Notes on C++ Programming

Chapter 1: Basics of C++ Programming

1.1 Introduction to C++

C++ is a **general-purpose, object-oriented programming language** developed by **Bjarne Stroustrup** at Bell Labs in 1979 as an extension of the C programming language. Originally called **“C with Classes”**, it introduced the concept of **objects and classes** into the already powerful C language.

C++ is widely used in system programming, game development, operating systems, browsers, compilers, and high-performance applications.

Key Features of C++:

- **Object-Oriented:** Supports classes, objects, inheritance, and polymorphism.
- **Portable:** Programs can run on multiple platforms with little or no modification.
- **Compiled Language:** Translated into machine code for fast execution.
- **Low-Level Manipulation:** Allows direct memory access using pointers.
- **Rich Library (STL):** Includes data structures and algorithms for efficiency.

1.2 Structure of a C++ Program

A simple C++ program:

```
#include <iostream>  // header file for input/output

using namespace std; // allows use of cout, cin without std::


int main() {

   cout << "Hello, World!";  // output statement

   return 0;              // exit status

}
```

Explanation:

1. #include <iostream> → Preprocessor directive, imports input-output functionality.

2. using namespace std; → Gives access to standard library objects like cout.
3. int main() → The entry point of a C++ program.
4. cout << "Hello, World!"; → Prints text to the console.
5. return 0; → Indicates successful program execution.

1.3 Compilation & Execution Process

Steps to run a C++ program:

1. **Write code** in a .cpp file.
2. **Compile** using a compiler (like g++, clang).
    - Example: g++ program.cpp -o program
3. **Run** the output file.
    - Example: ./program

The compiler translates C++ source code into machine language.

1.4 Data Types in C++

C++ supports different types of data:

Type	Description	Example
int	Integer numbers	10, -50
float	Single-precision decimal	3.14
double	Double-precision decimal	12.3456789
char	Single character	'A', 'z'
bool	Boolean values (true/false)	true, false
string (C++)	Sequence of characters	"Hello"

Example:

```
#include <iostream>

using namespace std;
```

```
int main() {
   int age = 20;
   float height = 5.9;
   char grade = 'A';
   bool isStudent = true;
   string name = "John";

   cout << "Name: " << name << endl;
   cout << "Age: " << age << endl;
   cout << "Height: " << height << endl;
   cout << "Grade: " << grade << endl;
   cout << "Is Student? " << isStudent << endl;
   return 0;
}
```

1.5 Variables, Constants, and Keywords

- **Variables** → Named storage for data.
- **Constants** → Fixed values that cannot be changed.
- **Keywords** → Reserved words like int, float, class.

Example:

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

int main() {
   const float PI = 3.14159;   // constant
   int radius = 5;           // variable
   float area = PI * radius * radius;
   cout << "Area of circle: " << area;
```

```cpp
    return 0;
}
```

1.6 Input and Output in C++

C++ uses cin for input and cout for output.

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

int main() {
    int a, b;
    cout << "Enter two numbers: ";
    cin >> a >> b;   // input
    cout << "Sum = " << a + b;  // output
    return 0;
}
```

1.7 Example Programs

1. Hello World Program

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

int main() {
    cout << "Hello, World!";
    return 0;
}
```

2. Swapping Two Numbers

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

```
int main() {
   int x = 10, y = 20, temp;
   cout << "Before swap: x=" << x << " y=" << y << endl;
   temp = x;
   x = y;
   y = temp;
   cout << "After swap: x=" << x << " y=" << y << endl;
   return 0;
}
```

3. Simple Calculator

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

int main() {
   int a, b;
   cout << "Enter two numbers: ";
   cin >> a >> b;
   cout << "Addition: " << a + b << endl;
   cout << "Subtraction: " << a - b << endl;
   cout << "Multiplication: " << a * b << endl;
   cout << "Division: " << (float)a / b << endl;
   return 0;
}
```

Chapter 2: Operators & Control Statements in C++

2.1 Operators in C++

Operators are **symbols** used to perform operations on variables and values.

Types of Operators:

1. Arithmetic Operators

Perform mathematical operations.

