https://simplifycpp.org

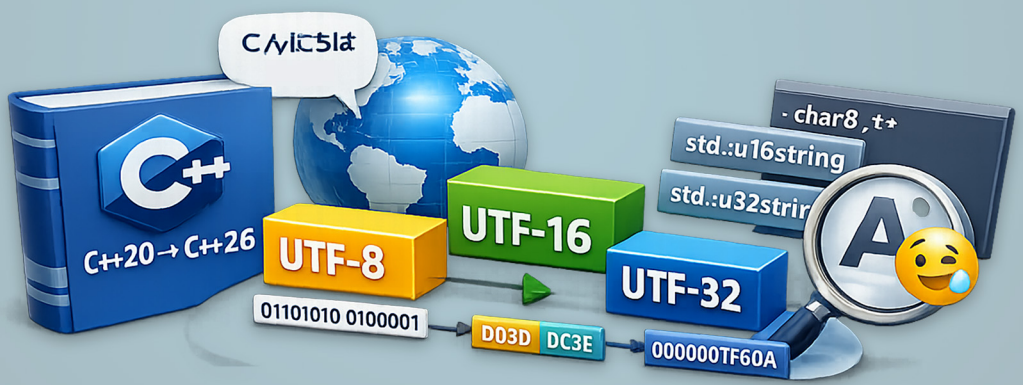
Unicode and Modern C++

Correct Text Handling from C++20 to C++26

Understanding Text Encoding in Modern C++

Reality, Limitations, and Solutions

DRAFT



Prepared by Ayman Alheraki

Unicode and Modern C++

Correct Text Handling from C++20 to C++26

Understanding Text Encoding in Modern C++: Reality, Limitations, and Solutions

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

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Author's Introduction

Why This Booklet Exists and Who It Is Written For

Why this booklet exists

Text is the most underestimated subsystem in modern software. In many codebases, "string handling" is treated as a solved problem until it fails in production: corrupted user names, broken search results, incorrect truncation, unreadable logs, mismatched identifiers, security bypasses, and platform-dependent bugs that appear only under specific languages and inputs. Unicode is not a niche requirement. It is a fundamental reality of modern computing:

- Operating systems, file systems, network protocols, databases, and web formats are built around Unicode text.
- Users expect correct behavior across languages, scripts, emojis, and complex writing systems.
- Internationalization is no longer optional; it is part of baseline product correctness.

C++ developers, especially those who build performance-critical systems, are frequently forced to confront text problems at a low level. Unlike ecosystems that hide text semantics behind a heavyweight runtime, C++ makes representation explicit and expects engineers to define encoding and semantic policies consciously. This booklet exists to turn that reality into an

advantage: when responsibilities are explicit, they can be engineered correctly, tested rigorously, and maintained over time.

What this booklet is (and is not)

This booklet is an engineering guide to Unicode correctness in modern C++ from C++20 through C++26. Its focus is not theoretical linguistics. It focuses on practical correctness properties that matter in real systems:

- distinguishing bytes, code units, code points, and grapheme clusters,
- choosing UTF-8 as a practical interchange and storage format,
- validating and handling ill-formed input at boundaries,
- transcoding reliably across OS and ABI boundaries (especially Windows UTF-16),
- understanding why indexing, slicing, and “length” are not trivial in Unicode,
- designing APIs that express encoding intent and semantic expectations,
- selecting and integrating professional Unicode libraries where the standard library stops,
- testing and maintaining Unicode behavior under evolving Unicode data and library upgrades.

This booklet is not a replacement for the Unicode Standard, its technical annexes, or comprehensive libraries such as ICU. Those are authoritative specifications and implementations; this booklet is a professional map that connects them to modern C++ engineering practice, explains the common failure modes, and presents reliable design patterns that scale.

The problem: why Unicode feels difficult in C++

Unicode becomes difficult in C++ mainly because developers incorrectly assume that a string is "a sequence of characters" with trivial indexing and length. In C++:

- `std::string` is a container of bytes; it does not promise any particular encoding.
- UTF-8 is variable-length, so byte indexing is not character indexing.
- UTF-16 uses surrogate pairs, so 16-bit indexing is not character indexing either.
- UTF-32 provides code points, but user-perceived characters (grapheme clusters) can still span multiple code points.
- comparisons that humans expect (case-insensitive, accent-insensitive, locale-aware) require Unicode algorithms and data, not byte comparisons.

These are not defects of C++. They are consequences of Unicode's real complexity and C++'s design priorities: explicitness, performance predictability, and long-term compatibility.

Who this booklet is written for

This booklet is written for engineers who:

- write C++ systems where correctness and performance both matter,
- integrate with modern file formats and network protocols that expect UTF-8,
- build cross-platform products that must behave consistently across operating systems and locales,
- design APIs and libraries that accept and produce text safely,
- maintain large codebases where text bugs are costly and hard to reproduce.

It is particularly useful for:

- C++ developers migrating legacy code that assumed ASCII or “local code pages”,
- engineers building backend systems (APIs, services, databases, logs, messaging),
- tool and infrastructure developers (compilers, build tools, editors, CLIs),
- UI and application developers who must handle grapheme-aware editing and rendering semantics.

Prerequisites and assumed background

The booklet assumes:

- familiarity with modern C++ (at least C++17 fundamentals),
- comfort with standard library containers, views, and resource management,
- basic understanding of byte sequences and binary I/O.

No prior Unicode expertise is assumed. The early chapters build the required terminology and correct mental model from first principles.

What you will be able to do after reading

After completing this booklet, you should be able to:

- correctly reason about text in terms of representation and semantics,
- select encodings appropriately by layer (storage, transport, interfaces, algorithms),
- validate UTF-8 input and define robust error-handling policies,
- avoid common UTF-8 and UTF-16 pitfalls in indexing, slicing, and iteration,

- design C++ APIs that make encoding intent explicit (especially using C++20 UTF-8 types),
- integrate Unicode libraries for normalization, segmentation, collation, and casing,
- build a Unicode test strategy that survives library upgrades and Unicode version changes,
- maintain long-lived Unicode-correct systems without fragile assumptions.

The philosophy of this booklet

This booklet follows a systems-engineering philosophy:

- **Text correctness is an engineering responsibility.** Unicode is not "handled automatically"; it must be designed.
- **Representation and semantics must be separated.** Code units are not characters; semantic operations are explicit and policy-driven.
- **Boundaries must be enforced.** Validation and transcoding belong at interfaces, not scattered across the codebase.
- **Policies must be documented and testable.** Normalization, collation, and casing choices are not trivia; they define product behavior.
- **Performance must be honest.** Hidden decoding costs and implicit conversions lead to both bugs and regressions.

The goal is not to make Unicode feel "simple" by hiding reality. The goal is to make Unicode manageable and reliable by exposing the right concepts, enforcing invariants, and applying the right tools at the right layers.

How to read this booklet

The recommended reading approach is:

- Read Chapters 1–3 carefully to internalize the terminology and the UTF-8 mental model.
- Use Chapters 4–6 as a reference when integrating with platform APIs and professional Unicode libraries.
- Treat Chapter 7 and the appendices as a checklist and long-term engineering guide for designing and maintaining Unicode-safe systems.

A short technical preview

To set expectations, here is the most important idea you will repeatedly see:

> In modern C++, strings store code units. Correct text behavior requires explicit policies and, when needed, decoding into semantic units such as code points and grapheme clusters.

```cpp
#include <string_view>

using BytesView = std::string_view;      // arbitrary bytes
using Utf8View  = std::u8string_view;    // UTF-8 code units by contract

void store_payload(BytesView bytes);     // no text promise
void render_label(Utf8View text);        // text contract (validate at boundary)
```

This single distinction prevents a large fraction of real-world Unicode defects.

Closing note

If you approach Unicode in C++ as a disciplined engineering problem—with explicit representation, explicit semantics, explicit policies, and strong testing—the topic becomes predictable. This booklet exists to provide that discipline in a form that is practical, rigorous, and directly applicable to modern C++20–C++26 systems.

Ayman Alheraki

Preface

Unicode, C++, and the Reality of Modern Text Processing

The ubiquity of text in software systems

Text is everywhere: user interfaces, network protocols, configuration files, logs, identifiers, metadata, search indexes, and persistence layers. What once appeared as simple sequences of characters has, in the era of global connectivity and multilingual usage, become an intricate substrate of cultural, linguistic, and semantic variation. This has profound implications for software engineers, especially those working in C++ where representation and semantics are not hidden by run-time layers but must be designed explicitly.

The transition from legacy code pages to Unicode marked a universal moment in computing: text could finally represent every language, every script, and every symbol needed in global systems. With this expressive power came unavoidable complexity. This booklet embraces that complexity and provides a rigorous, practical framework for handling it in C++ systems that must operate correctly, efficiently, and sustainably in the real world.

Why C++ developers face unique challenges with Unicode

Unlike some language ecosystems that embed text handling inside a managed run-time with implicit defaults, C++ places representation and behavior under the control of the engineer.

This is a deliberate design choice rooted in C++'s foundational goals:

- provide zero-overhead abstractions,
- expose representation costs explicitly,
- enable predictable performance across domains,
- avoid hidden costs and unspecified behaviors,
- preserve long-term compatibility across implementations.

These same design choices that give C++ its power also require engineers to confront text processing decisions that other environments may hide. Particularly:

- strings are code-unit containers, not inherently "text",
- encoding intent must be explicit in APIs and types,
- semantic text operations require deliberate decoding steps,
- normalization, collation, and segmentation semantics are outside the core language definition.

Understanding this reality is essential to designing systems that handle modern text correctly rather than by accident.

The gap between representation and semantics

Many engineers initially assume that a string in source code or a `std::string` universally represents "characters." In Unicode, however:

- code units may represent partial semantic units,
- scalar values (code points) require decoding,

- user-perceived characters (grapheme clusters) can span multiple code points,
- locale-specific rules influence collation and case mapping.

This gap between representation and semantics is not a peculiarity of C++; it reflects the inherent complexity of Unicode text. C++'s explicit model forces engineers to bridge this gap consciously and correctly.

The practical engineer's perspective

This booklet is written from the perspective of a practicing engineer:

- text correctness cannot be an afterthought,
- encoding decisions should be explicit, not assumed,
- APIs must state intent unambiguously,
- errors should be handled deterministically,
- semantic processing belongs in defined layers,
- performance trade-offs must be understood and documented.

These are not academic concerns; they are core to building software that:

- interoperates with JSON, HTTP, and database systems,
- behaves consistently across platforms,
- supports global languages and symbols,
- resists security vulnerabilities related to malformed text,
- scales with user expectations in a global environment.

The evolution of Unicode support in C++

The C++ language and standard library have evolved in response to the need for clearer and safer text handling:

- C++11 introduced distinct character types (`char16_t`, `char32_t`),
- C++20 introduced `char8_t` to express UTF-8 code units explicitly,
- C++20 and beyond clarified the relationship between encoding intent and types,
- C++23 and ongoing work toward C++26 refine interoperability and library support.

These changes do not magically solve all Unicode problems; they provide better tools for engineers to express intent and avoid common pitfalls.

What this booklet aims to accomplish

The primary goals of this booklet are:

1. **Clarify the difference between representation and semantics**, including what each encoding provides and what it does not.
2. **Explain why text is hard in C++**, not because of a defective language, but because of a complex problem space that requires explicit engineering responsibility.
3. **Provide a mental model and terminology** that engineers can apply consistently to avoid common mistakes.
4. **Offer practical guidelines and architectural patterns** for designing APIs and systems that handle text correctly.
5. **Introduce professional Unicode handling strategies** using appropriate libraries and test methodologies.

6. **Document pitfalls and best practices** so that Unicode defects are prevented, not merely discovered after they cause customer impact.

The approach is a blend of specification understanding, practical API design, and real-world engineering judgment.

Who benefits from this booklet

This booklet is intended for:

- C++ developers building systems that handle text in any nontrivial way,
- engineers migrating legacy C++ code to internationalized environments,
- library authors exposing text APIs,
- architects defining service boundaries and data interchange formats,
- teams concerned with correctness, security, and global user experience.

It is not a terse reference; it is a guide for engineers who must wrestle with the realities of text in production software.

How to use this booklet effectively

To make the most of this material:

- Read the early chapters for conceptual grounding; do not skip the terminology chapters.
- Use the API guidelines and architectural patterns as checklists when designing code.
- Treat the appendices as living artifacts to revisit when integrating new encodings or libraries.

- Build test suites against the pitfalls checklist to ensure long-term correctness.
- Reference specifications and library documentation when a behavior is unclear or ambiguous.

A final engineering reminder

Modern text processing is a domain where correctness and performance intersect with human expectations. C++ gives you the power to control both representation and behavior, but it also expects you to manage that control responsibly. This booklet equips you with the concepts, tools, and patterns to do exactly that: to build text handling that is correct, maintainable, and aligned with modern Unicode and C++ evolution.

Acknowledgements to the engineering community

The practices and insights in this booklet reflect years of evolution in:

- Unicode standards and technical reports,
- compiler and standard library evolution,
- open source library design and lessons learned from production systems,
- cross-platform engineering experience in text and localization.

The goal is not to replicate specifications, but to translate them into reliable engineering practice in the context of C++ and modern systems.

Embark on this journey

With the concepts in this preface established, this booklet invites you to explore the reality of Unicode text in C++ and to build systems that handle text with the rigor, clarity, and correctness that professional software demands.

Chapter 1

Why Unicode Is a Challenge in C++

1.1 Historical Background of Text Handling in C++

1.1.1 Context: C++ Inherited a Pre-Unicode World

C++ did not begin as a ”text language”; it began as a systems language that inherited C’s model of memory and I/O. The earliest and most influential design assumption was that *text is a sequence of bytes* and that the basic unit of storage and processing is `char`. That assumption worked for decades because much software operated in environments dominated by:

- ASCII for source code and program output,
- 8-bit ”extended” encodings and code pages for localized systems,
- terminals and files where byte-to-glyph interpretation was a property of the environment rather than the program.

Unicode breaks this historical model because Unicode text is not inherently ”byte text”. Unicode introduces concepts that are orthogonal to bytes: code points, variable-length

encodings (notably UTF-8 and UTF-16), combining marks, normalization, grapheme clusters (what users perceive as characters), and directionality. The reason Unicode is challenging in C++ is not that C++ cannot store Unicode, but that its historical abstractions encourage reasoning in terms of bytes and platform-defined character sets.

1.1.2 Phase 1: The Byte Era (C Heritage)

In the earliest era, `char` served two roles:

1. a small integer type used for raw bytes (binary data),
2. the fundamental unit of text.

C and early C++ libraries therefore shaped programmer habits around byte-based operations:

- ”length” means number of bytes before `'\0'`,
- indexing is constant time and returns a single byte,
- case operations and classification are defined in terms of a locale-dependent mapping of `unsigned char` values.

These operations do not match Unicode semantics. In UTF-8, a single user-perceived character may span multiple bytes. In UTF-16, it may span multiple 16-bit code units. In both, byte/code-unit indexing does not correspond to character indexing.

```
// Byte-era intuition breaks under UTF-8.
#include <string>
#include <iostream>

int main() {
    // UTF-8 bytes for "Å" (U+00C5) are typically 0xC3 0x85.
    std::string s = u8"Å";
```

```
    std::cout << "bytes = " << s.size() << "\n";
    std::cout << "first byte as int = " << (int)(unsigned char)s[0] << "\n";

    // s[0] is not "the character". It is the first UTF-8 code unit (byte).
}
```

Historically, the standard library could not assume an encoding for `std::string`, because the language was deployed across operating systems with different defaults (code pages, locale conventions, terminal encodings). This portability goal entrenched the ”narrow string” model: `std::string` is a sequence of `char` bytes whose interpretation depends on the program’s chosen conventions.

1.1.3 Phase 2: Locales and the Multibyte/Wide Split

As software globalized before Unicode became dominant, the ecosystem adopted *locale* and *multibyte* concepts:

- A locale specifies cultural rules and an associated encoding context.
- ”Multibyte” encodings allow a character to be represented by a variable number of bytes.
- A ”wide character” type was used as an intermediate representation for decoded characters.

C and C++ provided conversion functions between multibyte sequences and wide characters, and iostreams introduced locale facets to customize classification, collation, and code conversion. The intent was to give a generic framework to support many national encodings. In practice, this framework had long-standing limitations that matter historically:

- Locale configuration is environment-sensitive and often inconsistent across platforms.
- Some legacy encodings are stateful, making robust streaming conversion difficult.

- Locale-based text behavior is not a full Unicode text model (no normalization, grapheme segmentation, or modern security-aware comparisons).
- Locale state and facet use introduced complexity and poor ergonomics for everyday programming.

The historical outcome is that ”text correctness” became split across too many layers: the OS, the terminal, the locale, the stream facets, and application code. This fragmentation is one reason modern C++ practice tends to prefer explicit Unicode encodings (especially UTF-8) and explicit conversion boundaries, rather than implicit locale magic.

1.1.4 Phase 3: `wchar_t` and the Platform Divide

C++ inherited `wchar_t` as a ”wide character” type intended to hold a larger set of characters than `char`. Crucially, `wchar_t` was never standardized as ”Unicode code point”. Its size and meaning are implementation-defined, and two major traditions emerged:

- Many Windows environments use 16-bit wide characters (commonly aligning with UTF-16 code units).
- Many Unix-like environments use 32-bit wide characters (often able to store Unicode scalar values directly as UTF-32 code units).

This history created a portability trap:

- `std::wstring` is not a portable ”Unicode string”.
- Algorithms that assume 1 element = 1 character behave differently between UTF-16 and UTF-32.
- ”Length” and indexing in `std::wstring` do not reliably correspond to user-perceived characters.

```cpp
// wstring portability trap: the same literal can have different element counts.
#include <string>
#include <iostream>

int main() {
    std::wstring ws = L"  "; // U+10437 (outside the Basic Multilingual Plane)
    std::wcout << L"elements = " << ws.size() << L"\n";

    // On a UTF-16 wchar_t platform, size may be 2 (surrogate pair).
    // On a UTF-32 wchar_t platform, size is typically 1.
}
```

Historically, `wchar_t` was most successful not as a universal Unicode abstraction, but as a way to interoperate with platform APIs that had standardized around ”wide” interfaces. That pragmatic role persists today, but it does not solve Unicode processing as a portable, standard-only problem.

1.1.5 Phase 4: Standard Library Text Facilities Grew Around Code Units

The C++ standard library evolved robust containers and algorithms for sequences, but it did so around *code units*:

- `std::string` and `std::u8string` are sequences of 8-bit code units.
- `std::u16string` is a sequence of UTF-16 code units.
- `std::u32string` is a sequence of UTF-32 code units.

A core historical reality is that most standard algorithms operate on element boundaries:

- `std::sort`, `std::find`, `std::search` operate on code units, not grapheme clusters.
- Regular expressions and case-insensitive comparisons are not standardized as full Unicode-aware operations.

- ”Character classification” in the standard library is primarily locale-based and historically centered on narrow/wide character classifications, not Unicode properties.

This is a design trade: the standard library provides generic sequence tools, but Unicode text correctness requires additional semantic layers (decoding, validation, normalization, segmentation). Those layers historically remained outside the standard library because they are large, complex, and rapidly evolving compared to core language facilities.

1.1.6 Phase 5: C++11 Made Unicode Types Explicit (But Not Unicode Processing)

C++11 introduced distinct character types and literal prefixes that made encoding intent more explicit:

- UTF-16 and UTF-32 string literals (`u"..."`, `U"..."`),
- a UTF-8 literal prefix (`u8"..."`),
- character types suited for UTF-16/UTF-32 code units (`char16_t`, `char32_t`).

Historically, this was an important milestone: the language could now express Unicode-related encodings in source code without relying on platform-specific ”wide” assumptions.
But this era also made a limitation more visible: *types alone do not provide correct text handling*. Correct Unicode work needs:

- validation (rejecting invalid sequences),
- transcoding (UTF-8 $\leftrightarrow$ UTF-16 $\leftrightarrow$ UTF-32),
- normalization (canonical equivalence),
- segmentation (grapheme clusters, word boundaries),

- security-aware comparisons (avoiding confusable or ill-formed inputs).

The C++ standard library historically did not standardize these semantic operations in a complete way, so portable Unicode correctness remained primarily a library ecosystem problem rather than a language problem.

1.1.7 Phase 6: Deprecation of Older Conversion Mechanisms

A notable historical correction occurred when older standard conversion utilities (often used with locales and facets) were deprecated. This reflected a broad consensus that:

- the older approach was hard to use correctly,
- behavior could be inconsistent across implementations,
- the design did not align with modern UTF-8-first software development.

The practical consequence for modern C++ is that developers are expected to manage conversions explicitly, usually by:

- treating `std::string`/`std::u8string` as byte/code-unit sequences with explicit encoding contracts,
- converting at boundaries (file I/O, OS APIs, network protocols),
- performing Unicode-aware operations using dedicated, well-tested Unicode libraries when required.

1.1.8 Phase 7: C++20 and the Transition to Strongly Typed UTF-8

C++20 introduced a critical historical improvement: UTF-8 literals became strongly typed via `char8_t`. This changed UTF-8 from ”often stored in `char` by convention” to ”explicitly represented at the type level”.
Why this matters historically:

- It reduces accidental mixing of ”bytes” and ”UTF-8 text” in APIs.
- It encourages the design of interfaces that state encoding expectations explicitly.
- It forces legacy code that assumed `u8"..."` is a `const char*` to become explicit about conversions.

```
// C++20: u8 literals are char8_t-based, forcing explicit intent.
#include <string>
#include <string_view>

std::u8string     ok1 = u8"UTF-8 text";
std::u8string_view ok2 = u8"view";

// The following are intentionally not implicit in C++20:
// std::string s = u8"UTF-8 text";           // ill-formed (type mismatch)
// std::string_view v = u8"UTF-8 text";      // ill-formed (type mismatch)
```

This change did not magically solve Unicode, but historically it moved C++ closer to a world where encoding is part of an API contract rather than a hidden assumption.

1.1.9 Phase 8: C++23–C++26: Toward Clearer Boundaries (Not a Full Unicode Stack)

From C++23 through C++26, standard evolution has continued to improve surrounding capabilities that affect text handling:

- stronger and more uniform formatting/printing facilities,
- ongoing cleanup/removal of long-deprecated locale-related conversion machinery,
- emerging standard facilities to *identify* encodings (distinguishing what an encoding is, without claiming that the standard library fully processes text).

Historically, this direction signals that standard C++ is converging on a model where:

1. the standard library provides strong primitives and clear types for code units,

2. encoding and decoding boundaries are explicit and auditable,

3. full Unicode semantics (normalization, segmentation, confusable detection) are recognized as a specialized layer that may remain outside the core standard library or arrive incrementally.

1.1.10 The Central Historical Lesson

Unicode is challenging in C++ because C++ contains *multiple historical text models at once*:

- **byte sequences** (`char` as raw storage),
- **locale-dependent narrow text** (environment-selected meaning of `char`),
- **platform ”wide” text** (`wchar_t` with platform-defined width and semantics),
- **explicit Unicode code units** (`char8_t`/`char16_t`/`char32_t`),
- **user-perceived text** (grapheme clusters, normalization, directionality) that is not identical to code units.

This historical layering is not accidental; it reflects decades of backward compatibility, portability goals, and evolving industry practices. Modern C++ (C++20–C++26) improves correctness primarily by pushing developers toward *explicit encoding contracts* and *explicit boundaries*, rather than by pretending that a single built-in string type can represent ”text” correctly for all languages and all operations.

1.1.11 Practical Takeaway for the Rest of This Chapter

Because the historical foundation is code-unit based, this booklet will use a boundary-driven mental model:

1. Treat external data as bytes until an encoding contract is established.
2. Validate and decode at boundaries (files, OS APIs, network).
3. Perform internal processing with a representation chosen for the algorithm (UTF-8/UTF-16/UTF-32) and be explicit about what ”indexing” means.
4. When operations require true Unicode semantics (normalization, grapheme clusters), treat them as a separate layer with explicit tooling and explicit tests.

