length(), substr()

stoi(), to_string()

C++
CHEAT SHEET

SCALER Topics

std::min

std::max_element

01 Basics

Why C++?

- Efficient and low-level access
- Widely used in system programming

C++ vs C

C++ is an extension of C with object-oriented features

Data Types

Integer types

int, short, long, long long

Character types

char, wchar_t, char16_t, char32_t

Floating-point types

float, double, long double

Boolean

bool

Input and Output

console input

cin is used for input stream

console output

cout is used for output stream

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

int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "You entered: " << num;
    return 0;
}
```

• use iomanip library for formatting output • setw, setprecision, and fixed help control the width, precision, and formatting	`#include <iomanip>` `cout << fixed << setprecision(2) << 3.14159;`

• comment can be single line and multi-line	`// Single line comment`
	`/*`
	`Multi`
	`line`
	`comment`
	`*/`

Basic Operators

Logical Operators	Arithmetic Operators	Bitwise Operators
&& (AND)	+	& (AND)
|| (OR)	-	| (OR)
! (NOT)	*	^ (XOR)
	/	~ (NOT)
	%	<< (Left shift)
		>> (Right shift)

```cpp
// Logical AND (&&)
    int age = 25;
    bool isAdult = (age >= 18) && (age <= 60);
```

```cpp
// Logical OR (||)
    bool isStudent = true;
    bool discountEligible = isAdult || isStudent;
```

```cpp
// Logical NOT (!)
    bool canEnterClub = !isAdult;
```

Output:

```
Is Adult: 1
Discount Eligible: 1
Can Enter Club: 0
```

```cpp
// Addition
    int sum = 5 + 3;

// Subtraction
    int difference = 10 - 4;

// Multiplication
    int product = 6 * 2;

// Division
    double quotient = 15.0 / 4.0;

 // Modulo
    int remainder = 17 % 3;
```

Output:

```
Sum: 8
Difference: 6
Product: 12
Quotient: 3
Remainder: 2
```

```cpp
// Bitwise AND (&)
    int num1 = 12; // 1100 in binary
    int num2 = 10; // 1010 in binary
    int resultAnd = num1 & num2;
```

```cpp
// Bitwise OR (|)
    int resultOr = num1 | num2;
```

```cpp
// Bitwise XOR (^)
    int resultXor = num1 ^ num2;
```

```cpp
// Bitwise NOT (~)
    int resultNot = ~num1;
```

Output:

```
Bitwise AND: 8
Bitwise OR: 14
Bitwise XOR: 6
Bitwise NOT: -13
```

Control Flow

Switch

```cpp
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    // ...
    default:
        // code if none of the cases match
}
```

If else

```cpp
if (condition) {
    // code if condition is true
} else if (anotherCondition) {
    // code if anotherCondition is true
} else {
    // code if none of the conditions are true
}
```

Loops

+ for loop
+ while loop
+ do-while loop

```
for (int i = 0; i < 5; ++i) {
    // code executed 5 times
}

while (condition) {
    // code executed while the condition is true
}

do {
    // code executed at least once, then repeated while the condition is true
} while (condition);
```

Arrays

```
int numbers[] = {1,2,3,4,5}
```

```
#include <iostream>

int main() {
  // Declaring an array with a fixed size
  const int arraySize = 5;
  int numbers[arraySize] = {1, 2, 3, 4, 5};

  // Accessing and modifying elements
  std::cout << "Elements of the array: ";
  for (int i = 0; i < arraySize; ++i) {
    std::cout << numbers[i] << " ";
  }

  // Changing the value of an element
  numbers[2] = 10;

  // Output after modification
  std::cout << "\nModified array: ";
  for (int i = 0; i < arraySize; ++i) {
    std::cout << numbers[i] << " ";
  }

  return 0;
}
```

- a fixed-size, contiguous collection of elements of the same data type
- Array indices start from 0
- Declare an array using the syntax: `type name[size];`
- Access elements with array[index]
- Arrays have a fixed size specified during declaration

```
Elements of the array: 1 2 3 4 5
Modified array: 1 2 10 4 5
```

Functions

- Blocks of code that perform a specific task
- Enhance code modularity and reusability