Operator	Description	Example (a=10, b=3)	Result
+	Addition	a+b	13
-	Subtraction	a-b	7
*	Multiplication	a*b	30
/	Division	a/b	3
%	Modulus (remainder)	a%b	1

```
int a=10, b=3;
cout << "a+b=" << a+b;
```

---

2. Relational Operators

Used to compare values.

Operator	Meaning	Example (a=10, b=20)	Result
==	Equal to	a==b	false
!=	Not equal to	a!=b	true
>	Greater than	a>b	false
<	Less than	a<b	true
>=	Greater or equal	a>=b	false
<=	Less or equal	a<=b	true

---

3. Logical Operators

Combine conditions.

Operator	Meaning	Example	Result
&&	Logical AND	(a>5 && b<10)	true
`		`	Logical OR
!	Logical NOT	!(a>5)	false

---

4. Assignment Operators

Assign values to variables.

Operator	Example	Equivalent to
=	x=10	x=10
+=	x+=5	x=x+5
-=	x-=5	x=x-5
*=	x*=5	x=x*5
/=	x/=5	x=x/5

---

5. Increment & Decrement

- ++ → Increases value by 1.
- -- → Decreases value by 1.

Example:

```
int x = 5;

cout << x++; // prints 5, then x=6

cout << ++x; // x=7, prints 7
```

---

6. Bitwise Operators

Work on binary data.

Operator	Meaning	Example (a=5, b=3)	Result
&	AND	a & b (0101 & 0011)	0001=1

Operator	Meaning	Example (a=5, b=3)	Result
`	`	OR	` a
^	XOR	a ^ b	0110=6
<<	Left shift	a<<1 (0101 → 1010)	10
>>	Right shift	a>>1 (0101 → 0010)	2

---

7. Ternary Operator

Shorthand for if-else.

```
int age=20;

string result = (age>=18) ? "Eligible" : "Not Eligible";

cout << result;
```

2.2 Control Statements in C++

Control statements allow us to control the **flow of execution**.

---

1. Decision Making Statements

if statement

Executes a block if condition is true.

```
if(age>=18) {
   cout << "You can vote!";
}
```

---

if-else statement

Executes one block if condition is true, else another block.

```
if(age>=18) {
   cout << "You can vote!";
} else {
```

```
  cout << "You cannot vote!";
}
```

---

if-else if ladder

Multiple conditions.

```
if(marks >= 90) cout << "Grade A";
else if(marks >= 75) cout << "Grade B";
else if(marks >= 50) cout << "Grade C";
else cout << "Fail";
```

---

switch statement

Used when multiple cases exist.

```
int day = 3;
switch(day) {
  case 1: cout << "Monday"; break;
  case 2: cout << "Tuesday"; break;
  case 3: cout << "Wednesday"; break;
  default: cout << "Invalid day";
}
```

---

2. Looping Statements

for loop

Used when number of iterations is known.

```
for(int i=1; i<=5; i++) {
  cout << i << " ";
}
```

---

while loop

Used when number of iterations is not fixed.

```
int i=1;

while(i<=5) {
  cout << i << " ";
  i++;
}
```

---

do-while loop

Executes at least once.

```
int i=1;

do {
  cout << i << " ";
  i++;
} while(i<=5);
```

---

3. Jump Statements

- **break** → exits from loop/switch.
- **continue** → skips current iteration.
- **goto** → jumps to a labeled statement (not recommended).

Example:

```
for(int i=1; i<=5; i++) {
  if(i==3) continue; // skips 3
  if(i==5) break;    // stops loop
  cout << i << " ";
}
```

2.3 Example Programs

1. Find Maximum of Three Numbers

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

int main() {
   int a, b, c;
   cout << "Enter three numbers: ";
   cin >> a >> b >> c;

   if(a>=b && a>=c) cout << "Max = " << a;
   else if(b>=a && b>=c) cout << "Max = " << b;
   else cout << "Max = " << c;
   return 0;
}
```

---

2. Print Multiplication Table using Loop

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

int main() {
   int n;
   cout << "Enter a number: ";
   cin >> n;

   for(int i=1; i<=10; i++) {
      cout << n << " * " << i << " = " << n*i << endl;
   }
   return 0;
}
```