1.2 Encoding Neutrality as a Language Design Principle

1.2.1 Meaning of ”Encoding Neutrality” in C++

Encoding neutrality means the C++ language and its standard library avoid mandating a single, universal text encoding for `char`-based strings and I/O. Instead, C++ defines a framework of *character sets* and *translation phases* for source code, and provides *character types* and *string templates* that can represent many encodings—while leaving the selection of encodings for external data (files, terminals, network protocols, OS APIs) largely to the implementation and the program.
This is not an accident or omission; it is a deliberate design stance driven by:

- **Portability** across platforms with different historical defaults (code pages, UTF-8, UTF-16).
- **Backward compatibility** with decades of existing C and C++ code that treats `char` as ”bytes” or locale text.
- **Separation of concerns**: the core language provides types and rules; full Unicode text semantics are a large, evolving domain.
- **Systems-level reality**: programs often process both binary and text data through the same byte-oriented channels.

The cost is that ”text” in C++ is not a single built-in abstraction. Developers must explicitly manage *where bytes become text*, and what invariants are expected from that text (UTF-8 validity, normalization form, etc.).

1.2.2 Language-Level Foundations: Character Sets and Translation

C++ standardizes multiple conceptual character sets relevant to program text:

- **Basic source character set**: the minimal set required to write C++ source (letters, digits, punctuation, whitespace).
- **Translation character set**: the abstract character set after source translation (modern wording explicitly ties this to Unicode code space concepts).
- **Execution character sets**: the sets used at runtime for ordinary and wide character literals; their encodings are locale-specific and implementation-defined.

The key point for encoding neutrality is that C++ draws a strict boundary between:

1. **source code representation** (how the compiler interprets your source file and literals),
2. **runtime representation** (what bytes/code units exist in memory),
3. **external representation** (what encoding files/terminals/APIs use).

C++ historically permits implementations to choose encodings for the execution character sets, subject to constraints for the basic literal character set. This preserves portability, but it also means `std::string` does not inherently mean "UTF-8".

1.2.3 Type System Neutrality: Many Character Types, No Single "Text Type"

C++ provides multiple character types:

- `char` (narrow character / byte-sized code unit; interpretation is program-defined),
- `wchar_t` (wide character; size and meaning are platform-defined),
- `char8_t` (distinct UTF-8 code unit type since C++20),
- `char16_t` (UTF-16 code unit type),

- `char32_t` (UTF-32 code unit type).

This is encoding neutrality in practice: the language gives you *choices* and *precision* (especially after C++20), but it does not declare that one of these is the universal internal text type.

C++20: Strongly Typed UTF-8 as a Neutrality Refinement

C++20 introduced `char8_t` and made `u8"..."` literals produce `const char8_t[]`. This is best understood as a *neutrality refinement*:

- It does *not* force all narrow strings to be UTF-8.
- It *does* give UTF-8 an explicit, non-ambiguous type.
- It reduces accidental mixing of raw bytes and UTF-8 text in APIs.

```
// C++20: UTF-8 literals are char8_t-based, making intent explicit.
#include <string>
#include <string_view>

std::u8string_view sv = u8"     "; // UTF-8 code units, typed
// std::string_view  bad = u8"     "; // ill-formed in C++20 (type mismatch)
```

This change is historically important: it acknowledges that many programs were already using UTF-8 in `std::string` by convention, but the language could not safely assume that convention without breaking neutrality and existing code. `char8_t` provides a way to write UTF-8-aware interfaces without redefining `char`.

1.2.4 Library-Level Neutrality: Generic Strings and Code-Unit Algorithms

The standard library models ”strings” as sequences of *code units*:

- `std::basic_string<charT>` is a container of `charT` elements.

- `std::basic_string_view<charT>` is a non-owning view of `charT` elements.
- Most algorithms operate on element boundaries (code units), not on Unicode scalar values or grapheme clusters.

This design is deliberately encoding-neutral:

- It works for ASCII, UTF-8, Latin-1, Shift-JIS bytes (as code units), UTF-16 code units, UTF-32 code units, and many other representations.
- It avoids embedding Unicode-specific semantics into the fundamental containers and algorithms.

But the cost is semantic: code-unit operations do not equal text operations. For example, ”length” is the count of code units, not the count of user-perceived characters.

```
// Code-unit length is not "character count" for variable-width encodings.
#include <string>
#include <iostream>

int main() {
    std::string utf8 = u8"é"; // commonly 2 bytes in UTF-8: 0xC3 0xA9
    std::cout << utf8.size() << "\n"; // prints 2 on typical systems

    // size() counts code units (bytes), not Unicode scalar values, not graphemes.
}
```

1.2.5 I/O and Locales: Neutral by Necessity, Tricky by Reality

C++ iostreams and locales were designed in an era where many systems used non-Unicode encodings. The locale framework supports:

- cultural formatting (numbers, dates),

- character classification and collation rules,
- a mechanism (historically via facets) for conversions between external and internal representations.

From an encoding-neutral design standpoint, this is coherent: the standard cannot mandate a single external encoding for all operating systems, terminals, and file systems.
From a modern Unicode standpoint, this is a frequent source of confusion because:

- the ”current locale” is often implicit and environment-dependent,
- behavior can vary across standard library implementations,
- and many legacy conversion facilities were deprecated and are being removed in modern standards, pushing developers toward explicit conversions at boundaries.

1.2.6 C++23–C++26 Direction: Make Encodings Explicit Without Picking One

Modern C++ evolution has increasingly favored *explicitness*:

- Stronger typing for UTF-8 literals (`char8_t` in C++20).
- Continued cleanup/removal of older, fragile locale/Unicode conversion mechanisms (C++26 timeframe).
- New facilities aimed at *identifying* and naming encodings (e.g., `std::text_encoding` in the C++26 library feature set), which clarifies the environment without mandating that your program must use a single encoding everywhere.

This is encoding neutrality modernized: the standard moves toward a world where programs can:

1. represent text in the encoding that matches their domain needs,
2. explicitly tag and inspect encoding expectations at boundaries,
3. avoid ”magic” assumptions about what `char` means.

1.2.7 Neutrality vs. Correctness: What C++ Guarantees and What It Does Not

Encoding neutrality does *not* mean ”C++ is bad at Unicode.” It means C++ draws the line differently:

- C++ guarantees strong, efficient primitives for sequences of code units and bytes.
- C++ does not guarantee full Unicode text processing semantics in the core library (normalization, grapheme segmentation, confusables detection, full case folding).

As a result, a correct modern design treats Unicode as a *layer* on top of encoding-neutral primitives.

1.2.8 Practical Engineering Pattern: Explicit Boundaries and Encoding Contracts

Encoding-neutral design becomes robust when you adopt two explicit contracts:

Contract A: ”Bytes vs. Text”

- Use `std::byte` or `unsigned char`/`std::vector<std::byte>` for **opaque binary**.
- Use `std::u8string`/`std::u8string_view` for **UTF-8 code units** when you want typed UTF-8.

- Use `std::string`/`std::string_view` only when you have an explicit convention (e.g., "this is UTF-8" or "this is protocol-defined Latin-1") documented and enforced at the API boundary.

Contract B: "What does indexing mean?"

- Indexing into a string indexes **code units**.
- If your algorithm needs Unicode scalar values, decode first (e.g., to `char32_t` sequence).
- If your algorithm needs user-perceived characters, you need grapheme-cluster segmentation (a higher semantic layer).

```
// Encoding-neutral API pattern: accept code units + explicit contract, decode when needed.
#include <string_view>
#include <cstddef>

struct Utf8TextView {
    std::u8string_view v; // contract: must be valid UTF-8
};

struct BytesView {
    const std::byte* data;
    std::size_t size;
};

// Higher layers would add validation and decoding utilities.
```

1.2.9 Why This Principle Makes Unicode "Hard" in C++

Unicode is challenging in C++ primarily because encoding neutrality shifts responsibility:

- The language gives you *tools* (types, strings, views, algorithms) that are neutral and fast.

- The programmer must define and enforce *encoding contracts* and perform semantic operations explicitly.

In exchange, you gain:

- the ability to interoperate with any platform and any protocol,
- the ability to treat `char` as bytes when needed (systems programming),
- and the ability to adopt UTF-8 (or another encoding) as a project policy without being blocked by the language.

This section establishes the key mindset for the rest of the booklet:

> C++ is intentionally encoding-neutral; therefore, correct Unicode handling is achieved by explicit boundaries, explicit encoding contracts, and explicit semantic layers—not by assuming that a ”string” is inherently text.

1.3 Language Responsibility vs Library Responsibility

1.3.1 Overview of Responsibilities

In C++, text handling is divided into two conceptual domains:

- **Language Responsibility**: what the core C++ language defines, guarantees, and requires for text-related constructs.
- **Library Responsibility**: what the standard library provides in terms of facilities, abstractions, and algorithms that operate on text or encoding constructs.

Understanding the boundary between these two responsibilities is crucial to correctly handling Unicode in modern C++ (C++20 through C++26). This separation is intentional, rooted in historical design and portability goals that predate the widespread adoption of Unicode.

1.3.2 What the Language Defines

The C++ language specification defines the following key concepts related to text:

- **Character types** and literal forms: `char`, `wchar_t`, `char8_t`, `char16_t`, `char32_t`.
- **Translation and execution character sets**: categories of characters used by the compiler during translation and at runtime.
- **Literal encodings**: mapping of literal prefixes (`u8`, `u`, `U`, `L`) to types and code unit sequences.
- **Basic language rules** for string and character initialization, concatenation, and type conversions.

The language does not mandate:

- a single universal text encoding for `std::string` or other string types,
- a specific interpretation of textual content beyond code unit representation,
- built-in semantics for collation, normalization, or text segmentation,
- automatic transcoding or validation of external data.

The core language sets the stage upon which text-handling libraries operate. It provides the *syntax* and the minimal semantic guarantees needed to write programs that manipulate sequences of characters and bytes, but it does not define full text semantics.

1.3.3 What the Standard Library Provides

The C++ standard library complements the language with types and utilities that make text manipulation possible:

- `std::basic_string<charT>` and associated views for representing sequences of code units.
- I/O facilities (`std::istream`, `std::ostream`, file streams) for transferring data between external sources and program memory.
- Generic algorithms that operate on iterators over sequences of code units.

However, the library historically stopped short of providing full Unicode text semantics. There are no standard functions for:

- validating UTF-8 sequences,
- transcoding between encodings with error semantics,
- performing Unicode-aware collation or case folding,

- identifying grapheme clusters or performing normalization.

Instead, the library provides building blocks. Higher-level Unicode operations are left to:

- dedicated third-party libraries,
- explicit application-level code,
- or future extensions of the standard library.

1.3.4 Rationale Behind the Division

The division between language and library responsibility in C++ has multiple motivations:

Portability and Backward Compatibility C++ must interoperate with legacy systems, legacy C code, and a wide range of platform APIs. Mandating a single text encoding in the core language or library would break backward compatibility and limit portability.

Performance and Control C++ emphasizes zero-overhead abstractions. Text processing can be resource-intensive, and many applications (e.g., systems code) require precise control over when and how encoding operations occur. Offloading full Unicode semantics to libraries allows programs to pay only for the complexity they need.

Separation of Concerns The language provides a neutral set of character and string representations. The library provides generic data structures and algorithms. Complex text semantics (Unicode normalization, locale-aware collation) are orthogonal concerns that are better encapsulated in separate modules or libraries, offering specialization without burdening every text-handling program.

1.3.5 Language Guarantees About Text Representations

Although the language does not guarantee a specific encoding, it does guarantee:

- **Type safety**: distinct character types (`char8_t`, `char16_t`, `char32_t`) prevent accidental misuse of incompatible text representations.
- **Literal interpretation**: literal prefixes determine the type and code unit sequence in the translation phase.
- **Byte-level representation**: `char` is a contiguous type that can represent byte-oriented data.
- **Well-defined sequence behavior**: `std::basic_string` and related views provide deterministic iteration, capacity, and element access behavior.

These guarantees allow libraries to build higher-level text semantics on top of neutral primitives.

1.3.6 Library Expectations for Handling Text

The standard library assumes that:

- containers like `std::string` hold sequences of code units without enforcing encoding validity.
- algorithms such as `std::sort` or `std::find_if` operate on code-unit sequences without semantic knowledge of characters.
- I/O functions transfer sequences of bytes or code units as-is, leaving interpretation to callers.

For example, the length of a string container refers to the count of stored code units, not the number of Unicode code points or grapheme clusters.

```cpp
// Library concept: size() returns code-unit count.
#include <string>
#include <iostream>

int main() {
    std::u8string u8 = u8"€";
    // UTF-8 encoding of '€' is 3 bytes (0xE2 0x82 0xAC).
    std::cout << u8.size() << "\n"; // typically prints 3
}
```

In this example, the library is fulfilling its responsibility: counting code units. It does not infer or guarantee semantic glyph counts.

1.3.7 What the Language Does Not Guarantee

It is important to understand what the C++ language does not define:

- *Universal text semantics* (e.g., character boundaries, grapheme clusters).
- *Unified encoding for portable text* across all platforms.
- *Standardized Unicode text algorithms* as part of the core language or core library.

Because these concepts are outside the language guarantee, programmers must explicitly choose and implement conventions if their applications require them.

1.3.8 Implications for Modern C++ Development (C++20 to C++26)

From C++20 onward, the language has improved clarity in encoding intent:

- `char8_t` and UTF-8 literal types make encoding annotations explicit.
- Enhanced library support for formatting facilitates output of text in a consistent manner.

- New library facilities help identify encodings when interacting with external data.

However, these improvements remain within the realm of language and library primitives. They do not eliminate the need for explicit design decisions about encoding interpretation and text semantics.

1.3.9 Design Patterns Emerging from This Responsibility Split

Developers should adopt patterns that respect the boundaries:

- **Define encoding contracts at interfaces**: Document and enforce what encoding your APIs expect and produce.
- **Validate at boundaries**: When converting between external and internal representations, validate encoding correctness before further processing.
- **Use dedicated Unicode libraries for semantics**: For full Unicode behavior (normalization, case folding, segmentation), use established libraries rather than rely on the basic standard library.
- **Favor explicit types**: Prefer `std::u8string` and other clearly typed containers when working with known encodings.

1.3.10 Summary

The distinction between language responsibility and library responsibility in C++ text handling is fundamental to understanding why Unicode is challenging in C++. The language defines core types and rules without prescribing text semantics or a universal encoding. The standard library provides flexible containers and algorithms that work at the code-unit level. The result is a powerful, portable system that requires developers to be explicit about encoding contracts, validation, and semantic text operations. This separation underpins the rest of the discussion in this booklet on achieving correct text handling in modern C++ from C++20 through C++26.

1.4 Characters, Code Units, Code Points, and Grapheme Clusters

1.4.1 Why this section matters (C++ perspective)

One of the deepest sources of Unicode-related bugs in C++ is that the word *character* is used casually to mean several different things, each with different rules and different units of measurement. Many C++ APIs operate on *code units* (bytes or 16-bit units), while user expectations (cursor movement, deletion, column widths, ”one visible letter”) operate on *grapheme clusters*. Between these two sits the *Unicode scalar value / code point* model, which is often what developers intend when they say ”iterate characters”.
In modern C++ (C++20–C++26), the language gives you multiple character types (`char`, `wchar_t`, `char8_t`, `char16_t`, `char32_t`) and multiple literal prefixes (`u8`, `u`, `U`, `L`), but the standard library still fundamentally treats strings as sequences of *code units* and does not standardize Unicode normalization, grapheme segmentation, or locale-independent text shaping. Therefore, **you must decide what unit your algorithms are written for** and enforce that choice throughout your codebase.

1.4.2 Core definitions (precise, testable concepts)

Code unit

A **code unit** is the smallest storage unit of a particular Unicode encoding form:

- **UTF-8**: code unit is 8 bits (one byte).
- **UTF-16**: code unit is 16 bits.
- **UTF-32**: code unit is 32 bits.

In C++, a `std::basic_string<T>` is always a contiguous sequence of elements of type `T`. When `T` is `char` or `char8_t`, that sequence is a sequence of bytes. Whether those bytes represent UTF-8 is a *separate contract* you establish (at your interfaces, file I/O, network protocols, database schemas, etc.).

Practical takeaway Operations like indexing, `size()`, `substr()`, `find()`, and iterating a `std::string` are, by default, **code-unit operations**.

Code point (Unicode scalar value)

A **code point** is an integer value in the Unicode code space, conventionally written as `U+XXXX`. Not all code points are valid Unicode scalar values; Unicode reserves a range for surrogates, and scalar values exclude those surrogate code points.

Practical takeaway When developers say ”process the text by characters,” they often mean ”process by code points.” But even code points are not what users perceive as characters.

Character (ambiguous term)

In Unicode discussions, **character** is an overloaded word that may refer to:

- a *code unit* (common in byte-oriented code and many C++ string algorithms),
- a *code point* (common in Unicode-aware algorithms),
- a *grapheme cluster* (what users usually perceive as one displayed character),
- or even a *glyph* (rendering-dependent shape produced by fonts and shaping).

Rule In professional C++ text code, avoid the word *character* in API contracts unless you define the unit precisely.

Grapheme cluster (user-perceived character)

A **grapheme cluster** is a sequence of one or more code points that should be treated as a single user-perceived character for operations like cursor movement, backspace/delete, and ”counting visible characters.” Unicode segmentation defines *extended grapheme clusters* to cover modern writing systems and emoji sequences.
A grapheme cluster can be:

- one code point (e.g. Latin `U+0041` ”A”),
- multiple code points (e.g. `U+0065` ”e” + `U+0301` combining acute accent),
- multiple code points with joiners/modifiers (many emoji sequences, family emojis, skin-tone modifiers, flag sequences).

Critical consequence One grapheme cluster is not guaranteed to equal one code point, and one code point is not guaranteed to equal one code unit.

1.4.3 Three layers you must keep separate

A robust mental model is a three-layer stack:

1. **Storage layer (code units)**: bytes / 16-bit units / 32-bit units.
2. **Unicode layer (code points)**: decoded scalar values, validity rules, normalization.
3. **User interaction layer (grapheme clusters)**: cursor movement, editing, ”length” as humans expect.

In C++ terms:

- `std::string` or `std::u8string`: typically UTF-8 code units (bytes).

- `std::u16string`: UTF-16 code units.
- `std::u32string`: UTF-32 code units (often 1 code unit per code point for valid scalar values).

1.4.4 Concrete examples that expose common bugs

Example A: UTF-8 – one visible symbol may be many bytes

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

static void dump_bytes(std::string_view s) {
    std::cout << "bytes=" << s.size() << " : ";
    for (unsigned char c : s) {
        std::cout << std::hex << std::uppercase
                  << "0x" << static_cast<unsigned>(c) << ' ';
    }
    std::cout << std::dec << "\n";
}

int main() {
    // UTF-8 in a narrow literal depends on the source encoding and compiler options.
    // Using u8"" makes the literal UTF-8 by language rule (C++20 changes the element
    ↪  type).
    const char8_t* u8p = u8"€"; // U+20AC EURO SIGN, UTF-8 bytes: E2 82 AC
    std::u8string u8s = u8"€";

    std::cout << "u8string code-units=" << u8s.size() << "\n";
    // Print as bytes by reinterpreting to char for demonstration only.
```

```
    dump_bytes(std::string_view(reinterpret_cast<const char*>(u8s.data()), u8s.size()));
}
```

What this demonstrates:

- `u8"€"` encodes a single Unicode code point, but UTF-8 stores it using **3 code units (bytes)**.
- `size()` on a UTF-8 string is **byte count**, not character count.
- Indexing `u8s[0]` gives you **one byte**, not the full symbol.

Example B: One grapheme cluster may be multiple code points

Two visually identical strings may have different underlying sequences:

```
#include <cstdint>
#include <iomanip>
#include <iostream>
#include <string>

int main() {
    // "é" can be represented as:
    // 1) U+00E9 (LATIN SMALL LETTER E WITH ACUTE) -- NFC form
    // 2) U+0065 (e) + U+0301 (COMBINING ACUTE ACCENT) -- NFD form
    std::u32string nfc = U"\u00E9";
    std::u32string nfd = U"\u0065\u0301";

    std::cout << "nfc code-points=" << nfc.size() << "\n";
    std::cout << "nfd code-points=" << nfd.size() << "\n";

    auto dump_u32 = [](const std::u32string& s) {
        for (char32_t cp : s) {
            std::cout << "U+"
```

```
                    << std::hex << std::uppercase << std::setw(4)
                    << std::setfill('0') << static_cast<std::uint32_t>(cp)
                    << ' ';
        }
        std::cout << std::dec << "\n";
    };

    dump_u32(nfc);
    dump_u32(nfd);
}
```

What this demonstrates:

- Counting code points (`std::u32string::size()`) still does not equal "user-perceived characters."
- Normalization affects equality, searching, sorting, and "length" expectations.
- A grapheme cluster like e + combining accent is one user-perceived character but **two code points**.

Example C: UTF-16 surrogate pairs

UTF-16 uses one 16-bit code unit for many code points, but uses a **surrogate pair** (two code units) for code points beyond `U+FFFF`. Any algorithm that assumes one `char16_t` equals one code point will fail for such text.

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

int main() {
    // U+1F34C BANANA (emoji) is beyond U+FFFF, so UTF-16 needs two code units.
    std::u16string s = u"\U0001F34C";
    std::cout << "u16 code-units=" << s.size() << "\n";
}
```

1.4.5 C++20–C++26: what the language gives you (and what it does not)

Types signal *code unit width*, not "Unicode correctness"

- `char8_t` exists to represent UTF-8 code units explicitly and to type UTF-8 literals (`u8"..."`).
- `char16_t` and `char32_t` correspond to UTF-16 and UTF-32 code units for prefixed literals (`u"..."` and `U"..."`).
- `wchar_t` is platform-defined width and encoding (commonly UTF-16 on Windows, UTF-32 on many Unix-like systems), which makes it unsuitable as a portable Unicode boundary type.

The standard library does not provide grapheme segmentation

Even in C++26-era standard C++, you should assume:

- no standard facility for extended grapheme cluster boundaries,
- no standard Unicode normalization (NFC/NFD/NFKC/NFKD),
- no standard case folding suitable for Unicode identifiers or user text,
- no standard width measurement or shaping (font-dependent).

Therefore, whenever your requirement is "what the user sees" (backspace behavior, cursor movement, truncation by visible symbols), you must either:

- integrate a Unicode text-processing library, or
- implement a well-tested subset of the Unicode algorithms you need (segmentation, normalization, etc.) with conformance tests.

1.4.6 Design rules for professional C++ text algorithms

Rule 1: State the unit of iteration in API contracts

Every API that ”walks characters” must say one of:

- iterates **bytes / code units**,
- iterates **decoded Unicode scalar values (code points)**,
- iterates **extended grapheme clusters**.

Rule 2: Never index UTF-8 text by byte position unless you mean bytes

Byte indexing is valid only when your logic is byte-based (protocol framing, ASCII-only subset, binary blobs). For user text, it is usually incorrect.

Rule 3: Treat ”length” as a domain-specific concept

Possible meanings of ”length” include:

- number of bytes (storage, I/O),
- number of Unicode scalar values (some algorithms, normalization-sensitive),
- number of grapheme clusters (user-visible ”characters”),
- display width (terminal columns; depends on East Asian Width rules and fonts),
- rendered pixel width (GUI; depends on shaping and font metrics).

A correct system uses distinct names, e.g. `byte_length`, `codepoint_count`, `grapheme_count`, `display_columns`, `pixel_width`.

