C++ CheatSheet

1. Introduction to C++

C++ is a powerful, high-performance programming language used for system software, game development, and more. Learn its origins, features, and how it differs from C.

- What is C++?
- History and Features
- Difference Between C and C++
- Applications of C++
- Compiling and Running C++ Code

2. Basic Syntax

Understand the foundational structure of a C++ program.

- Structure of a C++ Program
- Header Files
- main() Function
- Comments
- Input/Output (cin, cout)
- using namespace std

3. Data Types and Variables

Explore how data is stored and manipulated in C++.

- Primitive Data Types (int, float, char, etc.)
- Type Modifiers (short, long, signed, unsigned)
- Variable Declaration and Initialization
- Constants (const, #define)
- Type Casting

4. Operators

Master arithmetic, logical, and bitwise operations.

- Arithmetic Operators (+, -, *, /, %)
- Relational Operators (==, >, <)
- Logical Operators (&&, ||, !)
- Assignment Operators (=, +=, -=)
- Bitwise Operators (&, |, <<, >>)
- Increment/Decrement (++, --)
- Ternary Operator
- Operator Precedence

5. Control Statements

Control program flow with conditional logic.

- if, else if, else
- switch Statement
- Nested Conditions

6. Loops

Repeat tasks efficiently using loops.

- for, while, do...while
- Loop Control (break, continue, goto)

7. Functions

Organize code into reusable blocks.

- Defining and Calling Functions
- Function Parameters and Return Types
- Default Arguments
- Function Overloading
- Inline Functions
- Recursion

8. Arrays and Strings

Work with collections of data.

- 1D and 2D Arrays
- Array Initialization and Traversal
- C-style Strings (char[])
- String Functions (strlen, strcpy, strcmp)
- string Class (from <string>)

9. Pointers

Manage memory addresses for advanced programming.

- Pointer Basics
- Pointer Arithmetic
- Pointers and Arrays
- Pointers to Functions
- Pointers to Structures
- nullptr and void*

10. References

Simplify variable aliases.

- Reference Variables
- Use in Functions
- Difference Between Pointers and References

11. Object-Oriented Programming (OOP)

Build modular code with classes and objects.

- Classes and Objects
- Access Specifiers (public, private, protected)
- Constructors and Destructors
- this Pointer
- static Members
- Friend Functions and Classes

12. Inheritance

Reuse and extend existing code.

- Types of Inheritance (Single, Multiple, Multilevel, etc.)
- Base and Derived Classes
- Constructor Order in Inheritance
- protected Inheritance

13. Polymorphism

Enable flexible code behavior.

- Function Overloading
- Operator Overloading
- Virtual Functions
- Pure Virtual Functions and Abstract Classes
- Runtime vs Compile-time Polymorphism

14. Encapsulation and Abstraction

Hide complexity and expose interfaces.

- Data Hiding
- Interface Implementation via Abstract Classes

15. File Handling

Read and write data to files.

- File Streams (ifstream, ofstream, fstream)
- Opening, Reading, Writing, Closing Files
- File Modes
- Error Handling in File I/O

16. Exception Handling

Gracefully manage runtime errors.

- try, catch, throw
- Custom Exception Classes
- Nested Try Blocks
- Exception Handling with Inheritance

17. Templates

Write generic, reusable code.

- Function Templates
- Class Templates
- Template Specialization

18. STL (Standard Template Library)

Leverage built-in data structures and algorithms.

- Containers: vector, list, deque, set, map, stack, queue
- Iterators
- Algorithms: sort, find, count, binary_search
- Function Objects (Functors)

19. Memory Management

Optimize dynamic memory usage.

- new and delete Operators
- Dynamic Memory Allocation
- Memory Leaks and Smart Pointers (unique_ptr, shared_ptr)

20. Preprocessor Directives

Control compilation with macros.

- #include, #define, #ifdef, #ifndef, #pragma

21. Namespace

Avoid naming conflicts.

- Declaring and Using Namespaces
- std:: Scope Resolution

22. Type Conversions

Safely convert between data types.

