Page No.
Date

Topics	Pg.no.
1. Dynamic Allocation & Empty class	53
2. classes and objects (Access modifiers)	54
3. Constructors	56
Constructor overloading	57
4. Destructors	60
5. Inheritence + (Diamond problem) (virtual keyword)	61
6. Polymorphism	65
Compile time → function overloading	65
Compile time → operator overloading	66
Early binding ← virtual function (Runtime) → Late binding	68
7. Encapsulation	70

20/06/2022

Page No. 51
Date

Oops Concept in C++ →

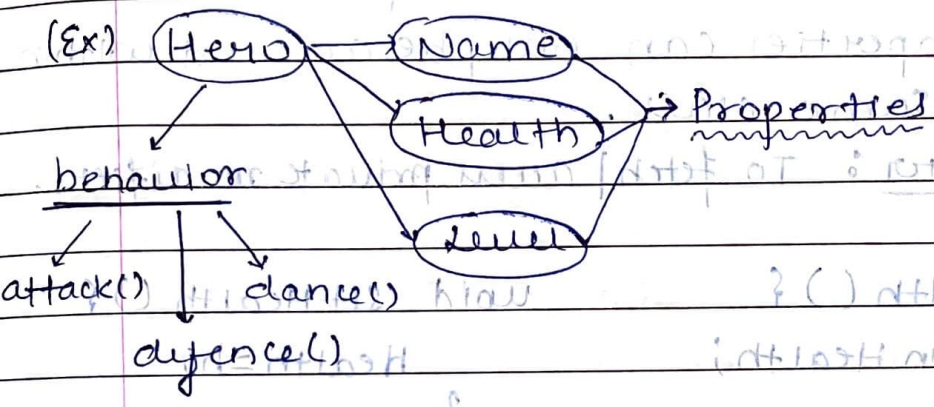
- **What is OOP ?**

Type of programming technique | paradigm where things revolve around object. (To bring program closer to real world)

Object → Entity → state | property
behaviour.

> increase • readability
• managibility
• compatibility

(Ex)



- **Class :** User Defined Datatype (blueprint for object)
(atype of template)

→ Empty class :
size → 1 byte (for identific^n)

- A class can be called from separate file or could be created in same.
If created in separated file then
#include "class-file.cpp"

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

class Hero {
    int Health;

};
int main() {
    Hero h1;

    return 0;
}
```

→ To access properties / Data members: using "." operator.

Access modifiers →
(1) Public
(2) Private
(3) Protected

* By default access modifier for class is "private".

* private properties can only be accessed within same class.

• Getter / Setter : To fetch / access private modifiers.

```
int getHealth() {
    return Health;
}

void setHealth() {
    Health = h;
}
```

→ Size of class = sum of datatypes size

→ Size of object = sum of max of datatype

Hero Ayush →
- Health
- ch

allocate 2 block in memory

- health (int) — 4
- ch (char) — 1

Size of Hero class = 5

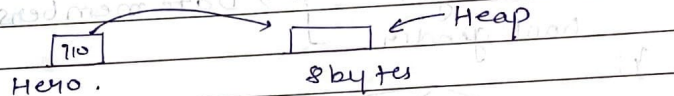
Size of Ayush object = 8

→ • Padding • Greedy alignment. (H/W ①)

Dynamic Allocation →

```
int *i = new int;
```

Ex `Hero *h = new Hero;`



Constructor
→ invoke at time of object creation
→ no return type
→ no input parameter

Ex: When we create class Hero and make an object Ayush.

Hero Ayush → Ayush.Hero()
↓
Call is invoked.
↓
It is called default constructor.

GFG

* Empty class → class that don't have any data members (but may have member func.)

Q. Why empty class in c++ takes one byte?
A. space allocated when class initiated.
So 1 byte allocated by compiler to give unique address identification.

Suppose, if a class doesn't have any size, what would be stored in the memory location
→ The non zero size ensure that the two objects will have different addresses.
(data members are diff than objects)

Page No.
Date

Apna College →

```
class Student {
    string name;
    int age;
    bool gender;
};
```
→ Data members

object of class