1.4.7 A minimal UTF-8 decoder (code-point iteration, not graphemes)

The following example shows how to iterate UTF-8 into Unicode scalar values. It is intentionally scoped: it demonstrates **code point iteration**, not grapheme segmentation, and includes basic validation to avoid accepting malformed sequences.

```
#include <cstdint>
#include <iostream>
#include <optional>
#include <string_view>
#include <vector>

struct Utf8DecodeResult {
    char32_t cp;
    std::size_t n; // number of bytes consumed
};

static bool is_cont(unsigned char b) { return (b & 0xC0u) == 0x80u; }

static std::optional<Utf8DecodeResult> decode_one(std::string_view s, std::size_t i) {
    if (i >= s.size()) return std::nullopt;

    const unsigned char b0 = static_cast<unsigned char>(s[i]);

    // 1-byte ASCII
    if (b0 <= 0x7Fu) {
        return Utf8DecodeResult{ static_cast<char32_t>(b0), 1 };
    }

    // Determine sequence length
    std::size_t need = 0;
    char32_t cp = 0;
```

```
    if ((b0 & 0xE0u) == 0xC0u) { need = 2; cp = b0 & 0x1Fu; }
    else if ((b0 & 0xF0u) == 0xE0u) { need = 3; cp = b0 & 0x0Fu; }
    else if ((b0 & 0xF8u) == 0xF0u) { need = 4; cp = b0 & 0x07u; }
    else {
        return std::nullopt; // invalid leading byte
    }

    if (i + need > s.size()) return std::nullopt;

    // Accumulate continuation bytes
    for (std::size_t k = 1; k < need; ++k) {
        const unsigned char bx = static_cast<unsigned char>(s[i + k]);
        if (!is_cont(bx)) return std::nullopt;
        cp = (cp << 6) | (bx & 0x3Fu);
    }

    // Reject surrogate code points and out-of-range
    if (cp >= 0xD800u && cp <= 0xDFFFu) return std::nullopt;
    if (cp > 0x10FFFFu) return std::nullopt;

    // Reject overlong encodings
    if (need == 2 && cp <= 0x7Fu) return std::nullopt;
    if (need == 3 && cp <= 0x7FFu) return std::nullopt;
    if (need == 4 && cp <= 0xFFFFu) return std::nullopt;

    return Utf8DecodeResult{ cp, need };
}

static std::vector<char32_t> utf8_to_codepoints(std::string_view s) {
    std::vector<char32_t> out;
    for (std::size_t i = 0; i < s.size(); ) {
        auto r = decode_one(s, i);
        if (!r) {
```

```
            // For production code, choose a policy:
            // - replace with U+FFFD, or
            // - reject the input, or
            // - treat as raw bytes.
            throw std::runtime_error("Invalid UTF-8");
        }
        out.push_back(r->cp);
        i += r->n;
    }
    return out;
}

int main() {
    // Example contains:
    // "A" (1 byte), "€" (3 bytes), " " (3 bytes), and an emoji (4 bytes)
    std::string_view s = "A€   ";

    auto cps = utf8_to_codepoints(s);
    std::cout << "bytes=" << s.size() << "\n";
    std::cout << "code_points=" << cps.size() << "\n";
}
```

Interpretation:

- `bytes` is the number of UTF-8 code units.
- `code_points` is the number of decoded Unicode scalar values.
- Neither of these is guaranteed to equal the number of **grapheme clusters**.

1.4.8 Where grapheme clusters become mandatory

If you implement any of the following features, you must operate on grapheme clusters (or on even higher-level shaping/layout abstractions), not on bytes or code points:

- Backspace/delete that removes ”one visible character”
- Cursor movement by ”one character”
- UI truncation by ”N characters”
- Text-field limits that users perceive as ”N characters”
- Highlighting, selection, and caret placement consistent with user expectations

1.4.9 Summary checklist (what to enforce in C++ code reviews)

- A **code unit** is storage; **code point** is a Unicode value; **grapheme cluster** is user-perceived.
- UTF-8: 1 code point uses 1–4 bytes; UTF-16: may use 1 or 2 `char16_t` code units; UTF-32: commonly 1 `char32_t` per code point for valid scalars.
- `std::string`/`std::u8string` operations are **code-unit algorithms** unless you build a higher-level layer.
- Never equate `size()` with ”number of characters” without stating the unit.
- If the requirement is user-visible editing, implement or use a **grapheme cluster** iterator.
- For comparisons and searching across diverse inputs, define a normalization policy (e.g., NFC) and apply it consistently at boundaries.

1.5 Common Unicode Misconceptions Among C++ Developers

1.5.1 Introduction

Unicode represents a broad and well-specified standard for encoding text from virtually all human languages. Despite this, many C++ developers hold misconceptions about how Unicode works and how it interacts with C++ language features and the standard library. These misconceptions often lead to bugs, incorrect assumptions, and inefficient solutions. In this section, we address the most widespread misunderstandings and clarify them with precise, modern explanations suited for C++20 through C++26 text handling.

1.5.2 Misconception 1: `std::string` means Unicode string

A frequent assumption is that `std::string` implicitly represents a Unicode string. In reality, `std::string` is a container of *bytes* (8-bit code units). The standard does not impose any encoding on those bytes. A `std::string` may hold ASCII, UTF-8, ISO-8859-1, or arbitrary binary data. UTF-8 is a common convention in modern C++ software, but it is not enforced by the language or standard library.

The consequence is that functions like `size()`, `operator[]` and iterators operate on *code units*, not on Unicode code points or grapheme clusters. Writing code that assumes one element in a `std::string` equals one user-visible character is incorrect for UTF-8 text containing multi-byte sequences.

1.5.3 Misconception 2: `wchar_t` makes Unicode automatic

Some developers believe that using `wchar_t` solves Unicode issues. In truth, `wchar_t` is a platform-dependent wide character type. On Windows, `wchar_t` is 16 bits and aligns with UTF-16 code units, but it still holds code units, not full Unicode abstraction. On most Unix-like systems, `wchar_t` is 32 bits (UTF-32 code units), which does allow storing a single Unicode

scalar in one element. However, even in UTF-32, `wchar_t` does not provide normalization, grapheme segmentation, case folding, or locale-independent operations. Therefore, using `wchar_t` alone does not automatically make text processing Unicode-correct or user-correct.

1.5.4 Misconception 3: Unicode is only about character encoding

While Unicode specifies encoding forms such as UTF-8, UTF-16, and UTF-32, Unicode encompasses much more than encoding. It defines:

- **Unicode scalar values** (also known as code points),
- **Normalization forms** (NFC, NFD, NFKC, NFKD),
- **Combining character behavior**,
- **Grapheme cluster boundaries**,
- **Bidirectional text rules**,
- **Case folding and collation**,
- **Emoji sequences and modifiers**.

Encoding is only the representation of code points in memory or storage. Proper Unicode handling often requires normalization and segmentation logic that the C++ standard library does not provide.

1.5.5 Misconception 4: One code point equals one user-perceived character

A single Unicode scalar value often represents what is informally called a “character.” However, what users perceive as a single character (for example for cursor movement, deletion, or counting) may consist of multiple code points. Typical examples include:

- A base letter plus one or more combining marks.
- Emoji sequences that use zero-width joiners and modifiers.
- Regional indicator sequences for flags and multi-code-point emoji.

Such sequences are called *grapheme clusters*. Algorithms that count code points or code units will *not* match user expectations for the number of displayed characters. Correct handling of user-perceived characters requires grapheme cluster segmentation, which the standard library does not supply.

1.5.6 Misconception 5: UTF-8 indexing is safe and intuitive

Another misconception is that indexing a UTF-8 encoded string by array index corresponds to indexing characters. With UTF-8, each code point may use between one and four code units. For example, non-ASCII symbols typically use multiple bytes. Therefore, `std::string::operator[]` returns a byte of a multi-byte sequence, not a complete code point or grapheme cluster. Any algorithm that treats `std::string::operator[]` as a character indexer will break on non-ASCII text.

1.5.7 Misconception 6: Normalized strings are automatically comparable

Unicode specification defines multiple equivalent representations for the same visual text. For example, a letter with an accent may be represented as a precomposed code point or as a base letter plus combining accent code point. Without normalization, byte-wise or code point-wise comparisons will consider these distinct, even though visually they appear identical. Correct Unicode string comparisons across arbitrary user input must apply a normalization form consistently at defined boundaries.

1.5.8 Misconception 7: Locale solves all Unicode needs

Locales in C++ provide language- and region-specific rules for formatting and parsing numbers, dates, and some character classification categories. They do not, however, provide Unicode normalization, grapheme segmentation, or comprehensive collation (sorting) across all scripts. Standard locales are limited in their Unicode handling and may vary by implementation. Developers should not rely on locales to provide full Unicode semantics.

1.5.9 Misconception 8: C++20 added full Unicode text processing

C++20 introduced enhancements such as explicit `char8_t` and UTF-8 literal support. While this clarifies encoding intentions, it does not equate to full Unicode text processing. The standard library still lacks fundamental text algorithms expected by Unicode-aware applications:

- Unicode normalization routines.
- Grapheme cluster iteration and segmentation.
- Unicode collation tailored to language norms.
- Case folding for Unicode.
- Bidi (bidirectional) text layout logic.

These features remain outside of the C++ standard and require external libraries or custom implementations.

1.5.10 Misconception 9: Surrogate pairs are irrelevant in modern C++

Surrogate pairs are a concept specific to UTF-16 encoding. Some developers assume that because modern applications often use UTF-8, surrogate logic is irrelevant. However, many

platforms, APIs, and libraries still use UTF-16 (for example, Windows APIs, certain text frameworks, and some file formats). Proper Unicode handling in mixed ecosystems requires understanding surrogate pairs and how they affect indexing, length calculation, and interoperability with non-UTF-8 APIs.

1.5.11 Misconception 10: Character width equals code point count

In terminals, text editors, and graphical user interfaces, what counts as one character in terms of display width can vary. East Asian scripts often assign double column width to certain characters. Emoji sequences may combine into single visible units with variable width. Thus, code point count, code unit count, and display column count are all distinct concepts. Algorithms that rely on code point counts to compute on-screen width will produce incorrect results for many scripts.

1.5.12 How to avoid these pitfalls in C++

To work correctly and portably with Unicode text in C++, follow these practical guidelines:

- Choose and document a text encoding (UTF-8 recommended) at API boundaries.
- Distinguish between code units, Unicode scalar values (code points), and grapheme clusters in both API design and implementation.
- Use specialized libraries for Unicode normalization and segmentation when user-perceived text operations are required.
- Normalize text at defined boundaries for comparison and storage.
- Avoid treating `std::string` as a sequence of visual characters.
- Use `char8_t`, `char16_t`, and `char32_t` appropriately to signal encoding and reduce ambiguity.

1.5.13 Summary

Incorrect assumptions about Unicode semantics are a leading source of bugs and security issues in text processing code. By dispelling these common misconceptions and adopting a precise mental model, C++ developers can write text-handling code that is robust, correct, and maintainable across languages and scripts. Modern C++ provides clearer encoding types and literal support, but full Unicode handling remains a higher-level concern that must be explicitly adopted and implemented.

1.6 Why Text Is Harder Than It Appears

1.6.1 The illusion: ”text is just bytes”

Text looks simple because it *renders* as a linear sequence of visible symbols. In code, however, text is a layered system with interacting rules:

- **Storage layer**: sequences of code units (bytes in UTF-8, 16-bit units in UTF-16, 32-bit units in UTF-32).
- **Unicode layer**: code points (Unicode scalar values), normalization, properties, and semantic equivalences.
- **User-perception layer**: grapheme clusters (what users perceive as one character), caret movement, deletion behavior.
- **Rendering layer**: glyph shaping, font fallback, ligatures, bidi layout, and line breaking.

A C++ program that treats text as a simple `std::string` of ”characters” often conflates these layers, producing incorrect results once it encounters non-ASCII input, mixed scripts, emojis, or combining marks.

1.6.2 Many encodings, many boundaries

Unicode is not a single encoding; it is a universal character set with multiple encoding forms (UTF-8/16/32). The same code point can occupy different numbers of code units depending on encoding.

Why this matters in C++ The standard library string APIs operate on code units. That is correct for the storage layer, but it is not what the user expects if you attempt to ”count characters” or truncate visible text.

UTF-8: the dominant interchange encoding, not a random byte string

UTF-8 is the most common encoding for interchange and storage because it is compact for ASCII-heavy text and compatible with byte-oriented systems. But UTF-8 is variable-length: one code point uses 1–4 bytes. That means:

- Indexing a UTF-8 `std::string` yields a **byte**, not a character.
- Truncating by bytes can split a code point, producing ill-formed UTF-8.
- Many ”simple” operations (length, substring, reverse) become encoding-sensitive.

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

static bool is_valid_utf8_prefix_byte(unsigned char b) {
    // Leading bytes are:
    // 0xxxxxxx (ASCII), 110xxxxx, 1110xxxx, 11110xxx
    return (b <= 0x7F) || ((b & 0xE0) == 0xC0) || ((b & 0xF0) == 0xE0) || ((b & 0xF8) ==
    ↪  0xF0);
}

int main() {
    // Contains a multi-byte UTF-8 character.
    std::string s = "A€B"; // If the source file is UTF-8; otherwise this is
    ↪  implementation-defined.
    std::cout << "bytes=" << s.size() << "\n";

    // Naive truncation: keep first 2 bytes
    std::string t = s.substr(0, 2);

    std::cout << "t bytes=" << t.size() << "\n";
    if (!t.empty() && !is_valid_utf8_prefix_byte(static_cast<unsigned char>(t[0]))) {
```

```
        std::cout << "t begins with an invalid UTF-8 leading byte\n";
    }

    // The real issue: t may end in the middle of a multi-byte sequence.
    // A robust UTF-8 truncation must truncate on a code point boundary.
}
```

Key point: even ”safe” standard operations like `substr` are only safe relative to *code units*. If you intend code point or grapheme semantics, you must add a decoding/segmentation layer.

1.6.3 One visible character may be multiple code points

Users interact with text as *grapheme clusters*: base characters plus combining marks, emoji sequences joined by special joiners, and other multi-code-point sequences. A user pressing backspace expects to delete one visible symbol, not one code unit or one code point.

C++ consequence Even if you decode UTF-8 to code points (e.g., into `char32_t`), iterating by code points is still not equivalent to iterating by user-perceived characters.

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

int main() {
    // "é" can be represented as either one code point or two code points.
    std::u32string nfc = U"\u00E9";        // precomposed
    std::u32string nfd = U"\u0065\u0301";  // decomposed: 'e' + combining acute

    std::cout << "nfc code_points=" << nfc.size() << "\n";
    std::cout << "nfd code_points=" << nfd.size() << "\n";

    auto dump = [](const std::u32string& s) {
```

```
        for (char32_t cp : s) {
            std::cout << "U+" << std::hex << std::uppercase
                      << std::setw(4) << std::setfill('0')
                      << static_cast<unsigned>(cp) << ' ';
        }
        std::cout << std::dec << "\n";
    };

    dump(nfc);
    dump(nfd);

    // Visually these may appear identical, but code-point sequences differ.
}
```

1.6.4 Normalization: ”equal” strings may not compare equal

Unicode allows multiple equivalent representations of the same text. Normalization defines canonical forms (e.g., NFC/NFD) so that equivalences become byte- or code-point-equal after conversion to a chosen form.

Where bugs happen

- Searching user input in stored text fails because one side is normalized and the other is not.
- Equality tests fail for visually identical strings.
- Security issues arise in identifier-like strings (confusables and mixed normalization).

C++ reality The standard library does not provide Unicode normalization; it must be applied explicitly at well-defined boundaries (input ingestion, storage, comparison, indexing).

1.6.5 Case conversion and comparison are not ”ASCII with extra letters”

Many developers assume case-insensitive operations are trivial: apply `tolower` and compare. That works only for ASCII in a fixed locale. Unicode case handling involves:

- **Locale-dependent rules** (language-specific casing).
- **Case folding** (a locale-independent comparison-oriented mapping).
- **One-to-many mappings** (case conversion can change string length).

C++ pitfall: `std::tolower` is not Unicode-aware `std::tolower` operates on `unsigned char` values (or `EOF`) and uses the currently imbued C locale facets; it is not a Unicode case folding solution. A robust Unicode case-insensitive comparison requires a Unicode-aware case folding algorithm plus (often) normalization.

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

int main() {
    // This demonstrates a common mistake: applying std::tolower to UTF-8 bytes.
    std::string s = "Straße"; // If UTF-8 source; otherwise implementation-defined.

    for (unsigned char& b : reinterpret_cast<unsigned char&>(*s.data()), *end =
    ↪  reinterpret_cast<unsigned char*>(s.data() + s.size()); &b != end; ++(&b)) {
        // This is intentionally incorrect and shown only as a warning pattern.
        // std::tolower expects a single character in the current locale, not UTF-8 bytes.
    }

    std::cout << "Do not apply std::tolower to UTF-8 bytes.\n";
}
```

Policy-level takeaway: define what ”case-insensitive” means in your domain and implement it with Unicode-aware components rather than byte-wise ctype functions.

1.6.6 Collation and sorting are not lexicographic byte order

Humans do not sort text by code point or byte values. Correct collation depends on language rules (and sometimes user preferences). A byte-wise sort of UTF-8 is not culturally meaningful; even code-point order is not equivalent to collation.

C++ reality

- `operator<` on `std::string` is byte-wise lexicographic comparison.
- `std::locale` facets can help for some narrow use cases but do not standardize full Unicode collation across platforms.

If your application promises human-friendly sorting, treat it as a specialized feature requiring dedicated collation logic and test datasets.

1.6.7 Bidirectional text breaks naive assumptions

Many scripts (Arabic, Hebrew) are written right-to-left, while numbers and embedded Latin segments are left-to-right. Correct display uses the Unicode bidirectional algorithm. This introduces counterintuitive behaviors:

- The logical order of code points differs from visual order.
- Cursor movement, selection, and editing semantics require bidi awareness.
- Mixed-direction strings can appear reordered if not handled correctly.

C++ implication Even if your internal storage is correct UTF-8, your UI layer must handle bidi, otherwise display and editing will be incorrect. This is not solvable by changing `char` to `wchar_t`.

1.6.8 Rendering: glyph shaping is not one glyph per code point

Complex scripts require shaping (context-sensitive glyph selection and positioning). In addition:

- Ligatures can map multiple characters to one glyph.
- Font fallback can change metrics and appearance.
- Emoji rendering is font- and platform-dependent.

A ”string length” is therefore not a reliable measure of on-screen width. Terminal width (columns) and GUI width (pixels) require dedicated measurement.

1.6.9 Line breaking, word boundaries, and ”what is a word?”

Splitting on spaces works for some languages but fails broadly:

- Some languages do not use spaces in the same way for word boundaries.
- Punctuation, combining marks, and joiners complicate segmentation.
- Emoji sequences and punctuation can be treated as units in UIs.

If your program highlights words, wraps lines, or counts ”words,” you need Unicode segmentation rules, not ASCII heuristics.

1.6.10 I/O and the operating system: correctness depends on boundaries

Even perfect internal Unicode logic can fail at system boundaries:

- File systems may store or compare names with normalization behavior that differs across platforms.
- Windows APIs often use UTF-16; many Unix-like systems use UTF-8 bytes with fewer guarantees.
- Terminal encodings and fonts affect display and input.
- Network protocols define their own encoding requirements; you must obey them explicitly.

Engineering rule Text correctness is achieved by **explicitly specifying encoding and normalization at every boundary**: file I/O, environment variables, command-line arguments, network payloads, database storage, UI input/output.

1.6.11 Security: text bugs can become vulnerabilities

Unicode mistakes are not just correctness issues:

- **Validation bypass**: different normalizations or encodings can evade filters.
- **Confusable identifiers**: visually similar characters can spoof names.
- **Truncation/overflow bugs**: byte counts mistaken for character counts can break limits.
- **Injection risks**: incorrect boundary handling can enable injection in downstream systems.

In security-sensitive code, treat text handling as part of your threat model: normalize, validate, and reject malformed encodings consistently.

1.6.12 Why C++ makes it easy to get wrong

C++ is a systems language with strong emphasis on performance and explicit control. That means:

- Strings are containers of code units, not semantic text objects.
- The standard library does not standardize full Unicode text algorithms.
- Multiple character types exist, but choosing the type does not automatically solve normalization, segmentation, bidi, or collation.
- Interoperability pressures (OS APIs, libraries, legacy encodings) push programs into mixed-encoding environments.

1.6.13 Practical mental model (what to decide before writing code)

Before implementing any text feature, answer these questions explicitly:

1. **Encoding**: What encoding is used at each boundary? (e.g., UTF-8 for files and network; UTF-16 for specific OS APIs)
2. **Validity policy**: Do you reject ill-formed input or replace with U+FFFD?
3. **Normalization policy**: Do you normalize on input (and to which form)?
4. **Unit of operation**: Are your algorithms byte-based, code-point-based, or grapheme-based?
5. **Comparison policy**: Do you need locale collation, or binary equality, or case-folded matching?
6. **UI semantics**: Does the feature require bidi, shaping, and grapheme-aware cursoring?

1.6.14 Summary

Text is hard because it combines variable-length encodings, multiple equivalent representations, user-perceived units that differ from stored units, locale- and language-dependent rules, bidirectional behavior, and rendering complexities. C++20–C++26 provides clearer tools to *express encoding intent* (notably UTF-8 literals and `char8_t`), but it does not turn `std::string` into a Unicode text object. Correct text handling in modern C++ requires deliberate design: explicit boundary contracts, explicit policies (validity/normalization/comparison), and Unicode-aware algorithms where human-facing behavior is required.

Chapter 2

Text Types and Character Representations in C++

2.1 `char` and `std::string`: What They Represent and What They Do Not

2.1.1 Introduction

In C++, `char` and `std::string` are the fundamental building blocks for text and byte storage. However, their semantics are often misunderstood in Unicode and modern text processing contexts. This section explains precisely what they represent, what assumptions they carry, and what they do not guarantee. Understanding these distinctions is critical when writing correct text handling code from C++20 through C++26.

2.1.2 `char`: storage unit, not Unicode character

The built-in type `char` in C++ is a fundamental integer type that occupies at least 8 bits. By definition, it is a *storage unit* for raw data. It has no inherent meaning regarding text encoding. When you see `char` in text code, it means *eight bits of storage*.

What `char` represents

- A container for an 8-bit value.
- An element in a sequence of raw bytes when used in `std::string`.
- A unit of storage for user-defined encoding formats, including UTF-8, ASCII, or binary protocols.

What `char` does not represent

- It does not inherently represent a Unicode code point.
- It does not guarantee that one `char` equals one user-visible character.
- It does not enforce any specific encoding.

Thus, `char` is best thought of as a byte in the context of modern C++: a minimal addressable unit of storage.

2.1.3 `std::string`: sequence of code units

The class template `std::basic_string<T>` provides a contiguous sequence of elements of type `T`. The alias `std::string` specializes this for `T = char`. Therefore, a `std::string` is a contiguous sequence of `char` elements.

What `std::string` represents

- A sequence of bytes (8-bit code units).
- A dynamically resizable container with well-defined semantics for accessing, iterating, and modifying its elements.
- A buffer for storing data of any kind: text in a known encoding, binary data, network frames, file contents, etc.

What `std::string` does not represent

- It does not guarantee Unicode semantics.
- It does not provide normalization, segmentation, or collation rules.
- It does not interpret the bytes as characters unless the programmer defines an encoding contract.

In practice, when using `std::string` for text, programmers adopt a *convention* such as UTF-8 encoding. However, this is a project-level contract, not something enforced by the language or the standard library.

2.1.4 Literal encodings and their impact on `std::string`

C++ provides literal prefixes that influence how string literals are encoded in the program text.

- `"..."` produces an array of `char`. The encoding is implementation-defined if not otherwise specified (commonly UTF-8 in modern toolchains with appropriate flags).
- `u8"..."` produces an array of `char8_t`, which is explicitly designated for UTF-8 code units.

- `u"..."` produces an array of `char16_t`, typically for UTF-16 code units.
- `U"..."` produces an array of `char32_t`, typically for UTF-32 code units.
- `L"..."` produces an array of `wchar_t`, whose width is implementation-defined.

When a string literal without a prefix is used to initialize a `std::string`, the compiler stores the literal's bytes into the resulting object. Whether these bytes represent ASCII, UTF-8, or some other encoding depends on the translation unit's encoding and build settings.

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

int main() {
    std::string s1 = "Hello";      // bytes only, encoding depends on source
    std::string s2 = u8"Hello";    // UTF-8 encoded sequence of char8_t,
                                   // implicitly cast to char during initialization
    std::cout << s1.size() << "\n";
    std::cout << s2.size() << "\n";
}
```

Key insight Even though `u8""` is UTF-8 by definition, once data is stored in `std::string`, the type remains a container of bytes. There is no Unicode functor applied implicitly during operations.

2.1.5 `char` signedness and portability

Different platforms define `char` as either signed or unsigned. This affects numerical interpretations but not storage layout:

- On some platforms, `char` is signed by default (`-128` to `127`).
- On others, `char` is unsigned (`0` to `255`).

This signedness does not influence how many bits `char` occupies, but it does influence how values beyond `0x7F` are promoted and compared in arithmetic and character classification functions.

2.1.6 `char8_t` and UTF-8 clarity

C++20 introduced `char8_t` as a distinct type to represent UTF-8 code units explicitly. This reduces ambiguity when interacting with APIs that accept or return sequences of bytes intended to be UTF-8 encoded.