```cpp
void greet() {
    cout << "Hello, World!" << endl;
}
```

```cpp
// Declaration
void myFunction(int x);

// Definition
void myFunction(int x) {
    cout << "Value: " << x << endl;
}
```

Declaration & Defintion

- Declare functions with their signature before using them
- Define the function to specify its behavior

Return type & Parameters

- Return type specified before function name
- Parameters are specified inside parentheses

Lambda Functions

- Anonymous functions defined using the lambda syntax
- Useful for short, local operations

```cpp
auto multiply = [](int a, int b) { return a * b;
cout << multiply(2, 3) << endl;
```

```cpp
void print(int x) {
    cout << "Integer: " << x << endl;
}

void print(double y) {
    cout << "Double: " << y << endl;
}
```

Function Overloading

- Define multiple functions with the same name but different parameters
- Compiler selects the appropriate function based on the context

Function Signature

A function signature in C++ consists of the function's return type, name, and parameter types, like this:

```
ReturnType functionName(ParameterType1 param1, ParameterType2 param2, ...);

# Example:
int add(int a, int b);
```

02 Object-Oriented Programming (OOP)

- A coding paradigm that uses objects to organize and structure code.
- OOP in C++ revolves around classes and objects.

Class	Object
• a blueprint for creating objects. • defines data members and member functions.	• an instance of a class. • represents a specific entity in a program.

- **Constructor:** Special member function that initializes objects when created.
- **Destructor:** Special member function that cleans up resources when objects are destroyed.
- **Member variables (attributes):** Data stored within objects.
- **Member functions (methods):** Operations that can be performed on objects.
- **Access specifiers:** Control visibility and accessibility of class members. There are three of them- **public**, **private**, **protected**.
- **Inheritance specifiers:** Control how members are inherited (public, protected, private).
- **Virtual functions**: Allow derived classes to override behavior in base classes.
- **Pure virtual functions:** Make a class abstract (cannot be instantiated directly).
- **Function overloading:** Multiple functions with the same name but different parameter lists.
- **Operator overloading:** Redefining the behavior of operators for user-defined types.

Pillars of OOP

Encapsulation Inheritance Polymorphism Abstraction

Inheritance

- Creates new classes (derived classes) that inherit properties and behaviors from existing classes (base classes).
- Promotes code reusability, extensibility, and hierarchical relationships.

Benefits

- **Code Reusability:** Avoids redundant code by reusing existing functionality.
- **Extensibility:** Easily add new features to existing class without modifying the core code.
- **Modeling Real-World Relationships:** Represents hierarchical relationships between entities effectively.

Key Concepts

- **Base Class (Parent Class):** The class being inherited from.
- **Derived Class (Child Class):** The class inheriting from the base class.
- **Inheritance Specifiers:** Control how members are inherited:
  - **public:** Inherited members retain their original access levels.
  - **protected:** Inherited public members become protected in the derived class.
  - **private:** Inherited members become inaccessible in the derived class.

```
class Animal {
public:
    void eat() {
        std::cout << "Eating...\n";
    }
};

