Gorgo Continuing C++

April 11, 2026

Contents

6. Advanced Strings

In *Gorgo Starting C++* you learned `std::string` for storing and manipulating text. `std::string` is versatile, but it owns its data — every copy creates a new allocation, and passing a string to a function that only needs to read it can be needlessly expensive. Sometimes you need pattern matching to validate input or extract structured data from text. Other times you need to convert between strings and numbers efficiently. In this chapter you will learn `std::string_view` for lightweight non-owning references to strings, `std::regex` for pattern matching, and the standard library's string-to-number and number-to-string conversion functions.

`std::string_view`

`std::string_view` (C++17, `#include <string_view>`) is a non-owning, read-only view of a character sequence. It stores a pointer and a length — no allocation, no copy:

```
void string_view(const char* str, size_t len);  // conceptually
```

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

void greet(std::string_view name)
{
    std::cout << "Hola, " << name << "!\n";
}

int main()
{
    std::string s = "Beyonce";
    greet(s);         // no copy — views into s
    greet("Shakira"); // no copy — views the string literal

    return 0;
}
```

```
Hola, Beyonce!
Hola, Shakira!
```

Without `string_view`, you would either pass `const std::string&` (which requires a `std::string` to exist) or `const char*` (which loses the length). `string_view` works with both — it is the best parameter type for functions that only need to read a string.

Common Operations

`string_view` supports most of the read-only operations you know from `std::string`:

```
std::string_view sv = "Lose Yourself";
std::cout << sv.size() << "\n";            // 13
std::cout << sv.substr(5, 8) << "\n";      // Yourself
std::cout << sv.find("Your") << "\n";      // 5
std::cout << sv.starts_with("Lose") << "\n";  // 1 (true)
std::cout << sv[0] << "\n";                // L
```

`substr()` on a `string_view` returns another `string_view` — no allocation. This is much cheaper than `std::string::substr()`, which creates a new string.

You can also narrow a view from either end:

```cpp
std::string_view sv = "  trimmed  ";
sv.remove_prefix(2);  // sv is now "trimmed  "
sv.remove_suffix(2);  // sv is now "trimmed"
```

Lifetime Dangers

Because `string_view` does not own its data, the underlying string must outlive the view:

```cpp
std::string_view dangerous()
{
    std::string s = "temporary";
    return s;  // BUG: s is destroyed, view becomes dangling
}
```

Trap: Never return a `string_view` to a local `std::string`. The string is destroyed when the function returns, and the view becomes a dangling pointer. Return `std::string` from functions that create new strings.

```cpp
// Also dangerous:
std::string_view sv;
{
    std::string temp = "gone soon";
    sv = temp;
}
// sv is dangling here — temp was destroyed
```

The rule is simple: use `string_view` for parameters and local variables when you know the source outlives the view. Store `std::string` when you need ownership.

Regular Expressions

The `<regex>` header provides pattern matching for strings. Regular expressions (regex) let you search for patterns instead of exact strings.

std::regex_match

`std::regex_match` checks whether an **entire** string matches a pattern:

```cpp
bool regex_match(const string& s, const regex& pattern);
bool regex_match(const string& s, smatch& match, const regex& pattern);
```

```cpp
#include <iostream>
#include <regex>
#include <string>

int main()
{
    std::regex email_pattern(R"(\w+@\w+\.\w+)");

    std::string s1 = "fan@correo.com";
    std::string s2 = "not-an-email";

    std::cout << std::regex_match(s1, email_pattern) << "\n";  // 1 (true)
    std::cout << std::regex_match(s2, email_pattern) << "\n";  // 0 (false)
```

```
    return 0;
}
```

```
1
0
```

The `R"(...)"` syntax is a **raw string literal** — backslashes are not escape characters, which makes regex patterns much easier to read.

std::regex_search

`std::regex_search` finds the first match **within** a string (it does not require the whole string to match):

```
bool regex_search(const string& s, smatch& match, const regex& pattern);
```

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

int main()
{
    std::string text = "Released in 2003, it sold 2 million copies by 2005";
    std::regex year_pattern(R"(\d{4})");
    std::smatch match;

    std::string::const_iterator start = text.cbegin();
    while (std::regex_search(start, text.cend(), match, year_pattern)) {
        std::cout << "Found year: " << match[0] << "\n";
        start = match.suffix().first;
    }

    return 0;
}
```

```
Found year: 2003
Found year: 2005
```

std::regex_replace

`std::regex_replace` replaces matches with a new string:

```
string regex_replace(const string& s, const regex& pattern, const string& replacement);
```