3. Check Prime Number using while Loop

```
#include <iostream>

using namespace std;


int main() {

   int n, i=2, flag=0;

   cout << "Enter a number: ";

   cin >> n;


   while(i<=n/2) {

      if(n%i==0) {

         flag=1; break;

      }

      i++;

   }

   if(flag==0) cout << n << " is Prime.";

   else cout << n << " is Not Prime.";

   return 0;

}
```

Chapter 3: Functions and Storage Classes in C++

3.1 Introduction to Functions

A **function** in C++ is a block of code designed to perform a specific task.
Functions make programs **modular, reusable, and easier to debug**.

Types of Functions:

1. **Library (Built-in) Functions** – already available in C++ (e.g., sqrt(), pow(), strlen()).
2. **User-defined Functions** – created by the programmer.

3.2 Function Syntax

```
return_type function_name(parameter_list) {
   // body of function
   return value;
}
```

Example:

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

int add(int a, int b) {   // function definition
   return a + b;
}

int main() {
   int result = add(5, 3);   // function call
   cout << "Sum = " << result;
   return 0;
}
```

3.3 Function Prototypes

A **function prototype** tells the compiler about the function before its actual definition.

```
int add(int, int);   // prototype

int main() {
```

```
   cout << add(2,3);
}

int add(int a, int b) {
   return a+b;
}
```

3.4 Categories of User-Defined Functions

Functions can be categorized by **parameters** and **return values**.

1. **No arguments, no return value**
2. void greet() {
3.   cout << "Hello World!";
4. }
5. **No arguments, with return value**
6. int getNumber() {
7.   return 10;
8. }
9. **With arguments, no return value**
10. void printSquare(int n) {
11.   cout << n*n;
12. }
13. **With arguments, with return value**
14. int square(int n) {
15.   return n*n;
16. }

3.5 Inline Functions

- Declared using the keyword inline.

- Used for **small functions** to reduce function call overhead.

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

inline int cube(int x) {
   return x*x*x;
}

int main() {
   cout << "Cube of 3 = " << cube(3);
}
```

3.6 Function Overloading

Two or more functions with the same name but **different parameter lists**.

```
int add(int a, int b) {
   return a+b;
}

double add(double a, double b) {
   return a+b;
}
```

3.7 Recursive Functions

A function that calls itself.

Example: Factorial

```
int factorial(int n) {
   if(n==0 || n==1) return 1;
   return n * factorial(n-1);
```

}

3.8 Storage Classes in C++

Storage classes define **scope, lifetime, and visibility** of variables.

Types of Storage Classes:

1. Automatic (auto)

- Default for local variables.
- Lifetime: Within the block.
- Scope: Local.

```
void test() {
   auto int x = 10;
   cout << x;
}
```

---

2. Register

- Stored in **CPU registers** (faster access).
- Used for frequently used variables.
- Scope: Local.

```
void fastCalc() {
   register int count;
   for(count=0; count<5; count++)
      cout << count << " ";
}
```

---

3. Static

- Retains value between function calls.
- Initialized only once.

```
void demo() {
```

```
    static int x = 0;
    x++;
    cout << x << " ";
}

int main() {
    demo(); // prints 1
    demo(); // prints 2
}
```

---

4. Extern

- Declares a global variable accessible across files.

file1.cpp

```
int num = 10;   // definition
```

file2.cpp

```
extern int num; // declaration
cout << num;
```

---

5. Mutable (used in classes)

- Allows modification of class members even in const objects.

```
class Student {
    mutable int age;
public:
    void setAge(int a) const {
        age = a;
    }
};
```

3.9 Examples

Example 1: Fibonacci using Recursion

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

int fib(int n) {
   if(n<=1) return n;
   return fib(n-1) + fib(n-2);
}

int main() {
   for(int i=0; i<5; i++) {
      cout << fib(i) << " ";
   }
}
```

Example 2: Static Variable Demo

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

void counter() {
   static int c = 0;
   c++;
   cout << "Count = " << c << endl;
}

int main() {
   counter();
```

```
    counter();
    counter();
}
```

