SimplifyCPP.org

Mastering REGULAR EXPRESSIONS in Modern C++

Practical std::regex from C++11 to C++23

Prepared by Ayman Alheraki

Mastering Regular Expressions in Modern C++

Practical ‘std::regex‘ from C++11 to C++23

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared by Ayman Alheraki

March 2026

Contents

Author's Introduction

Regular expressions are a fundamental tool for working with text in modern software systems. In C++, this capability was standardized starting from C++11 through the introduction of the `<regex>` library, which provides a complete framework for pattern matching, searching, extraction, and transformation of textual data.

The C++ regular expression system is built around a clear and structured model. A pattern is represented using `std::regex`, which encapsulates a compiled regular expression. This pattern is then applied to a target sequence, typically a `std::string`, using standard algorithms such as `regex_match`, `regex_search`, and `regex_replace`. The results of matching operations are stored in objects such as `std::match_results` (for example, `std::smatch`), which provide structured access to the full match and captured sub-expressions.

The default grammar used by the C++ standard library is a modified ECMAScript syntax, while additional grammars such as POSIX basic and extended forms are also supported. This allows developers to choose the most appropriate pattern style depending on the problem being solved.

Despite being part of the standard library, regular expressions in C++ are often misunderstood or misused. Many developers either avoid them due to perceived complexity or use them inefficiently by recompiling patterns repeatedly or writing patterns that are difficult to maintain. This booklet addresses these issues by focusing on practical usage grounded in the behavior defined by the standard.

The objective of this booklet is to present regular expressions as a precise engineering tool within Modern C++. The focus is not on theoretical completeness, but on correct usage, clarity of implementation, and real-world applicability. Each concept is introduced with direct examples that demonstrate how patterns are constructed, how matching works, and how results are extracted and transformed in real code.

All examples are designed to compile and run in a standard Windows development environment using modern C++ compilers that support C++11 through C++23. The code emphasizes readability, correctness, and practical relevance, allowing the reader to immediately apply the techniques in production scenarios.

By the end of this booklet, the reader will understand how to use `std::regex` effectively, how to avoid common pitfalls, and how to integrate regular expressions into modern C++ applications with confidence and control.

Ayman Alheraki

Preface

This booklet presents a focused and practical guide to using regular expressions in Modern C++, based on the standardized `<regex>` library introduced in C++11 and maintained through C++23. The scope of this guide is limited to the standard facilities provided by the C++ Standard Library, including pattern representation, matching algorithms, iterators, and transformation functions.

The C++ regular expression system operates on a well-defined model consisting of three main elements: a compiled pattern represented by `std::regex`, a target character sequence such as `std::string`, and a set of algorithms that apply the pattern to the target, including `regex_match`, `regex_search`, and `regex_replace`. Match results are stored in structured objects such as `std::match_results`, allowing direct access to full matches and captured sub-expressions.

This guide is intended for developers who are already familiar with C++ and want to use regular expressions effectively in real applications. It is especially relevant for engineers working with text processing, data extraction, validation, and transformation tasks, where pattern matching is a recurring requirement.

The content is designed to be concise and directly applicable. Each section focuses on a specific capability of the `<regex>` library, with examples that demonstrate how patterns are defined, applied, and evaluated in practice. All examples are compatible with modern C++ compilers on Windows environments and follow standard-compliant usage.

To use this guide effectively, it is recommended to:

- Read the chapters in order to understand the progression from basic concepts to practical applications.
- Compile and run each example to observe actual behavior and match results.
- Focus on understanding the distinction between full matching and partial searching, as this is central to correct usage.
- Reuse compiled `std::regex` objects in performance-sensitive code to avoid unnecessary overhead.

The goal of this booklet is to provide a clear and reliable reference for using `std::regex` as a practical tool in Modern C++, enabling the reader to write correct, maintainable, and efficient pattern-based code.

Understanding What Regular Expressions Solve in Real Software

1.1 Understanding the Core Problem That Regex Addresses

Modern software frequently needs to inspect, validate, extract, or transform text that follows recognizable patterns. A program may need to detect dates, identifiers, log messages, key-value pairs, version numbers, file names, or structured user input. In such cases, the problem is not only to compare text literally, but to describe a family of acceptable forms. Regular expressions solve this problem by allowing the programmer to express a text pattern in a compact and declarative way. Instead of checking one character at a time with many nested conditions, the programmer can define a rule such as:

- one or more digits,
- a word followed by a separator,
- a line that begins with a timestamp,
- an identifier that must start with a letter and continue with letters, digits, or underscores.

In Modern C++, this capability is provided by the standard `<regex>` library. The library gives the programmer a regular expression type, match-result types, algorithms for matching and

searching, and iterators for enumerating results. This makes regular expressions a standard part of text-oriented programming in C++.
The main software problem addressed by regex is therefore clear: how to describe and process variable textual patterns in a reliable and reusable form.

1.2 Situations Where Regex Is the Right Tool

Regex is the right tool when the input has a textual structure that can be described as a pattern and when the program needs one or more of the following operations:

- validate whether the input follows a required textual form,
- search within larger text for a smaller structured fragment,
- extract specific sub-parts from a match,
- replace or normalize text based on a rule,
- iterate over repeated occurrences of a pattern.

Regex is especially useful when the input is semi-structured rather than fully parsed into objects. Examples include lines from log files, configuration entries, imported text data, machine-generated reports, simple identifiers, and lightweight validation rules.
A good practical use of regex appears when the structure is easy to state as a pattern and difficult to maintain using manual character-by-character code.

1.2.1 Example: Finding an Order Identifier Inside Text

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

```
int main() {
    std::string text = "Customer confirmed Order ID: 48291 at 10:45 AM";
    std::regex pattern(R"(Order ID:\s*(\d+))");
    std::smatch match;

    if (std::regex_search(text, match, pattern)) {
        std::cout << "Full match: " << match[0] << '\n';
        std::cout << "Order number: " << match[1] << '\n';
    } else {
        std::cout << "No order identifier found.\n";
    }

    return 0;
}
```

This example is a good use of regex because the program is not comparing one fixed string only. It is searching for a structured fragment inside a larger sentence and extracting the numeric value.

1.2.2 Windows Compilation Example

```
cl /EHsc /std:c++17 regex_order.cpp
```

On modern Visual Studio toolchains, the language standard can be selected using the `/std:` option.

1.3 Situations Where Regex Should Be Avoided

Regex is powerful, but it should not be used everywhere. It becomes the wrong tool when the task is simpler than the pattern, when readability suffers, or when the input is better handled by another parsing strategy.

Regex should usually be avoided in the following situations:

- when a simple direct comparison is enough,
- when ordinary substring search is clearer,
- when the data is already structured and should be parsed formally,
- when the pattern becomes too complex to read or maintain,
- when performance-critical code repeatedly recompiles the same pattern unnecessarily.

For example, if a program only needs to check whether a file name ends with `.txt`, a simple string operation is often clearer than a regex. If a program must parse nested or highly recursive language syntax, regex alone is usually not the correct design.
The important engineering principle is not to use regex because it looks compact. It should be used because it models the problem clearly.

1.3.1 Example: A Task That Does Not Need Regex

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

int main() {
    std::string fileName = "report.txt";

    if (fileName.size() >= 4 &&
        fileName.substr(fileName.size() - 4) == ".txt") {
        std::cout << "Text file detected.\n";
    } else {
        std::cout << "Not a text file.\n";
    }
```

```
    return 0;
}
```

This solution is simpler, clearer, and easier to maintain than introducing a regular expression for the same task.

1.4 Real-World Scenarios Where Regex Is Commonly Used

Regex is commonly used in real software when text arrives in forms that are consistent enough to match patterns but not rich enough to justify a full parser.
Common real-world scenarios include:

- validating user input such as usernames, simple identifiers, and codes,
- extracting values from log lines,
- locating timestamps, IP-like fragments, or numbers in diagnostic text,
- reading key-value configuration lines,
- cleaning imported data before further processing,
- rewriting text into a normalized format,
- scanning documents for repeated textual patterns.

These scenarios appear in command-line utilities, desktop applications, support tools, automated scripts, testing systems, monitoring software, and internal engineering tools.

1.4.1 Example: Validating a Simple Username

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

bool is_valid_username(const std::string& name) {
    static const std::regex pattern(R"([A-Za-z][A-Za-z0-9_]{2,15})");
    return std::regex_match(name, pattern);
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_valid_username("Ayman_77") << '\n';
    std::cout << is_valid_username("77Ayman") << '\n';
    std::cout << is_valid_username("ab") << '\n';
    return 0;
}
```

This is a practical validation scenario because the input must follow a rule and the entire string must satisfy that rule.

1.4.2 Example: Parsing a Simple Log Line

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

int main() {
    std::string logLine = "[2026-03-24 14:33:10] ERROR Disk quota exceeded";
    std::regex pattern(R"(\[(.*?)\]\s+([A-Z]+)\s+(.*))");
    std::smatch match;
```

```
    if (std::regex_match(logLine, match, pattern)) {
        std::cout << "Timestamp: " << match[1] << '\n';
        std::cout << "Level: " << match[2] << '\n';
        std::cout << "Message: " << match[3] << '\n';
    } else {
        std::cout << "Log line format mismatch.\n";
    }

    return 0;
}
```

This example shows why regex is valuable in practice. One pattern can validate the general line format and extract the important components in a single step.

Chapter Summary

Regular expressions address a very practical software need: handling text that follows recognizable patterns. They are most useful when a program must validate, search, extract, or transform semi-structured text. They should be avoided when simpler string operations are sufficient or when the input requires a more formal parsing strategy. In real software, regex is commonly used for validation, log analysis, configuration handling, and data cleanup. For Modern C++ developers, understanding this boundary is essential before learning the detailed mechanics of `std::regex`.

Understanding the Design of the <regex> Library

2.1 Overview of the <regex> Header and Its Role in the Standard Library

The <regex> header provides the Standard Library facilities for regular-expression-based text processing in Modern C++. It supplies the pattern types, match-result types, core algorithms, iterator support, error handling, and grammar-selection flags needed to perform matching, searching, extraction, and replacement operations on character sequences.
In practical software terms, the role of this header is to give C++ programmers a standard and portable interface for expressing text patterns and applying them to input such as `std::string`, C-style strings, and related character ranges. Instead of writing many manual checks for characters, separators, and repetitions, the programmer can express a pattern once and reuse it through standard algorithms.
This library became part of standard C++ in C++11 and remains the standard regex facility through C++23. The programming model remains centered on three parts:

- a regular expression object that represents the pattern,
- an input sequence to test or search,

- an algorithm that applies the pattern and reports the result.

2.2 Understanding the Main Types Used in Regex Operations

2.2.1 The Role of `std::regex` as a Compiled Pattern

The type `std::regex` is the most commonly used regular expression type for narrow-character text. Conceptually, it represents a compiled pattern that can be constructed once and then reused many times. This is important because the pattern should normally be prepared before repeated matching or searching, especially in loops or repeated validations. A `std::regex` object can be constructed from a string literal, a raw string literal, or a `std::string`. In most real code, raw string literals are the clearest option because they reduce escaping complexity.

2.2.2 Example: Constructing and Reusing a Pattern

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

int main() {
    std::regex idPattern(R"([A-Z]{3}-\d{4})");

    std::vector<std::string> values = {
        "ABC-1024",
        "ZZ-9999",
        "QWE-7001"
    };

    for (const auto& value : values) {
```

```
        if (std::regex_match(value, idPattern)) {
            std::cout << value << " : valid\n";
        } else {
            std::cout << value << " : invalid\n";
        }
    }

    return 0;
}
```

In this example, one pattern object is created and reused for multiple inputs. This is clearer and more efficient than rebuilding the same regex object for every string.