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

int main()
{
    std::string text = "Call me at 555-1234 or 555-5678";
    std::regex phone(R"(\d{3}-\d{4})");

    std::string redacted = std::regex_replace(text, phone, "XXX-XXXX");
    std::cout << redacted << "\n";

    return 0;
}
```

```
Call me at XXX-XXXX or XXX-XXXX
```

Capture Groups

Parentheses in a regex create **capture groups** that let you extract parts of a match:

```cpp
#include <iostream>
#include <regex>
#include <string>

int main()
{
    std::string entry = "Nelly Furtado - Say It Right (2006)";
    std::regex pattern(R"((.+) - (.+) \((\d{4})\))");
    std::smatch match;

    if (std::regex_match(entry, match, pattern)) {
        std::cout << "Artist: " << match[1] << "\n";
        std::cout << "Title:  " << match[2] << "\n";
        std::cout << "Year:   " << match[3] << "\n";
    }

    return 0;
}
```

```
Artist: Nelly Furtado
Title:  Say It Right
Year:   2006
```

`match[0]` is the entire match, `match[1]` is the first group, `match[2]` the second, and so on.

Tip: `std::regex` can be slow — it compiles the pattern at run time. If you use the same pattern repeatedly, create the `std::regex` object once and reuse it. For performance-critical code, consider a dedicated regex library.

Common Regex Patterns

Pattern	Matches
`\d`	A digit (0-9)
`\w`	A word character (letter, digit, or underscore)
`\s`	Whitespace
`.`	Any character
`*`	Zero or more of the preceding
`+`	One or more of the preceding
`?`	Zero or one of the preceding
`{n}`	Exactly n of the preceding
`{n,m}`	Between n and m of the preceding
`[abc]`	Any one of a, b, or c
`^`	Start of string
`$`	End of string
`(...)`	Capture group

String Conversions

Converting between strings and numbers is one of the most common operations in programming. C++ provides several approaches, each with different trade-offs.

`std::stoi`, `std::stod`, and Friends

The `<string>` header provides functions to convert strings to numbers:

```
int stoi(const string& str, size_t* pos = nullptr, int base = 10);
long stol(const string& str, size_t* pos = nullptr, int base = 10);
long long stoll(const string& str, size_t* pos = nullptr, int base = 10);
float stof(const string& str, size_t* pos = nullptr);
double stod(const string& str, size_t* pos = nullptr);
```

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

int main()
{
    std::string s = "2003";
    int year = std::stoi(s);
    std::cout << year + 1 << "\n";  // 2004

    double pi = std::stod("3.14159");
    std::cout << pi << "\n";  // 3.14159

    // Parsing hex
    int hex_val = std::stoi("FF", nullptr, 16);
    std::cout << hex_val << "\n";  // 255

    return 0;
}
```

These functions throw `std::invalid_argument` if the string cannot be parsed and `std::out_of_range` if the value is too large for the target type.

`std::to_string`

`std::to_string` converts numbers to strings:

```
string to_string(int value);
string to_string(double value);
// ... and other numeric types
```

```
int bpm = 128;
std::string msg = "Playing at " + std::to_string(bpm) + " BPM";
std::cout << msg << "\n";  // Playing at 128 BPM
```

Tip: For formatted output, `std::format` (Chapter 9 of *Gorgo Starting C++*) is more flexible than `std::to_string`. Use `std::to_string` when you just need a plain number-to-string conversion.

`std::from_chars` and `std::to_chars` (C++17)

For high-performance, locale-independent conversions, C++17 provides `std::from_chars` and `std::to_chars` in `<charconv>`:

```
from_chars_result from_chars(const char* first, const char* last, int& value, int base = 10);
to_chars_result to_chars(char* first, char* last, int value, int base = 10);
```

```
#include <charconv>
#include <iostream>

int main()
{
    // String to number
    const char* str = "42";
    int value = 0;
    auto [ptr, ec] = std::from_chars(str, str + 2, value);
    if (ec == std::errc{}) {
        std::cout << "Parsed: " << value << "\n";
    }

    // Number to string
    char buf[20];
    auto [end, ec2] = std::to_chars(buf, buf + sizeof(buf), 2006);
    if (ec2 == std::errc{}) {
        std::cout << "Formatted: " << std::string_view(buf, end - buf) << "\n";
    }

    return 0;
}
```

```
Parsed: 42
Formatted: 2006
```

`from_chars` and `to_chars` never allocate memory, never throw exceptions, and are not affected by locale settings. They are the fastest standard conversion functions available.

Comparison

Feature	`stoi/to_string`	`from_chars/to_chars`
Locale-dependent	Yes	No
Throws exceptions	Yes	No (uses error codes)
Allocates memory	`to_string` does	Never
Speed	Good	Fastest
Ease of use	Easy	Verbose

For most code, `stoi` and `to_string` are fine. Reach for `from_chars/to_chars` when you need maximum performance or locale independence (e.g., parsing data files or network protocols).