Benefits of using `char8_t` over `char`

- Self-documenting intent: the type signals that each element is a UTF-8 code unit.
- Reduces accidental misuse of APIs that expect `char` for generic bytes.
- Improves overload resolution in templates, enabling compile-time distinction between generic byte buffers and UTF-8 text.

However, even with `char8_t`, `std::u8string` is still a sequence of code units. It does not provide higher-level Unicode manipulations out of the box.

2.1.7 `std::string_view` and non-ownership semantics

The type `std::string_view` represents a non-owning view into a sequence of `char` values. It provides constant-time slicing and affordable passing into functions without copying.

Important semantics of `std::string_view`

- It does not own the underlying storage.
- It does not manage lifetime; the programmer must ensure the referred data remains valid.

- It does not provide Unicode semantics; it views contiguous `char` elements.

These properties make `std::string_view` useful for efficient APIs, but text processing code must still interpret the underlying bytes according to the chosen encoding.

2.1.8 Comparisons: what `std::string` compares

By default, `std::string` comparisons (`operator==`, `operator<`, `compare()`) are *lexicographic comparisons of code unit sequences*, not Unicode-aware comparisons. Thus:

- Byte-wise equality does not imply Unicode equivalence if different normalization forms are used.
- Sorting using `operator<` does not produce culturally meaningful order for many languages.

If Unicode collation is required, the programmer must apply normalization and collation logic before comparison.

2.1.9 Interoperability with Unicode APIs

Many operating system APIs expect Unicode encodings. For example, POSIX interfaces often operate with UTF-8 byte sequences, while some native APIs (e.g., Windows wide char APIs) use UTF-16. Converting between UTF-8 stored in `std::string` and UTF-16 often requires explicit transcoding routines.

```
#include <string>
#include <locale>
#include <codecvt>

// Example: converting UTF-8 std::string to UTF-16 std::u16string
// The following pattern represents conceptual logic; production code
```

```
// should use a tested library for Unicode conversions.
std::u16string utf8_to_utf16(const std::string& s) {
    std::u16string result;
    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> convert;
    result = convert.from_bytes(s);
    return result;
}
```

The above pattern illustrates that text conversion requires explicit intent and mechanism. The standard library provides limited utilities; modern C++ codebases often use specialized libraries for comprehensive Unicode handling.

2.1.10 Summary: contracts over conventions

To avoid bugs and ambiguities in C++ text handling:

- Treat `char` as a storage unit without semantic text meaning.
- Treat `std::string` as a sequence of code units; assign it an encoding contract explicitly (e.g., UTF-8).
- Use `char8_t` and `std::u8string` to clarify UTF-8 usage.
- Do not assume that `size()`, indexing, or comparisons on `std::string` correspond to user-perceived characters.
- Apply explicit normalization and collation logic when semantic text operations are required.

By understanding what `char` and `std::string` *represent* and what they *do not*, C++ developers can build robust, correct text handling systems that meet modern Unicode requirements without hidden assumptions.

2.2 Signedness, Encoding Assumptions, and Portability Issues

2.2.1 Why this topic belongs in a Unicode booklet

Many Unicode failures in C++ are not caused by ”Unicode complexity” alone, but by subtle low-level issues:

- **Signedness of** `char` differs between platforms and compiler modes.
- **Character classification APIs** (like `std::isalpha`) have preconditions many developers violate.
- **Encoding assumptions** leak into algorithms (”one byte = one character”, ”`wchar_t` is Unicode”).
- **Portability gaps** arise from OS conventions (UTF-16 vs UTF-8), locale behavior, and build settings.

These issues lead to misparsing, incorrect comparisons, data corruption, security problems, and cross-platform bugs that only appear with non-ASCII input.

2.2.2 Signedness of `char`: what the language guarantees (and what it does not)

`char` is a distinct type with implementation-defined signedness

C++ defines three distinct character types: `char`, `signed char`, and `unsigned char`. Only `char` has *implementation-defined* signedness. On some platforms `char` behaves like `signed char`; on others it behaves like `unsigned char`. The choice can also be affected by compiler flags in some toolchains.

What is guaranteed

- `char` is at least 8 bits.
- `sizeof(char) == 1`.
- `char` can represent the basic execution character set values (at least the required set).

What is not guaranteed

- That `char` values are non-negative.
- That `char` can represent all byte values 0–255 as non-negative integers.

Why signedness becomes a real bug in text code

In UTF-8, any non-ASCII byte has the high bit set (values `0x80`–`0xFF`). If `char` is signed, these bytes become negative when converted to `int`. This causes trouble when:

- passing a `char` directly to `std::is*` and `std::to*` functions,
- indexing lookup tables with `char`,
- writing comparisons like `if (c <0)` to detect ”special” bytes,
- performing bitwise operations after implicit promotions.

2.2.3 The `std::ctype` family: a classic portability trap

The precondition many developers violate

The C and C++ character classification functions (e.g., `std::isalpha`, `std::isdigit`, `std::tolower`) have a critical precondition:

Their argument must be representable as `unsigned char` (or be `EOF` in the C interface).

Passing a negative `char` value that is not `EOF` is undefined behavior.

Correct pattern: cast to `unsigned char` first

```
#include <cctype>
#include <string>
#include <vector>

bool is_ascii_letter(unsigned char b) {
    // ASCII-only classification; independent of locale.
    return (b >= 'A' && b <= 'Z') || (b >= 'a' && b <= 'z');
}

std::string ascii_only_lower(std::string s) {
    for (char& ch : s) {
        unsigned char b = static_cast<unsigned char>(ch);
        // std::tolower is locale-dependent; this is an example of safe calling convention.
        ch = static_cast<char>(std::tolower(b));
    }
    return s;
}
```

Important note Even when used safely, these functions are **not Unicode-aware**. They classify and transform according to the current C locale (or facets), and they operate on single-byte character values, not UTF-8 sequences.

2.2.4 Encoding assumptions that break portability

Assumption 1: "`std::string` is UTF-8"

`std::string` is a container of `char` code units. The C++ standard does not require any particular encoding for those bytes. If your system uses UTF-8, that is a **project contract**, not a language guarantee.

How it breaks

- Source files not saved as UTF-8, or compiled without the intended source/execution character set options.
- Data read from external systems (files, DB, network) with different encodings.
- Incorrect mixing of "bytes" and "text" APIs.

Assumption 2: "`"..."` literals are UTF-8"

A narrow string literal `"..."` produces `const char[N]`. The mapping from source characters to literal bytes depends on translation phases and implementation settings. Some toolchains default to UTF-8 in many environments, but you must not rely on this for portability.

Portable intent expression

- Use `u8"..."` when you want UTF-8 by language rule (C++20: type is `const char8_t[N]`).
- Use `u"..."` for UTF-16 code units, and `U"..."` for UTF-32 code units.

```
#include <string>

int main() {
    // Explicitly UTF-8 by language rule (array of char8_t in C++20+)
    auto p = u8"     ";
```

```
    // Explicitly UTF-32 code units (often convenient for code-point demonstrations)
    std::u32string s32 = U"     ";
}
```

Assumption 3: "`wchar_t` is Unicode"

`wchar_t` is a wide character type whose size and encoding are implementation-defined.

- On Windows, `wchar_t` is typically 16 bits (UTF-16 code units).
- On many Unix-like systems, `wchar_t` is typically 32 bits (UTF-32 code units).

In both cases, `wchar_t` values are code units, not "characters" in the user-perceived sense. UTF-16 requires surrogate pairs for non-BMP code points, and neither UTF-16 nor UTF-32 inherently provides grapheme clustering, normalization, or collation.

2.2.5 Portability hazards across platforms and toolchains

Windows vs POSIX conventions

A common cross-platform pain point is that many Windows-facing APIs use UTF-16 code units, while POSIX environments commonly treat pathnames and environment strings as opaque byte sequences (with UTF-8 being a strong convention in many modern systems, but not a universal guarantee).

Consequence for C++ codebases

- You must define an internal canonical encoding (often UTF-8) and provide explicit boundary conversions.
- You must treat OS boundaries (file paths, command line, environment) as encoding-sensitive interfaces.

Locale variability

Locale affects:

- parsing and formatting (numbers, dates, punctuation),
- classification and case conversion via locale facets and C locale functions,
- collation in limited contexts.

Different systems ship different locale data and behavior. Even when the code is correct, results may differ. For Unicode semantics (normalization, graphemes, case folding), locale is not a substitute for a Unicode algorithm.

The execution character set and build flags

Portability depends on correctly aligning:

- source file encoding,
- compiler interpretation of source encoding,
- runtime expectations (file I/O, terminal, UI),
- external data encoding.

If any of these disagree, you can get mojibake (garbled text), failed comparisons, and data-loss conversions.

2.2.6 Byte vs text APIs: a robust boundary rule

A portable, maintainable codebase separates two worlds:

World A: bytes (`std::byte` / `unsigned char` buffers)

Used for:

- compression, encryption, hashing,
- network frames, binary file formats,
- serialization, checksums.

World B: text (explicit encoding contract)

Used for:

- user input/output,
- UI strings, messages, logs,
- protocol text fields, JSON, XML (with declared encodings).

Rule Never pass ”text bytes” through algorithms that assume ASCII classification unless you validate/limit to ASCII explicitly.

2.2.7 UTF-8 validation: why ”just trust the input” is unsafe

Portability and security both require deciding what to do with ill-formed byte sequences. A UTF-8 pipeline should define a policy:

- **Reject**: treat invalid UTF-8 as an error at boundaries.
- **Replace**: substitute the Unicode replacement character conceptually (U+FFFD) at decode time.

- **Preserve bytes**: treat input as raw bytes and defer interpretation (rarely correct for UI text).

Even when you do not implement full Unicode processing, validating UTF-8 at boundaries prevents downstream bugs and avoids undefined behavior in naive decoders.

2.2.8 Practical patterns for portable C++20–C++26 text code

Pattern 1: safe classification wrapper for ASCII-only logic

If you truly want ASCII-only behavior, express it explicitly and avoid locale traps:

```
#include <string_view>

bool is_ascii_digit(unsigned char b) { return b >= '0' && b <= '9'; }

bool is_ascii_identifier(std::string_view s) {
    if (s.empty()) return false;
    auto is_alpha_ = [](unsigned char b) {
        return (b >= 'A' && b <= 'Z') || (b >= 'a' && b <= 'z') || b == '_';
    };
    if (!is_alpha_(static_cast<unsigned char>(s[0]))) return false;
    for (unsigned char b : s) {
        if (!(is_alpha_(b) || is_ascii_digit(b))) return false;
    }
    return true;
}
```

Pattern 2: enforce encoding at API boundaries

Define a project rule such as:

- all internal text is UTF-8 in `std::string`,

- all external inputs are validated as UTF-8 on ingestion,
- all OS/API boundaries perform explicit conversions as required.

This converts ”portability” from guesswork into an engineering contract.

Pattern 3: treat `char` as ”byte” in UTF-8 text, but never as ”character”

In UTF-8:

- `char` / `char8_t` elements are code units (bytes),
- ”character” can mean code point or grapheme cluster depending on the feature,
- algorithms must explicitly decode or segment to operate at those higher levels.

2.2.9 Checklist: common portability failures to catch in review

- Passing `char` directly to `std::tolower`/`std::isalpha` without `unsigned char` cast.
- Using `std::string::size()` as ”character count” for UTF-8 user text.
- Truncating UTF-8 by bytes without checking code point boundaries.
- Assuming `"..."` literals are UTF-8 without an explicit policy/toolchain setting.
- Assuming `wchar_t` is portable Unicode.
- Mixing locale-sensitive classification with encoding-agnostic byte sequences.
- Comparing visually identical text without normalization policy.

2.2.10 Summary

Signedness, encoding assumptions, and portability issues are foundational to Unicode correctness in C++. `char` is a storage type with implementation-defined signedness; misusing it in classification or indexing can trigger undefined behavior and subtle bugs, especially with UTF-8 non-ASCII bytes. `std::string` is a byte container that does not enforce Unicode semantics, and narrow literals do not universally guarantee UTF-8 without an explicit contract. Portability requires disciplined boundary management: explicit encoding policies, safe casting conventions for classification APIs, and explicit conversions when crossing OS and library interfaces.

2.3 wchar_t: Platform Differences and Design Pitfalls

2.3.1 Introduction

In modern C++ text handling, wchar_t remains a legacy wide character type that many developers misunderstand. Although wchar_t appears to provide ”wide characters,” its semantics, size, and encoding representation differ across platforms. Overreliance on wchar_t often causes portability issues, hidden bugs, and incorrect assumptions about Unicode support. This section explains exactly what wchar_t is, how it behaves on different systems, and what design pitfalls to avoid.

2.3.2 What wchar_t actually is

wchar_t is a built-in fundamental integral type introduced in early C and C++ to represent “wide characters.” Unlike char, whose size is standardized to be at least 8 bits and typically exactly 8 bits, wchar_t has an implementation-defined size. Its main purpose historically was to provide a type capable of holding larger character sets than the narrow char, but the language standard does not mandate an encoding or a specific bit-width.

2.3.3 Platform differences: size and encoding

Windows

On Windows platforms (MSVC toolchain), wchar_t is typically 16 bits. It is conventionally used to store UTF-16 code units. While this aligns with many Windows APIs (file and registry APIs, window text, resource loading), it introduces the need to handle surrogate pairs for code points above U+FFFF. Therefore:

- wchar_t on Windows represents UTF-16 code units, not full Unicode code points.

- Algorithms that treat each `wchar_t` as one Unicode character will break for characters outside the Basic Multilingual Plane (BMP).
- Surrogate pair detection and handling is mandatory for correct text processing with UTF-16.

Unix and Unix-like systems

On most Unix and Unix-like systems (GCC and Clang on Linux and BSD), `wchar_t` is typically 32 bits. In practice this aligns with UTF-32 code units (i.e., one `wchar_t` element can hold a full Unicode scalar value without surrogate pairs). However:

- There is still no built-in handling of grapheme clusters, normalization, or collation.
- `wchar_t` is not automatically interoperable with UTF-8, which is the predominant encoding for text storage and interchange on modern Unix systems.

Other compilers and embedded systems

Compilers targeting embedded platforms, special environments, or non-mainstream toolchains may choose a different width for `wchar_t`. There is no requirement in the language that `wchar_t` use a particular number of bits. Some toolchains even treat `wchar_t` the same as `char`.

2.3.4 What `wchar_t` does not guarantee

`wchar_t` does not provide any of the following by itself:

- Unicode normalization semantics (NFC/NFD/NFKC/NFKD)
- Grapheme cluster segmentation
- Locale-independent classification or case folding

- Bidirectional text support
- Collation semantics beyond basic code unit comparison

Even if `wchar_t` can represent a code point in one element (as on many Unix systems), algorithms must still implement higher-level Unicode logic explicitly. Treating `wchar_t` as a full Unicode abstraction is a misconception.

2.3.5 Pitfalls and incorrect assumptions

Assumption: `wchar_t` equals a Unicode character

Developers often assume that one `wchar_t` equals one user-visible character. This is false even when it holds in a particular build environment:

- Combining marks may follow a base code unit, forming a grapheme cluster that consists of multiple code points.
- Surrogate pairs in UTF-16 hold a single Unicode scalar value across two `wchar_t` elements.
- A glyph in rendering may combine multiple grapheme clusters (ligatures) not evident in the underlying `wchar_t` sequence.

Pitfall: using `wchar_t` for portable Unicode APIs

The C++ standard library provides wide character I/O and conversion facets (e.g., `std::wcout`, `std::wstring_convert`), but these do not standardize Unicode text semantics. Portability issues arise when:

- Expecting consistent behavior across platforms with different `wchar_t` widths.

- Mixing narrow and wide APIs without explicit encoding contracts.
- Writing code that implicitly assumes UTF-16 or UTF-32 behavior without checking at compile time.

For example, code that iterates over `std::wstring` and treats each `wchar_t` as a code point will behave differently on Windows (UTF-16) versus Linux (UTF-32) when encountering non-BMP characters.

2.3.6 Interoperability with operating systems and libraries

Windows APIs

Windows native APIs (such as file I/O, message text, resource loading) predominantly use UTF-16 encoded `wchar_t` sequences. A C++ program that uses wide APIs directly must correctly handle surrogate pairs and ensure boundary safety when converting to or from UTF-8.

```
#include <string>
#include <windows.h>

// Example: converting UTF-8 std::string to UTF-16 std::wstring
std::wstring utf8_to_utf16(const std::string& utf8) {
    if (utf8.empty()) return {};
    int required = MultiByteToWideChar(CP_UTF8, 0, utf8.data(), int(utf8.size()), nullptr,
    ↪  0);
    if (required == 0) return {};
    std::wstring result(required, L' ');
    MultiByteToWideChar(CP_UTF8, 0, utf8.data(), int(utf8.size()), result.data(),
    ↪  required);
    return result;
}
```

This pattern demonstrates an explicit encoding conversion; relying on implicit promotions or assumptions about `wchar_t` layout will not produce portable code.

POSIX and Unix character interfaces

POSIX interfaces commonly use narrow character APIs designed around byte sequences. Modern Unix environments strongly favor UTF-8 for text, but the language and OS do not enforce this. Converting between `wchar_t` and UTF-8 requires explicit transcoding logic, and the wide APIs (e.g., `wcwidth`, `wcslen`) operate on code units without higher-level Unicode semantics.

2.3.7 Case conversion and classification with wide types

Standard facilities like `std::towlower`, `std::towupper`, and `std::iswalpha` operate on `wint_t` (which is typically a promoted `wchar_t`) according to the current locale. These functions categorize and transform wide code units in a locale-specific manner. However:

- They are not Unicode case folding by default.
- They operate on single code units, not considering sequences, combining marks, or code point boundaries.
- Their behavior depends on the current C locale, which varies across platforms and runtimes.

Using these functions without understanding locale semantics can introduce subtle bugs.

2.3.8 Alternatives and modern C++ strategies

Explicit Unicode encodings: `char8_t`**,** `char16_t`**,** `char32_t`

Modern C++ (since C++11 and refined in C++20) provides explicit character types to represent code units of specific encoding forms:

- `char8_t` for UTF-8 code units.

- `char16_t` for UTF-16 code units.
- `char32_t` for UTF-32 code units.

Using these types makes text encoding intentions explicit in interfaces, improving readability and reducing ambiguity about underlying representations. For example, `std::u8string` clearly conveys that the container holds UTF-8 code units.

Unicode utility libraries

Since the standard library does not provide comprehensive Unicode algorithms (normalization, segmentation, collation, case folding, bidi), many projects integrate dedicated libraries that support modern Unicode standards. Such libraries explicitly handle encoding forms and provide portable semantics across platforms. Integrating them avoids reinventing low-level text processing and reduces platform divergence.

2.3.9 Practical guidance for C++ text code

- Avoid using `wchar_t` as a default text type for Unicode.
- When interfacing with platform APIs that require wide strings, encapsulate conversion logic clearly and test on all target platforms.
- Treat `wchar_t` as a legacy, platform-dependent code unit type, not a portable Unicode abstraction.
- Prefer explicit encoding types (`char8_t`, `char16_t`, `char32_t`) for text storage and processing.
- Use reputable Unicode libraries for anything beyond storage and simple byte-sequence manipulation.

2.3.10 Summary

`wchar_t` is a wide character type whose size, encoding semantics, and runtime behavior vary across platforms. On Windows it typically represents UTF-16 code units, requiring surrogate pair handling; on many Unix systems it represents UTF-32 code units. However, `wchar_t` does not provide high-level Unicode semantics such as normalization, grapheme clustering, or collation. Relying on it as a Unicode type compromises portability and correctness. Modern C++ encourages explicit encoding types and clear encoding contracts, and production text handling code should use explicit transcoding layers and Unicode utility libraries to achieve correct and portable behavior.

2.4 `char8_t`, `char16_t`, and `char32_t`: Motivation and Semantics

2.4.1 Introduction

Modern C++ provides explicit character types that correspond to specific Unicode encoding code unit widths: `char8_t`, `char16_t`, and `char32_t`. These types were introduced to improve clarity, remove historical ambiguity, and enable safer text processing in multilingual, cross-platform systems. They promote self-documenting code, help distinguish text encoding from raw bytes, and inform API design.
In this section, we describe the motivation behind these types, the semantics they embody, and how they relate to Unicode encoding forms.

2.4.2 Why new character types were needed

`char` historically served both as a representation of bytes and sometimes text, depending on context. This dual role created ambiguity:

- Is `std::string` a container of raw binary data or Unicode text?
- What encoding do literal bytes represent?
- Should character classification and text algorithms use `char` directly?

To remove this ambiguity and align with the Unicode standard's encoding forms (UTF-8, UTF-16, UTF-32), C++ introduced specific character types corresponding to code unit widths.

2.4.3 `char8_t`: explicit UTF-8 code unit type

Motivation

UTF-8 has become the dominant encoding for text storage, interchange, and APIs in modern systems due to its backward compatibility with ASCII, byte-orientation, and widespread adoption. However, before C++20, UTF-8 text was commonly stored in `std::string`, but the type did not signal encoding intent:

- A function taking `std::string` might be handling binary blobs, ASCII text, or UTF-8 text.
- Mixing binary protocols and text protocols led to fragile code.
- Encoding assumptions were implicit rather than explicit.

Semantics

`char8_t` unambiguously denotes an 8-bit code unit in a UTF-8 encoded text sequence. A sequence of `char8_t` code units represents an encoded sequence of Unicode code points under the UTF-8 encoding scheme. This makes intent explicit at the type level.

Container alias

- `std::u8string` is an alias for `std::basic_string<char8_t>`.
- Literal prefix `u8""` produces a sequence of `char8_t` elements.

What `char8_t` gives you

- Type-level distinction between UTF-8 text and arbitrary byte sequences.
- Stronger overload resolution in templates involving text vs binary data.
- Clear API contracts with explicit encoding expectations.

What `char8_t` does not give you

- Built-in Unicode segmentation or normalization algorithms.
- Automatic iteration over code points or grapheme clusters.
- Collation or case folding.

2.4.4 `char16_t`: explicit UTF-16 code unit type

Motivation

UTF-16 is widely used in operating system APIs (notably on Windows and some mobile platforms) and some text storage formats. Like UTF-8, UTF-16 is a concrete encoding form with well-defined rules. Before C++11, wide character types like `wchar_t` were used to approximate wider code units, but their size varied by platform, which made code non-portable and encoding ambiguous.

Semantics

`char16_t` denotes a 16-bit code unit in a UTF-16 encoded sequence. UTF-16 uses single 16-bit units for most common Unicode scalar values, but employs *surrogate pairs* (two 16-bit units) to encode code points beyond `U+FFFF`. Therefore:

- `char16_t` code units map to UTF-16 code units.
- Two consecutive `char16_t` elements form a surrogate pair for non–Basic Multilingual Plane (non-BMP) code points.
- Algorithms operating on `char16_t` sequences must handle surrogate pairs for correctness.

Container alias

- `std::u16string` is an alias for `std::basic_string<char16_t>`.
- Literal prefix `u""` produces an array of `char16_t`.

Use cases

- Interfacing with UTF-16 APIs (Windows API, some text libraries).
- Storage or protocol formats that require UTF-16.
- Situations where explicit encoding clarity improves API design.

2.4.5 `char32_t`: explicit UTF-32 code unit type

Motivation

UTF-32 uses fixed-width 32-bit code units. Each code unit corresponds directly to a Unicode scalar value (excluding surrogate code points). Prior to C++11, developers used `wchar_t` for wide character representations, but its size and encoding were platform-dependent. `char32_t` provides a portable way to represent Unicode code points in fixed-width code units.

Semantics

`char32_t` unambiguously represents a 32-bit Unicode code unit capable of holding any valid Unicode scalar value. There is no surrogate logic in UTF-32: every `char32_t` element holds exactly one code point.

Container alias

- `std::u32string` is an alias for `std::basic_string<char32_t>`.
- Literal prefix `U""` produces an array of `char32_t`.

Use cases and trade-offs

- Ideal for internal logic that operates on Unicode code points directly.
- Fixed-width simplifies iteration over code points.
- Larger memory footprint compared to UTF-8 or UTF-16.
- Still requires explicit higher-level Unicode algorithms (normalization, segmentation).

2.4.6 Literal prefixes and type inference

C++ provides literal prefixes to match these character types:

- `u8"..."` yields an array of `char8_t`.
- `u"..."` yields an array of `char16_t`.
- `U"..."` yields an array of `char32_t`.