class Dog : public Animal { // Dog inherits from Animal
public:
    void bark() {
        std::cout << "Woof!\n";
    }
};
```

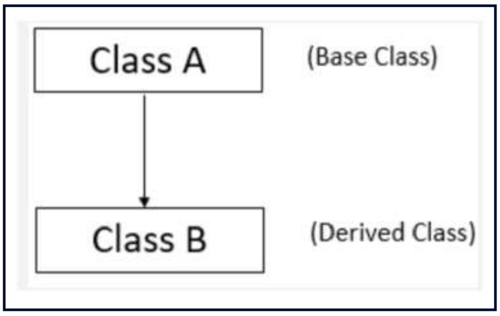
What to Remember?

- Use inheritance judiciously to avoid overly complex class hierarchies.
- Consider composition (has-a relationships) as an alternative when appropriate.
- Thoroughly understand access specifiers and their implications for inheritance.
- Virtual functions play a crucial role in polymorphism, a key OOP concept enabled by inheritance.

Types of Inheritance

Single Inheritance

- A derived class inherits from a single base class.
- Simplest form of inheritance.
- Depicted as a straight line in diagrams.



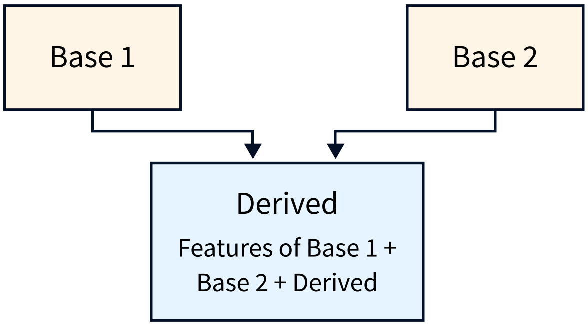
Properties

- A derived class can inherit from only one base class
- In a single inheritance relationship, the derived class directly inherits the attributes and behaviors of the single base class
- The base class is referred to as the parent class, while the derived class is referred to as the child class
- The derived class can access public and protected members of the base class directly
- The derived class can override base class methods and also add new methods and members of its own
- Objects of the derived class can be treated as objects of the base class, enabling polymorphic behavior.

```
class Animal {
public:
    void eat() {
        std::cout << "Eating...\n";
    }
};

class Dog : public Animal { // Dog inherits from Animal
public:
    void bark() {
        std::cout << "Woof!\n";
    }
};
```

Multiple Inheritance

- A derived class inherits from multiple base classes
- Can lead to ambiguity if inherited members have the same name
- Depicted as a diamond shape in diagrams



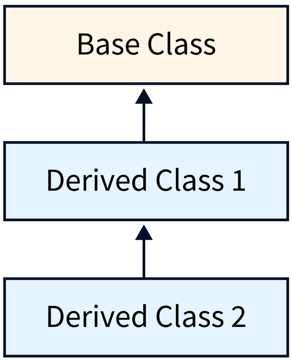
Properties

- Enables a derived class to inherit attributes and behaviors from multiple base classes
- Allows for greater flexibility in object-oriented design by combining functionalities from different classes
- Requires careful management of naming conflicts that may arise from shared member names among base classes
- Can result in increased complexity and potential ambiguity in the codebase
- Provides a mechanism for code reuse and promoting modular design
- Offers the potential for creating more specialized and complex class hierarchies

```cpp
class Shape {
public:
    virtual void draw() = 0;
};

class ColoredShape {
public:
    std::string color;
};

class Circle : public Shape, public ColoredShape { // Circle inherits from Shape and ColoredShape
public:
    void draw() override {
        std::cout << "Drawing a " << color << " circle...\n";
    }
};
```

Multilevel Inheritance

- A derived class inherits from another derived class, creating a chain of inheritance.
- Forms a hierarchical structure.
- Depicted as a vertical line in diagrams.



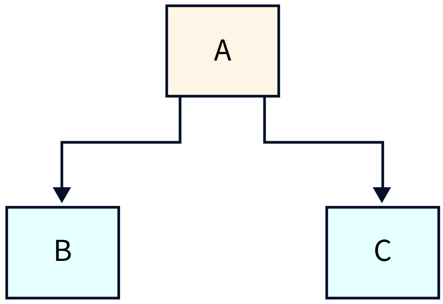
Properties

- Each derived class extends the functionality of its immediate base class
- Forms a hierarchical structure where classes are arranged in a parent-child relationship
- Indirect access to base class members is facilitated through the chain of inheritance
- Represented as a vertical line connecting derived classes to their immediate base classes in diagrams
- Allows for code reuse by enabling derived classes to inherit and extend the functionality of their base classes
- The depth of multilevel inheritance refers to the number of levels in the inheritance hierarchy