2.2.3 Understanding `std::smatch` and Match Result Objects

When a regex operation needs to return detailed information about a match, C++ uses match-result objects. For `std::string`-based operations, the most common type is `std::smatch`. It stores:

- the full match,
- captured sub-expressions,
- the text before the match,
- the text after the match.

The element at index `0` represents the full match. The following elements represent captured groups in the order they appear in the pattern.

2.2.4 Example: Accessing Full Match and Captured Groups

```
#include <iostream>
```

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

int main() {
    std::string text = "File: report_2026.txt";
    std::regex pattern(R"((\w+)_(\d{4})\.(\w+))");
    std::smatch match;

    if (std::regex_search(text, match, pattern)) {
        std::cout << "Full match : " << match[0] << '\n';
        std::cout << "Base name  : " << match[1] << '\n';
        std::cout << "Year       : " << match[2] << '\n';
        std::cout << "Extension  : " << match[3] << '\n';
    } else {
        std::cout << "No match found.\n";
    }

    return 0;
}
```

2.3 Understanding the Core Operations Provided by the Library

2.3.1 How `regex_match` Performs Full String Matching

The function `regex_match` checks whether the entire target sequence matches the pattern. This makes it suitable for validation tasks where the whole input must satisfy a rule. Typical uses include validating usernames, codes, identifiers, file naming rules, or strict data formats.

2.3.2 Example: Full Validation with `regex_match`

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

bool is_valid_code(const std::string& code) {
    static const std::regex pattern(R"([A-Z]{2}\d{6})");
    return std::regex_match(code, pattern);
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_valid_code("AB123456") << '\n';
    std::cout << is_valid_code("AB123") << '\n';
    std::cout << is_valid_code("XX999999") << '\n';
    return 0;
}
```

2.3.3 How `regex_search` Finds Patterns Inside Text

The function `regex_search` looks for a matching fragment inside a larger input sequence. The whole string does not need to match. This makes it suitable for extraction tasks such as finding timestamps, identifiers, error codes, or values embedded inside longer messages.

2.3.4 Example: Searching Inside a Larger Message

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

int main() {
```

```
    std::string line = "Warning: user ID 48291 exceeded the allowed quota.";
    std::regex pattern(R"(ID\s+(\d+))");
    std::smatch match;

    if (std::regex_search(line, match, pattern)) {
        std::cout << "Matched text : " << match[0] << '\n';
        std::cout << "Extracted ID : " << match[1] << '\n';
    } else {
        std::cout << "No identifier found.\n";
    }

    return 0;
}
```

2.3.5 How `regex_replace` Transforms Text

The function `regex_replace` creates transformed text by replacing matches with a replacement expression. It can perform simple substitutions or use captured groups to reorganize the output.

This is useful for text cleanup, data normalization, masking, and lightweight format conversion.

2.3.6 Example: Reformatting a Date

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

int main() {
    std::string isoDate = "2026-03-24";
    std::regex pattern(R"((\d{4})-(\d{2})-(\d{2}))");
```

```
    std::string reformatted =
        std::regex_replace(isoDate, pattern, "$3/$2/$1");

    std::cout << reformatted << '\n';
    return 0;
}
```

2.4 Understanding the Available Regex Grammars and Their Differences

2.4.1 Why ECMAScript Is the Default Grammar

In standard C++, the default grammar for `std::regex` is ECMAScript. This means that if no grammar flag is explicitly specified, the engine interprets the pattern according to the modified ECMAScript grammar defined for the standard regex library.

This default is important because it affects how escapes, character classes, repetition operators, and grouping behavior are interpreted. A pattern should therefore be written with the selected grammar in mind.

For most practical Modern C++ examples, ECMAScript is the best starting point because it is the default, widely recognized, and sufficient for common matching, searching, and replacement tasks.

2.4.2 Overview of Other Supported Grammars

The standard library also provides other grammar modes through syntax-option flags. These include:

- `basic`
- `extended`

- awk
- grep
- egrep

These grammars exist because regular expression syntax historically varies across systems and tool families. In practice, many application-level C++ programs stay with ECMAScript unless compatibility with another style is required.

2.4.3 Example: Explicitly Selecting the ECMAScript Grammar

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

int main() {
    std::string text = "abc123";
    std::regex pattern(
        R"([a-z]+\d+)",
        std::regex_constants::ECMAScript
    );

    if (std::regex_match(text, pattern)) {
        std::cout << "Matched under ECMAScript grammar.\n";
    } else {
        std::cout << "No match.\n";
    }

    return 0;
}
```

2.4.4 Windows Compilation Example

```
cl /EHsc /std:c++17 chapter2_regex.cpp
```

For newer language modes in recent MSVC toolchains, the project or command line can be configured with a later standard option when available.

Chapter Summary

The <regex> header provides the standard C++ framework for regular-expression-based text processing. Its design is centered on pattern objects such as `std::regex`, result containers such as `std::smatch`, and the core algorithms `regex_match`, `regex_search`, and `regex_replace`. The default grammar is ECMAScript, while other grammar modes are also available for specialized needs. For practical Modern C++ work, understanding these types and operations is essential before writing more advanced patterns.

Writing Correct and Readable Regex Patterns in C++

3.1 Understanding the Problem of Escaping in C++ String Literals

One of the first difficulties when writing regular expressions in C++ is that the pattern is itself written inside a C++ string literal. This means that two different interpretation layers are involved:

- the C++ compiler interprets escape sequences inside the string literal,
- the regex engine interprets the final text as a regular expression pattern.

Because of this, a backslash that is meant for the regex engine often must itself be escaped for the C++ compiler. For example, the regex token for a digit class is `\d`, but inside an ordinary C++ string literal it is commonly written as `"\\d"`.
This double interpretation is the source of many mistakes. A pattern may look correct as a regex idea, yet fail in C++ because the string literal changed the characters before the regex engine received them.

3.1.1 Example: Escaping in an Ordinary String Literal

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

int main() {
    std::string text = "Invoice 48291";
    std::regex pattern("\\d+");

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

    return 0;
}
```

In this example, the regex engine finally receives `\d+`, but the programmer had to write `"\\d+"` in C++ source code.

3.2 Using Raw String Literals to Simplify Regex Patterns

Raw string literals are usually the clearest way to write regex patterns in Modern C++. A raw string literal begins with `R"` and preserves backslashes and quotation marks more naturally. This removes most of the extra escaping that ordinary string literals require.
For regex work, raw string literals greatly improve readability. A pattern such as `\d{4}-\d{2}-\d{2}` becomes much easier to read when written directly as a raw string.

3.2.1 Example: Raw String Literal for a Date Pattern

```
#include <iostream>
```

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

int main() {
    std::string date = "2026-03-24";
    std::regex pattern(R"(\d{4}-\d{2}-\d{2})");

    if (std::regex_match(date, pattern)) {
        std::cout << "Valid date format.\n";
    } else {
        std::cout << "Invalid date format.\n";
    }

    return 0;
}
```

This form is easier to read because the pattern is written almost exactly as the regex logic is intended.

3.3 Defining Character Classes for Flexible Matching

Character classes allow a pattern to match one character from a defined set. They are useful when the exact character may vary within a known range. Common examples include digits, letters, spaces, or explicitly listed characters.
Typical forms include:

- `[0-9]` for one digit,
- `[A-Z]` for one uppercase letter,
- `[A-Za-z0-9_]` for one alphanumeric or underscore character,
- `\d` for a digit under the selected grammar,

- \w for a word character under the selected grammar.

Character classes are essential when validating identifiers, codes, file names, and many kinds of structured text.

3.3.1 Example: Matching a Product Code

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

int main() {
    std::string code = "ABX-2048";
    std::regex pattern(R"([A-Z]{3}-[0-9]{4})");

    if (std::regex_match(code, pattern)) {
        std::cout << "Product code accepted.\n";
    } else {
        std::cout << "Product code rejected.\n";
    }

    return 0;
}
```

3.4 Controlling Repetition Using Quantifiers

Quantifiers define how many times an element may repeat. They are one of the most important parts of practical regex writing because real input often contains repeated digits, words, separators, or optional pieces.
Common quantifiers include:

- * for zero or more,

- `+` for one or more,
- `?` for zero or one,
- `{n}` for exactly `n`,
- `{n,}` for at least `n`,
- `{n,m}` for from `n` to `m`.

These make patterns expressive and compact. Instead of writing several manual checks, a single quantified rule can describe valid repetition clearly.

3.4.1 Example: Username Validation with Quantifiers

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

bool is_valid_username(const std::string& name) {
    static const std::regex pattern(R"([A-Za-z][A-Za-z0-9_]{2,15})");
    return std::regex_match(name, pattern);
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_valid_username("Ayman_77") << '\n';
    std::cout << is_valid_username("ab") << '\n';
    std::cout << is_valid_username("7start") << '\n';
    return 0;
}
```

3.5 Anchoring Patterns to the Start and End of Text

Anchors constrain where a match may occur. The most common anchors are:

- ^ for the beginning of the sequence,
- $ for the end of the sequence.

Anchors are especially useful when the programmer wants to express a full rule explicitly, even when using a searching function. They help prevent accidental partial matches.
In validation tasks, anchors often make the intended pattern clearer to the reader.

3.5.1 Example: Anchored Identifier Rule

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

int main() {
    std::string id = "EMP-2026";
    std::regex pattern(R"(^[A-Z]{3}-\d{4}$)");

    if (std::regex_match(id, pattern)) {
        std::cout << "Identifier is valid.\n";
    } else {
        std::cout << "Identifier is invalid.\n";
    }

    return 0;
}
```

3.6 Using Alternation to Match Multiple Possible Patterns

Alternation allows a pattern to match one option from several alternatives. It is written with the vertical bar character `|`. This is useful when input may appear in more than one valid form. For example, a date separator may be `-` or `/`, or a log level may be `INFO`, `WARN`, or `ERROR`. Alternation should be written carefully because long chains of alternatives can reduce readability if not grouped well.