Using these prefixes signals the intended encoding at the literal level, making the code's encoding expectations explicit and unambiguous.

2.4.7 APIs and overload resolution clarity

Introducing distinct character types improves overload resolution in templates and generic code. For example:

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

void process_text(std::u8string_view text) {
    std::cout << "UTF-8 text\n";
}
```

```
void process_text(std::u16string_view text) {
    std::cout << "UTF-16 text\n";
}

void process_text(std::u32string_view text) {
    std::cout << "UTF-32 text\n";
}

int main() {
    process_text(u8"Hello");
    process_text(u"Hello");
    process_text(U"Hello");
}
```

The above demonstrates how clear encoding intentions lead to distinct overloads and reduce ambiguity in API usage.

2.4.8 Limitations and what the types do not solve

Although `char8_t`, `char16_t`, and `char32_t` improve clarity and signal encoding intent, they do not by themselves provide:

- Unicode normalization and canonical equivalence handling.
- Grapheme cluster segmentation for user-perceived characters.
- Locale-aware collation or case folding.
- Bidirectional text layout support.

These remain application-level concerns that must be addressed through libraries or explicit implementation.

2.4.9 Interoperability and conversion patterns

Because different APIs and systems expect different encodings, conversion between UTF-8, UTF-16, and UTF-32 is common. A robust conversion pipeline explicitly decodes from one code unit sequence into Unicode code points, then re-encodes into the target code unit width. A minimal conceptual example for code point iteration of UTF-8 to UTF-32 could be:

```
#include <string>
#include <vector>
#include <cstdint>

// Conceptual: decode UTF-8 code units (char8_t) to UTF-32 code points (char32_t)
std::u32string utf8_to_utf32(std::u8string_view input) {
    std::u32string output;
    for (size_t i = 0; i < input.size();) {
        unsigned char lead = static_cast<unsigned char>(input[i]);
        // decode sequence length based on leading byte pattern
        // append Unicode scalar to output
        // advance i by sequence length
    }
    return output;
}
```

This code snippet represents the pattern: decode from one encoding's code units to Unicode scalar values, then re-encode as needed. Actual implementations must handle error conditions and conform to Unicode canonical rules.

2.4.10 Best practices

- Use `char8_t` and `std::u8string` for UTF-8 text storage and interchange.
- Use `char16_t` and `std::u16string` when interfacing with UTF-16 APIs.

- Use `char32_t` and `std::u32string` for internal Unicode code point manipulation.
- Explicitly document encoding expectations at API boundaries.
- Combine these types with robust Unicode libraries for normalization, segmentation, and collation when needed.

2.4.11 Summary

`char8_t`, `char16_t`, and `char32_t` provide explicit, unambiguous types that correspond to UTF-8, UTF-16, and UTF-32 code units, respectively. They improve clarity, enhance overload resolution, and help express encoding contracts in modern C++. However, they are building blocks for text encoding; full Unicode semantics still require explicit handling of normalization, segmentation, and other higher-level text processing. Using these types correctly is foundational to reliable and portable Unicode text handling in C++20 through C++26.

2.5 `std::u8string`, `std::u16string`, and `std::u32string`

2.5.1 Introduction

Modern C++ provides specialized string types that map directly to explicit Unicode encoding forms: `std::u8string`, `std::u16string`, and `std::u32string`. These types improve clarity by associating the encoding form with the string type itself, removing ambiguity, and facilitating safer text handling. They help define precise API contracts regarding encoding assumptions and signal intent to programmers and compilers alike.
This section explains the motivation, semantics, usage, limitations, and best practices for these string types in Unicode-aware C++20 to C++26 code.

2.5.2 `std::u8string`: UTF-8 encoded text

Definition and semantics

`std::u8string` is an alias for `std::basic_string<char8_t>`. It represents a sequence of UTF-8 code units, where each element is a `char8_t`. UTF-8 is a variable-length encoding: each Unicode code point is encoded as one to four 8-bit code units.

Literal support

Using the UTF-8 literal prefix `u8""` generates an array of `char8_t` that can be used to initialize a `std::u8string`:

```
std::u8string s = u8"Hello, world";
```

This literal makes the encoding explicit rather than relying on the translator's source character set interpretation.

Typical usage scenarios

- Interchange formats and data storage where UTF-8 is the standard (e.g., JSON, XML, web APIs).
- Text processing in cross-platform systems where a compact, ASCII-compatible encoding is preferred.
- Interfacing with I/O layers that operate on UTF-8 sequences (network, files, protocols).

Semantics of `size()` and indexing

`std::u8string::size()` returns the number of UTF-8 code units, not the number of Unicode code points or user-perceived characters (grapheme clusters). Indexing with `operator[]` yields one code unit. Algorithms that require semantic character iteration must explicitly decode into code points or grapheme clusters.

```
std::u8string s = u8"Hello";
std::size_t code_units = s.size(); // number of 8-bit units
```

2.5.3 `std::u16string`: UTF-16 encoded text

Definition and semantics

`std::u16string` is an alias for `std::basic_string<char16_t>`. It holds UTF-16 code units, each 16 bits wide. UTF-16 encodes most common Unicode code points in one code unit, but code points above `U+FFFF` require two code units (a surrogate pair).

Literal support

The prefix `u""` produces an array of `char16_t`:

```
std::u16string s16 = u"Example UTF-16";
```

Typical usage scenarios

- Interfacing with operating system APIs that expect UTF-16 encoded text (e.g., Windows wide APIs).
- Text storage or export formats that specify UTF-16 encoding.
- Situations where fixed-width code units simplify some internal algorithms (while still handling surrogate logic).

Understanding surrogate pairs

UTF-16 uses surrogate pair encoding for code points outside the Basic Multilingual Plane (BMP). A surrogate pair consists of two 16-bit code units. Correct iteration and decoding must detect high and low surrogates to reconstruct the original Unicode code point.

```
std::u16string s16 = u"\U0001F600"; // UTF-16 surrogate pair
```

In the example above, the character above `U+FFFF` is encoded as two `char16_t` units.

2.5.4 `std::u32string`: UTF-32 encoded text

Definition and semantics

`std::u32string` is an alias for `std::basic_string<char32_t>`. It holds UTF-32 code units, each 32 bits wide. UTF-32 maps one code unit to one Unicode scalar value directly, simplifying iteration over code points without surrogate pairs.

Literal support

The prefix `U""` produces an array of `char32_t`:

```
std::u32string s32 = U"Example UTF-32";
```

Typical usage scenarios

- Internal logic that must iterate over Unicode scalar values without decoding overhead.
- Algorithms that require fixed-width code units and can tolerate larger memory footprint.
- Tools that perform Unicode analysis or transformation at the code point level.

2.5.5 Differences between the string types

These string types differ primarily in code unit width and encoding semantics:

- `std::u8string` uses 8-bit code units (UTF-8), variable length per code point.
- `std::u16string` uses 16-bit code units (UTF-16), surrogate pairs for non-BMP.
- `std::u32string` uses 32-bit code units (UTF-32), one code unit per Unicode scalar value.

Each provides different trade-offs between compactness, direct code point access, and compatibility with external APIs.

2.5.6 Conversions between string types

Often applications must convert between encoding forms. For example, converting UTF-8 to UTF-32 involves decoding each UTF-8 sequence into code points and storing them as `char32_t`. Conversions must validate input and handle ill-formed sequences appropriately.

```
#include <string>
#include <vector>

// Conceptual UTF-8 to UTF-32 conversion (without error handling for brevity)
std::u32string utf8_to_u32(std::u8string_view input) {
```

```
    std::u32string output;
    for (std::size_t i = 0; i < input.size(); ) {
        unsigned char lead = static_cast<unsigned char>(input[i]);
        // Determine length of the UTF-8 sequence
        // Decode into Unicode scalar
        // Append to output
        // Advance i
    }
    return output;
}
```

Such functions are fundamental building blocks for robust Unicode processing pipelines.

2.5.7 APIs and overload resolution

Because the character type is part of the string type, overload resolution differentiates functions that take different Unicode encodings:

```
void print(std::u8string_view s);
void print(std::u16string_view s16);
void print(std::u32string_view s32);

int main() {
    print(u8"UTF-8 text");
    print(u"UTF-16 text");
    print(U"UTF-32 text");
}
```

This clarity allows APIs to be explicit about the encoding they expect and process.

2.5.8 Size, indexing, and iteration semantics

For all these string types:

- `size()` returns the number of code units, not the number of code points or grapheme clusters.
- Indexing with `operator[]` yields one code unit.
- Iterators traverse code units sequentially; they do not provide semantic character iteration.

To iterate over Unicode scalar values or grapheme clusters, explicit decoding and segmentation logic is required.

2.5.9 Common pitfalls

- Using `size()` to measure user-perceived character count leads to incorrect results for multi-code-unit sequences.
- Assuming that indexing yields a logical character is only true for UTF-32 at the code point level, and even then grapheme clusters may combine multiple code points.
- Treating literal strings without prefixes as UTF-8 may lead to encoding mismatches depending on source encoding and toolchain settings.

2.5.10 Best practices

- Use `std::u8string` for UTF-8 text interchange and storage.
- Use `std::u16string` when interacting with APIs that specifically require UTF-16.
- Use `std::u32string` for internal code point manipulation or Unicode algorithm implementations.
- Always document encoding expectations at API boundaries.

- Validate and normalize text when converting between encodings or comparing across systems.

2.5.11 Summary

`std::u8string`, `std::u16string`, and `std::u32string` provide explicit, self-documenting types for UTF-8, UTF-16, and UTF-32 encoded text in C++20 and beyond. They clarify the intended encoding at the type level and improve API expressiveness. However, developers must still use explicit decoding, segmentation, and validation to work with semantic text units such as code points and grapheme clusters. Understanding the semantics and limitations of these string types is essential for correct, portable Unicode text handling in modern C++.

2.6 Memory, Performance, and Practical Trade-offs

2.6.1 Introduction

Choosing text representation in C++ has real consequences on memory utilization, runtime performance, API semantics, and interoperability. UTF-8, UTF-16, and UTF-32-based storage all have trade-offs. This section analyzes these trade-offs in detail to guide practical design decisions from C++20 through C++26.

2.6.2 Memory footprint: encoding form and code unit size

The choice of encoding directly affects memory usage:

- **UTF-8**: uses 1 to 4 bytes per Unicode code point. ASCII code points occupy one byte. Non-ASCII code points occupy more.
- **UTF-16**: uses one 16-bit code unit for BMP code points and two 16-bit code units (surrogate pairs) for supplementary planes.
- **UTF-32**: uses one 32-bit unit per Unicode scalar value, uniform width, but higher memory cost.

Practical impact Applications with predominantly ASCII or Western European text benefit from UTF-8's compact storage. Systems with frequent Asian characters may see more UTF-8 variable-length overhead, but still often less than UTF-32. UTF-32 ensures constant access cost to code points but multiplies memory use relative to UTF-8 and UTF-16 for typical text.

2.6.3 Access cost: indexing, iteration, and decoding

The type of encoding also impacts how efficiently text can be accessed:

- **UTF-8 indexing**: direct indexing yields a code unit. Determining the complete code point at a given offset requires scanning backward to the previous valid leading byte or decoding forward, adding overhead.
- **UTF-16 iteration**: iteration is straightforward for BMP code points, but surrogate pairs require detection and combination for correct code point decoding.
- **UTF-32 access**: each code unit represents exactly one Unicode scalar value, enabling O(1) access to code points by index if sequence of units is random access.

Iteration cost comparison

- UTF-8: iteration must decode variable-length sequences (1–4 code units).
- UTF-16: iteration must check for surrogate pairs.
- UTF-32: iteration is trivial at the code point level.

Thus, UTF-32 offers simplicity at the cost of memory; UTF-8 optimizes storage at the cost of more complex decoding.

2.6.4 Cache behavior and locality

Modern CPUs rely on memory locality for performance. UTF-8's compact layout often improves instruction and data cache usage, especially for predominantly ASCII text. In contrast, UTF-32's larger memory footprint means less text fits in the same cache lines. UTF-16 sits between these two extremes. For high throughput text processing (searching, scanning, transformation), cache friendliness can outweigh raw decoding cost.

2.6.5 Conversion overhead

Real-world systems often need to convert between encodings:

- UTF-8 ↔ UTF-16 for API boundaries (Windows vs cross-platform text).
- UTF-8 ↔ UTF-32 for internal algorithm work.

Conversion involves decoding sequences into code points and re-encoding to a target form. This cost is proportional to the number of code points processed and may include validation, boundary checking, and error handling.

```
std::u32string utf8_to_utf32(std::u8string_view input) {
    std::u32string output;
    output.reserve(input.size()); // rough upper bound
    for (std::size_t i = 0; i < input.size(); ) {
        unsigned char lead = static_cast<unsigned char>(input[i]);
        // decode sequence and append Unicode scalar
        // advance i
    }
    return output;
}
```

Correct conversion must check for malformed sequences, apply replacement policies, and handle edge cases such as overlong encodings.

2.6.6 String operations: comparison, search, and modification

Comparison

Lexicographic comparison of sequences of code units (as provided by `operator<` on `std::u8string`, `std::u16string`, and `std::u32string`) is efficient but not Unicode collation. Unicode collation involves language-specific rules, normalization, and possibly case folding. For correct human-facing comparisons, conversion to a canonical form and application of collation rules is necessary. This is orthogonal to storage type selection.

Search and find

Finding a substring or pattern in UTF-8 or UTF-16 encoded text must respect code point boundaries. A naive byte or code unit search can match inside a multi-unit sequence and produce incorrect results.

```
auto pos = utf8_str.find(u8"é"); // careful: u8"é" is multiple code units
```

Search must either operate on validated code unit boundaries or on decoded code points.

Modification

Inserting or erasing text segments in UTF-8 and UTF-16 requires consideration of code unit boundaries. Splitting a code point or surrogate pair leads to invalid sequences. Libraries or utility functions typically provide safe splice operations. In contrast, UTF-32's fixed width simplifies these operations but at a higher data cost.

2.6.7 Error handling and validation

Incorrectly encoded sequences are common in input data. Accepting malformed UTF-8 or UTF-16 without validation can lead to undefined behavior in algorithms that assume valid input. Validation introduces additional runtime cost but is essential for security and correctness.

- **Strict policy**: reject invalid sequences at input boundaries.
- **Replacement policy**: substitute invalid runs with a canonical replacement.
- **Lenient policy**: process as raw bytes with caution.

Balancing performance and robustness depends on application requirements. Security-critical systems favor strict validation despite cost.

2.6.8 Grapheme cluster iteration cost

User-perceived characters (grapheme clusters) do not align with code unit or code point boundaries. Grapheme cluster iteration requires recognizing combining marks, Zero Width Joiner sequences, and emoji sequences with modifiers. This iteration is algorithmically more complex and computationally heavier than simple code point traversal.

2.6.9 Algorithm complexity comparisons

- UTF-8 decoding: O(N) with variable-length interpretation per code unit.
- UTF-16 decoding with surrogates: O(N) with conditional checks.
- UTF-32 access: O(N) with one code unit per code point.
- Grapheme segmentation: O(N) with state machine for boundary detection.

Trade-offs between simplicity and accurate user semantics should guide algorithm selection.

2.6.10 Interoperability with APIs and runtimes

Some platforms use UTF-16 natively (e.g., Windows), which can make UTF-16 storage locally efficient for API calls without conversion. On POSIX and web ecosystems, UTF-8 is the de facto standard. Interoperability cost must factor into storage decisions. For example, frequent UTF-8 UTF-16 conversions reduce the benefit of storing text in UTF-8 for long-lived internal use.

2.6.11 Case folding and normalization performance

Normalization and case folding are expensive compared to simple code unit operations. Applying normalization at input boundaries or before comparisons ensures correctness but adds runtime cost. Strategies to reduce impact include:

- caching normalized representations,
- memoizing frequent results,
- deferring normalization until necessary.

2.6.12 Memory vs access speed trade-offs

UTF-32 offers maximal simplicity for direct code point access and indexing, but its larger memory footprint reduces cache utilization and increases allocation costs. UTF-8 balances memory but increases decode complexity. UTF-16 balances between these extremes but introduces surrogate handling.

2.6.13 Practical guidelines for C++ text handling

- Use UTF-8 (`std::u8string`) for external interfaces, storage, and interchange to minimize memory and maximize compatibility.
- Use UTF-16 (`std::u16string`) when interoperating with UTF-16 native APIs to avoid repeated conversion overhead.
- Use UTF-32 (`std::u32string`) for internal logic requiring direct code point manipulation or fixed-width iteration.
- Perform validation and normalization at API boundaries to contain performance cost.
- Avoid naive indexing and slicing on UTF-8 or UTF-16 without boundary checks.

2.6.14 Case study: performance impact in text processing

Consider a large text corpus consisting mainly of ASCII and occasional non-ASCII symbols. Using UTF-8 minimizes memory footprint and increases throughput due to cache friendliness.

However, if the application frequently indexes code points or maps UTF-8 to fixed positions, the decoding overhead may outweigh memory gains. In that case, a UTF-32 representation for internal use and UTF-8 for I/O may strike an optimal balance.

2.6.15 Summary

Memory usage, runtime characteristics, and practical design trade-offs are central to selecting text representation in C++. UTF-8 is compact and cache-friendly but requires decoding. UTF-16 eases API integration on certain platforms at moderate cost. UTF-32 simplifies direct access at a higher memory cost. Understanding these trade-offs enables engineers to design systems that balance performance, memory, correctness, and compatibility.

Chapter 3

UTF-8 as the Practical Standard in Modern C++

3.1 Why UTF-8 Became the Dominant Encoding

3.1.1 The core story in one sentence

UTF-8 became dominant because it solved the ”global text” problem while preserving the operational advantages of byte-oriented systems: it is ASCII-compatible, space-efficient for common text, self-synchronizing, endian-neutral, and fits naturally into existing protocols, filesystems, and programming-language infrastructures built around bytes.

3.1.2 The historical constraint: the world was already byte-oriented

Long before Unicode, the practical computing world standardized on:

- **8-bit bytes** as the unit of storage, transport, and I/O,
- **C strings** as null-terminated byte arrays,

- **protocols and file formats** designed around byte sequences,
- **ASCII** as the baseline for control characters, punctuation, and the English alphabet.

A ”universal character set” could only win if it integrated into this ecosystem without forcing a rewrite of every OS interface, every network protocol, every database, every text-processing tool, and every programming language runtime. UTF-8 is the Unicode encoding form that aligns most closely with that pre-existing reality.

3.1.3 ASCII compatibility: the decisive property

The invariant that changed everything

UTF-8 preserves ASCII exactly:

- Every ASCII byte `0x00..0x7F` encodes the same Unicode code point `U+0000..U+007F`.
- Any valid ASCII text is valid UTF-8 text without modification.

This single property allowed the world to migrate to Unicode incrementally:

- Existing ASCII-only systems continued working unchanged.
- Systems could become Unicode-capable simply by treating their bytes as UTF-8 and validating when needed.
- Developers could keep using byte-oriented APIs for many tasks (protocol framing, parsing ASCII keywords, etc.) while still supporting multilingual content.

Why UTF-16/UTF-32 could not replicate this advantage

UTF-16 and UTF-32 are not byte-compatible with ASCII:

- They require a fixed code-unit width (16/32 bits).

- They introduce endianness concerns for serialized data.
- They do not preserve the exact byte-level representation of ASCII in existing byte-based formats.

UTF-8 is the encoding that let Unicode ride on top of the existing byte-based world rather than replacing it.

3.1.4 Space efficiency: most real-world text is ASCII-heavy

In practice, many critical text streams contain large ASCII subsets:

- programming languages and source code,
- markup and data formats (JSON keys, XML tags),
- protocols (HTTP headers, SMTP commands),
- configuration files, logs, identifiers,
- mixed-language content with ASCII punctuation, digits, and common symbols.

UTF-8 stores ASCII in *one byte*. For such data, UTF-8 is typically more compact than UTF-16 and far more compact than UTF-32.

Trade-off truth UTF-8 is variable-length: non-ASCII code points cost 2–4 bytes. This is not a bug; it is the explicit trade-off that made global adoption possible:

- compact for the dominant subset (ASCII),
- still able to encode the full Unicode range,
- efficient on the wire and on disk.

3.1.5 Self-synchronization: robust scanning and recovery

UTF-8 is **self-synchronizing**. A decoder can find code point boundaries by examining byte patterns:

- Leading bytes have distinctive bit prefixes (`0xxxxxxx`, `110xxxxx`, `1110xxxx`, `11110xxx`).
- Continuation bytes are always `10xxxxxx`.

This has practical consequences:

- You can scan forward efficiently.
- You can often recover from errors by resynchronizing at the next plausible boundary.
- Many streaming use cases become simpler than with stateful legacy encodings.

```
#include <cstddef>
#include <cstdint>
#include <string_view>

// A minimal helper: "is this a UTF-8 continuation byte?"
constexpr bool is_utf8_cont(unsigned char b) {
    return (b & 0xC0u) == 0x80u; // 10xxxxxx
}

// Find a safe code point boundary at or before index i (best-effort).
// This is a boundary-finding aid, not full validation.
std::size_t utf8_rewind_to_boundary(std::string_view s, std::size_t i) {
    if (i > s.size()) i = s.size();
    while (i > 0) {
        unsigned char b = static_cast<unsigned char>(s[i]);
        if (!is_utf8_cont(b)) return i;
        --i;
```

```
    }
    return 0;
}
```

The point is not that UTF-8 eliminates the need for validation (it does not), but that its structure supports efficient boundary-aware processing in a byte stream.

3.1.6 Endianness neutrality and BOM avoidance

UTF-8 is inherently endian-neutral because it is defined in bytes. This matters for:

- network protocols (which are byte streams),
- files exchanged between different architectures,
- memory-mapped text data,
- serialization and stable binary formats.

UTF-16 and UTF-32 require an endianness decision when serialized and often rely on a BOM (byte order mark) or external metadata. BOM handling is operationally annoying:

- it complicates concatenation and streaming,
- it can pollute text if misinterpreted,
- it creates edge cases at file boundaries.

UTF-8 avoids this entire category of problems by design.

3.1.7 Fits the internet and modern data formats

UTF-8 aligned perfectly with the needs of the modern internet:

- Byte-oriented transport layers and framing.
- Huge volumes of mixed text and ASCII tokens.
- Backwards compatibility with existing ASCII infrastructure.

Even when a format is ”Unicode”, the operational choice for interchange is almost always UTF-8 because it is the most frictionless representation for networked systems.

3.1.8 Operational compatibility: POSIX-like systems and the ”bytes first” model

Many Unix/POSIX environments historically treat strings as sequences of bytes rather than as typed Unicode text. This is simultaneously a limitation and an advantage:

- **Limitation**: the OS will not enforce Unicode correctness.
- **Advantage**: UTF-8 can be adopted as a convention without changing the OS ABI.

UTF-8 works well in such environments because:

- it is stable in a byte stream,
- it is compatible with traditional tools that are ASCII-centric,
- it allows incremental adoption: validate and decode only where semantics require it.

3.1.9 C and C++ legacy: null-terminated strings and byte-based APIs

C's foundational string model is `char*` with `'\0'` termination, and C++ inherited a vast ecosystem built on that model. UTF-8 integrates with this naturally:

- ASCII `'\0'` remains a terminator.
- Many ASCII delimiters remain single-byte (quotes, braces, commas, colons, whitespace).
- Existing libraries that treat `char` as bytes remain valid at the transport/storage layer.

This does *not* mean UTF-8 makes text processing "easy"; it means UTF-8 makes Unicode-compatible storage and transport feasible within byte-based language ecosystems.

3.1.10 The C++20–C++26 angle: making UTF-8 intent explicit

`char8_t` as an intent type

C++20 introduced `char8_t` and `std::u8string` to represent UTF-8 code units explicitly:

- `u8"..."` yields UTF-8 code units with element type `char8_t`.
- `std::u8string` is `std::basic_string<char8_t>`.

This reinforces the modern reality: UTF-8 is a first-class practical encoding, and code should be able to express "this is UTF-8 text" rather than "this is some bytes that I hope are UTF-8".

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

void accept_utf8(std::u8string_view text);   // contract: UTF-8 code units
void accept_bytes(std::string_view bytes);   // contract: arbitrary bytes
```

```
int main() {
    accept_utf8(u8"UTF-8 text");
    accept_bytes("raw bytes");
}
```

Design benefit Encoding becomes part of the type-level API contract, reducing accidental misuse and enabling overload resolution and template constraints to distinguish bytes from UTF-8 text.