Try It: String Processing

Here is a program that exercises string_view, regex, and conversions. Type it in, compile with `g++ -std=c++23`, and experiment:

```
#include <charconv>
#include <cstring>
#include <iostream>
#include <regex>
#include <string>
```

```cpp
#include <string_view>
#include <vector>

// Uses string_view — no copies
bool starts_with_the(std::string_view s)
{
    return s.starts_with("The");
}

int main()
{
    // string_view
    std::vector<std::string> bands = {
        "The Strokes", "Arcade Fire", "The Killers", "Interpol"
    };

    std::cout << "Bands starting with 'The':\n";
    for (const auto& b : bands) {
        if (starts_with_the(b)) {
            std::cout << "  " << b << "\n";
        }
    }

    // Regex: parse "Artist - Song (Year)" entries
    std::regex entry_re(R"((.+?) - (.+?) \((\d{4})\))");
    std::vector<std::string> entries = {
        "Gorillaz - Feel Good Inc (2005)",
        "Yeah Yeah Yeahs - Maps (2003)",
        "Keane - Somewhere Only We Know (2004)"
    };

    std::cout << "\nParsed entries:\n";
    for (const auto& e : entries) {
        std::smatch m;
        if (std::regex_match(e, m, entry_re)) {
            std::cout << "  " << m[2] << " by " << m[1] << " (" << m[3] << ")\n";
        }
    }

    // from_chars
    const char* nums[] = {"120", "140", "160"};
    std::cout << "\nBPM values:\n";
    for (const char* n : nums) {
        int bpm = 0;
        auto [ptr, ec] = std::from_chars(n, n + std::strlen(n), bpm);
        if (ec == std::errc{}) {
            std::cout << "  " << bpm << " BPM = " << (60'000 / bpm) << " ms\n";
        }
    }

    return 0;
}
```

Try adding more regex patterns — for example, one that extracts hashtags from a string or validates a date

format.

Key Points

- **`std::string_view`** is a lightweight, non-owning reference to a string. It avoids copies and works with both `std::string` and `const char*`.
- `string_view::substr()` returns another view (no allocation), unlike `string::substr()`.
- Never return a `string_view` to a local `std::string` — the view becomes dangling.
- **`std::regex`** provides pattern matching: `regex_match` for full-string matches, `regex_search` for partial matches, and `regex_replace` for substitution.
- Capture groups in parentheses let you extract parts of a match.
- Use raw string literals (`R"(...)"`) for readable regex patterns.
- **`std::stoi/std::stod`** convert strings to numbers; they throw on invalid input.
- **`std::to_string`** converts numbers to strings.
- **`std::from_chars/std::to_chars`** (C++17) are the fastest conversion functions: no allocation, no exceptions, no locale dependency.

Exercises

1. **Think about it:** Why is `std::string_view` a better function parameter type than `const std::string&` for functions that only read the string? What is the main risk of using `string_view`?
2. **What does this print?**

```cpp
std::string_view sv = "Estoy aqui";
sv.remove_prefix(6);
std::cout << sv << "\n";
std::cout << sv.size() << "\n";
```

3. **Where is the bug?**

```cpp
std::string_view get_greeting()
{
    std::string s = "Buenos dias";
    return s;
}

std::cout << get_greeting() << "\n";
```

4. **What does this print?**

```cpp
std::regex pattern(R"(\d+)");
std::string text = "Track 7 of 12";
std::smatch match;

if (std::regex_search(text, match, pattern)) {
    std::cout << match[0] << "\n";
}
```

5. **Calculation:** What does `std::stoi("0xFF", nullptr, 16)` return?
6. **Think about it:** Why do `std::from_chars` and `std::to_chars` not use exceptions? What advantage does this give for performance-critical code?
7. **Where is the bug?**

```cpp
std::string s = "not a number";
int n = std::stoi(s);
std::cout << n << "\n";
```

8. **What does this print?**

```
std::string entry = "Daft Punk - One More Time (2000)";
std::regex re(R"((.+) - (.+) \((\d+)\))");
std::smatch m;
std::regex_match(entry, m, re);
std::cout << m[2] << "\n";
```

9. **Think about it:** The `<regex>` library can be slow for complex patterns. What alternatives exist in the C++ ecosystem for high-performance regex matching?

10. **Write a program** that takes a list of strings in the format "Name:Score" (e.g., "Alice:95", "Bob:87") stored in a `std::vector<std::string>`. Use `std::regex` or `string_view::find` to parse each entry, convert the score to an `int`, and print the name and score. Also print the average score.