```cpp
#include <iostream>

class Animal {
public:
  virtual void makeSound() = 0;
};

class Dog : public Animal {
public:
  void makeSound() override {
    std::cout << "Woof!" << std::endl;
  }
};

class Cat : public Animal {
public:
  void makeSound() override {
    std::cout << "Meow!" << std::endl;
  }
};
```

Hierarchical Inheritance

- Multiple derived classes inherit from a single base class, forming a tree-like structure
- Common way to model real-world classifications
- Depicted as branches from a single base class in diagrams



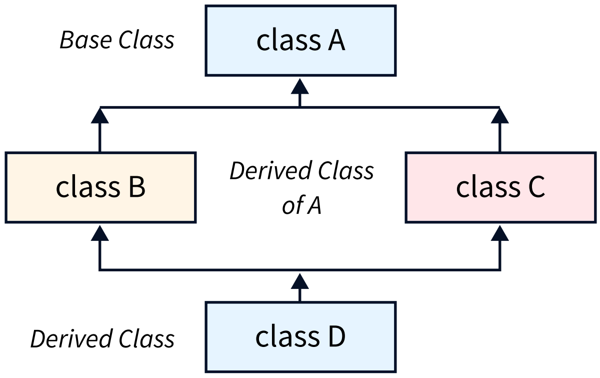
Properties

- Allows derived classes to inherit properties and behaviors from a common base class
- Provides a natural way to organize and classify objects based on shared characteristics
- Enables polymorphic behavior, where objects of different derived classes can be treated uniformly through pointers or references to the base class
- Allows for code reusability, as common attributes and methods can be defined in the base class and inherited by all derived classes
- Supports the "is-a" relationship, where derived classes represent specialized versions of the base class

```cpp
#include <iostream>

class Animal {
public:
  void eat() {
    std::cout << "Eating..." << std::endl;
  }

  void sleep() {
    std::cout << "Sleeping..." << std::endl;
  }
};

class Mammal : public Animal {
public:
  void giveBirth() {
    std::cout << "Giving birth..." << std::endl;
  }
};

class Dog : public Mammal {
public:
  void bark() {
    std::cout << "Woof!" << std::endl;
  }
};
```

Hybrid Inheritance

- Combines multiple inheritance patterns to create more complex relationships
- Can be used to address specific design needs
- Depicted using a combination of inheritance diagram elements



Properties

- Combines multiple inheritance patterns like hierarchical and non-hierarchical inheritance
- Allows creating complex relationships by inheriting from multiple base classes
- Facilitates addressing specific design requirements through a combination of inheritance
- Often depicted using a combination of inheritance diagram elements like arrows and class boxes
- Can lead to the diamond problem if not managed carefully
- Offers flexibility but can increase code complexity if overused
- Enables reusability of code by inheriting features from multiple sources

```
// Combination of multiple inheritance types
class HybridClass: public Base1, public Base2, protected Base3 {
    // ...
};
```

Encapsulation

- Wraps data and functions together within a class, creating a protective capsule.
- Hides internal details and complexities, exposing only a controlled interface.
- Promotes modularity, maintainability, and data security.

Benefits

- **Data Protection:** Prevents accidental or unauthorized access and modification.
- **Modularity:** Creates self-contained, reusable components.
- **Maintainability:** Easier to change implementation details without affecting other parts of the code.
- **Code Readability:** Improves code clarity by hiding implementation details.

```cpp
class Account {
  private:
    double balance; // Private member variable

  public:
    double getBalance() { // Public getter (accessor)
      return balance;
    }

    void deposit(double amount) { // Public mutator (setter)
      balance += amount;
    }
};
```

What to Remember?

- Encapsulation is a fundamental OOP principle that leads to well-structured and secure code.
- Use it thoughtfully to control visibility and protect data integrity.
- Provide clear and concise interfaces through public methods.

Abstraction

- Focuses on essential details and hides complex implementation, simplifying interactions.
- Presents a clear and concise interface to users while concealing inner workings.
- Promotes modularity, maintainability, and reusability.

Benefits

- **Simplicity:** Makes code easier to understand and use by hiding complexity.
- **Modularity:** Promotes code organization and reusability by creating self-contained components.
- **Maintainability:** Easier to modify implementation without affecting users of the interface.
- **Flexibility:** Allows for changes in implementation without affecting external code.