- Implicit and Explicit Casting
- static_cast, dynamic_cast, const_cast, reinterpret_cast

23. Lambda Expressions (C++11+)

Write concise inline functions.

- Syntax and Use Cases
- Capturing Variables

24. Multithreading (C++11+)

Build concurrent applications.

- Thread Creation using <thread>
- join() and detach()
- Mutexes and Locks
- Condition Variables

25. Modern C++ Features

Upgrade to C++11/17/20 standards.

- auto Keyword
- Range-Based For Loops
- Smart Pointers
- Move Semantics
- nullptr, enum class, constexpr
- Structured Bindings (C++17)

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

1. INTRODUCTION TO C++

1.1 What is C++?

- C++ is a general-purpose programming language developed as an extension of the C programming language. It supports both procedural and object-oriented programming, which means you can write both structured code and modular, reusable classes and objects.
- C++ is compiled and statically typed, which makes it fast and efficient. It is widely used for system software, game development, drivers, client-server applications, and embedded firmware.

1.2 History and Features

- C++ was developed by Bjarne Stroustrup in 1983 at Bell Labs. It started as "C with Classes" and later evolved into C++.

Key features of C++:

- Object-Oriented Programming (OOP)
- Strong type checking
- Function overloading and operator overloading
- Templates (Generic Programming)
- Standard Template Library (STL)
- Exception handling
- Low-level memory manipulation

1.3 Difference Between C and C++

Feature	C	C++
Programming Paradigm	Procedural	Procedural + Object-Oriented
Data Security	Less secure (no encapsulation)	More secure (supports encapsulation)
Code Reusability	No classes or objects	Supports inheritance and polymorphism
Standard Libraries	Limited	Rich STL for data structures and algorithms
Function Overloading	Not supported	Supported

1. INTRODUCTION TO C++

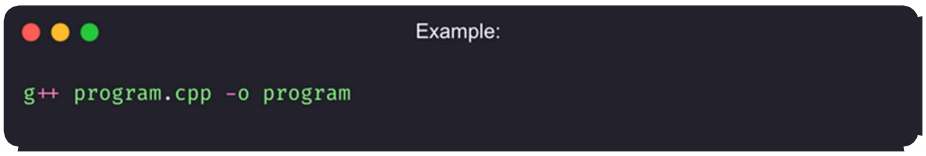
1.4 Applications of C++

- Game development (e.g., using Unreal Engine)
- Desktop applications
- Operating systems (Windows components)
- Embedded systems
- High-performance applications (finance, robotics)
- Compilers and interpreters

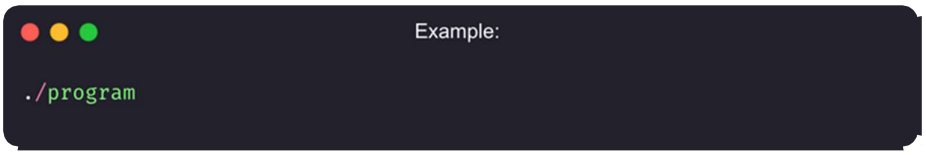
1.5 Compiling and Running C++ Code

To write and run a C++ program:

- Write the code in a file with .cpp extension.
- Use a compiler like g++ (GNU Compiler) to compile:



- Run the output file:



Popular IDEs for C++:

- Code::Blocks
- Dev C++
- Visual Studio
- Eclipse CDT

2. BASIC SYNTAX

2.1 Structure of a C++ Program

- A basic C++ program includes:

Example:

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

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

2.2 Header Files

Header files include pre-written code:

- #include <iostream>: For input/output operations
- Custom headers can be created using .h files

2.3 main() Function

- This is the entry point of every C++ program. The execution starts from here. It must return an int value.