3.1.11 What UTF-8 dominance does *not* imply

UTF-8 being dominant does not mean:

- **"UTF-8 makes indexing trivial"**: indexing returns bytes/code units, not code points or grapheme clusters.
- **"UTF-8 guarantees valid Unicode"**: input still must be validated; ill-formed sequences exist.
- **"UTF-8 solves user-perceived character handling"**: grapheme clusters, normalization, bidi, and shaping remain separate layers.
- **"UTF-8 is always best internally"**: some internal algorithms may prefer UTF-32 for code-point-centric processing.

UTF-8 is dominant primarily as an *interchange* and *boundary* encoding. Many robust systems use UTF-8 at boundaries and choose internal representations strategically.

3.1.12 A practical engineering model: UTF-8 at boundaries, explicit semantics inside

A production-grade model for modern C++ is:

1. **External boundary encoding**: UTF-8 for files, network, logs, configuration, and most storage.

2. **Validation policy**: reject or replace ill-formed sequences at boundaries.

3. **Internal processing unit**: choose based on algorithms:

    - keep UTF-8 for scanning/searching ASCII tokens and for memory locality,
    - decode to Unicode scalar values (UTF-32-like) for code-point algorithms,
    - segment to grapheme clusters for UI editing semantics.

4. **Conversion boundaries**: explicit, tested transcoding layers when needed (e.g., UTF-8 $\leftrightarrow$ UTF-16 APIs).

3.1.13 Summary

UTF-8 won because it was the only Unicode encoding form that fit the existing world without forcing a revolution:

- exact ASCII preservation enabled incremental migration,
- compactness matched real-world text distributions,
- self-synchronizing structure fit streaming and robust parsing,
- endian neutrality avoided BOM/endian complexity,
- byte-oriented compatibility aligned with the internet, POSIX, and C/C++ ecosystems,
- modern C++ reinforced this reality by adding `char8_t` and `std::u8string` for explicit UTF-8 intent.

Understanding *why* UTF-8 dominates is the foundation for understanding *how* to use it correctly: treat it as a code-unit encoding at boundaries, validate and decode explicitly for semantics, and never confuse bytes with user-perceived characters.

3.2 Variable-Length Encoding and Its Consequences

3.2.1 Introduction

UTF-8 is a variable-length encoding: a single Unicode scalar value (code point) is encoded using one to four 8-bit code units (bytes). This design is central to UTF-8's dominance, but it introduces consequences that C++ developers must internalize. Many bugs in "Unicode support" are not Unicode bugs at all; they are failures to respect the difference between *code units*, *code points*, and *grapheme clusters* when working with a variable-length encoding.
In modern C++ (C++20–C++26), the standard string types (`std::string`, `std::u8string`) remain sequences of code units. Therefore, the consequences of variable-length encoding are primarily consequences of *code-unit-based APIs* applied to *text with higher-level semantics*.

3.2.2 UTF-8 structure: why length varies

UTF-8 encodes scalar values using a prefix scheme:

- **1 byte** for `U+0000..U+007F` (ASCII): `0xxxxxxx`
- **2 bytes** for `U+0080..U+07FF`: `110xxxxx 10xxxxxx`
- **3 bytes** for `U+0800..U+FFFF` (excluding surrogates): `1110xxxx 10xxxxxx 10xxxxxx`
- **4 bytes** for `U+10000..U+10FFFF`: `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

Two properties matter operationally:

- **ASCII compatibility**: ASCII bytes are unchanged, enabling incremental migration.
- **Self-synchronization**: continuation bytes have a distinct `10xxxxxx` form.

3.2.3 Consequence 1: `size()` counts bytes, not characters

For UTF-8 strings, `size()` is the number of bytes (code units). It is not:

- the number of Unicode code points,
- the number of grapheme clusters (user-perceived characters),
- the display width in terminal columns,
- the rendered width in pixels.

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

int main() {
    // If the source file is UTF-8, this contains:
    // 'A' (1 byte), '€' (3 bytes), ' ' (3 bytes), '  ' (4 bytes)
    std::string s = "A€   ";
    std::cout << "bytes = " << s.size() << "\n";

    // Any attempt to interpret s.size() as "character count" is incorrect.
}
```

Engineering rule Never name a variable `charCount` if it is derived from `std::string::size()` on UTF-8 text. Use `byteCount`.

3.2.4 Consequence 2: indexing and slicing can break encoding validity

Indexing yields a byte, not a character

`s[i]` returns the *i-th byte*. If `i` points into the middle of a multi-byte UTF-8 sequence, `s[i]` is a continuation byte. Treating it as a character is meaningless.

Naive slicing can produce ill-formed UTF-8

`substr` operates on byte indices. Slicing at arbitrary byte offsets can split a multi-byte sequence, producing invalid UTF-8.

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

static bool is_cont(unsigned char b) { return (b & 0xC0u) == 0x80u; }

int main() {
    std::string s = "€"; // UTF-8: 3 bytes
    std::cout << "bytes=" << s.size() << "\n";

    std::string t = s.substr(0, 2); // splits the 3-byte sequence -> invalid UTF-8
    std::cout << "t bytes=" << t.size() << "\n";

    if (!t.empty() && is_cont(static_cast<unsigned char>(t.back()))) {
        std::cout << "t ends with a continuation byte -> likely invalid UTF-8 slice\n";
    }
}
```

Rule Byte slicing is valid only when the unit of meaning is bytes. For semantic text slices, compute code point boundaries (or grapheme boundaries) first.

3.2.5 Consequence 3: random access by ”character index” is not O(1)

In fixed-width encodings (like UTF-32), selecting the *k-th code point* is O(1) by index because each code point is one code unit. In UTF-8, selecting the k-th code point is generally O(n) because you must decode from the beginning (or from a cached index) to find boundaries.

Practical impact

- Repeated indexing in a loop (e.g., ”for i in 0..N, take i-th character”) is potentially quadratic if implemented naively.
- Efficient solutions either:
  - iterate once while decoding, or
  - build an index of code point boundaries, or
  - convert to a code-point representation for that processing stage.

3.2.6 Consequence 4: common algorithms become encoding-sensitive

Many classic string algorithms assume:

- each element is a logical character,
- reversing elements reverses characters,
- splitting by delimiters splits characters cleanly,
- trimming removes characters, not bytes.

In UTF-8, these assumptions fail unless your domain is ASCII-only.

Example: reversing a UTF-8 byte sequence is wrong

Reversing bytes reverses the encoding, not the text. The resulting byte sequence is almost always invalid UTF-8.

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

int main() {
```

```
    std::string s = "A€B";
    std::string r = s;
    std::reverse(r.begin(), r.end()); // byte-level reverse

    std::cout << "original bytes=" << s.size() << "\n";
    std::cout << "reversed bytes=" << r.size() << "\n";
    // r is not a valid UTF-8 reversal of the text.
}
```

Correct semantic reversal A correct reversal is defined over a chosen unit:

- reverse by **code points**: decode to scalars, reverse scalars, re-encode,
- reverse by **grapheme clusters**: segment into clusters, reverse clusters, reassemble.

3.2.7 Consequence 5: ”character classification” on bytes is incorrect

Functions like `std::isalpha` and `std::tolower` operate on single-byte character values in the current locale (with strict preconditions). Applying them to UTF-8 bytes is not Unicode classification. For non-ASCII bytes, you are classifying random fragments of an encoding sequence.

Two different problems

- **ASCII classification**: safe only after validating that the byte is ASCII.
- **Unicode classification**: requires decoding to code points and applying Unicode property tables (outside the standard library).

3.2.8 Consequence 6: validation becomes a first-class boundary concern

Variable-length encodings permit malformed sequences:

- invalid leading bytes,
- missing continuation bytes,
- overlong encodings,
- surrogate code points encoded in UTF-8 (invalid as scalar values),
- values above `U+10FFFF`.

A robust C++ system defines a boundary policy:

- validate and reject ill-formed UTF-8 at ingestion, or
- decode with replacement to keep pipelines flowing, or
- treat as bytes if the domain is not text.

3.2.9 Consequence 7: counting user-visible characters is harder than decoding

Even decoding UTF-8 to code points does not solve user-visible semantics. A user-visible ”character” is often an extended grapheme cluster that can include multiple code points (combining marks, ZWJ emoji sequences, regional indicator pairs, etc.). Therefore:

- byte count $\neq$ code point count,
- code point count $\neq$ grapheme cluster count,
- grapheme cluster count $\neq$ display width.

Implication for UI and text editing Cursor movement and backspace must follow grapheme cluster boundaries, not code point boundaries.

3.2.10 Consequence 8: performance trade-offs are nuanced

Variable-length decoding has a cost, but the full performance picture is subtle:

- UTF-8 can be faster end-to-end because it is **more compact**, improving cache locality and reducing I/O volume.
- Many real workloads are dominated by ASCII and delimiters, where UTF-8 processing is effectively byte processing.
- Workloads requiring heavy random access or repeated ”character indexing” may benefit from building indices or using UTF-32 internally.

3.2.11 Practical C++ patterns that respect UTF-8 variability

Pattern 1: treat UTF-8 strings as byte sequences until semantics are required

For tasks such as protocol framing, tokenizing ASCII keywords, scanning for ASCII punctuation, and splitting on known ASCII separators, it is valid and fast to operate at the byte level *provided you do not break UTF-8 sequences*. This typically means:

- only match ASCII bytes `< 0x80`,
- avoid slicing at arbitrary offsets; slice at known delimiter boundaries that are ASCII.

Pattern 2: decode once, process in one pass

When you need code point semantics (e.g., identifier rules, Unicode category checks), decode once and stream over code points rather than repeatedly indexing.

Pattern 3: build boundary indices for repeated random access

If you must support operations like ”jump to the k-th character” repeatedly, build an index of byte offsets to code point boundaries (or grapheme boundaries) once and reuse it.

3.2.12 Summary

UTF-8's variable-length design is the source of both its success and its complexity. The key consequences for modern C++ are:

- `size()` counts bytes, not characters.
- Indexing and slicing are byte-based and can break encoding validity.
- Random access by "character index" is not constant time.
- Many classic string algorithms must be redefined over code points or grapheme clusters.
- Classification and case conversion cannot be performed correctly on raw UTF-8 bytes.
- Validation and boundary policies are essential for correctness and security.

A C++20–C++26 codebase that treats UTF-8 as a byte encoding at storage boundaries, and explicitly decodes/segments when semantics are required, will be both correct and performant.

3.3 Iteration, Indexing, and Length Semantics in UTF-8

3.3.1 Introduction

UTF-8 is a variable-length encoding. Each Unicode scalar value can occupy one to four 8-bit code units. In C++, standard string types such as `std::string` and `std::u8string` represent sequences of these code units. Understanding the semantics of iteration, indexing, and length in this context is essential for correct and robust text processing.

This section explains the differences between code-unit iteration, code-point iteration, and grapheme-cluster iteration, and clarifies what `size()`, `operator[]`, and iterator operations mean for UTF-8.

3.3.2 Code units, code points, and grapheme clusters

In Unicode text processing, three distinct concepts are important:

- **Code units**: the smallest addressable units of a particular encoding. For UTF-8, code units are 8-bit unsigned values (`char` or `char8_t`).

- **Code points**: the abstract Unicode scalar values (`U+0000` to `U+10FFFF`) that represent characters in the Unicode repertoire.

- **Grapheme clusters**: user-perceived characters that may consist of one or more code points, including combining marks, sequences connected by zero-width joiners, and other multi-code-point constructs.

UTF-8 encodes each code point as a sequence of one to four code units. A grapheme cluster can consist of multiple code points, and therefore a variable number of code units.

3.3.3 Iteration over UTF-8

Code-unit iteration

Standard iteration over a `std::string` or `std::u8string` uses `begin()` and `end()`, which traverse code units:

```
std::u8string s = u8"Hi €";
for (char8_t cu : s) {
    // cu is one code unit
}
```

This iteration visits each byte in the UTF-8 sequence. It is efficient and appropriate for operations such as:

- raw byte I/O,
- scanning for ASCII delimiters,
- encoding-aware boundary detection,
- validating UTF-8 code-unit sequences.

However, code-unit iteration does *not* iterate over Unicode code points or user-perceived characters.

Code-point iteration

To iterate over Unicode scalar values, a UTF-8 decoder is required. Decoding involves reading one to four code units and assembling a single code point:

```
#include <cstdint>
#include <string>
#include <vector>
```

```
std::vector<char32_t> utf8_to_codepoints(std::u8string_view s) {
    std::vector<char32_t> cps;
    std::size_t i = 0;
    while (i < s.size()) {
        unsigned char lead = static_cast<unsigned char>(s[i]);
        // determine sequence length and decode
        // append decoded code point to cps
        // adjust i
    }
    return cps;
}
```

Code-point iteration is appropriate for:

- Unicode property checks (letter, digit, category),
- case mapping and normalization in code-point space,
- protocols or formats that define text at the code-point level.

Grapheme-cluster iteration

User-perceived characters can span multiple code points. For example:

- a base letter plus one or more combining marks,
- emoji sequences joined by zero-width joiner (ZWJ),
- regional indicator sequences for flags.

Correct grapheme-cluster iteration requires implementing or using a segmentation algorithm compliant with the Unicode Text Segmentation rules.
Note: The C++ Standard Library does not provide built-in grapheme segmentation; it must be added explicitly through libraries or custom logic.

3.3.4 Length semantics in UTF-8

`size()` and code units

For a UTF-8 string type, `size()` returns the number of code units:

```
std::u8string s = u8"Ø"; // 'Ø' occupies two code units
std::cout << s.size();   // prints 2
```

Code-unit length is useful for low-level memory management, storage allocation, and code-unit-based operations.
Important distinction: `size()` does not return the number of Unicode code points or grapheme clusters.

Code points vs grapheme clusters

To compute the number of code points, the UTF-8 sequence must be decoded. This produces a vector of Unicode scalar values, and its length gives the code-point count.
To compute the number of grapheme clusters, segmentation must be performed as defined by the Unicode Standard. The number of grapheme clusters corresponds to user-perceived character count, which is important for:

- cursor navigation in text editors,
- user-visible string metrics,
- enforcing length limits in user interfaces.

3.3.5 Indexing semantics

Indexing by code units

`operator[]` on `std::string` or `std::u8string` provides direct access to the *i-th code unit*:

```
std::u8string s = u8"abc";
char8_t cu = s[1]; // second code unit, not second code point
```

This access is fast but does not correspond to the second Unicode character. In a string containing multi-code-unit code points, `s[1]` may be a continuation byte.

Indexing by logical character positions

To index by code point or grapheme cluster positions, an indexing layer must be built:

```
std::u8string_view text = u8"école";
std::vector<std::size_t> codepoint_offsets;
std::size_t i = 0;
while (i < text.size()) {
    codepoint_offsets.push_back(i);
    // decode next UTF-8 code point to find next boundary
}
// codepoint_offsets[k] gives byte index of k-th code point
```

Such an index allows O(1) access to the k-th code point after an initial O(n) preprocessing. Similar indexing can be constructed for grapheme clusters.

3.3.6 Practical pitfalls and bugs

Using `size()` to report user-visible length

Reports such as ”This string has 5 characters” are incorrect if implemented as:

```
std::cout << s.size(); // counts bytes
```

Instead, clarify:

- **byteCount** for storage,
- **codePointCount** for Unicode scalar iteration,
- **graphemeClusterCount** for user-perceived characters.

Naive random access

A loop such as:

```
for (std::size_t i = 0; i < s.size(); ++i) {
    process(s[i]); // processes code units, not characters
}
```

is often incorrect for non-ASCII text. Instead, iterate over code points or grapheme clusters depending on semantic requirements.

3.3.7 Efficient iteration patterns

Single-pass processing

When semantic iteration is required, decode once and process in one pass. Avoid using `operator[]` inside loops to find code points repeatedly, as this leads to repeated decoding and suboptimal performance.

```
auto cps = utf8_to_codepoints(s); // decode once
for (char32_t cp : cps) {
    // process logical Unicode values
}
```

On-demand iteration with iterators

An iterator abstraction over code points or grapheme clusters encapsulates decoding and boundary semantics. A modern C++ design often wraps byte iterators with decoding logic to provide higher-level iteration without exposing raw code units.

3.3.8 APIs and interfaces

Design APIs to explicitly state the intended iteration semantic:

```cpp
void process_bytes(std::u8string_view bytes);     // code units
void process_codepoints(std::u32string_view cps); // Unicode scalar values
void process_graphemes(std::vector<std::u32string_view> gcs); // clusters
```

Such clarity prevents accidental misuse and communicates expectations to API consumers.

3.3.9 Summary

- In UTF-8, `size()`, `operator[]`, and default iterators operate on code units.
- Code-point iteration requires explicit decoding of variable-length sequences.
- Grapheme-cluster iteration requires segmentation beyond decoding.
- Indexing semantic characters requires building boundary indices.
- Efficient text processing separates storage-level operations from semantic-level iteration.

Understanding these differences is critical for writing correct, maintainable, and efficient Unicode text handling code in modern C++.

3.4 Searching, Slicing, and Substring Pitfalls

3.4.1 Introduction

UTF-8 is a byte-based, variable-length encoding. In C++20–C++26, both `std::string` and `std::u8string` are containers of code units (bytes or `char8_t` units) and their operations (`find`, `substr`, iterators, `erase`, `insert`) are defined in terms of *code units*. This is correct at the storage level, but it becomes a trap when developers implicitly assume *character semantics*. This section explains why searching and slicing UTF-8 text is subtle, how common bugs happen, and how to design safer APIs and algorithms.

3.4.2 The three different ”units” you might mean

When you say ”search” or ”substring”, you must decide what unit you mean:

- **Code units (bytes)**: UTF-8 encoding units; what C++ strings store.
- **Code points (Unicode scalar values)**: decoded Unicode values.
- **Grapheme clusters (user-perceived characters)**: what users perceive as one character.

C++ standard string algorithms are code-unit algorithms. If your task is code point or grapheme based, you must decode/segment explicitly.

3.4.3 Searching pitfalls

Pitfall 1: treating `find()` as ”find character”

`find()` searches for a sequence of code units. For ASCII text, ”one byte = one character” often makes this *look* like character search. For non-ASCII text, a single code point is multiple bytes, and a grapheme cluster may be multiple code points. So:

- searching for a code point requires searching for its UTF-8 byte sequence,
- searching for a user-visible character requires grapheme-level logic (often beyond simple matching),
- searching case-insensitively requires Unicode-aware case folding (not provided by `find`).

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

int main() {
    std::string s = "A€B";   // '€' is 3 UTF-8 bytes
    std::string needle = "€"; // also 3 bytes if source is UTF-8

    auto pos = s.find(needle);
    std::cout << "pos=" << pos << "\n";      // pos is a byte index
    std::cout << "bytes=" << s.size() << "\n"; // byte length, not character length
}
```

Key point Even when `find` succeeds, the returned position is a **byte offset**. If you interpret it as a ”character index”, your downstream slicing and indexing will likely be wrong.

Pitfall 2: false matches when scanning bytes naively

Many developers scan bytes and compare to ASCII constants (e.g., `'A'`) to find characters. This is safe *only* if you are explicitly working in ASCII. In UTF-8, multi-byte sequences contain bytes that can coincidentally match ASCII values, especially if you compare signed `char` without casting, or treat bytes as locale-dependent characters.

Safe ASCII-only scanning If your delimiter is ASCII (e.g., `','`, `'/'`), scanning bytes is valid because ASCII bytes are preserved in UTF-8. But you must not treat non-ASCII bytes as standalone characters.

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

std::vector<std::string_view> split_on_ascii_comma(std::string_view s) {
    std::vector<std::string_view> out;
    std::size_t start = 0;
    for (std::size_t i = 0; i < s.size(); ++i) {
        unsigned char b = static_cast<unsigned char>(s[i]);
        if (b == static_cast<unsigned char>(',')) {
            out.emplace_back(s.data() + start, i - start);
            start = i + 1;
        }
    }
    out.emplace_back(s.data() + start, s.size() - start);
    return out;
}
```

Pitfall 3: searching without normalization

Unicode permits multiple canonically equivalent sequences for the same text. A search that compares code units will not match equivalent-but-differently-normalized strings. This causes surprising failures in:

- user input matching stored text,
- file name matching across systems,
- identifier matching in multilingual contexts.

Implication If your domain requires canonical equivalence, normalize both haystack and needle to a chosen normalization form at defined boundaries before searching.

Pitfall 4: case-insensitive search done with ASCII rules

Lowercasing bytes and then searching is incorrect for UTF-8. Proper Unicode case-insensitive matching generally requires:

- decoding to code points,
- Unicode case folding,
- often normalization (depending on policy),
- then performing a search in that transformed domain.

3.4.4 Slicing pitfalls

Pitfall 1: `substr()` can create ill-formed UTF-8

`substr(pos, count)` slices by byte offsets. If `pos` is not a code point boundary, or `pos+count` lands in the middle of a multi-byte sequence, the result is ill-formed UTF-8.

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

static bool is_cont(unsigned char b) { return (b & 0xC0u) == 0x80u; }

int main() {
    std::string s = "€";        // 3 bytes
    std::string t = s.substr(1); // starts at a continuation byte -> invalid UTF-8

    unsigned char first = static_cast<unsigned char>(t[0]);
    std::cout << "t[0] is continuation? " << (is_cont(first) ? "yes" : "no") << "\n";
}
```

Rule Never slice UTF-8 by arbitrary byte indices unless you have proven they are code point boundaries (or you intentionally operate on raw bytes).

Pitfall 2: truncation by bytes corrupts text

Truncating a UTF-8 string to enforce a length limit (e.g., 20 characters) by taking the first 20 bytes is wrong. You may:

- cut a multi-byte code point,
- cut in the middle of a grapheme cluster (e.g., base + combining mark),
- cut in the middle of an emoji ZWJ sequence (changing meaning).

Policy decision If your limit is:

- **bytes**: truncate by bytes, but keep UTF-8 validity if required.
- **code points**: decode and count code points, then slice on boundary.
- **graphemes**: segment into grapheme clusters and count those.

Pitfall 3: deleting by byte index breaks invariants

`erase` and `insert` operate on code unit iterators or indices. Deleting a range that splits code points yields invalid UTF-8. This is especially common when:

- indices come from UI caret positions (which are grapheme-based),
- indices come from code point iteration but are applied as byte indices incorrectly,
- indices come from mixed encodings or different normalization forms.

3.4.5 Substring pitfalls and boundary correctness

”character position” is not ”byte offset”

Many systems store a cursor position as ”the 10th character.” In UTF-8 storage, that cursor must be mapped to a byte offset. Without a mapping layer:

- substring extraction is wrong,
- highlights are wrong,
- validation fails,
- edits corrupt UTF-8.

Correct architecture Maintain explicit boundary data when needed:

- an index from code point number to byte offset (for code point operations),
- an index from grapheme cluster number to byte offset (for UI operations).

3.4.6 Safe boundary detection primitives

Detecting UTF-8 continuation bytes

A minimal building block for boundary reasoning is recognizing continuation bytes:

```
constexpr bool is_utf8_continuation(unsigned char b) {
    return (b & 0xC0u) == 0x80u; // 10xxxxxx
}
```

A code point boundary in UTF-8 is any byte that is *not* a continuation byte.

Rewinding to a likely code point boundary

This is useful for safe truncation by bytes while preserving code point boundaries:

```cpp
#include <string_view>
#include <cstddef>

constexpr bool is_utf8_continuation(unsigned char b) {
    return (b & 0xC0u) == 0x80u;
}

// Best-effort: given a target byte length, rewind to the nearest boundary <= limit.
std::size_t utf8_safe_prefix_length(std::string_view s, std::size_t limit) {
    if (limit >= s.size()) return s.size();
    while (limit > 0 && is_utf8_continuation(static_cast<unsigned char>(s[limit]))) {
        --limit;
    }
    return limit;
}
```

Limitations This preserves code point boundaries but does not preserve grapheme clusters. It also does not validate the entire string. It is a boundary utility, not a complete UTF-8 correctness solution.

3.4.7 Practical guidelines for C++20–C++26

Guideline 1: be explicit about what ”substring” means

Define and document:

- substring by **byte offsets** (storage semantics),
- substring by **code point indices** (Unicode scalar semantics),

- substring by **grapheme indices** (user-visible semantics).