```
class Account {
  private:
    double balance; // Hidden implementation detail

  public:
    double getBalance() { // Part of the public interface
      return balance;
    }

    void deposit(double amount) { // Part of the public
interfaceImplementation details for depositing money
    }
}
```

What to Remember?

- Use access specifiers to carefully control visibility and protect internal implementation.
- Abstraction can be achieved through classes, header files, and access specifiers in C++.
- Consider interface-based programming techniques for enhanced abstraction and modularity.

Polymorphism

- Allows objects to take multiple forms.
- Objects of different classes can be treated as objects of a common base class.
- Allows the same function call to invoke different behaviors depending on the object's actual type.
- Enables flexible and adaptable code.

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

class Circle : public Shape {
public:
  void draw() override {
    std::cout << "Drawing a circle...\n";
  }
};

class Square : public Shape {
public:
  void draw() override {
    std::cout << "Drawing a square...\n";
  }
};
```

Key Concepts

- **Virtual Functions:**
  Makes code easier to understand and use by hiding complexity.
- **Function Overriding (Runtime Polymorphism):**
  The decision of which function to call is made at runtime based on the object's dynamic type.
- **Function Overloading (Compile-Time Polymorphism):**
  Multiple functions with the same name but different parameter lists within the same scope.
- **Operator Overloading:**
  Redefining the behavior of operators for user-defined types.

Benefits

- **Code Flexibility:** Write code that works with objects of different types without knowing their exact types at compile time.
- **Extensibility:** Easily add new classes without modifying existing code that uses polymorphism.
- **Code Reuse:** Promotes code reuse by allowing common functionality to be defined in a base class and specialized in derived classes.

What to Remember?

- Virtual functions are essential for runtime polymorphism.
- Use pure virtual functions to create abstract base classes that cannot be instantiated directly.
- Function overloading provides compile-time polymorphism for different parameter combinations.
- Operator overloading can enhance code readability and expressiveness for user-defined types.
- Polymorphism can lead to complex code structures if overused.
- Use it thoughtfully and consider alternatives like composition when appropriate.

Function Overloading	Function Overriding
• Multiple functions with the same name but different parameter lists within the same scope. • Resolved at compile time (static polymorphism). • Allows for different implementations based on the number and types of arguments passed.	• A derived class redefines a function inherited from its base class. • Requires virtual functions in the base class. • Resolved at runtime (dynamic polymorphism). • Allows derived classes to provide specialized behavior while maintaining a common interface.

Function Overloading:

```
int add(int x, int y) {
    return x + y;
}

double add(double x, double y) {
    return x + y;
}
```

Function Overriding:

```
class Animal {
public:
    virtual void makeSound() {
        std::cout << "Generic animal sound\n";
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Woof!\n";
    }
};
```

- Together, they contribute to the core concept of polymorphism in OOP, allowing code to work with objects of different types without knowing their exact types at compile time.
- Use function overloading for concise code and flexibility in argument handling.
- Use function overriding for specialized behavior in derived classes while maintaining a common interface.

03 Memory Management

Pointer:

- A variable that stores the memory address of another variable
- Allows direct manipulation of memory
- Often used for dynamic memory allocation

Reference:

- An alias or alternative name for an existing variable
- Once initialized, cannot be made to refer to another variable
- A safer alternative to pointers for certain use cases

```
int* ptr;             // Declaration of integer pointer
int num = 10;
ptr = &num;           // Initialization with the address of 'num'
```

```
int value = *ptr;  // Dereferencing pointer to get the value
```

```
int num = 5;
int& ref = num;     // Reference variable 'ref' referring to 'num'
```

Dynamic Memory Allocation

new

- new is an operator used for dynamic memory allocation
- used to create an object or an array on the heap

delete

- delete is an operator used for deallocating memory
- prevents memory leaks

Pointer declaration and initialization

```
int* dynamicNum = new int; //Allocating memory for an integer
*dynamicNum = 20; // Assigning a value
delete dynamicNum; //Deallocating memory
```

Memory Leak

- Occurs when allocated memory is not deallocated

Avoiding Memory Leaks

- Always use delete for each new
- Prefer using smart pointers (std::unique_ptr, std::shared_ptr) for automatic memory management.