Example 3: Function Overloading for Area

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

int area(int side) {
    return side*side;
}

int area(int l, int b) {
    return l*b;
}

double area(double r) {
    return 3.14*r*r;
}

int main() {
    cout << "Square Area = " << area(5) << endl;
    cout << "Rectangle Area = " << area(4,6) << endl;
    cout << "Circle Area = " << area(3.0);
}
```

Chapter 4: Object-Oriented Programming (OOP) Concepts in C++

4.1 Introduction to OOP

Object-Oriented Programming (OOP) is a **programming paradigm** based on the concept of **objects**.
C++ introduced OOP features on top of C, making it one of the most powerful languages for software development.

Core ideas of OOP:

1. **Encapsulation** – Wrapping data & methods into a single unit (class).
2. **Abstraction** – Showing essential details while hiding unnecessary implementation.
3. **Inheritance** – Acquiring properties from existing classes.
4. **Polymorphism** – One entity behaving differently in different contexts.

4.2 Classes and Objects

- **Class** → blueprint for objects.
- **Object** → instance of a class.

Example: Simple Class

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

class Student {
public:
  string name;
  int age;

  void display() {
    cout << "Name: " << name << ", Age: " << age << endl;
  }
```

```
};

int main() {
   Student s1;      // Object creation
   s1.name = "Raj";
   s1.age = 21;
   s1.display();
}
```

4.3 Encapsulation

- **Data + Functions = Class**
- Protects data from outside interference using **access specifiers**:
  - public → accessible everywhere
  - private → accessible only within class
  - protected → accessible within class & derived class

Example:

```
class Account {
private:
   double balance;

public:
   void setBalance(double b) {
      if(b >= 0) balance = b;
   }
   double getBalance() {
      return balance;
   }
};
```

4.4 Abstraction

- Focus on **what an object does**, not **how it does it**.
- Achieved using **abstract classes** and **interfaces (pure virtual functions)**.

Example:

```
class Shape {
public:
  virtual void draw() = 0;  // Pure virtual function
};

class Circle : public Shape {
public:
  void draw() override {
    cout << "Drawing Circle" << endl;
  }
};
```

4.5 Inheritance

Inheritance allows creating a new class from an existing class.

Types of Inheritance in C++:

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

Example: Single Inheritance

```
class Animal {
```

```
public:
   void eat() {
       cout << "Eating..." << endl;
   }
};

class Dog : public Animal {
public:
   void bark() {
       cout << "Barking..." << endl;
   }
};

int main() {
   Dog d;
   d.eat();
   d.bark();
}
```

Example: Multiple Inheritance

```
class A {
public:
   void displayA() { cout << "Class A" << endl; }
};

class B {
public:
   void displayB() { cout << "Class B" << endl; }
```

```cpp
};

class C : public A, public B {
};

int main() {
  C obj;
  obj.displayA();
  obj.displayB();
}
```

4.6 Polymorphism

Polymorphism = "many forms"
Two types:

1. **Compile-time (Static Polymorphism)** – Function Overloading & Operator Overloading
2. **Run-time (Dynamic Polymorphism)** – Virtual Functions & Function Overriding

---

Example: Function Overriding (Run-time Polymorphism)

```cpp
class Parent {
public:
  virtual void show() {
    cout << "Parent class" << endl;
  }
};

class Child : public Parent {
public:
  void show() override {
```

```
        cout << "Child class" << endl;
    }
};

int main() {
    Parent* p;
    Child c;
    p = &c;
    p->show();  // Calls Child version
}
```

4.7 Constructors and Destructors

- **Constructor**: Special function called when object is created.
- **Destructor**: Special function called when object is destroyed.