3.6.1 Example: Accepting Two Date Separator Styles

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

int main() {
    std::string value1 = "2026-03-24";
    std::string value2 = "2026/03/24";

    std::regex pattern(R"(\d{4}(-|/)\d{2}(-|/)\d{2})");

    std::cout << std::boolalpha;
    std::cout << std::regex_match(value1, pattern) << '\n';
    std::cout << std::regex_match(value2, pattern) << '\n';

    return 0;
}
```

3.7 Capturing Parts of a Match Using Groups

Parentheses create groups. Groups are useful for two main reasons:

- they control structure and precedence inside the pattern,
- they capture sub-parts of the match for later access.

When a match succeeds, the complete match is stored as element `0`, and captured groups are stored in increasing order beginning from element `1`. This allows the program to extract meaningful parts such as year, month, day, prefix, extension, or identifier number.

3.7.1 Example: Capturing Date Components

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

int main() {
    std::string date = "2026-03-24";
    std::regex pattern(R"((\d{4})-(\d{2})-(\d{2}))");
    std::smatch match;

    if (std::regex_match(date, match, pattern)) {
        std::cout << "Year  : " << match[1] << '\n';
        std::cout << "Month : " << match[2] << '\n';
        std::cout << "Day   : " << match[3] << '\n';
    } else {
        std::cout << "Format mismatch.\n";
    }

    return 0;
}
```

3.8 Understanding Greedy Behavior and Its Impact on Matching

Many regex repetitions are greedy by default. This means they try to consume as much text as possible while still allowing the overall match to succeed. Greedy behavior is useful in many cases, but it can also produce larger matches than the programmer intended.
This matters when a pattern is used inside larger text and a repeated wildcard such as `.*` appears. Such patterns can easily consume more text than expected.
The practical lesson is simple: greedy patterns should be used carefully, especially when delimiters or repeated structures are involved.

3.8.1 Example: Greedy Matching Consuming Too Much

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

int main() {
    std::string text = "<tag>first</tag><tag>second</tag>";
    std::regex pattern(R"(<tag>.*</tag>)");
    std::smatch match;

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

    return 0;
}
```

A greedy repetition may consume from the first opening tag to the last closing tag in the sequence. This is often broader than the intended result.

3.8.2 Example: Narrower Matching by Constraining the Pattern

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

int main() {
    std::string text = "<tag>first</tag><tag>second</tag>";
    std::regex pattern(R"(<tag>[^<]*</tag>)");
    std::smatch match;

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

    return 0;
}
```

This version is often clearer because it limits the repeated region instead of allowing an unrestricted wildcard.

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter3_regex_patterns.cpp
```

Recent Visual Studio toolchains can also be configured for newer standard modes in the project settings or through the corresponding `/std:` compiler option.

Chapter Summary

Writing correct regex patterns in C++ requires attention to both regex syntax and C++ string literal rules. Ordinary string literals often require double escaping, while raw string literals

usually provide a much clearer representation of the intended pattern. Character classes, quantifiers, anchors, alternation, and groups form the core building blocks of practical pattern design. Greedy behavior must also be understood carefully because it can significantly change the scope of a match. Good regex style in Modern C++ is therefore not only about correctness, but also about readability and maintainability.

Choosing Between Full Matching and Partial Searching

4.1 Understanding the Difference Between Full Match and Partial Match

The most important distinction in practical regex usage is the difference between matching the entire input and matching only part of the input. In the C++ standard regex library, `regex_match` is designed for full matching, while `regex_search` is designed for partial matching inside a larger character sequence. This distinction is fundamental because it changes the meaning of the same pattern depending on which algorithm is used.
A full match means that the complete target text must satisfy the pattern. A partial match means that a matching fragment may appear anywhere inside the target text. In real software, confusing these two operations is one of the most common reasons for incorrect validation and extraction logic.

4.1.1 Example: Same Pattern, Two Different Meanings

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

```
int main() {
    std::string text = "User ID: 48291";
    std::regex pattern(R"(\d+)");

    std::cout << std::boolalpha;
    std::cout << "regex_match  : "
              << std::regex_match(text, pattern) << '\n';
    std::cout << "regex_search : "
              << std::regex_search(text, pattern) << '\n';

    return 0;
}
```

In this example, `regex_match` returns false because the full string is not composed only of digits. By contrast, `regex_search` returns true because a digit sequence exists inside the larger text.

4.2 When to Use `regex_match` for Strict Validation

The function `regex_match` should be used when the program is validating whether the whole input conforms to a required rule. It tests whether a regular expression matches the entire target character sequence, making it the correct choice for strict validation tasks.
Typical uses include checking usernames, product codes, invoice numbers, simple date formats, file naming rules, and other structured values where partial success is not acceptable. If even one unwanted extra character exists before or after the intended structure, the validation should fail, and that is exactly what `regex_match` provides.

4.2.1 Example: Strict Validation of a Product Code

```
#include <iostream>
```

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

bool is_valid_product_code(const std::string& code) {
    static const std::regex pattern(R"([A-Z]{3}-\d{4})");
    return std::regex_match(code, pattern);
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_valid_product_code("ABC-2048") << '\n';
    std::cout << is_valid_product_code("ABC-2048-EXTRA") << '\n';
    std::cout << is_valid_product_code("ID ABC-2048") << '\n';
    return 0;
}
```

This example demonstrates strict validation. Only the exact full form is accepted.

4.2.2 Example: Strict Validation of a Simple Date Form

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

int main() {
    std::regex pattern(R"(\d{4}-\d{2}-\d{2})");

    std::cout << std::boolalpha;
    std::cout << std::regex_match(std::string("2026-03-24"), pattern) << '\n';
    std::cout << std::regex_match(std::string("Date: 2026-03-24"), pattern) << '\n';
    return 0;
}
```

The first string is fully valid according to the rule. The second contains a valid date fragment, but it is not itself a full date string, so `regex_match` rejects it.

4.3 When to Use `regex_search` for Pattern Discovery

The function `regex_search` should be used when the goal is to discover whether a matching fragment exists anywhere inside a larger text. It attempts to match a regular expression to any part of a character sequence, making it suitable for extraction and discovery tasks.
This makes `regex_search` the right choice for scanning log lines, finding IDs in messages, locating dates in reports, detecting keywords, or extracting values embedded in larger text blocks. It is also the normal choice when captured groups are used to extract structured information.

4.3.1 Example: Discovering an Identifier Inside a Message

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

int main() {
    std::string message = "Warning: user ID 48291 exceeded quota.";
    std::regex pattern(R"(ID\s+(\d+))");
    std::smatch match;

    if (std::regex_search(message, match, pattern)) {
        std::cout << "Full match: " << match[0] << '\n';
        std::cout << "ID value  : " << match[1] << '\n';
    } else {
        std::cout << "No identifier found.\n";
    }
```

```
    return 0;
}
```

This is a search problem, not a full validation problem. The message contains useful information inside a larger sentence.

4.3.2 Example: Finding a Date Fragment in a Log Line

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

int main() {
    std::string logLine = "[INFO] Backup completed at 2026-03-24 21:10:45";
    std::regex pattern(R"((\d{4}-\d{2}-\d{2}))");
    std::smatch match;

    if (std::regex_search(logLine, match, pattern)) {
        std::cout << "Found date: " << match[1] << '\n';
    } else {
        std::cout << "No date found.\n";
    }

    return 0;
}
```

Here, only a fragment of the larger line is relevant, so `regex_search` is the natural choice.

4.4 Practical Examples Comparing Both Approaches in Real Code

In practical engineering work, the choice between `regex_match` and `regex_search` depends on the intent of the operation. If the intent is to answer the question, *Is this whole string valid according to the rule?*, then `regex_match` is correct. If the intent is to answer the question, *Does this text contain something that follows the rule?*, then `regex_search` is correct.
A useful rule of thumb is:

- use `regex_match` for validation,
- use `regex_search` for discovery and extraction.

This simple guideline avoids many design mistakes in production code.

4.4.1 Example: Validation Versus Discovery

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

int main() {
    std::regex pattern(R"([A-Z]{2}\d{6})");

    std::string value1 = "AB123456";
    std::string value2 = "Invoice contains code AB123456 for shipment";

    std::cout << std::boolalpha;

    std::cout << "Value 1 with regex_match  : "
              << std::regex_match(value1, pattern) << '\n';
```

```
    std::cout << "Value 1 with regex_search : "
              << std::regex_search(value1, pattern) << '\n';

    std::cout << "Value 2 with regex_match  : "
              << std::regex_match(value2, pattern) << '\n';
    std::cout << "Value 2 with regex_search : "
              << std::regex_search(value2, pattern) << '\n';

    return 0;
}
```

The first string is itself a code, so both operations succeed. The second string only contains a code fragment, so `regex_search` succeeds while `regex_match` fails.

4.4.2 Example: Extracting Data After a Search

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

int main() {
    std::string line = "Order status changed. Order ID: 73145";
    std::regex pattern(R"(Order ID:\s*(\d+))");
    std::smatch match;

    if (std::regex_search(line, match, pattern)) {
        std::cout << "Whole fragment : " << match[0] << '\n';
        std::cout << "Extracted value: " << match[1] << '\n';
    }

    return 0;
}
```

This example highlights another practical reason to prefer `regex_search` in discovery scenarios: the matched fragment and captured groups can be extracted directly through `std::match_results`. The standard library stores the entire match at index `0` and captured sub-expressions at later indices.

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter4_match_vs_search.cpp
```

For Visual Studio on Windows, the `/std:` compiler option is used to select the language standard mode.

Chapter Summary

The distinction between full matching and partial searching is central to correct regex design in Modern C++. `regex_match` is used when the entire input must satisfy the rule, making it appropriate for strict validation. `regex_search` is used when the program needs to find a matching fragment inside larger text, making it appropriate for discovery and extraction. Choosing the correct algorithm is often more important than changing the pattern itself, because the same regex can produce very different behavior depending on whether full matching or partial searching is intended.

Extracting Structured Data Using Capturing Groups

5.1 Understanding How Match Results Store Extracted Data

In the C++ standard regex library, a successful regex operation can return a match result object that stores structured information about what was found in the target text. For `std::string`-based work, the most common result type is `std::smatch`. This type is an alias of `std::match_results<std::string::const_iterator>` and is designed to preserve the full match together with every captured subexpression.
A match result does not store only one text fragment. It stores a sequence of submatches. The first element represents the entire matched text, while later elements represent the captured groups written in parentheses inside the pattern. This design makes regex useful not only for finding text, but also for extracting structured fields from a raw input line.