Reference Variable

```
std::unique_ptr<int> smartPtr = std::make_unique<int>(10);
//Memory is automatically released when smartPtr goes out of scope
```

04 STL (Standard Template Library)

Container

Vectors

- Dynamic arrays with automatic resizing
- Access elements using [] or at()
- Functions: push_back(), pop_back(), size(), empty()

Lists

- Doubly-linked lists
- Efficient insertion/deletion at any position
- Functions: push_back(), push_front(), pop_back(), pop_front()

Queues

- FIFO data structure
- Functions: push(), pop(), front(), back()

Stacks

- LIFO data structure
- Functions: push(), pop(), top()

Iterators

- Pointers-like objects for iterating containers
- Functions: begin(), end(), advance(), next(), prev()

Code Examples

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);

    // Accessing elements using []
    std::cout << "Using []: " << vec[1] << std::endl;

    // Accessing elements using at()
    std::cout << "Using at(): " << vec.at(2) << std::endl;

    return 0;
}
```

```cpp
#include <list>
#include <iostream>

int main() {
    std::list<int> myList;
    myList.push_back(1);
    myList.push_back(2);
    myList.push_back(3);

    // Iterating through the list
    for (const auto& element : myList) {
        std::cout << element << " ";
    }

    std::cout << std::endl;

    return 0;
}
```

```cpp
#include <queue>
#include <iostream>

int main() {
    std::queue<int> myQueue;
    myQueue.push(1);
    myQueue.push(2);
    myQueue.push(3);

    // Accessing the front element
    std::cout << "Front element: " << myQueue.front() << std::endl;

    // Popping the front element
    myQueue.pop();

    return 0;
}
```

```cpp
#include <stack>
#include <iostream>

int main() {
    std::stack<int> myStack;
    myStack.push(1);
    myStack.push(2);
    myStack.push(3);

    // Accessing the top element
    std::cout << "Top element: " << myStack.top() << std::endl;

    // Popping the top element
    myStack.pop();

    return 0;
}
```

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Using iterators
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " ";
    }

    std::cout << std::endl;

    return 0;
}
```

Algorithms

Sorting

- sort(begin, end): Sorts elements in ascending order
- Custom sorting: provide a comparison function

Searching

- find(begin, end, value): Finds element in range
- binary_search(begin, end, value): Checks if value exists in a sorted range

Algorithms with Containers

- Algorithms work with iterators
- Example: sort(vec.begin(), vec.end())
- Custom operations using function objects

The <algorithm> header in C++ provides a collection of useful functions for performing operations on sequences of elements.

Useful Functions

- **Sorting:**
  - std::sort: Sorts elements in a range.
  - std::stable_sort: Sorts elements in a range, preserving the relative order of equal elements.
- **Searching:**
  - std::find: Searches for an element in a range.
  - std::binary_search: Checks if a value exists in a sorted range.
- **Min and Max:**
  - std::min: Returns the smaller of two values.
  - std::max: Returns the larger of two values.
  - std::min_element: Finds the minimum element in a range.
  - std::max_element: Finds the maximum element in a range.
- **Counting and Accumulating:**
  - std::count: Counts the occurrences of a value in a range.
  - std::accumulate: Computes the sum of a range.
- **Permutations:**
  - std::next_permutation: Generates the next lexicographically greater permutation of a range.
  - std::prev_permutation: Generates the next lexicographically smaller permutation of a range.

- **Transformations:**
  - std::transform: Applies a function to each element in a range and stores the result.
  - std::replace: Replaces all occurrences of a value in a range with another value.
- **Partitioning:**
  - std::partition: Partitions a range into two groups based on a condition.
- **Other Operations:**
  - std::reverse: Reverses the order of elements in a range.
  - std::rotate: Rotates the elements in a range.
  - std::unique: Removes consecutive duplicate elements in a range.