Guideline 2: keep UTF-8 valid at boundaries

If your system uses UTF-8 as the canonical external encoding:

- validate at ingestion,
- ensure any slicing/editing preserves UTF-8 validity,
- convert to code points for Unicode algorithms when needed.

Guideline 3: normalize when your domain requires canonical equivalence

If searches must behave as users expect across canonical variants:

- normalize the stored representation (or store normalized alongside original),
- normalize user input before searching,
- document the chosen normalization form and policy.

Guideline 4: do not use locale/ASCII transforms for Unicode search

Avoid byte-wise `tolower` and locale-dependent ctype logic for UTF-8. Use Unicode-aware transformations (case folding, normalization) at the code point level.

3.4.8 Summary

Searching and substring operations in UTF-8 become tricky because C++ string operations are code-unit-based while human text semantics are code-point- and grapheme-based. The most common pitfalls are:

- interpreting byte indices as character positions,

- slicing at arbitrary byte offsets and creating invalid UTF-8,
- implementing truncation limits in bytes while intending ”characters”,
- ignoring normalization and case-folding requirements for user-facing search.

Robust modern C++ designs treat UTF-8 as the boundary encoding, keep boundaries explicit, decode when semantic operations are required, and apply normalization and segmentation policies deliberately.

3.5 Validation and Error Handling in UTF-8 Text

3.5.1 Introduction

UTF-8 text is ubiquitous in modern systems. Unlike fixed-width encodings, UTF-8 allows a wide range of valid and invalid code-unit sequences. Because UTF-8 is variable-length and self-synchronizing, it is possible for malformed sequences to appear in real-world text due to:

- truncated or corrupted byte streams,
- incorrect slicing or splicing of byte sequences,
- legacy systems that intermix code pages,
- data stored without an explicit encoding contract.

Robust C++ software must validate UTF-8 input at defined boundaries and handle errors according to domain-specific policies. Failure to do so leads to subtle bugs, security vulnerabilities, data corruption, and user experience inconsistencies.
This section presents modern best practices, explicit validation logic, and error handling strategies suited for use in C++20–C++26 codebases.

3.5.2 What does UTF-8 validation mean?

A **well-formed** UTF-8 sequence satisfies the Unicode Standard's definition of valid UTF-8:

- No overlong encodings (each code point is encoded with the shortest legal sequence).
- No surrogate code points appear in UTF-8 (their use is explicitly disallowed).
- Continuation bytes follow a valid leading byte and match the continuation format (`10xxxxxx`).

- No code units exist that cannot be part of any valid UTF-8 sequence.
- No values beyond `U+10FFFF` are encoded.

Validation verifies that every byte and multi-byte sequence conforms to these rules.

3.5.3 Why validation is essential

Defensive correctness

Malformed sequences can cause:

- incorrect decoding (yielding invalid code points),
- out-of-bounds memory access in naive decoders,
- infinite loops or algorithm invariants failure,
- downstream logic misinterpretation.

Security robustness

Many security vulnerabilities originate from invalid input not being checked before use:

- bypassing filters,
- buffer overruns,
- integer overflows during decoding,
- denial-of-service due to pathological sequences.

Semantic guarantees

Applications that enforce semantics on text (such as identifiers, protocols, or user interfaces) must establish that their input conforms to a valid Unicode encoding.

3.5.4 When to validate

Validation should occur:

- at all external boundaries (file input, network protocols, API calls),
- before decoding bytes to higher-level representations (code points, grapheme clusters),
- when crossing trust domains (user input vs internal storage),
- when storing or forwarding text to other systems.

3.5.5 Decoding with validation

A robust UTF-8 decoder both decodes and validates. It does not assume that the input is already correct.

```
#include <cstddef>
#include <optional>
#include <string_view>
#include <vector>

// A UTF-8 decoder that validates each sequence.
struct Utf8DecodeResult {
    char32_t code_point;
    std::size_t length; // number of bytes consumed
};

static bool is_continuation_byte(unsigned char b) {
    return (b & 0xC0u) == 0x80u; // 10xxxxxx
}

static std::optional<Utf8DecodeResult> decode_utf8_at(std::string_view s, std::size_t
↪  index) {
```

```
    if (index >= s.size()) return std::nullopt;

    unsigned char lead = static_cast<unsigned char>(s[index]);
    if (lead <= 0x7Fu) {
        return Utf8DecodeResult{ static_cast<char32_t>(lead), 1 };
    }

    std::size_t expected_len = 0;
    char32_t cp = 0;

    if ((lead & 0xE0u) == 0xC0u) {
        expected_len = 2;
        cp = lead & 0x1Fu;
    } else if ((lead & 0xF0u) == 0xE0u) {
        expected_len = 3;
        cp = lead & 0x0Fu;
    } else if ((lead & 0xF8u) == 0xF0u) {
        expected_len = 4;
        cp = lead & 0x07u;
    } else {
        // Invalid leading byte
        return std::nullopt;
    }

    if (index + expected_len > s.size()) {
        // Truncated sequence
        return std::nullopt;
    }

    for (std::size_t i = 1; i < expected_len; ++i) {
        unsigned char c = static_cast<unsigned char>(s[index + i]);
        if (!is_continuation_byte(c)) {
            // Missing or invalid continuation byte
```

```
            return std::nullopt;
        }
        cp = (cp << 6) | (c & 0x3Fu);
    }

    // Reject overlong encodings
    if ((expected_len == 2 && cp <= 0x7Fu)
        || (expected_len == 3 && cp <= 0x7FFu)
        || (expected_len == 4 && cp <= 0xFFFFu)) {
        return std::nullopt;
    }

    // Reject surrogate code points
    if (cp >= 0xD800u && cp <= 0xDFFFu) {
        return std::nullopt;
    }

    // Reject out-of-range
    if (cp > 0x10FFFFu) {
        return std::nullopt;
    }

    return Utf8DecodeResult{ cp, expected_len };
}
```

3.5.6 Validation strategies

Strict validation

Reject sequences that deviate from a well-formed UTF-8 pattern at the earliest point of detection. This approach is valuable when:

- correctness is paramount,

- data integrity must be enforced,
- protocols require strict conformance.

Replacement policy

Convert invalid sequences to a special placeholder (in Unicode contexts, commonly U+FFFD). This allows processing to continue without rejecting the entire input. A replacement policy is useful in:

- user interface text rendering,
- logging untrusted input,
- applications tolerant of non-conforming data.

Lenient processing

Some systems operate on raw bytes and treat invalid sequences as literal data. This is rare in Unicode-aware contexts and should be explicitly documented to avoid semantic ambiguity.

3.5.7 Defining boundary policies

A boundary policy specifies what happens at each interface where text crosses from one context into another:

- Before decoding network buffers to code points,
- Before storing text in databases,
- When normalizing and comparing user input,
- When converting between encoding forms (UTF-8 $\leftrightarrow$ UTF-16).

A typical boundary policy includes:

- validation as early as possible,
- normalization to a chosen canonical form,
- error handling (reject, replace, preserve),
- logging or reporting parse issues.

3.5.8 Error reporting and propagation

Validation errors should not be silently ignored. A strategy for error reporting includes:

- returning an error code or status,
- throwing an exception in contexts that support it,
- returning an optional or expected type carrying diagnostic data,
- logging context and position of error for debugging.

3.5.9 Utilities for boundary-safe prefixes

To safely truncate UTF-8 text without breaking a code point boundary:

```
#include <string_view>
#include <cstddef>

static bool is_utf8_continuation(unsigned char b) {
    return (b & 0xC0u) == 0x80u;
}

std::size_t safe_utf8_prefix(std::string_view s, std::size_t max_bytes) {
```

```
    if (max_bytes >= s.size()) return s.size();
    while (max_bytes > 0 && is_utf8_continuation(
                static_cast<unsigned char>(s[max_bytes]))) {
        --max_bytes;
    }
    return max_bytes;
}
```

This utility ensures that the prefix returned ends on a probable code point boundary, though it does not validate the entire prefix.

3.5.10 Integration with segmentation and normalization

Validation is a prerequisite for:

- code-point decoding and iteration,
- normalization (canonically equivalent sequences),
- grapheme cluster segmentation,
- collation and case folding.

Failure to validate before these operations often leads to undefined behavior or incorrect results.

3.5.11 Performance considerations

Validation adds cost, but it can be structured to minimize overhead:

- combine validation and decoding passes,
- stop at the first invalid sequence for strict policies,
- use iterators that produce valid code-point streams,
- apply validation once at boundaries, not repeatedly.

3.5.12 Best practices for C++20–C++26

- Validate UTF-8 text as soon as it crosses an untrusted boundary.
- Document the policy (strict vs replacement vs lenient).
- Provide utility functions that return clear status and location of invalid data.
- Integrate validation with higher-level Unicode processing (normalization/segmentation).
- Avoid assuming that input is well-formed without verification.

3.5.13 Summary

UTF-8 validation is essential for correctness, security, and robust interoperability. Because UTF-8 is variable-length and self-synchronizing, malformed sequences can appear naturally in systems that do not enforce encoding contracts. C++20–C++26 provides the tools to build explicit validation at input boundaries, decode safely, and handle errors according to policy. A disciplined validation strategy protects text pipelines from subtle bugs and security risks while enabling correct higher-level Unicode text processing.

3.6 Common Anti-Patterns in UTF-8 Processing

3.6.1 Introduction

UTF-8 is the practical standard encoding for modern systems, but it is also a frequent source of bugs in C++ codebases because it is *byte-oriented* and *variable-length*. In C++20–C++26, `std::string` and `std::u8string` remain sequences of code units; the standard library does not automatically provide Unicode-aware operations such as normalization, grapheme segmentation, or collation.

This section catalogs the most common UTF-8 anti-patterns in C++ code, explains why they are wrong, and provides correct alternatives and design guidance.

3.6.2 Anti-pattern 1: ”one byte equals one character”

What it looks like

```
std::string s = /* UTF-8 text */;
for (std::size_t i = 0; i < s.size(); ++i) {
    char c = s[i]; // treated as a "character"
}
```

Why it is wrong

- In UTF-8, non-ASCII code points use 2–4 bytes.
- `s[i]` yields a **code unit** (byte), not a code point or grapheme.
- Many bytes in UTF-8 are continuation bytes and cannot be interpreted independently.

Correct alternative

- If you need **bytes**: keep the loop, but treat the value as a byte, not a character.

- If you need **code points**: decode UTF-8 while iterating.
- If you need **graphemes**: segment the decoded stream into grapheme clusters.

3.6.3 Anti-pattern 2: using `size()` as "character count"

What it looks like

```
std::size_t characters = s.size(); // wrong for UTF-8
```

Why it is wrong

`size()` returns the number of code units (bytes), not the number of Unicode scalar values, and not the number of user-perceived characters.

Correct alternative

Name the value correctly:

```
std::size_t byteCount = s.size();
```

If you need code point count, decode and count. If you need grapheme count, segment and count.

3.6.4 Anti-pattern 3: slicing and truncating by byte offsets without boundary checks

What it looks like

```
std::string prefix = s.substr(0, 10); // may cut mid-codepoint
```

Why it is wrong

`substr` slices bytes. Cutting a multi-byte sequence produces ill-formed UTF-8. Even if code point boundaries are preserved, truncation may split grapheme clusters.

Correct alternative: boundary-safe byte truncation

If you must limit by bytes but keep UTF-8 validity, truncate to a code point boundary:

```
#include <string_view>
#include <cstddef>

constexpr bool is_utf8_cont(unsigned char b) { return (b & 0xC0u) == 0x80u; }

// Best-effort: ensures the returned length ends on a non-continuation byte boundary.
std::size_t utf8_safe_prefix_len(std::string_view s, std::size_t maxBytes) {
    if (maxBytes >= s.size()) return s.size();
    while (maxBytes > 0 && is_utf8_cont(static_cast<unsigned char>(s[maxBytes]))) {
        --maxBytes;
    }
    return maxBytes;
}
```

If you need grapheme-safe truncation (user-visible character count), you must segment into grapheme clusters first.

3.6.5 Anti-pattern 4: using `std::tolower`/`std::isalpha` on UTF-8 bytes

What it looks like

```
for (char& c : s) {
    c = static_cast<char>(std::tolower(c)); // wrong and possibly undefined
}
```

Why it is wrong

- The ctype APIs operate on single-byte character values in the current locale, not on UTF-8 sequences.
- Passing negative `char` values (when `char` is signed) violates preconditions and can be undefined behavior.
- Lowercasing UTF-8 bytes corrupts multi-byte sequences.

Correct alternative

- For ASCII-only transforms, restrict to ASCII bytes and cast to `unsigned char` before calling `std::tolower`.
- For Unicode case-insensitive behavior, decode to code points and apply Unicode case folding (requires dedicated Unicode data/algorithms).

3.6.6 Anti-pattern 5: reversing a UTF-8 string by bytes

What it looks like

```
std::reverse(s.begin(), s.end()); // reverses bytes
```

Why it is wrong

Byte reversal breaks multi-byte code point sequences and produces invalid UTF-8.

Correct alternative

Define what ”reverse” means:

- reverse by code points (decode, reverse scalars, re-encode),

- reverse by grapheme clusters (segment, reverse clusters, reassemble).

3.6.7 Anti-pattern 6: mixing UTF-8 text and arbitrary bytes in `std::string` without contracts

What it looks like

```
std::string payload = read_from_network();
std::string text = payload; // assumes UTF-8
```

Why it is wrong

`std::string` does not carry encoding. Without a contract, code reviews cannot tell whether the data is text or binary. Bugs occur when binary blobs are processed as text or vice versa.

Correct alternative

- Use distinct types or aliases to separate ”bytes” from ”UTF-8 text”.
- In C++20+, use `std::u8string` / `std::u8string_view` for UTF-8 code units where practical.

```
using Bytes = std::string_view;
using Utf8  = std::u8string_view;

void process_bytes(Bytes b);
void process_utf8(Utf8 t);
```

3.6.8 Anti-pattern 7: assuming ”`find` gives character positions”

What it looks like

```
auto pos = s.find(needle);
```

```
auto charIndex = pos; // treated as character index (wrong)
```

Why it is wrong

`find` returns a byte offset. Treating it as a character index breaks subsequent slicing, highlighting, and UI positions.

Correct alternative

Either:

- keep it as a byte offset and slice at byte boundaries, or
- map byte offsets to code point/grapheme indices via decoding/segmentation indices.

3.6.9 Anti-pattern 8: comparing user-visible strings by raw code units without normalization policy

What it looks like

```
if (a == b) { /* considered equal */ } // raw byte equality
```

Why it is wrong

Unicode allows multiple canonically equivalent representations. Byte equality is not the same as user-perceived equality when normalization differs. This affects:

- user input matching,
- search and filtering,
- filenames and identifiers across systems.

Correct alternative

Define a normalization policy at boundaries:

- normalize stored text to a chosen form,
- normalize queries similarly before comparison,
- document where and when normalization occurs.

3.6.10 Anti-pattern 9: using `wchar_t` or wide I/O as a "Unicode solution"

What it looks like

```
std::wstring ws = /* ... */;
std::wcout << ws;
```

Why it is wrong

- `wchar_t` size and encoding are platform-dependent (often UTF-16 on Windows, UTF-32 on Unix-like systems).
- Wide I/O behavior depends on locale and runtime configuration and is not a portable Unicode pipeline.
- Wide strings still do not solve normalization, grapheme segmentation, or collation.

Correct alternative

Use explicit encoding forms at boundaries (commonly UTF-8) and apply explicit conversion layers when interfacing with OS APIs that require UTF-16.

3.6.11 Anti-pattern 10: decoding UTF-8 without rejecting overlong sequences and invalid ranges

What it looks like

```
// A naive decoder that just combines bits without validation.
```

Why it is wrong

A decoder must reject:

- overlong encodings (security and correctness risk),
- surrogate values (invalid as Unicode scalar values),
- values above `U+10FFFF`,
- missing/incorrect continuation bytes.

Failing to reject these can create security bypasses and inconsistent behavior.

Correct alternative

Always validate while decoding, and define an error policy:

- strict rejection,
- replacement,
- or explicit byte-preserving handling.

3.6.12 Anti-pattern 11: trusting input is UTF-8 without validation at boundaries

What it looks like

```
std::string userText = read_untrusted();
use_as_text(userText); // assumes valid UTF-8
```

Why it is wrong

Untrusted inputs regularly contain invalid sequences, mixed encodings, or truncated data. Downstream processing may fail, or vulnerabilities may appear in decoders and text rendering layers.

Correct alternative

Validate once at the boundary, then treat internal text as valid by invariant (documented and enforced).

3.6.13 Anti-pattern 12: implementing text UI logic in bytes

What it looks like

```
cursor += 1;        // moves by byte, not character
backspace();        // deletes one byte
selection_length(); // measured in bytes
```

Why it is wrong

UI editing semantics are grapheme-based. Deleting one byte can corrupt UTF-8, deleting one code point can still split grapheme clusters, and cursor movement must respect grapheme cluster boundaries.

Correct alternative

Represent cursor positions and editing operations in terms of grapheme cluster boundaries (or at least code point boundaries if domain permits), and map them to byte offsets.

3.6.14 Practical review checklist

When reviewing UTF-8 code, flag:

- any use of `size()` labeled as ”characters”,
- slicing/truncation without boundary checks,
- ctype usage on `char` without `unsigned char` cast,
- attempts to lowercase/uppercase by bytes,
- reversing bytes to reverse ”text”,
- UI logic implemented in bytes,
- missing validation at input boundaries,
- missing normalization policy where user-facing equality/search is required.

3.6.15 Summary

UTF-8 bugs in C++ usually come from treating code units as characters and applying byte-based APIs to problems that require Unicode semantics. The anti-patterns listed here are repeat offenders because they often ”work” on ASCII and fail on real-world multilingual text. Correct UTF-8 processing requires explicit encoding contracts, boundary validation, boundary-aware slicing, decoding for semantic operations, and (when needed) normalization and grapheme segmentation policies.

Chapter 4

UTF-16 and UTF-32: Use Cases and Limitations

4.1 UTF-16 and Its Relationship with Windows APIs

4.1.1 Overview of UTF-16 in Unicode

UTF-16 (16-bit Unicode Transformation Format) is a variable-length encoding defined by the Unicode Standard that represents Unicode code points using either one or two 16-bit code units. Code points in the Basic Multilingual Plane (BMP) (`U+0000..U+FFFF` except the surrogate range) are encoded as a single 16-bit code unit; supplementary code points (`U+10000..U+10FFFF`) are encoded as a *surrogate pair* of two 16-bit units. UTF-16 inherits from an older fixed-width encoding called UCS-2 but extends it to cover the full set of Unicode code points using surrogate pairs. UTF-16 remains widely used in several environments, including many high-level languages and systems. :contentReference[oaicite:0]index=0

4.1.2 Windows native text representation

Microsoft Windows historically adopted Unicode early in its history, initially using UCS-2 and later transitioning to UTF-16 as the Unicode standard evolved. As a result:

- The Win32 API (and many related OS subsystems) natively represent Unicode text using UTF-16 code units (little-endian in Windows memory).
  :contentReference[oaicite:1]index=1

- Core API calls that manipulate "wide" strings use UTF-16 and are typified by wide character types in C and C++ (`wchar_t`, `LPWSTR`, `LPCWSTR`).
  :contentReference[oaicite:2]index=2

- Many foundational APIs provide two variants: an "ANSI" (`A`) version that operates on legacy code pages and a "wide" (`W`) version that operates on UTF-16.
  :contentReference[oaicite:3]index=3

This design reflects decisions made when Windows introduced Unicode support well before UTF-8 became dominant outside specialized systems. At that time, UCS-2 (the precursor to UTF-16) was widely available, and distinguishing textual interfaces on a 16-bit code unit basis was a pragmatic compromise. :contentReference[oaicite:4]index=4

4.1.3 Windows API string types and UTF-16

In the Windows API:

- `wchar_t` is defined as a 16-bit integral type and is the native code unit for UTF-16 text.
  :contentReference[oaicite:5]index=5

- `LPWSTR` and `LPCWSTR` denote pointers to wide (UTF-16) strings passed to or returned from APIs. :contentReference[oaicite:6]index=6

- The generic data type `TCHAR` and function macros `_TCHAR`, `TEXT()`, and `CreateFile` map to either ANSI or wide variants depending on compilation settings, but modern documentation strongly encourages explicit Unicode (wide) usage. :contentReference[oaicite:7]index=7

These wide APIs provide operations for filesystem paths, registry keys, message text, graphical interfaces, and more, ensuring that applications can represent the full Unicode repertoire when interacting with system services.

4.1.4 UTF-16 vs UTF-8 on Windows

Windows retains native UTF-16 support in its APIs. Until recently, UTF-8 support was limited or experimental at the OS level; wide APIs were the norm for system text and file names. UTF-8 support is available in newer Windows builds through explicit configuration (such as application manifests enabling UTF-8 code pages), but the conventional Win32 encoding remains UTF-16. :contentReference[oaicite:8]index=8
Legacy code pages ("ANSI" variants) still appear in some APIs, but when Unicode support is enabled, the wide UTF-16 versions of functions are the preferred interface. UTF-8 can interoperate with Win32 APIs by converting to and from UTF-16 at the boundary.

4.1.5 Implications for C++ development on Windows

Because Windows APIs use UTF-16:

- C++ code interacting with OS services often needs to convert between UTF-8 (common in application logic) and UTF-16 (required for APIs). Typically, conversion routines such as `WideCharToMultiByte` and `MultiByteToWideChar` are used to map between UTF-16 and UTF-8 or legacy encodings. :contentReference[oaicite:9]index=9

- Native wide string types (`std::wstring`) often serve as intermediate representations for API calls. However, care must be taken when storing or manipulating `wchar_t`

sequences, as such code is platform-dependent; on non-Windows systems, `wchar_t` may have different width and encoding semantics. :contentReference[oaicite:10]index=10

- File names, registry strings, window captions, and other OS text fields delivered via wide APIs are encoded as UTF-16. Passing UTF-8 application text directly to these APIs without conversion will not produce correct results. Developers must apply explicit transcoding at the boundary. :contentReference[oaicite:11]index=11

4.1.6 Surrogate pairs and supplementary planes

UTF-16 uses surrogate pairs (two 16-bit code units) to encode code points outside the BMP (`U+10000..U+10FFFF`). On Windows this means:

- APIs operate on arrays of 16-bit code units but do not inherently interpret surrogate pair boundaries; the developer must ensure correct handling when iterating, measuring length, or modifying strings. :contentReference[oaicite:12]index=12
- Higher-level Unicode logic (normalization, grapheme segmentation, collation) remains the responsibility of the application or a dedicated library; the OS ABI itself does not supply these behaviors. :contentReference[oaicite:13]index=13

Thus, UTF-16 wide strings on Windows function as byte sequences of 16-bit elements at the ABI level, and semantic correctness requires explicit Unicode logic when necessary.

4.1.7 Conversion patterns in modern C++

To integrate UTF-8-centric application logic with Windows APIs:

```
#include <string>
#include <windows.h>

// Convert UTF-8 to UTF-16 (wstring)
```

```
std::wstring utf8_to_utf16(std::string_view utf8) {
    int needed = MultiByteToWideChar(CP_UTF8, 0,
        utf8.data(), int(utf8.size()), nullptr, 0);
    std::wstring utf16(needed, L'\\0');
    MultiByteToWideChar(CP_UTF8, 0,
        utf8.data(), int(utf8.size()), utf16.data(), needed);
    return utf16;
}

// Convert UTF-16 to UTF-8
std::string utf16_to_utf8(std::wstring_view utf16) {
    int needed = WideCharToMultiByte(CP_UTF8, 0,
        utf16.data(), int(utf16.size()), nullptr, 0, nullptr, nullptr);
    std::string utf8(needed, '\\0');
    WideCharToMultiByte(CP_UTF8, 0,
        utf16.data(), int(utf16.size()), utf8.data(), needed, nullptr, nullptr);
    return utf8;
}
```

These conversion helpers are common in modern C++ Windows applications that maintain UTF-8 as the internal encoding and convert only when calling or returning from OS services.