```
Student a;
a.name = 'Tim';
a.age = 12;
a.gender = 0;
```

```
Student b; Student c;
b.name = 'Sim'; c.name = 'jim';
b.age = 21;  c.age = 20;
b.gender = 1; c.gender = 0;
```

- Object : Instnce of a class

instnce.

class Student (datatype (user defined)) → Object a, Object b, Object c

GFG

BASIC DEFINITIONS →

1. **Class** : User defined datatype which holds its own data members and member functions and those can be accessed and used by creating an instnce of that class
   Class is a blueprint for object.
   - Data members → data variables
   - member functions → manipulate data members
   - Together define properties / Behaviour of class.

Page No. 55
Date

1 byte

* When class is defined no memory is allocated but when it is instantified, memory is allocated

When a program is executed the objects interact by sending messages to one another. (without having to know details of each other's data or code)

↓

It is sufficient to know the type (modifiers) of message accepted and type of response returned by the objects.

Syntax Defining (class) & Declaring (objects) →

User defined name
Keyword

```
class ClassName {
    Access specifier;   // public, private, protected
    Data members;       // Var to be used
    member functions() { // methods to access data members
    }
};                      // class name ends with semicolon
```

Declaring objects :

```
ClassName ObjectName;
```

To access data members and member funct^n we use (.) dot operator

Ex. Obj.printName(), Obj.Name

Accessing of data members (by default private) depends on access control

- 3 Access modifiers → public, private, protected

There are 2 ways to define member function.

① Inside class definition.

② Outside class definition (use scope resolution ::)

```
class Geeks {
public:
    string geekname;
    int id;

    void printname();
    void printid(){
        cout << id << endl;
    }
};

void Geeks::printname(){
    cout << geekname;
}
```

member function defined outside class.

- Constructors → ① Default (zero argument)
  ② Parameterized
  ③ copy.
  - > name of constructor is same as classname
  - > member function of class whose name is same as class.
  - > does not have a return type as do not have a return value.
  - > Prototype: [<class name>(list of parameters);]

- ① Default constructor (zero argument constructor) → Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

But, when ② parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating simple object like Student S (without parameters) will flash error.

Parameterized → used to initialize various data elements | objects with diff values when they are created.

used to overload constructors

Constructor overloading → (more than one constructor but diff arg)

more than one constructor in a class with same name, as long as each has different list of arguments.

Constructor is called depending upon number and type of arguments passed.

while creating objects, arguments must be passed to let compiler know which constructor needs to be called.

→ data members → member funct" → create object (with/without parameter)
define define
access data members

③ Copy constructor Initializes an object using another object of the same class.

It takes reference to an object of the same class as an argument.

```
Sample (Sample &t) {
    id = t.id
}
```

Implicit copy constructor

```
int main() {
    Sample obj1;
    obj1.Init(10);
    obj1.display();
    Sample obj2(obj1);
    obj2.display();
    return 0;
}
```

Explicit copy constructor

```
class Sample {
    int id;
public:
    void Init(int x)
    {
        id = x;
    }
    Sample() { } // Default
    Sample (Sample &t)
    {
        id = t.id;
    } // copy
    void display()
    {
        cout << id;
    }
};
```

(Same driver code)

④ strcpy() → lib funct" used to copy one string to another.

After copying src string to dest string, strcpy() func returns a pointer to the destination string..

```
char* strcpy (char* dest, const char* src);
```

—x—

When is copy constructor called?

(1) when an object of class is returned by value
(2) obj of class is passed (to funct") by value as an argument
(3) obj is constructed based on another object of same class.
(4) compiler generates a temporary obj

Copy elision (optimized) : (prevents making extra copies).

