Lecture: Strings in C++

1. Introduction

- Today, we'll explore one of the most essential and widely used data types in programming — strings. In C++, strings allow us to work with sequences of characters, such as names, messages, or entire documents.
- While C-style character arrays are available, C++ provides a much more powerful and user-friendly tool: the `std::string` class from the Standard Template Library (STL).

1. Introduction

By the end of this lecture, you will:

- Understand what `std::string` is and how it differs from C-style strings.
- Know how to declare, initialize, and manipulate strings.
- Be familiar with common string operations and member functions.
- Learn about string input/output and best practices.

Let’s get started!

2. What Is a String?

In C++, a string is a sequence of characters stored as an object of the std::string class, defined in the `<string>` header.

- Unlike C-style strings (null-terminated character arrays like `char str[] = "Hello";`), `std::string` is a class, which means:
- It manages memory automatically.
- It supports dynamic resizing.
- It comes with many built-in methods for manipulation.
- It prevents common errors like buffer overflows (if used correctly).

2. What Is a String?

- To use `std::string`, include the header:

```
#include <string>
```

- And don't forget to use the standard namespace (or prefix with `std::`):

```
using namespace std;
// OR use std::string  explicitly
```

3. Declaring and Initializing Strings

```
// 1. Empty string
string s1;

// 2. Initialize with a string literal
string s2 = "Hello";

// 3. Direct initialization
string s3("World");
```

3. Declaring and Initializing Strings

- ```
  // 4. Copy initialization
  string s4(s2); // or auto s4 = s2;

  // 5. From a part of another string
  string s5(s2, 1, 3); // starts at index 1, takes 3
                       // chars → "ell"

  // 6. Repeating a character
  string s6(5, 'a'); // "aaaaa"
  ```

  ☑ Note: All these are valid and safe. No need to worry about array bounds!
  ```

4. Input and Output of Strings

- **Using `cin` and `cout`**

```
string name;
cout << "Enter your name: ";
cin >> name; // Reads until whitespace
cout << "Hello, " << name << "!" << endl;
```

⚠ Problem: `cin >>` stops reading at the first whitespace (space, tab, newline). So it cannot read full sentences.

Solution: `getline()`

4. Input and Output of Strings

Use getline() to read entire lines, including spaces:

```
string sentence;
cout << "Enter a sentence: ";
getline(cin, sentence);
cout << "You entered: " << sentence << endl;
```

Tip: If using getline() after cin, you may need to clear the input buffer with `cin.ignore()`.

```
int age;
cin >> age;
cin.ignore(); // Skip the newline left in buffer
getline(cin, sentence);
```

5. Common String Operations
5.1 Length and Capacity

The std::string class provides many useful member functions.

```
string s = "C++ Programming";


cout << s.length();    // 15 - number of
                       // characters
cout << s.size();      // 15 - same as length()
cout << s.empty();     // false - checks if
                        //string is empty
```

5.2 Accessing Characters

```
s[0] = 'c';  // Modify first char → "c++
             // Programming"
char c = s.at(1);// Safer access (throws
                 // exception if out of range)
```

⚠ s[i] does not check bounds. Use .at(i) for safety.

5.3 Concatenation

```
string a = "Hello";
string b = "World";

string c = a + " " + b;   // "Hello World"
a += " there!";           // a becomes "Hello
                          // there!"
```

5.4 Substring Extraction

```
string sub = s.substr(4, 5);
// Extract 5 characters starting at index 4
```

5.5 Finding Substrings

```
size_t pos = s.find("Prog"); // Returns position
                            // (index) or
                            // string::npos
if (pos != string::npos) {
    cout << "Found at position: " << pos <<
endl;
} else {
    cout << "Not found" << endl;
}
```

5.5 Finding Substrings

```
s.find(s0,pos) // search starts at position pos
```

Other search functions:

- rfind() – last occurrence
- find_first_of() – any character from a set
- find_last_not_of() – etc.

5.6 Replacing and Erasing

```
s.replace(6, 4, "Fun"); // Replace 4 chars at
                        // pos 6 with "Fun"
s.erase(5, 3);          // Remove 3 characters
                        //starting at index 5
```

6. Comparing Strings

- Use comparison operators (==, !=, <, >, etc.):

```
string s1 = "apple";
string s2 = "banana";

if (s1 < s2) {
    cout << "apple comes before banana alphabetically" << endl;
}
```

Or use .compare() method (returns 0 if equal, negative if less, positive if greater):

```
if (s1.compare(s2) == 0) { /* equal */ }
```

7. Conversions Between Strings and Numbers

Sometimes you need to convert between strings and numeric types.

String to Number

```
string numStr = "123";
int x = stoi(numStr);       // string to int
double d = stod("3.14");    // string to double
long l = stol("1000");
```

- 🛑 Throws invalid_argument or out_of_range on error.
- **Number to String**