Example:

```
class Demo {
public:
    Demo() { cout << "Constructor called" << endl; }
    ~Demo() { cout << "Destructor called" << endl; }
};

int main() {
    Demo d; // constructor invoked
} // destructor invoked automatically
```

4.8 Operator Overloading (Intro)

Operator overloading allows redefining operators for user-defined classes.

```
class Complex {
```

```cpp
    int real, imag;
public:
    Complex(int r=0, int i=0) : real(r), imag(i) {}
    Complex operator+(Complex c) {
        return Complex(real+c.real, imag+c.imag);
    }
    void display() { cout << real << "+" << imag << "i"; }
};

int main() {
    Complex c1(3,4), c2(1,2), c3;
    c3 = c1 + c2;
    c3.display();
}
```

4.9 Advantages of OOP in C++

- **Modularity** – Code is divided into objects.
- **Reusability** – Inheritance promotes reuse.
- **Extensibility** – Easy to extend systems.
- **Security** – Data hiding protects data.
- **Flexibility** – Polymorphism allows dynamic behavior.

Chapter 5: Arrays, Strings, and Pointers in C++

5.1 Introduction

Data in C++ can be stored as:

- **Arrays** – Collection of elements of the same type.
- **Strings** – Sequence of characters ending with '\0'.
- **Pointers** – Variables that store the memory address of another variable.

These three are **foundations** for working with structured data and dynamic memory in C++.

5.2 Arrays in C++

Definition

An **array** is a collection of elements of the same data type stored in **contiguous memory locations**.

Types of Arrays

1. **One-dimensional Array**
2. **Two-dimensional Array**
3. **Multi-dimensional Array**

Example: 1D Array

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

int main() {
  int marks[5] = {90, 85, 88, 92, 75};
  for(int i=0; i<5; i++) {
    cout << "Mark[" << i << "] = " << marks[i] << endl;
  }
  return 0;
```

```
}
```

Example: 2D Array

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

int main() {
  int matrix[2][3] = { {1,2,3}, {4,5,6} };
  for(int i=0; i<2; i++) {
    for(int j=0; j<3; j++) {
      cout << matrix[i][j] << " ";
    }
    cout << endl;
  }
}
```

Operations on Arrays

- Traversal
- Insertion
- Deletion
- Searching (Linear, Binary)
- Sorting (Bubble, Selection, Quick, etc.)

5.3 Strings in C++

Definition

A string is an array of characters terminated by '\0' (null character).

C++ provides two ways to use strings:

1. **C-Style Strings** (char str[20])

2. **C++ String Class** (#include <string>)

Example: C-Style String

```
#include <iostream>

#include <cstring>

using namespace std;


int main() {

   char name[20] = "Hello";

   cout << "Length: " << strlen(name) << endl;

   strcat(name, " World");  // Concatenation

   cout << "Message: " << name << endl;

}
```

Example: C++ String Class

```
#include <iostream>

#include <string>

using namespace std;


int main() {

   string str1 = "C++ ";

   string str2 = "Programming";

   string result = str1 + str2;

   cout << "Result: " << result << endl;

   cout << "Length: " << result.length() << endl;

}
```

Common String Operations

- Concatenation (+, append)
- Substring extraction (substr)
- Searching (find)
- Comparison (==, <, >)

5.4 Pointers in C++

Definition

A **pointer** is a variable that stores the **memory address** of another variable.

Declaring a Pointer

```
int x = 10;
int *ptr = &x;  // ptr stores address of x
```

Example: Pointer Basics

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

int main() {
   int a = 5;
   int *p = &a;
   cout << "Value of a: " << a << endl;
   cout << "Address of a: " << &a << endl;
   cout << "Pointer p: " << p << endl;
   cout << "Value pointed by p: " << *p << endl;
}
```

5.5 Pointers and Arrays

- Array name itself is a pointer to the first element.

Example:

```
int arr[5] = {10, 20, 30, 40, 50};

int *ptr = arr;


for(int i=0; i<5; i++) {

   cout << *(ptr+i) << " ";  // Access using pointer arithmetic

}
```

5.6 Dynamic Memory Allocation

C++ provides operators for dynamic memory management:

- new → allocate memory
- delete → deallocate memory

Example:

```
#include <iostream>

using namespace std;


int main() {

   int *ptr = new int(25);

   cout << "Value: " << *ptr << endl;

   delete ptr;  // free memory

}
```

5.7 Pointer to Pointer

- Pointers can point to other pointers.

