Object Oriented Programming with C++

Short notes for B.Tech CSE / IT · BCA · MCA

Uncodemy Handwritten Notes · free for students

I. Introduction to OOP

- **Procedural programming** (C): program = functions working on data. Data is often global and unsafe.
- **Object-oriented programming**: program = objects that hold data and functions together.
- C++ was developed by Bjarne Stroustrup as "C with Classes". It supports both styles.

➤ Features of OOP

Feature	Meaning
Class	Blueprint / user-defined type
Object	Instance of a class
Encapsulation	Wrap data and functions in one unit, hide data
Abstraction	Show only essential details
Inheritance	New class reuses an existing class
Polymorphism	One name, many forms
Message passing	Objects talk by calling methods

Procedural (C)	OOP (C++)
Top-down approach	Bottom-up approach
Focus on functions	Focus on data / objects
No access control	public, private, protected
Hard to reuse	Reuse via inheritance

2. C++ Basics

```
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;                  // input
    cout << "x = " << x << endl;  // output
    return 0;
}
```

- `cin` with `>>` (extraction), `cout` with `<<` (insertion).
- **Reference variable**: another name for an existing variable. `int &r = x;` Must be initialised; cannot be re-bound.
- **Scope resolution operator** `::` accesses a global variable or defines a member outside the class.
- `new` and `delete` allocate and free memory: `int *p = new int[5]; delete[] p;`

➤ Functions in C++

- **Inline function**: the compiler may replace the call with the body. Good for small functions.
- **Default arguments**: `void f(int a, int b = 10);` Defaults must be given from the right.
- **Function overloading**: same name, different parameter list (number or type). Return type alone is not enough.

> **EXAMPLE** `int area(int s);` and `double area(double r);` can both exist. `area(5)` calls the int version.

Pointer	Reference
Can be NULL	Cannot be null
Can point elsewhere later	Bound once
Uses `*` to access	Used like the variable itself

3. Classes and Objects

```
class Student {
  private:
    int roll;
  public:
    void set(int r) { roll = r; }
    int  get()      { return roll; }
};
Student s; s.set(5);
```

Access specifier	Same class	Derived class	Outside
private	Yes	No	No
protected	Yes	Yes	No
public	Yes	Yes	Yes

- Default access in a `class` is `private`; in a `struct` it is public.
- **Static data member**: one copy shared by all objects. Defined outside the class. **Static member function** can use only static members and has no `this`.
- **this pointer**: points to the object that called the member function.
- **Friend function**: not a member, but can access private data. Declared with `friend` inside the class.
- **const member function** `int get() const;` cannot change the object.

4. Constructors and Destructors

- **Constructor**: same name as the class, no return type, called automatically when an object is created.

Type	Example
Default	`A()`
Parameterized	`A(int x)`
Copy	`A(const A &obj)`

- If you write no constructor, the compiler gives a default one. If you write any constructor, no default is generated automatically.
- **Copy constructor** is called when: an object is initialised from another, passed by value, returned by value.

- Default copy is a **shallow copy** (copies pointer values). Write your own copy constructor for a **deep copy** when the class owns heap memory.
- **Initializer list**: `A(int x) : a(x) {}`. Needed for const and reference members.
- **Destructor** `~A()`: no arguments, no return type, cannot be overloaded. Called when the object goes out of scope.

REMEMBER Objects are destroyed in the reverse order of their construction.

EXAMPLE `A a1; A a2;` at the end of main, the destructor runs for a2 first, then a1.

5. Inheritance

- A **derived class** gets members of a **base class**. Syntax: `class B : public A { };`

Type	Structure
Single	A → B
Multiple	A, B → C
Multilevel	A → B → C
Hierarchical	A → B, A → C
Hybrid	Mix, e.g. diamond shape

➤ Visibility after inheritance

Base member	public inheritance	protected inheritance	private inheritance
public	public	protected	private
protected	protected	protected	private
private	Not accessible	Not accessible	Not accessible

- Constructors run base first, then derived. Destructors run derived first, then base.
- **Diamond problem**: B and C inherit A, D inherits B and C, so D has two copies of A. Fix: **virtual base class** `class B : virtual public A`.

```
   A
  / \
 B   C      class B : virtual public A {};
  \ /       class C : virtual public A {};
   D        class D : public B, public C {};
```

6. Polymorphism

Compile-time (static)	Run-time (dynamic)
Function overloading	Function overriding with `virtual`
Operator overloading	Base pointer calling derived function
Templates	Uses vtable (virtual table)
Early binding	Late binding

➤ Operator overloading

```
class Complex {
  public:
    int re, im;
    Complex operator+(const Complex &c) {
        Complex t; t.re = re + c.re; t.im = im + c.im;
        return t;
    }
};
```

- Cannot overload: `::`, `.`, `.*`, `?:`, `sizeof`.
- `<<` and `>>` for cout / cin are overloaded as **friend** functions.

➤ Virtual functions

```
class Shape { public: virtual void draw() { cout << "Shape"; } };
class Circle : public Shape {
  public: void draw() { cout << "Circle"; }
};
Shape *p = new Circle();
p->draw();      // prints Circle (without virtual, prints Shape)
```

- **Pure virtual function**: `virtual void draw() = 0;`. A class with one is an abstract class and cannot have objects.
- **Virtual destructor**: make the base destructor virtual so `delete basePtr;` also runs the derived destructor.
- Constructors cannot be virtual.

7. Templates and Exception Handling

➤ Templates (generic programming)

```
template <typename T>
T maxOf(T a, T b) { return (a > b) ? a : b; }
// maxOf(3, 7) -> 7, maxOf(2.5, 1.5) -> 2.5
```

- **Function template** and **class template** (`template <class T> class Stack { ... };`).

➤ Exception handling

```
try {
    if (b == 0) throw "Divide by zero";
    cout << a / b;
}
catch (const char *msg) { cout << msg; }
catch (...)             { cout << "Other error"; }
```

- `try` holds risky code, `throw` raises an exception, `catch` handles it. `catch(...)` catches everything and must be last.

8. File Handling and STL

➤ Files (fstream)

- `ofstream` (write), `ifstream` (read), `fstream` (both).
- Modes: `ios::in`, `ios::out`, `ios::app`, `ios::binary`, `ios::trunc`.

```
ofstream f("data.txt");
f << "Hello";
f.close();
```

➤ STL (Standard Template Library)

Part	Examples
Containers	vector, list, deque, set, map, stack, queue, priority_queue
Iterators	begin(), end(), used to move through containers
Algorithms	sort, find, reverse, count, binary_search

- `vector`: dynamic array, O(1) access, push_back amortised O(1).
- `map`: sorted key-value pairs (usually red-black tree), O(log n). `unordered_map`: hash table, average O(1).

> **TIP** `sort(v.begin(), v.end());` sorts a vector in ascending order.

9. Most Asked Questions (practice)

1. Explain the main features of OOP. Procedural vs object-oriented programming.
2. What is a class and object? Explain access specifiers.
3. Types of constructors. Explain the copy constructor and deep vs shallow copy.

4. Explain types of inheritance with examples. What is the diamond problem?
5. Compile-time vs run-time polymorphism.
6. Write a program to overload the + operator for complex numbers.
7. What are virtual functions and pure virtual functions? What is an abstract class?
8. Explain friend functions and static members with examples.
9. Explain function templates and class templates.
10. Explain exception handling using try, throw and catch.

TIP University exams expect short programs: constructor/destructor order, operator overloading, virtual function output and templates. Practise writing them neatly with comments.