Code Examples

```cpp
void example_operations() {

  // Sorting a vector
  vector<int> vec = {3, 1, 4, 1, 5, 9, 2, 6};
  sort(vec.begin(), vec.end());

  // Output: vec becomes {1, 1, 2, 3, 4, 5, 6, 9}

  // Searching an element
  auto it = find(vec.begin(), vec.end(), 4);

  if (it != vec.end()) {
    cout << "Position of 4 is " << distance(vec.begin(), it) << endl;
  } else {
    cout << "4 not found" << endl;
  }

  // Output: Position of 4 is 4

  // Counting and accumulating elements
  int count_value = count(vec.begin(), vec.end(), 1); // count_value becomes 2
  int sum = accumulate(vec.begin(), vec.end(), 0);    // sum becomes 31

  // Permutations of a vector
  next_permutation(vec.begin(), vec.end());
  prev_permutation(vec.begin(), vec.end());

  // Output:
  // - Initial permutation: 11234569
  // - Next permutation: 1 1 2 3 4 596
  // - Previous permutation: 11234569

  // Transformations on a vector
  vector<int> result;
  transform(vec.begin(), vec.end(), back_inserter(result), [](int x) { return x * 2; });

  // Original vector becomes: {2, 2, 4, 6, 8, 10, 12, 18}

  replace(vec.begin(), vec.end(), 1, 0);

  // Original vector becomes: {0, 0, 2, 3, 4, 5, 6, 9}

  // Partitioning a vector
  auto partition_point = partition(vec.begin(), vec.end(), [](int x) { return x < 2; });

  // partition_point points to: {2, 4, 6, 1, 1, 5, 3, 9}

  // Other operations on vectors
  reverse(vec.begin(), vec.end());

  // Original vector becomes: {5, 4, 4, 4, 3, 2, 2, 1}

  rotate(vec.begin(), vec.begin() + 2, vec.end());

  // Vector becomes: {4, 4, 3, 2, 2, 1, 5, 4}

  vec.erase(unique(vec.begin(), vec.end()), vec.end());

  // Vector becomes: {4, 3, 2, 1, 5, 4}
}
```

Strings

Strings are typically represented as sequences of characters stored in contiguous memory locations.

The standard C++ library provides the std::string class, which abstracts the complexity of managing memory and provides a convenient interface for string manipulation.

```
#include <iostream>
#include <string>

int main() {
    // Creating strings
    std::string str1 = "Hello, ";
    std::string str2 = "C++ Strings!";

    // Concatenation
    std::string result = str1 + str2;

    // Accessing individual characters
    char thirdChar = result[2];

    // Length of the string
    std::size_t length = result.length();

    // Output
    std::cout << "Concatenated String: " << result << std::endl;
    std::cout << "Third Character: " << thirdChar << std::endl;
    std::cout << "Length of String: " << length << std::endl;

    return 0;
}
```

Output:

```
Concatenated String: Hello, C++ Strings!
Third Character: l
Length of String: 23
```

Useful Functions

- **C++ String Class**
  - std::string class for string manipulation
  - Functions: length(), substr(), find()
  - Concatenation using + or append()
- **String Manipulation**
  - Convert to uppercase/lowercase: toupper(), tolower()
  - String comparison: compare()
  - Conversion to/from other data types: stoi(), to_string()

Math Functions

- Use **<cmath>** header for mathematical functions
- **Common functions:** sqrt(), pow(), abs(), sin(), cos()

```cpp
#include <cmath>

int main() {
    // sqrt() - Square root
    double num1 = 25.0;
    double result1 = sqrt(num1);
    std::cout << "Square root of " << num1 << " is: " << result1 << std::endl;

    // pow() - Power
    double base = 2.0;
    double exponent = 3.0;
    double result2 = pow(base, exponent);
    std::cout << base << " raised to the power of " << exponent << " is: " << result2 << std::endl;

    // abs() - Absolute value
    int num2 = -10;
    int result3 = abs(num2);
    std::cout << "Absolute value of " << num2 << " is: " << result3 << std::endl;

    // sin() - Sine
    double angle = 30.0; // in degrees
    double result4 = sin(angle * M_PI / 180.0); // convert degrees to radians
    std::cout << "Sine of " << angle << " degrees is: " << result4 << std::endl;

    // cos() - Cosine
    double result5 = cos(angle * M_PI / 180.0); // convert degrees to radians
    std::cout << "Cosine of " << angle << " degrees is: " << result5 << std::endl;

    return 0;
}
```

Output:

```
Square root of 25 is: 5
2 raised to the power of 3 is: 8
Absolute value of -10 is: 10
Sine of 30 degrees is: 0.5
Cosine of 30 degrees is: 0.866025
```