5.1.1 Example: Storing a Full Match and Submatches

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

```
int main() {
    std::string text = "User: Ayman, ID: 48291";
    std::regex pattern(R"(User:\s*(\w+),\s*ID:\s*(\d+))");
    std::smatch match;

    if (std::regex_search(text, match, pattern)) {
        std::cout << "Number of stored parts: " << match.size() << '\n';
        for (std::size_t i = 0; i < match.size(); ++i) {
            std::cout << "match[" << i << "] = " << match[i] << '\n';
        }
    } else {
        std::cout << "No match found.\n";
    }

    return 0;
}
```

In this example, `match[0]` stores the whole matched fragment, while `match[1]` and `match[2]` store the captured name and numeric identifier.

5.2 Accessing the Full Match and Individual Captured Groups

Captured groups are accessed by index. Index `0` always refers to the whole match. Each group written inside parentheses is assigned the next index in left-to-right order. This makes it possible to convert one unstructured line into smaller logical fields.
This model is especially useful when processing configuration lines, dates, identifiers, filenames, and machine-generated text.

5.2.1 Example: Accessing Captured Parts of a Date

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

int main() {
    std::string date = "2026-03-24";
    std::regex pattern(R"((\d{4})-(\d{2})-(\d{2}))");
    std::smatch match;

    if (std::regex_match(date, match, pattern)) {
        std::cout << "Full date : " << match[0] << '\n';
        std::cout << "Year      : " << match[1] << '\n';
        std::cout << "Month     : " << match[2] << '\n';
        std::cout << "Day       : " << match[3] << '\n';
    } else {
        std::cout << "Date format mismatch.\n";
    }

    return 0;
}
```

This approach is clearer than manual slicing because the structure of the input is declared directly in the pattern.

5.3 Using Prefix and Suffix to Analyze Surrounding Text

A match result object also exposes the text before and after the matched fragment through `prefix()` and `suffix()`. This is useful when the program needs not only the extracted value, but also the surrounding context in which it appeared.

The prefix is the part of the target sequence before the current match. The suffix is the part after the current match. This behavior is part of the standard `match_results` design and is valuable when scanning logs, command output, or mixed-content lines.

5.3.1 Example: Inspecting Prefix and Suffix

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

int main() {
    std::string line = "Status=OK; Order ID: 73145; Source=Desktop";
    std::regex pattern(R"(Order ID:\s*(\d+))");
    std::smatch match;

    if (std::regex_search(line, match, pattern)) {
        std::cout << "Prefix : " << match.prefix() << '\n';
        std::cout << "Match  : " << match[0] << '\n';
        std::cout << "Value  : " << match[1] << '\n';
        std::cout << "Suffix : " << match.suffix() << '\n';
    } else {
        std::cout << "Order identifier not found.\n";
    }

    return 0;
}
```

This style is useful when later processing depends on what appears before or after the extracted field.

5.4 Handling Optional and Missing Groups Safely

Not every captured group is guaranteed to participate in every successful match. A pattern may contain optional parts, and when one of those parts is absent, the corresponding submatch is present as a valid object but does not contain matched text. For this reason, optional groups should be checked carefully before use.
In practice, a safe approach is to inspect whether the submatch is marked as matched before treating it as a meaningful value.

5.4.1 Example: Optional Group for a File Extension

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

int main() {
    std::string name = "archive";
    std::regex pattern(R"((\w+)(\.(\w+))?)");
    std::smatch match;

    if (std::regex_match(name, match, pattern)) {
        std::cout << "Base name : " << match[1] << '\n';

        if (match[3].matched) {
            std::cout << "Extension : " << match[3] << '\n';
        } else {
            std::cout << "Extension : <none>\n";
        }
    } else {
        std::cout << "Input format mismatch.\n";
    }
```

```
    return 0;
}
```

This is a practical safety rule: optional groups should never be assumed to contain meaningful text unless they are checked.

5.5 Building Structured Data Extraction from Raw Text

The real strength of capturing groups appears when a program transforms raw text into structured values. Instead of treating an input line as a single string, the program can divide it into logical parts such as timestamp, level, message, code, year, or user identifier.
This is often sufficient for lightweight parsing tasks where a full parser would be unnecessary.

5.5.1 Example: Extracting Fields from a Log Line

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

struct LogRecord {
    std::string timestamp;
    std::string level;
    std::string message;
};

int main() {
    std::string logLine =
        "[2026-03-24 21:10:45] ERROR Disk quota exceeded";

    std::regex pattern(R"(\[(.*?)\]\s+([A-Z]+)\s+(.*))");
```

```
    std::smatch match;

    if (std::regex_match(logLine, match, pattern)) {
        LogRecord record;
        record.timestamp = match[1].str();
        record.level = match[2].str();
        record.message = match[3].str();

        std::cout << "Timestamp : " << record.timestamp << '\n';
        std::cout << "Level     : " << record.level << '\n';
        std::cout << "Message   : " << record.message << '\n';
    } else {
        std::cout << "Log line format mismatch.\n";
    }

    return 0;
}
```

This example shows how regex groups can serve as a bridge between raw text and structured program data.

5.5.2 Example: Extracting Key-Value Pairs

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

int main() {
    std::vector<std::string> lines = {
        "host=server01",
        "port=5432",
        "mode=readonly"
```

```
    };

    std::regex pattern(R"((\w+)=(.+))");

    for (const auto& line : lines) {
        std::smatch match;
        if (std::regex_match(line, match, pattern)) {
            std::cout << "Key   : " << match[1] << '\n';
            std::cout << "Value : " << match[2] << '\n';
            std::cout << '\n';
        }
    }

    return 0;
}
```

This is a common real-world use case in configuration processing and diagnostic tools.

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter5_capturing_groups.cpp
```

Recent Visual Studio toolchains can also be configured for newer language modes through the corresponding project setting or compiler option.

Chapter Summary

Match result objects allow Modern C++ programs to treat regex operations as structured extraction rather than simple yes-or-no testing. The full match is stored at index `0`, captured groups are stored at later indices, and surrounding text can be inspected through `prefix()` and `suffix()`. Optional groups should be checked safely before use. When applied carefully,

capturing groups make it possible to transform raw text into meaningful structured data with compact and readable code.

Transforming and Cleaning Text Using Regex Replacement

6.1 Understanding How `regex_replace` Works Internally

The function `regex_replace` applies a regular expression to a target character sequence and produces a new string in which each selected match is replaced according to a replacement format. In practical terms, the operation follows a simple model:

- scan the input text for matches,
- copy unchanged text segments to the output,
- insert replacement text for each match,
- continue until the full input has been processed.

This means that `regex_replace` does not modify the original string in place. It generates a transformed result based on the pattern, the replacement format, and any match flags that affect behavior.
In Modern C++, this is one of the most useful regex operations because it combines searching and rewriting into a single standard-library call.

6.1.1 Example: Replacing All Digits with a Marker

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

int main() {
    std::string text = "User42 has 3 pending tasks.";
    std::regex pattern(R"(\d+)");

    std::string result = std::regex_replace(text, pattern, "[NUMBER]");

    std::cout << result << '\n';
    return 0;
}
```

This example replaces each digit sequence with the same fixed text.

6.2 Using Captured Groups to Build Dynamic Replacement Strings

A major strength of `regex_replace` is that the replacement string can refer to captured groups from the match. This allows the program to rebuild text using parts of the original input in a different order or format.
When the pattern contains parentheses, each captured group can be referenced in the replacement format. This makes it possible to:

- reorder fields,
- extract only selected parts,
- insert separators,

- normalize the output structure.

This is especially useful when rewriting dates, filenames, names, identifiers, and simple structured text.

6.2.1 Example: Reordering a Date

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

int main() {
    std::string isoDate = "2026-03-24";
    std::regex pattern(R"((\d{4})-(\d{2})-(\d{2}))");

    std::string result = std::regex_replace(
        isoDate,
        pattern,
        "$3/$2/$1"
    );

    std::cout << result << '\n';
    return 0;
}
```

The replacement format reuses the captured groups to change the order from year-month-day to day/month/year.

6.2.2 Example: Rewriting a Name

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

```
int main() {
    std::string name = "Alheraki, Ayman";
    std::regex pattern(R"((\w+),\s*(\w+))");

    std::string result = std::regex_replace(
        name,
        pattern,
        "$2 $1"
    );

    std::cout << result << '\n';
    return 0;
}
```

This is a practical example of using captured groups to build a new string layout.

6.3 Formatting and Rewriting Text Based on Patterns

Regex replacement is often used not merely to remove text, but to reformat it into a cleaner or more consistent representation. The pattern identifies the structure, while the replacement string expresses the desired output form.
Typical tasks include:

- converting date layouts,
- normalizing file naming conventions,
- masking sensitive values,
- adding separators or labels,
- reducing inconsistent spacing.

This kind of transformation is very common in support tools, import-cleanup utilities, validation pipelines, and desktop utilities that prepare text for later processing.

6.3.1 Example: Masking Part of an Account Number

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

int main() {
    std::string account = "Account: 9876543210";
    std::regex pattern(R"((Account:\s*)\d{6}(\d{4}))");

    std::string result = std::regex_replace(
        account,
        pattern,
        "$1******$2"
    );

    std::cout << result << '\n';
    return 0;
}
```

This keeps the label and the last four digits while masking the middle part.

6.3.2 Example: Normalizing Repeated Spaces

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

int main() {
    std::string text = "C++     regex    can   clean   text.";
```

```
    std::regex pattern(R"(\s+)");

    std::string result = std::regex_replace(text, pattern, " ");

    std::cout << result << '\n';
    return 0;
}
```

This is one of the most common data-cleaning patterns.

6.4 Practical Examples for Cleaning and Normalizing Input Data

Real software often receives imperfect text from logs, imported files, copied reports, or user-entered forms. Regex replacement helps transform that raw text into a more stable format before further processing.

The best use cases are those where the structure is repetitive enough to describe with a pattern and the output can be expressed as a simple rewrite rule.

6.4.1 Example: Cleaning a Key-Value Assignment

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

int main() {
    std::string line = "   port    =    5432   ";
    std::regex pattern(R"(\s*([A-Za-z_]\w*)\s*=\s*(.*?)\s*)");

    std::string result = std::regex_replace(
```

```
        line,
        pattern,
        "$1=$2"
    );

    std::cout << result << '\n';
    return 0;
}
```

This converts irregular spacing into a normalized key-value form.

6.4.2 Example: Removing Non-Digit Separators from a Phone Number

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

int main() {
    std::string phone = "(555) 120-9988";
    std::regex pattern(R"([^\d])");

    std::string compact = std::regex_replace(phone, pattern, "");

    std::cout << compact << '\n';
    return 0;
}
```

This is useful when imported text must be reduced to a machine-friendly canonical form.