```
int x = 10;

int *p = &x;

int **q = &p;


cout << **q; // 10
```

5.8 Pointers and Functions

Pointers are often used to pass variables by reference.

Example:

```
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
   int x=5, y=10;
   swap(&x, &y);
   cout << "x=" << x << ", y=" << y << endl;
}
```

5.9 Arrays, Strings, and Pointers Together

Example: Reverse a String Using Pointer

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

int main() {
   char str[] = "Hello";
   char *p = str;
   int len = strlen(str);

   for(int i=len-1; i>=0; i--) {
```

```
    cout << *(p+i);
  }
}
```

5.10 Summary

- Arrays store elements of the same type in sequence.
- Strings can be handled using C-style arrays or C++ string class.
- Pointers provide direct memory access and dynamic memory allocation.
- Combining arrays, strings, and pointers allows **efficient memory handling** in C++.

Chapter 6: Operator Overloading and Templates in C++

6.1 Introduction

C++ is not just an extension of C — it adds **object-oriented features** and **powerful abstractions**. Two such features are:

1. **Operator Overloading** → allows programmers to redefine operators for user-defined data types.
2. **Templates** → allow writing generic and reusable code for multiple data types.

These make C++ a flexible and efficient programming language for building **generalized libraries and frameworks**.

6.2 Operator Overloading in C++

Definition

Operator overloading allows C++ operators (+, -, *, [], (), etc.) to be redefined for **user-defined classes**.

- Improves **code readability**.
- Enables **natural usage** of objects with operators.
- Implemented using **special functions** called operator functions.

Syntax

```
return_type operator symbol (parameters) {
   // implementation
}
```

Example 1: Overloading + Operator

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

class Complex {
```

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

   // Overloading +
   Complex operator + (Complex const &obj) {
      return Complex(real + obj.real, imag + obj.imag);
   }

   void display() {
      cout << real << " + " << imag << "i" << endl;
   }
};

int main() {
   Complex c1(3,4), c2(1,2);
   Complex c3 = c1 + c2;   // uses overloaded +
   c3.display();  // 4 + 6i
}
```

Example 2: Overloading Unary ++ Operator

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

class Counter {
   int value;
public:
   Counter(int v=0): value(v) {}
```

```
  Counter operator ++() {  // prefix
    ++value;
    return *this;
  }

  void display() { cout << "Value: " << value << endl; }
};

int main() {
  Counter c1(5);
  ++c1;   // invokes operator++
  c1.display(); // Value: 6
}
```

Common Operators That Can Be Overloaded

- Arithmetic: +, -, *, /, %
- Relational: ==, !=, <, >
- Logical: &&, ||
- Increment/Decrement: ++, --
- Subscript: []
- Function Call: ()
- Stream Insertion/Extraction: <<, >>

Example 3: Overloading << and >> for I/O

```
#include <iostream>

using namespace std;
```

```
class Student {
  string name;
  int age;
public:
  Student(string n="", int a=0): name(n), age(a) {}

  friend ostream& operator << (ostream &out, const Student &s) {
    out << "Name: " << s.name << ", Age: " << s.age;
    return out;
  }

  friend istream& operator >> (istream &in, Student &s) {
    in >> s.name >> s.age;
    return in;
  }
};

int main() {
  Student s1;
  cout << "Enter name and age: ";
  cin >> s1;
  cout << s1 << endl;
}
```

👉 This shows how operator overloading makes **objects behave naturally like built-in types**.

6.3 Function Overloading vs Operator Overloading

- **Function Overloading** → Same function name, different parameters.

- **Operator Overloading** → Same operator symbol, different meaning for objects.

Both are **polymorphism** techniques in C++.

6.4 Templates in C++

Definition

Templates allow writing **generic functions and classes** that work with any data type.

- Introduced to support **code reusability**.
- Saves time by avoiding rewriting functions for different data types.
- Supports **compile-time polymorphism**.

6.4.1 Function Templates

Syntax