Example:

```cpp
int main() {
    // code
    return 0;
}
```

2.4 Comments

- Used for code readability and documentation.
- Single-line:

Example:

```cpp
// This is a comment
```

- Multi-line:

Example:

```cpp
/* This is
   a multi-line comment */
```

2. BASIC SYNTAX

2.5 Input/Output (cin, cout)

- cout: Used for output

Example:

```
cout << "Hello";
```

- cin: Used for input

Example:

```
int age;
cin >> age;
```

2.6 using namespace std

- This avoids prefixing std:: before cout, cin, etc.

Example:

```
using namespace std;
```

- Without this, you'd need to write std::cout, std::cin.

3. DATA TYPES AND VARIABLES

3.1 Primitive Data Types

Type	Description	Size
int	Integer	4 bytes
float	Decimal	4 bytes
double	Large decimal	8 bytes
char	Single character	1 byte
bool	Boolean	1 byte

3.2 Type Modifiers

- Modify data range and storage:
- short, long, signed, unsigned

Example:

```
unsigned int age = 25;
long double salary = 50000.99;
```

3.3 Variable Declaration and Initialization

- Declaring:

Example:

```
int a;
```

- Initializing:

Example:

```
int a = 5;
```

- Multiple declarations:

Example:

```
int x = 10, y = 20;
```

3. DATA TYPES AND VARIABLES

3.4 Constants

- const: Value cannot be changed

Example:

```
const float PI = 3.14;
```

- #define: Preprocessor directive

Example:

```
#define MAX 100
```

3.5 Type Casting

- Changing one data type to another:

Example:

```
int x = 10;
float y = (float)x;
```

- Or using C++ style:

Example:

```
float y = static_cast<float>(x);
```

4. OPERATORS

4.1 Arithmetic Operators

- Used for calculations:
- +, -, *, /, %

Example:

```
int a = 10, b = 3;
int sum = a + b;
```

4.2 Relational Operators

- Used to compare values:
- ==, !=, >, <, >=, <=

4.3 Logical Operators

- Used to combine conditions:
- && (AND), || (OR), ! (NOT)

4.4 Assignment Operators

- Used to assign values:
- =, +=, -=, *=, /=, %=

Example:

```
x += 5; // same as x = x + 5
```

4.5 Bitwise Operators

- Operate on bits:
- & (AND), | (OR), ^ (XOR), ~ (NOT), <<, >>

Example:

```
int a = 5 << 1; // left shift
```

4.6 Increment/Decrement

- ++, --

Example:

```
x++; // post-increment
++x; // pre-increment
```

4. OPERATORS

4.7 Ternary Operator

- Short form of if-else:

Example:

```
int max = (a > b) ? a : b;
```

4.8 Operator Precedence

- Defines the order of evaluation.

Example:

```
int result = 10 + 5 * 2; // Output: 20
```

- Multiplication has higher precedence than addition.

5. CONTROL STATEMENTS

5.1 if, else if, else

- Used to make decisions.

Example:

```
int age = 20;
if (age < 18) {
    cout << "Minor";
} else if (age == 18) {
    cout << "Just an adult";
} else {
    cout << "Adult";
}
```

5.2 switch Statement

- Used for multiple selections.

Example:

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

5.3 Nested Conditions

- You can place if or switch inside another.

Example:

```
if (age > 18) {
    if (age > 60) {
        cout << "Senior";
    } else {
        cout << "Adult";
    }
}
```

6. LOOPS

6.1 for Loop

- Used when the number of iterations is known.

Example:

```
for (int i = 0; i < 5; i++) {
    cout << i << endl;
}
```

- Structure:

Example:

```
for(initialization; condition; increment/decrement) {
    // code
}
```

6.2 while Loop

- Used when the number of iterations is not known.

Example:

```
int i = 0;
while (i < 5) {
    cout << i << endl;
    i++;
}
```

6.3 do...while Loop

- Executes at least once, even if the condition is false.

Example:

```
int i = 0;
do {
    cout << i << endl;
    i++;
} while (i < 5);
```

6.4 Loop Control Statements

- break
- Exits the loop immediately.