6.4.3 Example: Normalizing a List Separator Style

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

```cpp
#include <string>

int main() {
    std::string text = "red ; blue,green ;  yellow";
    std::regex pattern(R"(\s*[,;]\s*)");

    std::string result = std::regex_replace(text, pattern, ", ");

    std::cout << result << '\n';
    return 0;
}
```

This example converts mixed separators and inconsistent spacing into one unified style.

6.4.4 Example: Rewriting Log Prefixes

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

int main() {
    std::string logLine = "[2026-03-24 21:10:45] ERROR Disk quota exceeded";
    std::regex pattern(R"(\[(.*?)\]\s+([A-Z]+)\s+(.*))");

    std::string result = std::regex_replace(
        logLine,
        pattern,
        "Time=$1 | Level=$2 | Message=$3"
    );

    std::cout << result << '\n';
    return 0;
}
```

This shows how regex replacement can turn raw text into a cleaner reporting format.

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter6_regex_replace.cpp
```

Recent Visual Studio toolchains can also be configured for newer standard modes through the corresponding project setting or compiler option.

Chapter Summary

The function `regex_replace` transforms text by scanning for matches and constructing a new output string from unchanged segments and replacement text. When captured groups are used, replacement strings become dynamic and expressive, allowing fields to be reordered or selectively preserved. This makes regex replacement a practical tool for text cleanup, normalization, masking, and lightweight rewriting tasks in Modern C++ programs.

Processing Multiple Matches in Large Text Inputs

7.1 Iterating Over All Matches Using `sregex_iterator`

When a pattern may appear more than once in the same input, calling `regex_search` only once is not sufficient. In such cases, the standard library provides `regex_iterator` and its string specialization `sregex_iterator`. This iterator repeatedly applies the regular expression to the input range and yields a sequence of `match_results` objects, one for each match found.

This makes `sregex_iterator` the standard tool for scanning a full text block and collecting all occurrences of a repeated pattern. Each iterator dereference gives access to the full match and all captured groups for the current occurrence.

7.1.1 Example: Iterating Over All Numbers in Text

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

int main() {
    std::string text = "IDs: 104, 275, 389, 450";
```

```
    std::regex pattern(R"(\d+)");

    std::sregex_iterator it(text.begin(), text.end(), pattern);
    std::sregex_iterator end;

    for (; it != end; ++it) {
        std::cout << "Match: " << (*it).str() << '\n';
    }

    return 0;
}
```

This is the correct standard approach when the goal is to find every occurrence rather than only the first one.

7.2 Extracting Repeated Patterns from Large Text Blocks

In practical software, repeated patterns often appear inside logs, reports, configuration dumps, and imported text data. A regex iterator allows the program to process these matches one by one without manually updating the search position.
This is especially useful when each match contains structured sub-parts that must be extracted and processed independently.

7.2.1 Example: Extracting All Email-Like Fragments

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

int main() {
    std::string text =
```

```
        "Contacts: admin@example.com, support@company.org, dev.team@lab.net";

    std::regex pattern(R"(([A-Za-z0-9._%+-]+)@([A-Za-z0-9.-]+\.[A-Za-z]{2,}))");

    std::sregex_iterator it(text.begin(), text.end(), pattern);
    std::sregex_iterator end;

    for (; it != end; ++it) {
        std::cout << "Full address : " << (*it)[0] << '\n';
        std::cout << "Local part   : " << (*it)[1] << '\n';
        std::cout << "Domain       : " << (*it)[2] << '\n';
        std::cout << '\n';
    }

    return 0;
}
```

Each match result behaves like the structure returned by `regex_search`, but the iterator automatically advances to the next occurrence.

7.2.2 Example: Extracting Repeated Log Records

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

int main() {
    std::string logs =
        "[2026-03-24 08:00:01] INFO Service started\n"
        "[2026-03-24 08:02:15] WARN Cache miss\n"
        "[2026-03-24 08:03:41] ERROR Disk full\n";

    std::regex pattern(R"(\[(.*?)\]\s+([A-Z]+)\s+(.*))");
```

```
    std::sregex_iterator it(logs.begin(), logs.end(), pattern);
    std::sregex_iterator end;

    for (; it != end; ++it) {
        std::cout << "Time    : " << (*it)[1] << '\n';
        std::cout << "Level   : " << (*it)[2] << '\n';
        std::cout << "Message : " << (*it)[3] << '\n';
        std::cout << '\n';
    }

    return 0;
}
```

This technique is useful when a large text block contains many lines with the same structural form.

7.3 Using `regex_token_iterator` for Tokenization Tasks

The standard library also provides `regex_token_iterator`, which is designed for iterating over selected submatches or over the text between matches. Its string specialization is commonly used for tokenization tasks.
This iterator is especially helpful when the goal is not to process the whole match, but instead:

- extract one specific captured group from each match,
- split text using a regex delimiter,
- iterate over the unmatched fragments between separators.

A special submatch index value of `-1` instructs the iterator to return the text between matches, making it useful for splitting operations.

7.3.1 Example: Extracting Only One Captured Group from Each Match

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

int main() {
    std::string text =
        "Name=Ayman; Name=Sara; Name=Omar;";

    std::regex pattern(R"(Name=(\w+))");

    std::sregex_token_iterator it(text.begin(), text.end(), pattern, 1);
    std::sregex_token_iterator end;

    for (; it != end; ++it) {
        std::cout << "Captured name: " << *it << '\n';
    }

    return 0;
}
```

Here, the iterator returns only the first captured group from each match, not the whole matched fragment.

7.4 Practical Techniques for Splitting and Filtering Text

A very common text-processing task is splitting input into tokens. With `regex_token_iterator`, the delimiter is described as a regex, and the iterator yields the segments between delimiters. This is more flexible than splitting on a single fixed character because the separator can include spaces, commas, semicolons, tabs, or combinations of them.

7.4.1 Example: Splitting a Comma-Separated List with Optional Spaces

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

int main() {
    std::string text = "red, blue,green,   yellow";
    std::regex delimiter(R"(\s*,\s*)");

    std::sregex_token_iterator it(text.begin(), text.end(), delimiter, -1);
    std::sregex_token_iterator end;

    for (; it != end; ++it) {
        std::cout << "Token: [" << *it << "]\n";
    }

    return 0;
}
```

This example splits the text and removes the separator from the result.

7.4.2 Example: Splitting Mixed Separators

This style is useful when the input is inconsistent and one delimiter rule must cover several separator forms.

7.4.3 Example: Filtering File Extensions from a Text Block

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

```
int main() {
    std::string text =
        "report.txt image.png notes.txt archive.zip draft.txt";

    std::regex pattern(R"((\w+)\.(txt))");

    std::sregex_iterator it(text.begin(), text.end(), pattern);
    std::sregex_iterator end;

    for (; it != end; ++it) {
        std::cout << "Text file base name: " << (*it)[1] << '\n';
    }

    return 0;
}
```

This is a simple filtering pattern where only selected matches are processed.

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter7_multiple_matches.cpp
```

Recent Visual Studio toolchains can also be configured for newer language modes through the corresponding project setting or compiler option.

Chapter Summary

When a pattern may appear many times in the same input, `sregex_iterator` is the standard C++ tool for visiting every match in sequence. It is especially useful for large text blocks where each occurrence must be examined or extracted. For tokenization and splitting tasks, `regex_token_iterator` provides a flexible solution by returning selected submatches or the

text between matches. Together, these iterators make the standard regex library suitable not only for one-time matching, but also for practical large-scale text scanning and filtering.

Writing Efficient and Performant Regex Code in C++

8.1 Understanding the Cost of Compiling Regex Patterns

In the C++ standard library, a regular expression is represented by `std::regex`, which stores a compiled form of the pattern. Constructing a `std::regex` object is not a trivial operation. The implementation must parse the pattern, validate syntax, and prepare an internal representation suitable for matching.

This means that pattern construction has a noticeable cost compared to simple string operations. If a pattern is constructed repeatedly, especially inside loops or frequently executed functions, the total cost can become significant.

For this reason, the design model of the library assumes that a pattern is typically constructed once and reused many times.

8.1.1 Example: Repeated Compilation Inside a Loop

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

```
int main() {
    std::vector<std::string> data = {
        "ID 1001", "ID 2048", "ID 3055"
    };

    for (const auto& value : data) {
        std::regex pattern(R"(\d+)");
        if (std::regex_search(value, pattern)) {
            std::cout << value << '\n';
        }
    }

    return 0;
}
```

In this example, the pattern is compiled again for every iteration. This is inefficient because the pattern does not change.

8.2 Reusing Compiled Patterns to Improve Performance

A better approach is to construct the pattern once and reuse it. This reduces repeated parsing and preparation cost and aligns with the intended usage of the standard library.
Patterns can be reused by storing them in local static variables, function-level static variables, or objects that live across multiple calls.

8.2.1 Example: Reusing a Compiled Pattern

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

```cpp
#include <vector>

int main() {
    std::vector<std::string> data = {
        "ID 1001", "ID 2048", "ID 3055"
    };

    std::regex pattern(R"(\d+)");

    for (const auto& value : data) {
        if (std::regex_search(value, pattern)) {
            std::cout << value << '\n';
        }
    }

    return 0;
}
```

This version avoids repeated compilation and is more efficient for repeated operations.

8.2.2 Example: Static Pattern for Repeated Validation

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

bool is_valid_id(const std::string& value) {
    static const std::regex pattern(R"([A-Z]{2}\d{6})");
    return std::regex_match(value, pattern);
}

int main() {
    std::cout << std::boolalpha;
```

```
    std::cout << is_valid_id("AB123456") << '\n';
    std::cout << is_valid_id("XX999999") << '\n';
    return 0;
}
```

The pattern is created once and reused across all calls.

8.3 Avoiding Common Performance Mistakes in Loops

Performance problems with regex are often caused by incorrect placement of pattern construction or unnecessary repeated work. Common mistakes include:

- constructing `std::regex` inside tight loops,
- recompiling the same pattern in frequently called functions,
- performing repeated full scans of large text without need,
- using complex patterns where simpler logic would suffice.

A practical guideline is to separate pattern construction from pattern usage. The pattern should be created before the loop, and the loop should only apply it.

8.3.1 Example: Incorrect and Correct Loop Usage

```
// Incorrect
for (const auto& line : lines) {
    std::regex pattern(R"(ERROR)");
    if (std::regex_search(line, pattern)) {
        // process
    }
}
```

```
// Correct
std::regex pattern(R"(ERROR)");
for (const auto& line : lines) {
    if (std::regex_search(line, pattern)) {
        // process
    }
}
```

The second version avoids repeated construction and is the recommended approach.

8.4 Recognizing When Regex Becomes a Bottleneck

Although regex is powerful, it is not always the fastest option. In some cases, especially when processing very large datasets or running inside performance-critical code paths, regex operations can become a bottleneck.
Situations where this may occur include:

- scanning very large text repeatedly,
- using overly complex patterns,
- performing many matches in tight loops,
- using regex where a simple comparison would be sufficient.

When performance is critical, it is important to measure behavior in real code and consider whether the pattern can be simplified or replaced.

8.4.1 Example: Regex vs Simple Search

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

```
int main() {
    std::string text = "System ERROR occurred";

    // Simple search
    if (text.find("ERROR") != std::string::npos) {
        std::cout << "Found ERROR\n";
    }

    return 0;
}
```

In this case, a simple substring search is more efficient and easier to read than a regex.

8.5 Choosing Simpler Alternatives When Regex Is Not Needed

A key principle in Modern C++ design is to choose the simplest tool that correctly solves the problem. Regex should be used when pattern flexibility is required, but not when the task is straightforward.
Simpler alternatives include:

- direct string comparison,
- `std::string::find`,
- prefix and suffix checks,
- manual parsing when structure is fixed and simple.

Using simpler operations improves readability and often reduces runtime overhead.

8.5.1 Example: Checking a File Extension Without Regex

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

bool is_text_file(const std::string& name) {
    return name.size() >= 4 &&
           name.substr(name.size() - 4) == ".txt";
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_text_file("report.txt") << '\n';
    std::cout << is_text_file("image.png") << '\n';
    return 0;
}
```

This solution is simpler and avoids unnecessary regex overhead.