```
template <typename T>
T functionName(T a, T b) {
   // implementation
}
```

Example: Generic Swap Function

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

template <typename T>
void mySwap(T &x, T &y) {
   T temp = x;
   x = y;
   y = temp;
}
```

```
int main() {
   int a=5, b=10;
   mySwap(a,b);
   cout << "a=" << a << ", b=" << b << endl;

   double x=1.2, y=3.4;
   mySwap(x,y);
   cout << "x=" << x << ", y=" << y << endl;
}
```

6.4.2 Class Templates

Syntax

```
template <class T>
class ClassName {
   T data;
public:
   ClassName(T d) : data(d) {}
   void show() { cout << data << endl; }
};
```

Example: Class Template

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

template <class T>
class Box {
   T value;
public:
```

```
    Box(T v): value(v) {}

    void display() { cout << "Value: " << value << endl; }

};


int main() {

    Box<int> b1(10);

    Box<string> b2("Hello Templates");

    b1.display();

    b2.display();

}
```

6.5 Template Specialization

Sometimes we need a **different implementation** for a particular data type.
This is done using **template specialization**.

Example: Specialization for char

```
#include <iostream>

using namespace std;


template <class T>

class Print {

public:

    void show(T x) {

        cout << "Generic: " << x << endl;

    }

};


// Specialization for char
```

```
template <>
class Print<char> {
public:
  void show(char c) {
    cout << "Character: " << c << endl;
  }
};

int main() {
  Print<int> p1;
  p1.show(100);

  Print<char> p2;
  p2.show('A');
}
```

6.6 Advantages of Operator Overloading and Templates

- **Operator Overloading**
  - Increases readability.
  - Provides natural syntax for user-defined types.
  - Enhances **abstraction** in OOP.
- **Templates**
  - Code reusability.
  - Type-safety with generic programming.
  - Used in **STL (Standard Template Library)**.

6.7 Real-Life Applications

- **Operator Overloading**

    - Complex number operations.
    - Matrix manipulations.
    - Overloaded << and >> used in input/output.

- **Templates**
    - Standard Template Library (STL) uses templates for **vectors, lists, maps, stacks, queues**.
    - Generic sorting and searching algorithms.
    - Implementation of data structures independent of data type.

6.8 Summary

- Operator Overloading → makes custom objects behave like built-in types.
- Function Overloading vs Operator Overloading → both are forms of polymorphism.
- Templates → allow **generic programming** with functions and classes.
- Specialization allows **custom behavior** for specific data types.
- STL heavily relies on **templates and operator overloading**.

Chapter 7: File Handling and Exception Handling in C++

7.1 Introduction

File handling and exception handling are two essential components of real-world programming in C++.

- **File Handling** → enables reading, writing, and storing data permanently on secondary storage (hard disk, SSD, etc.).
- **Exception Handling** → provides a mechanism to handle runtime errors gracefully, preventing program crashes.

These features make C++ practical for building **robust, production-level software**.

7.2 File Handling in C++

7.2.1 Why File Handling?

- Data stored in variables or arrays is **temporary** and lost when the program ends.
- File handling allows **permanent storage** of data.
- Supports **large-scale data processing**.
- Used in databases, text editors, compilers, and OS components.

7.2.2 File Stream Classes in C++

C++ provides classes for file operations inside the **<fstream>** header:

1. **ifstream** → input file stream (for reading).
2. **ofstream** → output file stream (for writing).
3. **fstream** → supports both reading and writing.

7.2.3 File Opening Modes

Mode	Meaning
ios::in	Open file for reading
ios::out	Open file for writing

Mode	Meaning
ios::app	Append to end of file
ios::trunc	Discards file contents if file exists
ios::binary	Opens file in binary mode

7.2.4 Writing to a File

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

int main() {
   ofstream outFile("example.txt"); // create/open file
   if (!outFile) {
      cout << "File could not be opened!" << endl;
      return 1;
   }
   outFile << "Hello, this is C++ file handling!" << endl;
   outFile << "File handling makes data permanent.";
   outFile.close();
   cout << "Data written successfully." << endl;
   return 0;
}
```

✔ This creates example.txt and writes text into it.