```
for (int i=0; i<=2; i++) {
    G.print();
    cout << "\n";
}
```

RVO (Return value optimization) is a technique that gives compiler some additional power to terminate the temporary object created.

Shallow copy (faster)
→ pointer points to the same copy of objects of class

Deep copy (comparatively slow)
→ It creates copy of each object inside of the class.

Page No. 59
Date

- Default copy constructor → shallow copy (pointers get copied not the location they are pointing to)

- Copy constructor → deep copy (pointers and location pointing both get copied)

- Destructors → (does not have arguments)
  invoked automatically whenever an object is going to be destroyed.
  → Destructor is the last function going to be called before an object is destroyed.
  → should be declared in public section of class.
  → Destroys class objects created by constructor
  → Same name as class preceded by (~)
  Syntax: ~ClassName (inside class)
  → Not possible to define more than one destructor
  → only one way to destroy hence can't be overloaded.

  * → automatically called when object goes out of scope (} ends / program ends / delete operator called)
  → release memory space occupied by objects created by constructor
  * → In destructor, objects are destroyed in the reverse of an object creation
  → Do not return any value.
  → To call destructor outside class
  Syntax: ClassName :: ~ClassName()
  Destructor is also called when a block containing local variables ends.

* Destructors can be virtual and virtual destructors ensure that object of derived class is destructed properly.
⇒ (Deleting derived class obj. using pointer of base class type that has non virtual destructor is undefined.)
Q. when do we need to write a user-defined destructor?
A. Compiler constructs default destructor unless we have dynamically allocated m/o or pointer in class.
↓
to release m/o before the class instance is destroyed. Done to avoid m/o leak.

* Destructors can be virtual.

INHERITENCE IN C++ →
capability of class to derive properties and characteristics from another class (new classes "Derived classes" are created from existing class. Also called child class and base class called "parent class".

Sub class → that inherits → Derived class
Super class → whose inherited → base class

Helps avoiding repetition of code by inheriting the common properties

Syntax:
class Derived : Access specifier BaseClassName
{
// Body
};

* Derived class do not inherit access to private data members

* fflush() → to clear the output buffer and move the buffered data to console / disk.

When base class inherited privately, public members of base class become private member of derived class. They are inaccessible to objects of derived class.

modes of inheritance → Public mode, Protected mode, Private mode

```
# class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};
class B : public A {
    // x is public
    // y is protected
    // z not accessible from B
};
class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};
class D : private A {
    // x is private
    // y is private
    // z is not accessible from D
};
```


Types of Inheritance :
1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Class A (base) → Class B (derived) → Single Inheritance

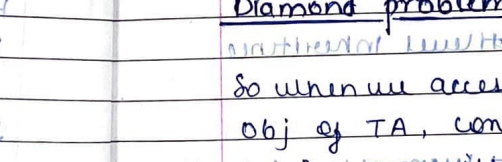
```
class A { ... };
class B : public A {
    ...
};
```

2. Class A, Class B → Class C → Multiple Inheritance

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

This order determines which constructor to be called first.

Diamond problem :



So when we access obj of TA, constructor of Person will be called twice (one with student and then faculty).

Problem: This causes ambiguity.

Solution: "Virtual" keyword we make faculty and student as virtual base classes

default constructor
just using virtual will call the class not its parameter

To call parameterized constructor of the grandparent class, the constructor has to be called in son's class

Ex. class TA : public faculty, public Student {
public:
TA(int x) : Student(x), faculty(x), Person(x) {
cout<< "TA :: TA(int) called" <<endl;
};

* It is not allowed to call grandparent's constructor directly, it has to be called through parent class.
It is only allowed if "virtual" keyword is used.

3. class C (Base 2)
(Base 1) class B
(Indirect) class A
→ multilevel inheritance

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

4. class A
class B, class C, class D, class E
→ Hierarchical inheritance

5. class F, class G → class B, class E → class A, class C
→ Hybrid (virtual) inheritance

6. multipath inheritance : special case of hybrid inheritance
(Diamond problem case)
To remove ambiguity ; use "virtual"
(or)
use scope resolution operator, to specify the path from which data member will be accessed.

POLYMORPHISM : IN C++ → (many forms)
(different behaviour in different situations).
* polymorphism is ability to take many forms. It occurs if there is hierarchy of classes that are all related to each other by inheritance.

compile time, Run time
function overloading, operator overloading, virtual functions.

1. Compile time polymorphism (func / operator overloading)

(a) function overloading → multiple func^n with same name, diff. parameters.

* overloading functions are good because you don't have to find a new name for new method you want to code.

Page No.
Date

• Conditions for function overloading →
(1) parameters should have different type
(2) parameters should have diff. number
(3) parameter should have diff sequence.

(b) operator overloading → give special meaning to an existing operator without changing their original meaning.

(Ex) '+' can add two built in datatypes but applying it on user defined data type generates error.
This is where we redefine the meaning of '+' operator such that it adds user defined datatypes

Redefining doesn't mean operator loses its property, instead they have been given additional meaning along with their existing ones.