Example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    cout << i << endl;
}
```

6. LOOPS

6.4 Loop Control Statements

- continue
- Skips the current iteration.

Example:

```
for (int i = 0; i < 5; i++) {
    if (i == 2) continue;
    cout << i << endl;
}
```

- goto (Not Recommended)
- Transfers control to a labeled statement.

Example:

```
int x = 1;
label:
cout << x << endl;
x++;
if (x <= 5) goto label;
```

7. FUNCTIONS

7.1 Defining and Calling Functions

- Definition:

Example:

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

- Calling:

Example:

```
int result = add(5, 3);
```

7.2 Function Parameters and Return Types

- Parameters can be passed by value or reference.
- Functions must declare a return type (int, void, etc.).

7.3 Default Arguments

- Allows default values for parameters.

Example:

```
void greet(string name = "User") {
    cout << "Hello, " << name;
}
```

7.4 Function Overloading

- Same function name with different parameter types or numbers.

Example:

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

7.5 Inline Functions

- Used to reduce function call overhead by copying code at compile-time.

Example:

```
inline int square(int x) {
    return x * x;
}
```

7. FUNCTIONS

7.6 Recursion

- Function calling itself.

Example:

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

8. ARRAYS AND STRINGS

8.1 1D and 2D Arrays

- 1D Array

Example:

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

- 2D Array

Example:

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

8.2 Array Initialization and Traversal

- Initialization:

Example:

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

- Traversal:

Example:

```
for (int i = 0; i < 3; i++) {
    cout << arr[i] << " ";
}
```

8.3 C-style Strings (char[])

- Null-terminated character arrays.

Example:

```
char name[] = "John";
```

8.4 String Functions

strlen

- Returns the length of the string.

Example:

```
strlen(name);
```

8. ARRAYS AND STRINGS

8.4 String Functions

strcpy

- Copies one string to another.

Example:

```
char dest[20];
strcpy(dest, name);
```

strcmp

- Compares two strings.

Example:

```
strcmp(str1, str2);
```

8.5 string Class (from <string>)

- Modern way to handle strings.

Example:

```
#include <string>
string s = "Hello";
s.length();
s.append(" World");
```

9. POINTERS

9.1 Pointer Basics

- A pointer stores the memory address of a variable.

Example:

```
int x = 10;
int* ptr = &x;
```

9.2 Pointer Arithmetic

- Used to traverse arrays.

Example:

```
int arr[] = {1, 2, 3};
int* p = arr;
p++; // points to arr[1]
```

9.3 Pointers and Arrays

- Pointers can access array elements.

Example:

```
int arr[3] = {10, 20, 30};
int* p = arr;
cout << *(p + 1); // 20
```

9.4 Pointers to Functions

- Pointer storing function address.

Example:

```
void greet() {
    cout << "Hello";
}

void (*ptr)() = greet;
ptr();
```

9.5 Pointers to Structures

Example:

```
struct Person {
    int age;
};

Person p = {25};
Person* ptr = &p;
cout << ptr->age;
```

9. POINTERS

9.6 nullptr and void*

nullptr

- A null pointer value (C++11+).

Example:

```
int* p = nullptr;
```

void*

- Generic pointer; can point to any data type.

Example:

```
void* ptr;
int x = 10;
ptr = &x;
```

10. REFERENCES

10.1 Reference Variables

- Alias for another variable.

Example:

```
int a = 5;
int& ref = a;
ref = 10; // a becomes 10
```

10.2 Use in Functions

- Used for efficient parameter passing and modifying values.

Example:

```
void update(int& x) {
    x = 100;
}
```

10.3 Difference Between Pointer and Reference

Feature	Pointer	Reference
Syntax	int* p = &x;	int& r = x;
Null allowed	Yes	No
Reassignment	Can be reassigned	Cannot be reassigned
Use in arrays	Common	Not commonly used

11. OBJECT-ORIENTED PROGRAMMING (OOP)

11.1 Classes and Objects

- A class is a blueprint for objects. An object is an instance of a class.

Example:

```
class Car {
public:
    string brand;
    void start() {
        cout << "Car started";
    }
};

int main() {
    Car myCar;
    myCar.brand = "Toyota";
    myCar.start();
}
```

11.2 Access Specifiers

Control access to class members.

- public: Accessible from anywhere
- private: Accessible only within the class
- protected: Accessible in the class and its derived classes

Example:

```
class Demo {
private:
    int a; // Not accessible outside
public:
    int b; // Accessible
};
```

11.3 Constructors and Destructors

- Constructor: Initializes objects, same name as class.
- Destructor: Cleans up when object is destroyed.

Example:

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

11. OBJECT-ORIENTED PROGRAMMING (OOP)

11.4 this Pointer

- Refers to the current object.

Example:

```
class Demo {
    int x;
public:
    void set(int x) {
        this→x = x;
    }
};
```

11.5 static Members

- Shared by all objects of a class.

Example:

```
class Demo {
public:
    static int count;
};

int Demo::count = 0;
```

11.6 Friend Function and Friend Class

- Allows access to private members from outside.

Example:

```
class A {
    friend void show(A);
private:
    int val = 10;
};

void show(A obj) {
    cout << obj.val;
}
```

12. INHERITANCE

12.1 Types of Inheritance

- Single: One base → one derived
- Multilevel: A → B → C
- Multiple: Multiple bases → one derived
- Hierarchical: One base → many derived
- Hybrid: Combination of above

12.2 Base and Derived Class

Example:

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

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

12.3 Constructor Order in Inheritance

- Base class constructor runs before derived class constructor.

Example:

```
class A {
public:
    A() { cout << "A constructor"; }
};

class B : public A {
public:
    B() { cout << "B constructor"; }
};
```

12.4 protected Inheritance

- In protected inheritance, public and protected members become protected in the derived class.

Example:

```
class A {
protected:
    int x;
};

class B : protected A {
    void set() {
        x = 10; // valid
    }
};
```

13. POLYMORPHISM

13.1 Function Overloading

- Same name, different parameters.

Example:

```
int sum(int a, int b);
float sum(float a, float b);
```

13.2 Operator Overloading

- Redefining operators for user-defined types.

Example:

```
class Point {
public:
    int x;
    Point operator+(Point obj) {
        Point temp;
        temp.x = x + obj.x;
        return temp;
    }
};
```

13.3 Virtual Functions

- Used for runtime polymorphism.

Example:

```
class Base {
public:
    virtual void show() { cout << "Base"; }
};

class Derived : public Base {
public:
    void show() override { cout << "Derived"; }
};
```

13.4 Pure Virtual Functions and Abstract Classes

- An abstract class has at least one pure virtual function.

Example:

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

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

13. POLYMORPHISM

13.5 Runtime vs Compile-time Polymorphism

Type	Example	When Resolved
Compile-time	Function/Operator Overloading	At compile time
Runtime	Virtual functions	At run time

14. ENCAPSULATION AND ABSTRACTION

14.1 Data Hiding

- Achieved by keeping data members private and accessing via public methods.

Example:

```
class BankAccount {
private:
    int balance;
public:
    void setBalance(int b) { balance = b; }
    int getBalance() { return balance; }
};
```

14.2 Interface Implementation via Abstract Classes

- Abstract class acts as an interface (no object can be created from it).

Example:

```
class Interface {
public:
    virtual void method() = 0;
};
```

15. FILE HANDLING

15.1 File Streams

Used to handle files.

- ifstream: Read from files
- ofstream: Write to files
- fstream: Both read/write

15.2 Opening, Reading, Writing, Closing Files

- Write to File

Example:

```cpp
#include <fstream>
ofstream file("data.txt");
file << "Hello World";
file.close();
```

- Read from File

Example:

```cpp
#include <fstream>
string line;
ifstream file("data.txt");
while (getline(file, line)) {
    cout << line << endl;
}
file.close();
```

15.3 File Modes

Example:

```cpp
fstream file("data.txt", ios::out | ios::app);
```

15. FILE HANDLING

15.4 Error Handling in File I/O

Example:

```
ifstream file("missing.txt");
if (!file) {
    cout << "File not found!";
}
```

16. EXCEPTION HANDLING

16.1 try, catch, throw

- Used to handle run-time errors and prevent program crashes.

Example:

```
try {
    int x = 0;
    if (x == 0) throw "Divide by zero";
}
catch (const char* msg) {
    cout << "Error: " << msg;
}
```

- try: Block where exceptions might occur.
- throw: Used to throw an exception.
- catch: Catches and handles the exception.

16.2 Custom Exception Classes

- You can define your own exception types using classes.

Example:

```
class MyException : public exception {
public:
    const char* what() const noexcept override {
        return "Custom Exception Occurred";
    }
};

try {
    throw MyException();
} catch (MyException &e) {
    cout << e.what();
}
```

16.3 Nested Try Blocks

- Try blocks can be nested to handle exceptions at different levels.

Example:

```
try {
    try {
        throw 100;
    } catch (int x) {
        cout << "Inner Catch: " << x << endl;
        throw; // rethrow
    }
} catch ( ... ) {
    cout << "Outer Catch";
}
```

16. EXCEPTION HANDLING

16.4 Exception Handling with Inheritance

- Derived exception classes are caught by base-class handlers only if no specific handler is found.

Example:

```
class BaseEx {};
class DerivedEx : public BaseEx {};

try {
    throw DerivedEx();
} catch (BaseEx&) {
    cout << "Caught Base";
}
```

17. TEMPLATES

17.1 Function Templates

- Allow functions to work with different data types using a single template.

Example:

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

17.2 Class Templates

- Create classes that can handle any data type.

Example:

```
template <class T>
class Box {
    T data;
public:
    void set(T val) { data = val; }
    T get() { return data; }
};
```

17.3 Template Specialization

- Define a special version of a template for specific data types.

Example:

```
template<>
class Box<string> {
public:
    void set(string val) {
        cout << "String: " << val;
    }
};
```

18. STL (STANDARD TEMPLATE LIBRARY)

18.1 Containers

- Vector: Dynamic array
- List: Doubly linked list
- Deque: Double-ended queue
- Set: Sorted unique elements
- Map: Key-value pairs
- Stack: LIFO structure
- Queue: FIFO structure

Example with vector:

```
#include <vector>
vector<int> v = {1, 2, 3};
v.push_back(4);
```

18.2 Iterators

- Used to traverse containers.

Example:

```
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << *it;
```

18.3 Algorithms

- STL provides built-in algorithms:
- sort(), find(), count(), binary_search()

Example:

```
sort(v.begin(), v.end());
```

18.4 Function Objects (Functors)

- Objects that act like functions.

Example:

```
struct Square {
    int operator()(int x) {
        return x * x;
    }
};

Square sq;
cout << sq(5); // Output: 25
```

19. MEMORY MANAGEMENT

19.1 new and delete Operators

- Used for dynamic memory allocation and deallocation.

Example:

```
int* p = new int;  // Allocation
*p = 10;
delete p;          // Deallocation
```

19.2 Dynamic Memory Allocation

- Allocating memory for arrays:

Example:

```
int* arr = new int[5];
delete[] arr;
```

19.3 Memory Leaks and Smart Pointers

- Memory leaks happen if dynamically allocated memory is not deallocated.

Smart Pointers:

- unique_ptr: Single ownership
- shared_ptr: Multiple ownership

Example:

```
#include <memory>
unique_ptr<int> p1(new int(10));
shared_ptr<int> p2 = make_shared<int>(20);
```

20. PREPROCESSOR DIRECTIVES

20.1 #include

- Used to include header files.

Example:

```
#include <iostream>
```

20.2 #define

- Used to define constants or macros.

Example:

```
#define PI 3.1416
```

20.3 #ifdef, #ifndef

- Conditional compilation:

Example:

```
#define DEBUG
#ifdef DEBUG
    cout << "Debug mode";
#endif
```

20.4 #pragma

- Used to control compiler-specific features.

Example:

```
#pragma once // Ensures file is included only once
```

21. NAMESPACE

21.1 Declaring and Using Namespaces

- Namespaces help avoid name conflicts by grouping identifiers.

Example:

```cpp
namespace MySpace {
    int val = 100;
    void display() {
        cout << "Inside MySpace";
    }
}
```

- Usage:

Example:

```cpp
MySpace::display();
cout << MySpace::val;
```

21.2 std:: Scope Resolution

- The C++ Standard Library is under the std namespace.

Example:

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

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

- Or, without using namespace:

Example:

```cpp
std::cout << "Hello";
```

22. TYPE CONVERSIONS

22.1 Implicit and Explicit Casting

Implicit:

Done by compiler automatically.

Example:

```
int a = 5;
double b = a; // Implicit
```

Explicit (C-style):

Example:

```
double d = 9.8;
int x = (int)d; // Explicit
```

22.2 static_cast, dynamic_cast, const_cast, reinterpret_cast

Example:

```
float f = 3.5;
int a = static_cast<int>(f);

Base* b = new Derived();
Derived* d = dynamic_cast<Derived*>(b);
```

23. LAMBDA EXPRESSIONS (C++11+)

23.1 Syntax and Use Cases

- A lambda is an anonymous function.

Example:

```
auto greet = []() {
    cout << "Hello";
};
greet(); // Output: Hello
```

- With parameters:

Example:

```
auto sum = [](int a, int b) → int {
    return a + b;
};
```

23.2 Capturing Variables

Example:

```
int x = 5;
auto print = [x]() {
    cout << x; // capture by value
};

int y = 10;
auto print2 = [&y]() {
    y++;
    cout << y; // capture by reference
};
```

24. MULTITHREADING (C++11+)

- Requires <thread> header.

24.1 Thread Creation using <thread>

Example:

```cpp
#include <thread>

void printMsg() {
    cout << "Running in thread";
}

int main() {
    thread t1(printMsg);
    t1.join(); // Wait for thread to finish
}
```

24.2 join() and detach()

- join(): Main thread waits for this thread.
- detach(): Thread runs independently.

Example:

```cpp
t1.join();   // Blocking
t1.detach(); // Non-blocking
```

24.3 Mutexes and Locks

- Used to avoid race conditions.

Example:

```cpp
#include <mutex>
mutex m;

void safeFunction() {
    m.lock();
    // critical section
    m.unlock();
}
```

- Better way:

Example:

```cpp
lock_guard<mutex> lock(m);
```

24. MULTITHREADING (C++11+)

24.4 Condition Variables

- Used for thread communication.

Example:

```
condition_variable cv;
mutex m;
bool ready = false;

cv.wait(lock, [](){ return ready; });
cv.notify_one(); // or cv.notify_all();
```

25. MODERN C++ FEATURES

25.1 Auto Keyword

- Type is inferred automatically.

Example:

```
auto x = 5;       // int
auto y = 3.14;    // double
```

25.2 Range-Based For Loops

Example:

```
vector<int> v = {1, 2, 3};
for (auto val : v) {
    cout << val;
}
```

25.3 Smart Pointers

- Smart pointers automatically manage memory.

Example:

```
unique_ptr<int> p1 = make_unique<int>(10);
shared_ptr<int> p2 = make_shared<int>(20);
```

25.4 Move Semantics

- Transfers ownership instead of copying.

Example:

```
string a = "Hello";
string b = move(a); // a becomes empty
```

25.5 nullptr, enum class, constexpr

- nullptr: Safer than NULL
- enum class: Strongly typed enums
- constexpr: Compile-time constants

Example:

```
constexpr int size = 10;
enum class Color { RED, BLUE };
int* ptr = nullptr;
```

25.6 Structured Bindings (C++17)

- Break a tuple or pair into variables.

Example:

```
tuple<int, string> t = {1, "Hello"};
auto [id, name] = t;
```