05 File Handling

Reading from and Writing to Files

- Input and Output File Streams
- ifstream for reading, ofstream for writing
- fstream for both
- Example: ifstream infile("file.txt")

File Modes and Operations

- Modes: ios::in, ios::out, ios::app, ios::binary
- Opening files: open(), closing: close()
- Reading/writing: >>, <<, getline()

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

int main() {
    // Opening a file
    std::ofstream outputFile("example.txt"); // Create or open a file for writing

    if (outputFile.is_open()) {
        // Writing data to the file
        outputFile << "Hello, this is an example file.\n";
        outputFile << "Writing some data to demonstrate file handling in C++.\n";

        // Closing the file
        outputFile.close();
        std::cout << "File closed successfully." << std::endl;
    } else {
        std::cerr << "Error opening the file!" << std::endl;
    }

   return 0;
}
```

06 Error Handling

try-catch block

- Used to enclose code that might throw exceptions

```cpp
try{
    // Code that may throw exceptions
}

Catch (ExceptionType1 ex1){
    //Handle exception type 1
}

Catch (ExceptionType2 ex2){
    //Handle exception type 2
}
// ...
```

throw Statement

- Used to throw an exception explicitly

```
throw SomeException("Error Message")
```

Custom Exception Classes

- Define custom exception classes by inheriting from std::exception or its derived classes

```
class CustomException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Custom exception occurred";
    }
};
```

07 Miscellaneous

Preprocessor Directives

- **#include:** Used to include header files
  Example: #include <iostream>
- **#define:** Used for macro definitions
  Example: #define PI 3.1415
- **#ifdef and #ifndef:** Used for conditional compilation
  Example:

```
#ifdef DEBUG
// Code for debugging
#endif
```

Namespace Declaration

- Used to group entities into a named scope
- **Example:**

```
namespace MyNamespace {
    // Code here
}
```

Using Directive

- Reduces the need for fully qualified names
- **Example:**

```
using namespace std;
```

Type Casting

Static Cast:

- Used for implicit conversions

```
float floatNumber = 3.14;
int intNumber = static_cast<int>(floatNumber);
```

Dynamic Cast:

- Used for safe downcasting in polymorphic classes

```
Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
if (derivedPtr) {
    // Code for Derived class
}
```

Const Cast:

- Used to add or remove const qualifier

```
const int* constPtr = someFunction();
int* nonConstPtr = const_cast<int*>(constPtr);
```

08 Best Practices and Tips

Coding Standards	Performance Tips
Naming Conventions • Use meaningful names for variables, functions, and classes • Follow a consistent naming convention • Example: CamelCase or snake_case • Prefix member variables with "m_" or use a naming convention to distinguish them **Indentation** • Use a consistent and readable indentation style • Example: 4 spaces or tabs • Maintain a clean and organized code structure with proper alignment	**Avoid Unnecessary Copies** • Use references and const references to avoid unnecessary object copies **Prefer Stack Allocation** • Allocate memory on the stack for small, short-lived objects to improve performance **Optimize Loops** • Minimize loop overhead by precomputing loop conditions or using iterators effectively **Smart Memory Management** • Utilize smart pointers • Example: std::unique_ptr, std::shared_ptr **Profile Code** • Identify bottlenecks using profiling tools before optimizing code

09 Useful C++ Libraries and Frameworks

STL (Standard Template Library)

- Powerful collection of template classes and functions
- Common Components:
  Vectors, lists, queues, stacks, maps, algorithms

Boost C++ Libraries

- Set of high-quality libraries to extend C++ functionality
- Notable Libraries:
  + Boost.Asio for asynchronous programming
  + Boost.Serialization for object serialization

Eigen

- Template library for linear algebra
- Features:
  + Efficient matrix and vector operations
  + Supports dense and sparse matrices

OpenCV (Open Source Computer Vision Library)

- Widely used for computer vision and image processing tasks
- Key Features:
  + Image and video processing
  + Object detection and recognition

QT

- Cross-platform C++ framework for GUI development
- Widgets for desktop and mobile applications
- Signal and slot mechanism for event handling