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter8_performance.cpp
```

Recent Visual Studio toolchains can also be configured for newer standard modes through the corresponding project setting or compiler option.

Chapter Summary

Efficient use of regex in Modern C++ depends on understanding that pattern construction has a cost and should be minimized. Patterns should be compiled once and reused whenever possible. Avoid placing regex construction inside loops and avoid unnecessary repeated scans.

When performance is critical, consider whether regex is the correct tool or whether simpler string operations provide a clearer and faster solution. Proper usage leads to code that is both efficient and maintainable.

Solving Real-World Problems Using Regex in C++

9.1 Parsing and Analyzing Log Files

Log files are one of the most common places where regular expressions are useful. A log line often follows a repeated textual structure that contains fields such as a timestamp, severity level, and free-form message. In such cases, a regex pattern can both validate the general structure and extract the useful fields into separate values.
A practical design is to use capturing groups for each logical field. This keeps the extraction logic compact and readable.

9.1.1 Example: Parsing a Simple Log Line

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

struct LogRecord {
    std::string timestamp;
    std::string level;
    std::string message;
```

```
};

int main() {
    std::string line =
        "[2026-03-24 21:10:45] ERROR Disk quota exceeded";

    std::regex pattern(R"(\[(.*?)\]\s+([A-Z]+)\s+(.*))");
    std::smatch match;

    if (std::regex_match(line, match, pattern)) {
        LogRecord record;
        record.timestamp = match[1].str();
        record.level = match[2].str();
        record.message = match[3].str();

        std::cout << "Timestamp: " << record.timestamp << '\n';
        std::cout << "Level    : " << record.level << '\n';
        std::cout << "Message  : " << record.message << '\n';
    } else {
        std::cout << "Log format mismatch.\n";
    }

    return 0;
}
```

This pattern is suitable because the structure is regular and each part can be captured directly.

9.1.2 Example: Extracting Multiple Log Entries from a Large Block

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

```
int main() {
    std::string logs =
        "[2026-03-24 08:00:01] INFO Service started\n"
        "[2026-03-24 08:02:15] WARN Cache miss\n"
        "[2026-03-24 08:03:41] ERROR Disk full\n";

    std::regex pattern(R"(\[(.*?)\]\s+([A-Z]+)\s+(.*))");

    std::sregex_iterator it(logs.begin(), logs.end(), pattern);
    std::sregex_iterator end;

    for (; it != end; ++it) {
        std::cout << "Time    : " << (*it)[1] << '\n';
        std::cout << "Level   : " << (*it)[2] << '\n';
        std::cout << "Message : " << (*it)[3] << '\n';
        std::cout << '\n';
    }

    return 0;
}
```

9.2 Extracting Emails, Numbers, and Identifiers from Text

Many real applications need to discover useful values inside larger text rather than validate the entire string. In such situations, `regex_search` is usually the correct operation for one match, while `sregex_iterator` is suitable for repeated matches.

9.2.1 Example: Extracting an Email-Like Address

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

```
int main() {
    std::string text = "Please contact support@example.com today.";
    std::regex pattern(R"(([A-Za-z0-9._%+-]+)@([A-Za-z0-9.-]+\.[A-Za-z]{2,}))");
    std::smatch match;

    if (std::regex_search(text, match, pattern)) {
        std::cout << "Full address: " << match[0] << '\n';
        std::cout << "Local part  : " << match[1] << '\n';
        std::cout << "Domain      : " << match[2] << '\n';
    } else {
        std::cout << "No email-like address found.\n";
    }

    return 0;
}
```

9.2.2 Example: Extracting All Numbers from a Text Block

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

int main() {
    std::string text = "Orders: 104, 275, 389, 450";
    std::regex pattern(R"(\d+)");

    std::sregex_iterator it(text.begin(), text.end(), pattern);
    std::sregex_iterator end;

    for (; it != end; ++it) {
        std::cout << "Number: " << (*it).str() << '\n';
    }
```

```
    return 0;
}
```

9.2.3 Example: Extracting a Structured Identifier

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

int main() {
    std::string line = "Shipment code AB123456 was accepted.";
    std::regex pattern(R"(([A-Z]{2}\d{6}))");
    std::smatch match;

    if (std::regex_search(line, match, pattern)) {
        std::cout << "Identifier: " << match[1] << '\n';
    } else {
        std::cout << "No identifier found.\n";
    }

    return 0;
}
```

9.3 Parsing Key-Value Data and Configuration Lines

Key-value text is common in configuration files, command output, and small data-exchange formats. Regex is useful here because the line structure is simple and repetitive. A pattern can separate the key from the value while tolerating spaces around the separator.

9.3.1 Example: Parsing a Key-Value Line

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

int main() {
    std::string line = "port = 5432";
    std::regex pattern(R"(\s*([A-Za-z_]\w*)\s*=\s*(.+)\s*)");
    std::smatch match;

    if (std::regex_match(line, match, pattern)) {
        std::cout << "Key   : " << match[1] << '\n';
        std::cout << "Value : " << match[2] << '\n';
    } else {
        std::cout << "Configuration line mismatch.\n";
    }

    return 0;
}
```

9.3.2 Example: Parsing Multiple Configuration Lines

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

int main() {
    std::vector<std::string> lines = {
        "host = server01",
        "port = 5432",
        "mode = readonly"
```

```
    };

    std::regex pattern(R"(\s*([A-Za-z_]\w*)\s*=\s*(.+)\s*)");

    for (const auto& line : lines) {
        std::smatch match;
        if (std::regex_match(line, match, pattern)) {
            std::cout << "Key   : " << match[1] << '\n';
            std::cout << "Value : " << match[2] << '\n';
            std::cout << '\n';
        }
    }

    return 0;
}
```

9.4 Validating User Input with Clear and Maintainable Patterns

Validation is one of the best uses of `regex_match`, because it checks whether the entire input satisfies the intended rule. The most maintainable validation patterns are usually the simplest ones that clearly express the constraint.

9.4.1 Example: Validating a Username

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

bool is_valid_username(const std::string& name) {
    static const std::regex pattern(R"([A-Za-z][A-Za-z0-9_]{2,15})");
```

```
    return std::regex_match(name, pattern);
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_valid_username("Ayman_77") << '\n';
    std::cout << is_valid_username("7Ayman") << '\n';
    std::cout << is_valid_username("ab") << '\n';
    return 0;
}
```

9.4.2 Example: Validating a Product Code

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

bool is_valid_product_code(const std::string& code) {
    static const std::regex pattern(R"([A-Z]{3}-\d{4})");
    return std::regex_match(code, pattern);
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_valid_product_code("ABC-2048") << '\n';
    std::cout << is_valid_product_code("ABC-20") << '\n';
    std::cout << is_valid_product_code("X-2048") << '\n';
    return 0;
}
```

These patterns are readable because they state the structural rule directly.

9.5 Cleaning and Normalizing Data Before Processing

Before data is stored, parsed, or compared, it is often useful to normalize formatting. The standard `regex_replace` function is suitable for this because it can transform matching fragments while preserving the rest of the text.

9.5.1 Example: Reducing Repeated Spaces

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

int main() {
    std::string text = "C++     regex    can   clean   text.";
    std::regex pattern(R"(\s+)");

    std::string result = std::regex_replace(text, pattern, " ");

    std::cout << result << '\n';
    return 0;
}
```

9.5.2 Example: Normalizing a Phone Number to Digits Only

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

int main() {
    std::string phone = "(555) 120-9988";
    std::regex pattern(R"([^\d])");
```

```
    std::string compact = std::regex_replace(phone, pattern, "");

    std::cout << compact << '\n';
    return 0;
}
```

9.5.3 Example: Normalizing Mixed Separators

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

int main() {
    std::string text = "red ; blue,green ;  yellow";
    std::regex pattern(R"(\s*[,;]\s*)");

    std::string result = std::regex_replace(text, pattern, ", ");

    std::cout << result << '\n';
    return 0;
}
```

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter9_real_world_regex.cpp
```

Recent Visual Studio toolchains can also be configured for newer language modes through the corresponding project setting or compiler option.

Chapter Summary

Regex becomes most valuable when it is tied to practical software work. It can parse repeated log formats, extract email-like addresses and identifiers from larger text, process key-value configuration lines, validate structured user input, and normalize imperfect data before later processing. In Modern C++, these tasks are handled through the standard library by combining `std::regex`, match results, iterators, and replacement operations in a clear and maintainable way.

Avoiding Common Errors When Working with Regex

10.1 Understanding Common Escaping Mistakes in C++ Strings

One of the most common regex errors in C++ comes from forgetting that the pattern is written inside a C++ string literal. This creates two interpretation layers:

- the C++ compiler interprets the string literal,
- the regex engine interprets the final string as a pattern.

Because of this, a backslash intended for the regex engine often needs to be escaped for the C++ compiler. For example, a digit class written as `\d+` in regex is commonly written as `"\\d+"` in a normal C++ string literal. Raw string literals avoid most of this confusion and usually make regex code easier to read and maintain.

10.1.1 Example: Incorrect Style and Better Style

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

```cpp
#include <string>

int main() {
    std::string text = "User 48291";

    std::regex normalStyle("\\d+");
    std::regex rawStyle(R"(\d+)");

    std::cout << std::boolalpha;
    std::cout << std::regex_search(text, normalStyle) << '\n';
    std::cout << std::regex_search(text, rawStyle) << '\n';

    return 0;
}
```

The second form is usually easier to understand because the regex pattern appears almost exactly as intended.