```
Class Name operator + (Class Name const &obj)
{
    Complex res;
    res.real = real + obj.real;
    res.Imag = Imag + obj.Imag;
    return res;
}
```

(redefined the operator)

∴ Now compiler $C_3 = C_1 + C_2$ won't give error.

Page No. 67
Date

* List of operators that can't be overloaded
(1) sizeof (return size; not evaluated in runtime)
(2) typeid (purpose is to uniquely identify)
(3) Scope resolution (::) (syntatically not possible)
(4) Class member access operators
. (dot) ; .* (pointer to member operator)
(5) Ternary or conditional (?:)

* Operators that can be overloaded
(i) Unary
(ii) Binary
(iii) Special operators ([], (), etc)
Subscript
Func call
(iv) Assignment
(v) Bit-wise
(vi) Logical
(vii) Relational
(viii) De-Referencing
(ix) Dynamic m/o allocation / de allocn.

* Important things about operator overloading :
(1) for it to work, atleast one of the operand must be user-defined class
(2) Compiler automatically creates a default assignment operator with every class. (same as copy constructor)
(3) Conversion operator convert one type to another (Overloaded conv. operator must be member method)
(4) Single arg. constructor works as conversion constructor

* In C++ calling a virtual funcⁿ means if we call a member function, then it would cause a diff func to be executed instead depending on what type of object invoked it.

2. Runtime polymorphism → (by virtual functions)

virtual func declared within base class and is redefined (overriden) in derived class.

↳ Ensures that the correct function is called for an object regardless type of reference (or pointer) used for the function call.

Rules of virtual funcⁿ →

(1) Can't be static but can be friend of another class.

(2) To achieve runtime polymorphism, virtual func should be accessed using pointer or reference of base class type.

(3) Prototype should be same in base and derived class.

(4) Not mandatory for derived class to override; in that case the base class version of the function is executed.

(5) Class may have virtual destructor but not a virtual constructor.

(6) Virtual funcⁿ in base class might not be used.

ex. of virtual func() →

```
class base {
public: virtual func1() { }
       func2() { } };          ← precedence since derived below
class derived : public base {
public: func1() { }            ← override
       func2() { } };          ← these stub will be executed
```

If a virtual function is defined in a base class, there is no necessity of redefining it in the derived class

Page No. 69

Compile time (early binding) vs Runtime (late binding) behaviour of virt. func.

Late binding is done acc with the content of pointer. (i.e locatⁿ pointed to by pointer)

Early binding done acc to type of pointer.

In the prev example

func1 → late binding (as ptr is pointing towards it)

func 2 → early binding (as it is being printed from base class and ptr is also of base type.).

~~* Virtual func are resolved late, at runtime.~~

→ The technique of virtual function falls under runtime polymorphism.

~~virtual functions are not called on their own unless derived or overridden.~~

working of virtual functions

(1) VPTR (virtual pointer)

If object of class is created then the VPTR is inserted as a data member of class to point to VTABLE (virtual table) of that class.

VTABLE → static array of function pointers.

(Store the address of each virtual function contained in that class).

25/06/2022

Page No.
Date

ENCAPSULATION →

(binding together the data and functions that manipulate them)

we cannot access any function from class directly. we need an object to access that function which is using the member, the variable of that class.

* Encapsulation occurs only if the function we are making inside the class make use of all member variable.

E.g. we make a private variable x in a class which is then only accessed using get() & set() function present inside the class.

↓

Here x, get() & set() are binded together which is called encapsulation

* Role of access specifiers in encapsulation →

The process of implementing encapsulation can be subdivided into two steps:

1. Data members should be labeled as private using private access specifiers
2. member function which manipulate the data members should be labeled as public using the public access specifiers.

Page No. 71
Date

ABSTRACTION →

(displaying only essential information and hiding the details)

(i) Abstraction using classes
(ii) Abstraction in Header files.

(1) class can decide which data member will be visible to outside world or not
(2) In header file e.g when we add math.h header file and call function pow() we do not actually know the underlying algo

* Access specifiers are main pillar of abstraction.
(i) members declared public in class can be accessed from anywhere in the program
(ii) private members can only be accessed from within class.

• Advantages:
> avoid writing low level code
> avoid code duplication & increases reusability
> Yes security of an application or program as only important details provided to the user.
> can change internal implementation independently without affecting users.