```
int x = 42;
string s = to_string(x); // "42"
double pi = 3.14159;
string sp = to_string(pi); // "3.141590" (6 decimal places by
                           // default)
```

8. Iterating Over Strings

You can loop through each character:

```
string text = "Hello";
// Classic for loop
for (int i = 0; i < text.length(); ++i) {
    cout << text[i] << endl;
}
// Range-based for loop (C++11 and later)
for (char c : text) {
    cout << c << endl;
}
// Using iterators
for (auto it = text.begin(); it != text.end(); ++it) {
    cout << *it << endl;
}
```

8. Iterating Over Strings

You can also modify characters:

```
for (char &c : text) { // Note: reference!
    c = toupper(c);
}
```

9. Important Notes and Best Practices

☑ Advantages of std::string:

- Automatic memory management.
- Safe operations (no manual buffer handling).
- Rich API for searching, modifying, comparing.
- Integrates well with other STL components.

9. Important Notes and Best Practices

🔧 Best Practices:

- Always `include <string>`.
- Prefer `getline()` for full-line input.
- Use `.empty()` instead of `length() == 0`.
- Use `const string&` when passing strings to functions to avoid copying:

```
void print(const string& str) {
    cout << str << endl;
}
```

10. Example: Simple String Program

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

int main() {
    string name, city;

    cout << "Enter your name: ";
    getline(cin, name);

    cout << "Enter your city: ";
    getline(cin, city);

    string message = "Hello, " + name + "! Welcome to " + city + ".";

    cout << message << endl;
    cout << "Your message has " << message.length() << " characters." << endl;

    // Find space
    size_t pos = message.find(' ');
    if (pos != string::npos) {
        cout << "First space at index: " << pos << endl;
    }

    return 0;
}
```

11. Summary

- `std::string` is the modern way to handle text in C++.
- It's safer, easier, and more powerful than C-style strings.
- Key operations: concatenation, substring, find, replace, length, iteration.
- Use `getline()` for reading full lines.
- Always validate input and handle exceptions when converting strings to numbers.

12. Exercises

- Write a program that counts the number of vowels in a string.
- Reverse a string without using extra space (modify in place).
- Check if a string is a palindrome.
- Split a sentence into words and store them in a vector.
- Convert a string to title case (first letter of each word uppercase).

Header Files in C++

1. Introduction

We’re going to talk about one of the most important and widely used features in C++: header files.

You’ve probably seen lines like this in your code:

```
#include <iostream>
#include "myclass.h"
```

But what exactly are header files? Why do we need them? And how should we use them correctly?

1. Introduction

By the end of this lecture, you will understand:

- What header files are and why they exist.
- The difference between .h and .cpp files
- How #include works
- Best practices for writing safe and reusable headers.
- Common pitfalls and how to avoid them.

Let’s get started!

2. What Is a Header File?

A header file (usually with extension .h or .hpp) is a file that contains declarations — not definitions — of functions, classes, variables, templates, constants, and type aliases.

Its main purpose is to share interface information between different source files.

Think of a header as a "contract" or "blueprint":

"Here’s what I can do. If you want to use me, include my header."

2. What Is a Header File?

Example: `math_utils.h`

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

// Function declaration
int add(int a, int b);
// Class declaration
class Calculator {
public:
    double multiply(double x, double y);
    double divide(double x, double y);
};
#endif // MATH_UTILS_H
```

2. What Is a Header File?

This tells other files:

"There is a function called add, and a class called Calculator with these methods."

The actual implementation goes into a .cpp file.

3. Declaration vs. Definition

It's crucial to understand the difference:

TERM	MEANING	WHERE IT BELONGS
Declaration	Says something exists (compiler trusts you)	Header file (.h)
Definition	Actually creates the object or implements the function	Source file (.cpp)

3. Declaration vs. Definition

```
extern int x;              // ☑ Declaration
int x = 10;                // ☑ Definition

void foo();                // ☑ Declaration
void foo() { }             // ☑ Definition

class MyClass;             // ☑ Forward declaration
class MyClass { };         // ☑ Definition
```

⚠ You can have many declarations, but only one definition (ODR — One Definition Rule).

4. Why Do We Need Header Files?

Imagine a project with multiple .cpp files:

main.cpp
utils.cpp
parser.cpp
If main.cpp wants to call a function from utils.cpp, how does it know about it?

☑ Answer: By including the corresponding header file.

Without headers, the compiler wouldn’t know what functions are available → compilation errors.

Headers enable modular programming: separate concerns, reuse code, and compile files independently.

5. How `#include` works

- When you write:

```
#include "myheader.h"
```

The preprocessor copies the entire content of `myheader.h` and inserts it directly into your source file before compilation.

So this:

```
#include "math_utils.h"
int main() {
    return add(2, 3);
}
```

Becomes (after preprocessing):

```
// [Content of math_utils.h inserted here]
int add(int a, int b);
int main() {
    return add(2, 3);
}
```

Now the compiler knows about add().

5. How `#include` works

Two styles of `#include`:

SYNTAX	PURPOSE
#include <header>	For standard or system headers (<iostream>, <vector>) — searches in system directories
#include "header.h"	For your own headers — first looks in current directory, then system paths

6. The One Definition Rule (ODR)

C++ has a strict rule:

Every class, template, inline function, and variable must have exactly one definition in the entire program.

But what happens if two .cpp files include the same header?

👉 Without protection, you might get multiple definitions, leading to linker errors.

That's where **include guards** come in.

7. Include Guards (Header Guards)

Include guards prevent a header file from being included more than once in the same translation unit.

```
#ifndef MYCLASS_H
#define MYCLASS_H

// Your declarations here
class MyClass {
    // ...
};

#endif // MYCLASS_H
```

7. Include Guards (Header Guards)

How it works:

First time: `MYCLASS_H` is not defined → define it and include content.

Second time: `MYCLASS_H` is already defined → skip everything until `#endif`.

Modern alternative: `#pragma once`

```
#pragma once

// Declarations...
```

☑ #pragma once is simpler and widely supported (but technically not standard until C++23).

☑ Both work — choose based on team preference or portability needs.

8. What should go into a header file?

☑ Safe to put in headers:

- Function declarations
- Class declarations
- Template definitions (must be in header!)
- inline functions
- Constants (constexpr, const)
- Type aliases (using, typedef)
- #include directives
- Inline variables (C++17)

8. What should go into a header file?

❌ Avoid in headers (unless necessary):

- Function definitions (except inline, templates)
- Global variable definitions
- using namespace std; (pollutes global scope)
- Large amounts of code that slows down compilation

9. Example Project Structure

```
project/
│
├── main.cpp
├── calculator.h
├── calculator.cpp
└── utils.h
```

calculator.h

```
#ifndef CALCULATOR_H
#define CALCULATOR_H
class Calculator {
public:
    int add(int a, int b);
    int subtract(int a, int b);
};
#endif
```

9. Example Project Structure

calculator.cpp

```
#include "calculator.h"
int Calculator::add(int a, int b) {
    return a + b;
}
int Calculator::subtract(int a, int b) {
    return a - b;
}
```

main.cpp

```
#include <iostream>
#include "calculator.h"  // Now we can use Calculator
int main() {
    Calculator calc;
    std::cout << calc.add(5, 3) << std::endl;
    return 0;
}
```

9. Example Project Structure

```
g++ main.cpp calculator.cpp -o program
./program  # Output: 8
```

10. Special Cases: Templates and Inline Functions

Because the compiler needs to generate code for each type used:

```
// vector.h
template<typename T>
class MyVector {
    T data[100];
public:
    void push(T value);  // Implementation often here
};
```

Even if implemented outside class, still in header:

```
template<typename T>
void MyVector<T>::push(T value) {
    // ...
}
```

10. Special Cases: Templates and Inline Functions

- inline Functions in Headers

Allowed and common:

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

Multiple translations units can include it — linker merges them safely.

11. Best Practices

- ☑ Use descriptive names: student.h, network_manager.h
- ☑ Always use include guards or #pragma once
- ☑ Keep headers minimal — only what users need
- ☑ Avoid deep header dependencies
- ☑ Prefer forward declarations when possible:

```
// Instead of #include "BigClass.h"
class BigClass;  // Forward declare if pointer/reference used
void process(const BigClass* obj);
```

- ☑ Group related declarations in one header

12. Common Mistakes

❌ Putting non-inline function definitions in headers:

```
// bad.h
void foo() { } // ❌ Multiple definition error if
               // included twice!
```

❌ Missing include guards: → Compiler errors or duplicate symbols.

❌ Circular includes:

```
// a.h
#include "b.h"
// b.h
#include "a.h"
```

☑ Fix: Use forward declarations instead.

13. Summary

- Header files (.h/.hpp) contain declarations shared across multiple source files.
- They allow modular design and independent compilation.
- Use #include to insert their content into .cpp files.
- Always protect headers with include guards or #pragma once.
- Follow the One Definition Rule.
- Templates and inline functions belong in headers.
- Headers are not for implementation — save that for .cpp files.

Conclusion

- Header files are the glue that holds large C++ projects together. Used correctly, they promote clean architecture, code reuse, and faster builds.
- Now you know not just *how* to use headers, but *why* they exist and *what problems* they solve.
- Keep your interfaces clear, your implementations hidden, and happy coding!