10.2 Avoiding Incorrect Use of Matching Functions

Another common mistake is confusing `regex_match` with `regex_search`. In the C++ standard library, `regex_match` tests whether the entire target string matches the pattern, while `regex_search` looks for a matching part inside a larger string.
This means:

- use `regex_match` for strict validation,
- use `regex_search` for finding and extracting data inside larger text.

Choosing the wrong function can make otherwise correct patterns behave incorrectly.

10.2.1 Example: Same Pattern, Wrong Function Choice

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

int main() {
    std::string text = "Order ID: 73145";
    std::regex pattern(R"(\d+)");

    std::cout << std::boolalpha;
    std::cout << "regex_match  : "
              << std::regex_match(text, pattern) << '\n';
    std::cout << "regex_search : "
              << std::regex_search(text, pattern) << '\n';

    return 0;
}
```

Here, `regex_match` returns false because the whole string is not only digits, while `regex_search` succeeds because a digit sequence exists inside the larger text.

10.3 Simplifying Overly Complex and Hard-to-Read Patterns

A regex can be technically correct and still be a poor engineering choice if it is too hard to read, maintain, or debug. Overly dense patterns often cause mistakes because the structure becomes difficult to verify.
A good pattern should:

- express only the needed rule,
- avoid unnecessary wildcards,

- use clear groups,
- remain readable enough to be reviewed later.

In many cases, a slightly longer but clearer pattern is better than a compact pattern that is difficult to understand.

10.3.1 Example: Broad Pattern Versus Clearer Pattern

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

int main() {
    std::string line = "[2026-03-24] ERROR Disk full";

    std::regex broadPattern(R"((.*))");
    std::regex clearPattern(R"(\[(\d{4}-\d{2}-\d{2})\]\s+([A-Z]+)\s+(.*))");

    std::smatch match;

    if (std::regex_match(line, match, clearPattern)) {
        std::cout << "Date    : " << match[1] << '\n';
        std::cout << "Level   : " << match[2] << '\n';
        std::cout << "Message : " << match[3] << '\n';
    }

    return 0;
}
```

The clearer pattern describes the intended structure directly and makes extraction easier.

10.3.2 Example: Replacing a Greedy Wildcard with a Safer Form

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

int main() {
    std::string text = "<tag>first</tag><tag>second</tag>";

    std::regex greedy(R"(<tag>.*</tag>)");
    std::regex safer(R"(<tag>[^<]*</tag>)");

    std::smatch match;

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

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

    return 0;
}
```

The safer version is more precise because it constrains what may appear between the opening and closing tags.

10.4 Identifying and Fixing Performance Issues

Regex performance problems often come from how the code is written, not only from the pattern itself. One of the most common mistakes is constructing the same `std::regex` object

repeatedly inside a loop. Since a regex object represents a compiled pattern, repeated construction adds avoidable cost.

A better design is to construct the regex once and reuse it.

10.4.1 Example: Inefficient and Efficient Pattern Usage

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

int main() {
    std::vector<std::string> lines = {
        "ERROR Disk full",
        "INFO Service started",
        "ERROR Quota exceeded"
    };

    std::cout << "Inefficient style:\n";
    for (const auto& line : lines) {
        std::regex pattern(R"(ERROR)");
        if (std::regex_search(line, pattern)) {
            std::cout << line << '\n';
        }
    }

    std::cout << "\nEfficient style:\n";
    std::regex reusedPattern(R"(ERROR)");
    for (const auto& line : lines) {
        if (std::regex_search(line, reusedPattern)) {
            std::cout << line << '\n';
        }
    }
```

```
    return 0;
}
```

The second style is preferred because the pattern is compiled once and reused for all inputs.

10.4.2 Example: Using a Simpler Alternative When Regex Is Not Needed

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

int main() {
    std::string text = "System ERROR occurred";

    if (text.find("ERROR") != std::string::npos) {
        std::cout << "Found ERROR\n";
    }

    return 0;
}
```

If the task is only to find a fixed substring, a simple string operation is usually clearer and more efficient than regex.

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter10_common_regex_errors.cpp
```

Recent Visual Studio toolchains can also be configured for newer language modes through the corresponding project setting or compiler option.

Chapter Summary

Most regex mistakes in Modern C++ come from four sources: escaping confusion inside ordinary string literals, choosing the wrong matching function, writing patterns that are too broad or too complex, and introducing unnecessary performance cost through repeated pattern construction. These problems are best avoided by using raw string literals where possible, selecting `regex_match` or `regex_search` according to the real task, keeping patterns readable, and reusing compiled regex objects instead of rebuilding them repeatedly.

Building a Practical Reference for Daily Use

11.1 A Collection of Commonly Used Regex Patterns

A practical regex reference is most useful when it contains patterns that appear often in real software. In Modern C++, these patterns are typically stored in `std::regex` objects and applied through `regex_match`, `regex_search`, or `regex_replace`. The examples below use the default ECMAScript grammar supported by the standard library.

11.1.1 Example: Common Validation and Extraction Patterns

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

int main() {
    std::vector<std::pair<std::string, std::regex>> patterns = {
        {"Digits only", std::regex(R"(\d+)")},
        {"Simple date YYYY-MM-DD", std::regex(R"(\d{4}-\d{2}-\d{2})")},
        {"Username", std::regex(R"([A-Za-z][A-Za-z0-9_]{2,15})")},
```

```
        {"Product code", std::regex(R"([A-Z]{3}-\d{4})")},
        {"Identifier", std::regex(R"([A-Z]{2}\d{6})")},
        {"Key-value line", std::regex(R"(\s*([A-Za-z_]\w*)\s*=\s*(.+)\s*)")},
        {"Email-like address",
         std::regex(R"(([A-Za-z0-9._%+-]+)@([A-Za-z0-9.-]+\.[A-Za-z]{2,}))")}
    };

    std::string value = "support@example.com";

    for (const auto& item : patterns) {
        std::cout << item.first << " : "
                  << std::boolalpha
                  << std::regex_match(value, item.second) << '\n';
    }

    return 0;
}
```

Useful daily patterns include:

- `R"(\d+)"` for one or more digits,
- `R"([A-Za-z][A-Za-z0-9_]{2,15})"` for a simple username,
- `R"((\d{4})-(\d{2})-(\d{2}))"` for a simple date,
- `R"([A-Z]{3}-\d{4})"` for a product code,
- `R"(\s*([A-Za-z_]\w*)\s*=\s*(.+)\s*)"` for key-value lines,
- `R"(([A-Za-z0-9._%+-]+)@([A-Za-z0-9.-]+\.[A-Za-z]{2,}))"` for an email-like address.

These patterns are intentionally practical rather than overly broad. A daily reference should prefer clarity and maintainability over trying to cover every possible text variation.

11.2 A Simple and Readable Syntax Cheat Sheet

The most useful regex reference is not a long theoretical catalog. It is a short set of syntax elements that developers use repeatedly in real code.
Common syntax elements include:

- `.` for any character in the selected grammar context,
- `[abc]` for one character chosen from a set,
- `[A-Z]` for one uppercase letter,
- `\d` for a digit,
- `\w` for a word character,
- `\s` for whitespace,
- `*` for zero or more,
- `+` for one or more,
- `?` for zero or one,
- `{n}` for exactly `n`,
- `{n,m}` for from `n` to `m`,
- `^` for the beginning of the sequence,
- `$` for the end of the sequence,
- `( )` for grouping and capturing,
- `|` for alternation.

11.2.1 Example: Small Syntax Demonstration

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

int main() {
    std::string value = "AB123456";

    std::regex exactIdentifier(R"(^[A-Z]{2}\d{6}$)");
    std::regex partialDigits(R"(\d+)");
    std::regex alternative(R"(ERROR|WARN|INFO)");

    std::cout << std::boolalpha;
    std::cout << "Exact identifier : "
              << std::regex_match(value, exactIdentifier) << '\n';
    std::cout << "Contains digits   : "
              << std::regex_search(value, partialDigits) << '\n';
    std::cout << "Matches log level : "
              << std::regex_match(std::string("WARN"), alternative) << '\n';

    return 0;
}
```

A readable cheat sheet should help the programmer answer practical questions quickly:

- Do I need a full match or a partial search
- Do I need one character, many characters, or an optional part
- Do I need grouping for extraction
- Do I need anchors to restrict the pattern

11.3 Best Practices for Writing Maintainable Regex in C++

Maintainable regex code depends on more than syntax correctness. The C++ standard library gives powerful matching tools, but real quality comes from how patterns are written, stored, and reused.

A practical set of best practices includes:

- prefer raw string literals for readability,
- construct a pattern once and reuse it when possible,
- use `regex_match` for strict validation,
- use `regex_search` for discovery and extraction,
- keep patterns as narrow and explicit as possible,
- avoid unnecessary wildcards such as `.*` when a safer character class is clearer,
- test optional groups carefully before using them,
- choose simpler string operations when regex is not needed.

11.3.1 Example: Readable and Reusable Validation Pattern

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

bool is_valid_username(const std::string& name) {
    static const std::regex pattern(R"([A-Za-z][A-Za-z0-9_]{2,15})");
    return std::regex_match(name, pattern);
}
```

```
int main() {
    std::cout << std::boolalpha;
    std::cout << is_valid_username("Ayman_77") << '\n';
    std::cout << is_valid_username("7Ayman") << '\n';
    return 0;
}
```

11.3.2 Example: Prefer Simpler String Logic When Appropriate

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

bool is_text_file(const std::string& name) {
    return name.size() >= 4 &&
           name.substr(name.size() - 4) == ".txt";
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_text_file("report.txt") << '\n';
    std::cout << is_text_file("image.png") << '\n';
    return 0;
}
```

11.3.3 Example: Safer Pattern Instead of a Broad Wildcard

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

int main() {
    std::string text = "<tag>first</tag><tag>second</tag>";
```

```
    std::regex safer(R"(<tag>[^<]*</tag>)");
    std::smatch match;

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

    return 0;
}
```

This style is easier to maintain because it states the intended structure more precisely than a broad greedy wildcard.

Windows Compilation Example

