiple inheritance



→ creating the derived class from other derived class of base class

with function

& goto

tion should

chical, &

rived class

```
class A {
  public int a;
};
class B : public virtual A {
public int b;
};
class C : virtual public A {
 public int c;
};
class D : public B, C {
  int d;
  public:
   D() {
      a=10;
      b=20;
      c=30;
      d=40;
   }
  void sum() {
      cout<<(a+b+c+d);
  }
};
int main() {
    D obj;
    obj.sum();→100
return 0;
}
```

<u>copy constructor</u>

→copying the values from one object to another object

→it is a type of constructor that initializes an object by using another object of the same class

→for the copy constructor we should have class object as a parameter

```
class Test{
  int x;
  public:
    Test(int i){
      x=i;
    }
    Test(Test &obj){
      x=obj.x;
    }
    void display(){
      cout<<x;
    }
};
int main(){
    Test t1(10);
    Test t2(t1);
    t2.display();->10
    return 0;
}
```