7.2.5 Reading from a File

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main() {
  ifstream inFile("example.txt"); // open file for reading
  if (!inFile) {
    cout << "File not found!" << endl;
    return 1;
  }
  string line;
  while (getline(inFile, line)) {
    cout << line << endl;
  }
  inFile.close();
  return 0;
}
```

✔ Reads contents line by line and displays on screen.

7.2.6 File Handling with Objects

C++ allows writing objects directly to files using write() and read() methods.

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

class Student {
  char name[30];
  int age;
public:
  void input() {
```

```
        cout << "Enter name and age: ";
        cin >> name >> age;
    }
    void show() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Student s1;
    s1.input();

    ofstream outFile("student.dat", ios::binary);
    outFile.write((char*)&s1, sizeof(s1));
    outFile.close();

    Student s2;
    ifstream inFile("student.dat", ios::binary);
    inFile.read((char*)&s2, sizeof(s2));
    inFile.close();

    s2.show();
}
```

✔ Demonstrates **binary file handling** using objects.

7.2.7 Advantages of File Handling

- **Permanent storage** of data.
- Supports **large datasets**.

- Enables data sharing between programs.
- Essential for databases, logs, and backups.

7.3 Exception Handling in C++

7.3.1 What is an Exception?

- An **exception** is an error or unexpected event that occurs during program execution.
- Without handling, exceptions can cause program **termination** or **crashes**.
- Exception handling ensures **smooth recovery** from such errors.

7.3.2 Exception Handling Keywords

1. **try** → block of code that may generate an exception.
2. **catch** → block that handles the exception.
3. **throw** → used to signal (raise) an exception.

7.3.3 Basic Example

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

int main() {
  int a, b;
  cout << "Enter two numbers: ";
  cin >> a >> b;

  try {
    if (b == 0)
      throw "Division by zero error!";
    cout << "Result: " << a / b << endl;
```

```
    }
    catch (const char* msg) {
        cout << "Exception: " << msg << endl;
    }
}
```

✔ Prevents **division by zero crash**.

7.3.4 Multiple Catch Blocks

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

int main() {
    try {
        throw 10; // throwing integer
    }
    catch (int x) {
        cout << "Caught an integer: " << x << endl;
    }
    catch (...) {
        cout << "Caught an unknown exception" << endl;
    }
}
```

✔ The catch(...) block catches **any exception type**.

7.3.5 Exception Handling with Classes

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

```
int main() {
   try {
      throw runtime_error("Something went wrong!");
   }
   catch (runtime_error &e) {
      cout << "Exception: " << e.what() << endl;
   }
}
```

✔ Standard library exceptions like **runtime_error, out_of_range, bad_alloc** are commonly used.

7.3.6 Nested Try-Catch Blocks

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

int main() {
   try {
      try {
         throw "Inner Exception";
      }
      catch (const char* msg) {
         cout << "Caught inside: " << msg << endl;
         throw; // rethrowing
      }
   }
   catch (const char* msg) {
      cout << "Caught outside: " << msg << endl;
```

```
  }
}
```

✔ Exceptions can be **re-thrown** and handled at multiple levels.

7.4 File Handling + Exception Handling (Combined Example)

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

int main() {
  try {
    ifstream file("data.txt");
    if (!file)
      throw runtime_error("File not found!");

    string line;
    while (getline(file, line)) {
      cout << line << endl;
    }
  }
  catch (runtime_error &e) {
    cout << "Exception: " << e.what() << endl;
  }
}
```

✔ Combines **file I/O** and **error handling** in one program.

7.5 Advantages

- **File Handling**

    - Data persistence.
    - Large-scale storage.
    - Essential for real-world apps.
- **Exception Handling**
    - Prevents crashes.
    - Improves reliability.
    - Makes debugging easier.

7.6 Summary

- File handling in C++ uses **ifstream, ofstream, fstream**.
- Supports **text and binary files**.
- Exception handling uses **try, catch, throw**.
- Supports multiple and nested catches.
- Standard exceptions improve **robustness**.
- Together, they make programs **safe and practical for real-world use**.