```
cl /EHsc /std:c++17 chapter11_reference.cpp
```

Recent Visual Studio toolchains support selecting newer language modes through the `/std:` compiler option in the command line or project settings. The ‘<regex>‘ algorithms documented by Microsoft include full matching, searching, replacing, and iterator-based processing, which together form the core daily-use toolkit for standard C++ regex work.

Chapter Summary

A practical regex reference for daily use should focus on clear patterns, a short syntax reminder, and a small set of maintainability rules. The most valuable habits are to write readable raw-string patterns, choose the correct matching algorithm, reuse compiled regex objects, avoid unnecessarily broad expressions, and prefer simpler string operations when the problem does not require full regex power.

Conclusion

Key Lessons and Practical Takeaways from Using Regex in C++

Regular expressions in Modern C++ provide a standardized and powerful mechanism for matching, searching, extracting, and transforming text using the `<regex>` library. When used correctly, they allow developers to express complex textual rules in a compact and maintainable form.

The most important lesson is understanding the distinction between full matching and partial searching. The function `regex_match` must be used when the entire input is expected to follow a strict rule, while `regex_search` is the correct tool for discovering patterns inside larger text. Choosing the correct function is often more important than changing the pattern itself.

Another key principle is treating `std::regex` as a compiled object. Pattern construction has a cost, and repeated construction inside loops should be avoided. A pattern should be created once and reused whenever possible, especially in performance-sensitive code.

Clarity of pattern design is equally important. Raw string literals should be preferred because they reduce escaping complexity and make patterns easier to read. Patterns should be written to reflect the intended structure clearly, avoiding unnecessary wildcards and overly complex expressions that are difficult to maintain.

Capturing groups are a central feature for practical use. They allow a program to convert raw text into structured data by extracting meaningful parts such as identifiers, dates, keys, or values. Combined with `match_results`, this enables direct access to the full match, submatches, and surrounding text.

Regex iterators provide an efficient way to process multiple matches in large inputs. The use of `sregex_iterator` enables scanning entire text blocks, while `regex_token_iterator` supports tokenization and splitting tasks based on flexible patterns.

The use of `regex_replace` allows transformation of text using captured groups and replacement formats. This makes it suitable for normalization, formatting, masking, and cleanup operations before further processing.

At the same time, regex should not be used unnecessarily. For simple tasks such as fixed substring detection or suffix checking, standard string operations are often clearer and more efficient. A good design always selects the simplest correct solution.

Example: Combining Validation and Extraction

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

int main() {
    std::string input = "User: Ayman, ID: 48291";
    std::regex pattern(R"(User:\s*(\w+),\s*ID:\s*(\d+))");
    std::smatch match;

    if (std::regex_search(input, match, pattern)) {
        std::cout << "Name : " << match[1] << '\n';
        std::cout << "ID   : " << match[2] << '\n';
    } else {
        std::cout << "Input format mismatch.\n";
```

```
    }

    return 0;
}
```

This example demonstrates how a single pattern can locate structured information and extract it into usable program data.

Example: Reusing a Pattern Efficiently

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

int main() {
    std::vector<std::string> lines = {
        "ERROR Disk full",
        "INFO Service started",
        "ERROR Quota exceeded"
    };

    std::regex pattern(R"(ERROR)");

    for (const auto& line : lines) {
        if (std::regex_search(line, pattern)) {
            std::cout << line << '\n';
        }
    }

    return 0;
}
```

This reinforces the principle of compiling once and reusing the pattern across multiple inputs.

Final Note

The standard <regex> library in C++ provides a complete set of tools for working with text patterns. Its effective use depends on understanding its design, selecting the correct operations, writing clear patterns, and applying them with attention to performance and maintainability.

When these principles are followed, regular expressions become a reliable and practical tool in Modern C++ software development.

Appendices

Additional Patterns, Tools, and Extended Examples

This appendix provides a compact practical reference for additional regex patterns, standard tools, and extended examples that are useful in day-to-day Modern C++ work. The focus remains on the standard `<regex>` library and on examples that compile cleanly in a Windows environment.

Additional Practical Patterns

The following patterns are frequently useful in real software:

- Simple year-month-day date: `R"((\d{4})-(\d{2})-(\d{2}))"`
- Simple username: `R"([A-Za-z][A-Za-z0-9_]{2,15})"`
- Product code: `R"([A-Z]{3}-\d{4})"`
- Two-letter identifier with digits: `R"([A-Z]{2}\d{6})"`
- Key-value pair: `R"(\s*([A-Za-z_]\w*)\s*=\s*(.+)\s*)"`
- Repeated spaces: `R"(\s+)"`

- Mixed separators such as comma or semicolon: `R"(\s*[,;]\s*)"`
- Digits only: `R"(\d+)"`
- Simple file extension extraction: `R"((\w+)\.(\w+))"`

These forms are intentionally simple and readable. In practical C++ work, a maintainable pattern is usually better than an overly broad pattern that attempts to cover every possible variation.

Standard Tools Worth Remembering

The standard regex toolkit in C++ is centered on a small set of components:

- `std::regex` for storing the compiled pattern
- `std::smatch` for storing full matches and captured groups when working with `std::string`
- `regex_match` for full-string validation
- `regex_search` for finding a matching fragment inside larger text
- `regex_replace` for rewriting text
- `sregex_iterator` for iterating over all matches
- `sregex_token_iterator` for extracting selected groups or splitting text

A practical habit is to think of these tools as a small workflow:

- define a readable pattern,
- choose validation, search, replacement, or iteration,

- extract only the parts that matter,
- reuse the pattern when the same rule is applied repeatedly.

Extended Example: Extracting Every Identifier from a Report

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

int main() {
    std::string report =
        "Accepted IDs: AB123456, CD654321, EF111222";

    std::regex pattern(R"(([A-Z]{2}\d{6}))");

    std::sregex_iterator it(report.begin(), report.end(), pattern);
    std::sregex_iterator end;

    for (; it != end; ++it) {
        std::cout << "Identifier: " << (*it)[1] << '\n';
    }

    return 0;
}
```

This example demonstrates iterator-based extraction from a larger text block.

Extended Example: Rewriting a Key-Value File into Normalized Form

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

```
#include <vector>

int main() {
    std::vector<std::string> lines = {
        " host   =   server01 ",
        " port=5432",
        " mode   = readonly "
    };

    std::regex pattern(R"(\s*([A-Za-z_]\w*)\s*=\s*(.*?)\s*)");

    for (const auto& line : lines) {
        std::string normalized =
            std::regex_replace(line, pattern, "$1=$2");
        std::cout << normalized << '\n';
    }

    return 0;
}
```

This is useful when imported text contains inconsistent spacing around separators.

Extended Example: Splitting Text with Multiple Delimiters

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

int main() {
    std::string text = "red; blue, green   yellow";
    std::regex delimiter(R"([,;\s]+)");

    std::sregex_token_iterator it(text.begin(), text.end(), delimiter, -1);
```

```
    std::sregex_token_iterator end;

    for (; it != end; ++it) {
        if (!it->str().empty()) {
            std::cout << "Token: " << *it << '\n';
        }
    }

    return 0;
}
```

This is a compact standard-library way to split text when one fixed separator is not enough.

Extended Example: Extracting Structured Log Fields

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

struct LogRecord {
    std::string date;
    std::string level;
    std::string message;
};

int main() {
    std::string line =
        "[2026-03-24 21:10:45] ERROR Disk quota exceeded";

    std::regex pattern(R"(\[(.*?)\]\s+([A-Z]+)\s+(.*))");
    std::smatch match;

    if (std::regex_match(line, match, pattern)) {
```

```
        LogRecord record;
        record.date = match[1].str();
        record.level = match[2].str();
        record.message = match[3].str();

        std::cout << "Date    : " << record.date << '\n';
        std::cout << "Level   : " << record.level << '\n';
        std::cout << "Message : " << record.message << '\n';
    } else {
        std::cout << "Format mismatch.\n";
    }

    return 0;
}
```

This is a useful example because it combines validation and structured extraction in one step.

Extended Example: Cleaning a Phone Number Before Storage

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

int main() {
    std::string phone = "(555) 120-9988";
    std::regex pattern(R"([^\d])");

    std::string cleaned = std::regex_replace(phone, pattern, "");

    std::cout << cleaned << '\n';
    return 0;
}
```

A normalized value is often easier to store, compare, and validate later.

Short Practical Checklist

When writing regex code in Modern C++, the following checklist is useful:

- Prefer raw string literals for readability.
- Use `regex_match` only when the whole input must match.
- Use `regex_search` when the pattern may appear inside larger text.
- Reuse `std::regex` objects instead of rebuilding them unnecessarily.
- Use captured groups only when extracted parts are actually needed.
- Keep patterns narrow and explicit whenever possible.
- Prefer ordinary string operations when the task is simple.

Windows Compilation Examples

```
cl /EHsc /std:c++17 appendix_regex_examples.cpp
```

```
cl /EHsc /std:c++20 appendix_regex_examples.cpp
```

Recent Visual Studio toolchains can also be configured for newer standard modes through the project properties or the corresponding `/std:` compiler option.

Appendix Summary

This appendix extends the main chapters with additional patterns, a compact reminder of the core standard tools, and a set of practical examples for extraction, normalization, tokenization, and structured parsing. Together, these examples form a useful working reference for everyday regex tasks in Modern C++.

References

Official Documentation and Further Reading Sources

This section provides a concise list of authoritative sources for the C++ <regex> library and related language features. These sources represent the official definitions, standard behavior, and widely accepted references for Modern C++ development from C++11 through C++23.

C++ Standard Draft and Specification

- The C++ Standard Draft, section on regular expressions, defines:
  - basic_regex and its specializations
  - match_results and sub_match
  - algorithms such as regex_match, regex_search, and regex_replace
  - iterators including regex_iterator and regex_token_iterator
  - syntax option flags and grammar definitions
- The draft also specifies behavior for ECMAScript, basic, extended, awk, grep, and egrep grammars.

Microsoft C++ Standard Library Documentation

- Documentation for `<regex>` in the Microsoft C++ Standard Library describes:
  - construction and usage of `std::regex`
  - use of `std::smatch` and match result access
  - behavior of `regex_match`, `regex_search`, and `regex_replace`
  - usage of iterators for multiple matches
  - practical examples aligned with Windows development environments
- The Microsoft documentation also explains compiler options such as `/std:` for selecting the language standard.

C++ Reference Documentation

- Comprehensive reference descriptions for all regex components:
  - `std::regex` and pattern construction
  - `std::match_results` and group access
  - matching and searching algorithms
  - replacement rules and formatting behavior
  - iterator-based processing of multiple matches
- Detailed explanations of syntax elements, including character classes, quantifiers, anchors, and grouping behavior.

Compiler and Toolchain Documentation

- Official compiler documentation for Windows environments describes:

- enabling Modern C++ language standards using `/std:` options
- standard library support for `<regex>`
- build and compilation behavior using command-line tools

Example: Minimal Verified Usage of the Standard Library

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

int main() {
    std::string text = "Example 12345";
    std::regex pattern(R"(\d+)");
    std::smatch match;

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

    return 0;
}
```

Windows Compilation Example

```
cl /EHsc /std:c++17 references_example.cpp
```

Reference Summary

The sources listed in this section represent the official and authoritative foundation for using regular expressions in Modern C++. They define the behavior of the `<regex>` library, describe its algorithms and types, and provide guidance for correct and portable usage. For

reliable development, it is recommended to base all regex-related code on these standard definitions and documented behaviors.