4.1.8 Summary

UTF-16 is the native Unicode encoding format for Windows APIs, deeply embedded in the platform's ABI and legacy design. C++ programs targeting Windows must understand that:

- Windows wide APIs use `wchar_t` sequences in UTF-16 little-endian format as the canonical Unicode representation. :contentReference[oaicite:14]index=14
- Interfacing between UTF-8 application logic and Windows APIs requires explicit conversion. :contentReference[oaicite:15]index=15

- Surrogate pairs and encoding boundaries are not handled automatically by the ABI; semantic Unicode logic must be implemented or provided by libraries. :contentReference[oaicite:16]index=16
- Legacy APIs and code page interfaces exist but should be avoided in favor of Unicode (UTF-16) or UTF-8 with boundary conversions when possible. :contentReference[oaicite:17]index=17

Understanding the relationship between UTF-16 and Windows APIs is essential for correct, portable Unicode handling in modern C++ applications that target the Windows platform.

4.2 Surrogate Pairs and Their Impact on Text Algorithms

4.2.1 Why surrogate pairs exist

UTF-16 is a 16-bit code-unit encoding. It represents Unicode scalar values using:

- **one 16-bit code unit** for values in the Basic Multilingual Plane (BMP), excluding the surrogate range, and
- **two 16-bit code units** (a *surrogate pair*) for values outside the BMP (`U+10000..U+10FFFF`).

The surrogate mechanism allows UTF-16 to encode the full Unicode range without abandoning the 16-bit code-unit model that many platforms and ABIs historically adopted.

4.2.2 Surrogate ranges and terminology

Surrogates are reserved code points that do *not* represent characters by themselves. They are used only as halves of a pair:

- **High surrogates**: `0xD800..0xDBFF`
- **Low surrogates**: `0xDC00..0xDFFF`

A valid surrogate pair is a high surrogate followed immediately by a low surrogate. Any other occurrence is ill-formed UTF-16 (with an important real-world caveat discussed later).

4.2.3 How a surrogate pair encodes a code point

A surrogate pair encodes a 20-bit value that maps to the supplementary planes:

- Let `H` be the high surrogate, `L` the low surrogate.

- Compute:

$$\texttt{U} = 0x10000 + ((\texttt{H} - 0xD800) \ll 10) + (\texttt{L} - 0xDC00)$$

This produces a Unicode scalar value `U` in `0x10000..0x10FFFF`.

4.2.4 The core algorithmic reality: UTF-16 is still variable-length

A central misconception is treating UTF-16 as ”fixed-width characters” because code units are 16 bits. In fact, UTF-16 is variable-length at the code point level:

- BMP scalar values: 1 code unit
- Supplementary scalar values: 2 code units (surrogate pair)

Therefore, many algorithms that appear correct on BMP-only data fail on real-world text with emoji, many historic scripts, and numerous modern symbols.

4.2.5 Impact 1: length and indexing semantics

Code-unit length is not code-point length

`std::u16string::size()` counts **code units**, not Unicode scalar values, and certainly not user-perceived characters.

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

int main() {
    // "  " is U+1F600 (supplementary plane).
    // In UTF-16 it occupies 2 code units (a surrogate pair).
    std::u16string s = u"  ";
    std::cout << "code units = " << s.size() << "\n"; // prints 2
}
```

Indexing is code-unit indexing

`s[i]` yields the i-th 16-bit unit. If i lands on a high surrogate, you have only half of a character; if it lands on a low surrogate, it is an invalid starting point for decoding a scalar value.

Consequence Any algorithm that assumes "`s[i]` is a character" is wrong for supplementary characters.

4.2.6 Impact 2: iteration and decoding correctness

Correct code-point iteration requires surrogate-aware decoding

A correct iterator over Unicode scalar values must detect surrogate pairs.

```
#include <cstdint>
#include <optional>
#include <string_view>

constexpr bool is_high_surrogate(char16_t x) {
    return x >= 0xD800 && x <= 0xDBFF;
}
constexpr bool is_low_surrogate(char16_t x) {
    return x >= 0xDC00 && x <= 0xDFFF;
}

struct U16DecodeResult {
    char32_t scalar;      // Unicode scalar value (or a policy-specific substitute)
    std::size_t units;    // number of char16_t consumed: 1 or 2
    bool well_formed;     // whether decoding consumed a well-formed scalar value
};

std::optional<U16DecodeResult> decode_utf16_at(std::u16string_view s, std::size_t i) {
    if (i >= s.size()) return std::nullopt;
```

```
    char16_t a = s[i];

    // Non-surrogate BMP scalar value (excluding surrogate range)
    if (!is_high_surrogate(a) && !is_low_surrogate(a)) {
        return U16DecodeResult{ static_cast<char32_t>(a), 1, true };
    }

    // High surrogate must be followed by low surrogate
    if (is_high_surrogate(a)) {
        if (i + 1 >= s.size()) {
            return U16DecodeResult{ 0xFFFD, 1, false }; // truncated pair
        }
        char16_t b = s[i + 1];
        if (!is_low_surrogate(b)) {
            return U16DecodeResult{ 0xFFFD, 1, false }; // malformed pair
        }
        char32_t u = 0x10000;
        u += (static_cast<char32_t>(a) - 0xD800) << 10;
        u += (static_cast<char32_t>(b) - 0xDC00);
        return U16DecodeResult{ u, 2, true };
    }

    // Lone low surrogate is malformed in well-formed UTF-16
    return U16DecodeResult{ 0xFFFD, 1, false };
}
```

Algorithmic consequence Any code-point iteration over UTF-16 must do at least:

- a surrogate-range test, and
- conditional consumption of two code units.

4.2.7 Impact 3: substring, slicing, and editing operations

Slicing by code units can split surrogate pairs

`subs_tr` and range erasures operate on code units. If you slice in the middle of a surrogate pair, you create ill-formed UTF-16.

```
#include <string>

int main() {
    std::u16string s = u"A  B";    // '  ' is two code units
    std::u16string bad = s.substr(1, 1); // likely isolates half of the surrogate pair
}
```

Consequence Text algorithms that implement ”cut”, ”paste”, ”delete one character”, ”cursor left/right” must operate on:

- code point boundaries (at minimum), and
- grapheme cluster boundaries for user-facing editing semantics.

Boundary-safe slicing by code point

A minimal boundary check for UTF-16 code point boundaries:

- A boundary cannot be between a high surrogate and a low surrogate.

```
#include <string_view>
#include <cstddef>

constexpr bool is_high_surrogate(char16_t x) { return x >= 0xD800 && x <= 0xDBFF; }
constexpr bool is_low_surrogate (char16_t x) { return x >= 0xDC00 && x <= 0xDFFF; }

// Returns true if i is a code point boundary in UTF-16 (best-effort).
```

```cpp
bool is_u16_codepoint_boundary(std::u16string_view s, std::size_t i) {
    if (i == 0 || i >= s.size()) return true;
    // Not a boundary if it splits a surrogate pair: high at i-1 and low at i.
    return !(is_high_surrogate(s[i - 1]) && is_low_surrogate(s[i]));
}
```

This check is necessary but not sufficient for full validation; it prevents splitting valid pairs but does not repair malformed sequences.

4.2.8 Impact 4: searching and pattern matching

Naive ”character search” is ill-defined

Searching in UTF-16 by code unit sequences can work only if you understand what you are searching for:

- Searching for a BMP scalar can be a single code unit match.
- Searching for a supplementary scalar requires matching two code units (a surrogate pair).

Common bug A search routine that assumes one code unit equals one character may:

- match only half a surrogate pair (meaningless),
- miscompute match positions in terms of ”characters”,
- break subsequent slicing or highlighting.

Regular expressions and ”dot” semantics

Regex engines that operate on UTF-16 code units may treat:

- ”.” as one code unit (splitting surrogate pairs), unless the engine is Unicode-aware at the code point level,

- character classes as code units unless explicitly specified.

Practical takeaway When using a regex engine in a UTF-16 environment, you must know whether it interprets the input as:

- 16-bit code units, or
- Unicode scalar values (code points), or
- grapheme clusters (rare).

4.2.9 Impact 5: case mapping, normalization, and grapheme clusters

Surrogate correctness is only the first step. Even with perfect surrogate handling:

- **Case mapping** can expand or change code point sequences.
- **Normalization** can transform text into canonically equivalent but different code point sequences.
- **Grapheme clusters** can span multiple code points (and thus multiple code units), even entirely within the BMP.

Implication ”Delete one character” in a user interface is *not* ”delete one UTF-16 code unit” and is not always ”delete one code point”.

4.2.10 Impact 6: performance and complexity trade-offs

Handling surrogate pairs adds branching and sometimes additional memory traffic. The real cost depends on the text profile:

- For BMP-heavy text: most code points are single code units; decoding is cheap.

- For emoji-heavy or historic-script-heavy text: surrogate pairs are common; decoding and indexing cost increases.

Indexing consequence Random access by code point index in UTF-16 is not guaranteed $O(1)$ unless you build an index of boundaries, because some code points consume two code units.

4.2.11 Impact 7: ill-formed UTF-16 in real systems

Strictly speaking, well-formed UTF-16 must not contain unpaired surrogates. However, real systems sometimes allow unpaired surrogates to appear in data (notably in some platform-specific contexts such as certain filesystem namespaces). This creates a pragmatic requirement:

- Some software must round-trip such sequences losslessly (treating them as code units rather than valid Unicode text).

- Other software must reject them to maintain the invariant ”internal text is valid Unicode”.

Engineering decision Define a **boundary policy**:

- **Strict Unicode policy**: reject unpaired surrogates at ingestion.

- **Replacement policy**: convert ill-formed sequences to a replacement scalar (commonly U+FFFD) when decoding.

- **Lossless code-unit policy**: treat the data as a 16-bit code-unit string for round-tripping, and only decode where permitted.

4.2.12 Recommended design rules for C++ text algorithms over UTF-16

1. **Name your units**: distinguish `codeUnits`, `codePoints`, and `graphemes` in variable names and APIs.
2. **Never index UTF-16 as "characters"**: `s[i]` is a code unit.
3. **Never slice blindly**: ensure boundaries do not split surrogate pairs; use code point or grapheme segmentation where needed.
4. **Validate at boundaries**: decide strict/replacement/lossless behavior and document it.
5. **Separate storage from semantics**: store UTF-16 when required by platform APIs, but decode to scalar values for Unicode-aware processing stages.
6. **Treat UI editing as grapheme-based**: surrogate-correct code point iteration is necessary but not sufficient for user-facing text behavior.

4.2.13 Summary

Surrogate pairs make UTF-16 a variable-length encoding at the code point level. Their existence impacts nearly every non-trivial text algorithm:

- length and indexing are code-unit-based,
- iteration requires surrogate-aware decoding,
- slicing and editing can corrupt text if they split pairs,
- searching and regex behavior depends on whether engines operate on code units or code points,
- higher-level semantics (normalization, case mapping, grapheme clusters) remain separate and can still defeat naive "character" logic,

- real systems may contain ill-formed surrogate sequences, requiring explicit boundary policies.

A robust modern C++ design treats UTF-16 as a code-unit storage format, enforces explicit decoding and validation at boundaries, and chooses code point or grapheme semantics deliberately based on the problem domain.

4.3 UTF-32: Fixed-Width Encoding and Its Costs

4.3.1 Introduction

UTF-32 is a Unicode Transformation Format that uses one 32-bit code unit (typically `char32_t` in C++) for every Unicode scalar value. In contrast to UTF-8 and UTF-16, which are variable-length, UTF-32 is a fixed-width encoding at the code-point level. This yields conceptual simplicity: each code unit directly corresponds to one Unicode scalar value (excluding surrogate code points, which do not exist in UTF-32). However, this simplicity comes at measurable memory and processing costs. Understanding these costs and trade-offs is essential for text handling in modern C++ applications.

4.3.2 UTF-32 semantics

Fixed-width guarantee

In UTF-32:

- each element is exactly one 32-bit code unit,
- each code unit represents one Unicode scalar value (`U+0000..U+10FFFF`),
- surrogate code points are never encoded,
- decoding is trivial: the code unit is the code point.

This means:

- indexing by code point is trivial and constant time ($O(1)$),
- code point iteration does not require variable-length decoding logic,
- storage semantics are unambiguous.

This fixed-width property simplifies many algorithms conceptually, but it also masks the complexity of higher-level semantics such as normalization, grapheme segmentation, and case folding, which operate on sequences of code points rather than individual units.

4.3.3 Memory footprint: substantial compared to UTF-8/UTF-16

The most obvious cost of UTF-32 is memory:

- every Unicode scalar value occupies one 32-bit unit,
- typical UTF-8 text (especially ASCII-heavy) uses far fewer bytes,
- typical UTF-16 text uses either 2 bytes or 4 bytes per code point.

This becomes significant in large text datasets or memory-constrained environments. For example, an ASCII string that occupies `N` bytes in UTF-8 occupies `4N` bytes in UTF-32.

4.3.4 Performance trade-offs

Iteration and indexing

Because each code point is one 32-bit unit:

- accessing the `k`-th code point is trivial and constant time,
- a single pass over a UTF-32 sequence requires no boundary detection,
- pointers and iterators move predictably by fixed strides.

This can simplify algorithm implementation and reduce branching overhead per code point. However, this performance advantage must be weighed against increased memory traffic due to the larger data representation.

Cache behavior and locality

The cost of larger code units is seen in cache efficiency:

- more memory must be brought into cache for the same logical text,
- L1/L2/L3 cache utilization is reduced compared to UTF-8,
- memory bandwidth is consumed more heavily, especially in large texts.

For workloads that scan entire texts, memory bandwidth can become a dominating factor, making UTF-32 slower in real throughput even if per-code-point logic is simpler.

4.3.5 Use cases where UTF-32 excels

Algorithmic simplicity

Algorithms that operate on code points without needing encoding boundaries benefit from UTF-32:

- code-point arithmetic (e.g., direct numeric properties),
- simple indexing and random access,
- clear mapping between array index and Unicode scalar value.

UTF-32 can reduce implementation complexity compared to UTF-8, where variable-length decoding is required to find the next code point or to index into a sequence.

Text analysis and transformation pipelines

Many text analysis tasks require multiple passes over code points:

- building frequency tables,

- applying Unicode property tests,
- performing case folding or normalization,
- pattern matching at the code-point level.

Using UTF-32 as an intermediate representation after decoding can reduce redundant decoding work. For example, if a pipeline repeatedly accesses code points, it may be more efficient to decode to UTF-32 once and then operate on the resulting sequence.

4.3.6 Higher-level semantics remain separate

Although UTF-32 simplifies many encoding-level tasks, it does not inherently provide:

- normalization (canonical or compatibility),
- grapheme cluster segmentation,
- collation (language-aware sorting),
- bidirectional text layout,
- combining mark handling for user-perceived character boundaries.

These semantic layers operate on sequences of code points and require additional logic beyond fixed-width encoding.

4.3.7 Conversion costs

In real systems, UTF-32 is rarely the native encoding for external I/O. Therefore, conversions are often necessary:

- **UTF-8 to UTF-32**: decode each code point and store in 32-bit units,

- **UTF-16 to UTF-32**: decode surrogate pairs and BMP values to scalar values,
- **UTF-32 to UTF-8 or UTF-16**: encode each scalar value into variable-length sequences.

These conversions add overhead, which must be considered when UTF-32 is used as an intermediate format in pipelines or for algorithmic operations.

```
#include <string>
#include <vector>
#include <cstddef>
#include <optional>

// Conceptual: decode a UTF-8 buffer into a UTF-32 vector of code points
std::vector<char32_t> utf8_to_utf32(std::u8string_view input) {
    std::vector<char32_t> result;
    for (std::size_t i = 0; i < input.size(); ) {
        unsigned char lead = static_cast<unsigned char>(input[i]);
        // decode a code point from UTF-8
        // append to result
        // advance i by the length of the code point encoding
    }
    return result;
}
```

Such conversion routines are common in Unicode libraries and frameworks that need a stable, uniform code-point representation for internal algorithms.

4.3.8 Memory reduction strategies

Selective decoding

Instead of storing an entire text in UTF-32, decode only portions needed for a particular algorithm, while retaining the original UTF-8 or UTF-16 data as storage.

Hybrid models

Some systems store:

- frequently accessed substrings or segments in UTF-32,
- long-term storage in UTF-8,
- conversion layers at API boundaries.

This hybrid approach balances memory cost with algorithmic simplicity for intensive text operations.

4.3.9 Comparative summary

- **UTF-8**: compact for ASCII-heavy data, widespread I/O support, requires variable-length decoding for code-point semantics.
- **UTF-16**: variable-length at the code-point level (surrogate pairs), native in many platform APIs (notably Windows), moderate memory footprint.
- **UTF-32**: fixed-width at the code-point level, simple indexing and iteration, highest memory footprint, conversion cost for external interfaces.

4.3.10 Practical guidance for C++ text systems

- Use UTF-8 at external boundaries (files, networks, interop).
- Decode to UTF-32 when algorithms benefit from direct code-point access and the memory footprint is acceptable.
- Use UTF-16 as needed for platform interfaces that require it (e.g., Windows native APIs).

- Do not assume that encoding alone solves higher-level semantics such as grapheme clusters or normalization.
- Explicitly document encoding choices and conversion points in the system architecture.

4.3.11 Summary

UTF-32 is the simplest encoding in terms of code-point representation: one 32-bit code unit per Unicode scalar value. This fixed-width property simplifies certain algorithms and random access patterns, but it comes with significant costs in memory and cache behavior. It remains a valuable internal representation in pipelines and algorithms that can tolerate its footprint and where the simplicity of direct indexing outweighs the overhead of conversion and storage. In modern C++20–C++26 systems, UTF-32 is a tool in the encoding toolkit, used where appropriate but not assumed to replace UTF-8 for interchange or UTF-16 for platform interop.

4.4 Performance, Cache Behavior, and Memory Overhead

4.4.1 Introduction

When engineers debate UTF-8 vs UTF-16 vs UTF-32, the discussion often starts with ”decoding cost” and ends there. In real systems, performance is dominated by *memory behavior*: cache locality, bandwidth, allocation patterns, and the frequency of conversions at boundaries. UTF-16 and UTF-32 change the physical layout of text in memory, which can shift the bottleneck from CPU cycles to memory traffic (or vice versa).
This section presents a practical, system-level view of performance for UTF-16 and UTF-32, emphasizing cache behavior, memory overhead, and the algorithmic patterns that become fast or slow depending on representation.

4.4.2 Three layers of cost

For text workloads, cost typically decomposes into three interacting layers:

1. **Representation cost (storage)**: bytes moved through memory hierarchy per logical unit of text.

2. **Traversal cost (iteration/indexing)**: CPU work to advance through logical units (code points, graphemes).

3. **Semantic cost (Unicode algorithms)**: normalization, case folding, segmentation, collation; these often dominate regardless of encoding.

UTF-16 and UTF-32 primarily change (1) and (2). Many teams overestimate (2) and underestimate (1).

4.4.3 Memory overhead: what you pay per code point

Code units vs code points

- UTF-16 uses 16-bit code units. Code points in the BMP generally use 1 unit; supplementary code points use 2 units (surrogate pair).
- UTF-32 uses a single 32-bit code unit per Unicode scalar value.

Approximate memory ratios for typical text

Let N be the number of Unicode scalar values, and let p be the fraction of code points outside the BMP (requiring surrogate pairs in UTF-16). Then:

$$\text{UTF-16 code units} \approx N(1+p)$$

$$\text{UTF-16 bytes} \approx 2N(1+p)$$

$$\text{UTF-32 bytes} = 4N$$

So UTF-32 is always exactly 4 bytes per scalar value, while UTF-16 ranges from approximately 2 bytes per scalar value (BMP-heavy) toward 4 bytes per scalar value (supplementary-heavy).

Key observation In many practical corpora (source code, logs, JSON, config), BMP text dominates; surrogate pairs are present but not usually the majority. For such corpora, UTF-16 tends to be closer to 2 bytes per scalar value than 4, while UTF-32 is fixed at 4.

Hidden overhead: allocations and metadata

The encoding is only part of the memory story. Strings also carry:

- object headers (size, capacity, pointer) and potential small-string optimization storage,
- allocator metadata and fragmentation overhead,

- alignment padding (often more visible for 16-bit and 32-bit element types),
- additional indexing structures if you build code-point or grapheme indices.

If your design creates many small strings (tokenization, substring extraction, AST nodes), object and allocator overhead can dominate encoding overhead.

4.4.4 Cache behavior: why compact encodings often win in throughput

Caches are measured in bytes, not characters

CPU caches store fixed byte counts. If text is wider in memory:

- fewer logical symbols fit per cache line,
- more cache lines are touched for a full scan,
- memory bandwidth becomes the limiter sooner.

Implication Even if UTF-32 eliminates variable-length decoding, it may run *slower* in end-to-end scans because it moves more data through caches and memory.

Streaming scans vs random access

Streaming scan workloads (tokenizing, searching delimiters, counting code points, validating encoding):

- often become bandwidth-limited;
- benefit from compact representations (fewer bytes moved);
- are friendly to prefetching and sequential access.

Random access workloads (jump to the k-th code point repeatedly, frequent indexing by logical position):

- benefit from fixed-width code-point indexing (UTF-32);
- or from precomputed indices over UTF-16/UTF-8;
- may suffer if representation forces repeated decoding from the start.

Engineering rule If you mostly *scan* text, compactness usually beats fixed-width simplicity. If you mostly *index* by code point repeatedly, fixed-width or indexing structures become important.

4.4.5 Branching and decoder overhead: the cost you see first (but not always the cost that matters)

UTF-16 decoding requires surrogate checks; UTF-32 decoding is trivial; UTF-8 decoding is variable-length. However:

- surrogate checks are predictable if most code units are BMP;
- branch prediction becomes expensive when the surrogate rate is high or data is adversarial;
- decoding cost may be dwarfed by memory traffic for large scans.

UTF-16 occupies a middle ground:

- It reduces decoding complexity compared to UTF-8 for many BMP-heavy texts.
- It still requires surrogate-aware iteration and boundary handling.
- It often increases memory traffic compared to UTF-8 but less than UTF-32.

4.4.6 Algorithmic consequences that directly affect performance

Length and indexing semantics

- `std::u16string::size()` is in *code units*, not code points.
- `std::u32string::size()` is in *code points* (scalar values), but still not grapheme clusters.

If a subsystem repeatedly converts between ”logical positions” and code-unit indices, it may become a hotspot. Common fixes:

- keep byte/code-unit offsets as the canonical internal coordinate,
- or build an index from code point number to code-unit offset (and update it incrementally).

Substring and copying behavior

Substrings can silently become expensive if they allocate and copy:

- UTF-32 substrings copy 4 bytes per code point,
- UTF-16 substrings copy 2 bytes per code unit, and may copy twice for surrogate-heavy segments,
- repeated substring extraction can turn linear scans into quadratic copying overhead.

Guidance:

- prefer views (`std::basic_string_view<char16_t>` / `char32_t>`) for slicing without copying,
- when editing, consider rope/piece-table/gap-buffer designs rather than repeated concatenation.

Normalization and case folding dominate

If your workflow requires normalization and Unicode case folding for correctness, the dominant costs often become:

- table lookups,
- multi-code-point expansions,
- additional allocations,
- multi-pass transformations.

In such cases, the difference between UTF-16 and UTF-32 is frequently secondary. Choose the representation that minimizes conversions and memory churn in your pipeline.

4.4.7 Conversion overhead and boundary strategy

UTF-16 is strongly associated with Windows APIs; UTF-8 dominates interchange formats; UTF-32 is often an internal working representation. Conversions are unavoidable in mixed environments. The key performance question is not ”is conversion expensive?”, but:

> How often do you convert, how much do you convert, and can you avoid repeated conversions?

Common high-cost anti-pattern:

- decode UTF-8 to UTF-16 for one API call, then immediately convert back, inside a loop.

Preferred approach:

- pick a canonical internal representation for the subsystem,
- convert at boundaries once,
- cache converted results when repeatedly used.

4.4.8 Measuring what matters: a portable micro-benchmark scaffold

The following code is a *measurement scaffold* you can paste into a C++20 project to compare:

- scanning code units,
- surrogate-aware decoding for UTF-16,
- trivial iteration for UTF-32.

It is intentionally simple and standard-only. Real benchmarks must control inputs, disable I/O, warm caches, and run multiple trials.

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

static constexpr bool is_high_surrogate(char16_t x) {
    return x >= 0xD800 && x <= 0xDBFF;
}
static constexpr bool is_low_surrogate(char16_t x) {
    return x >= 0xDC00 && x <= 0xDFFF;
}

// Count Unicode scalar values in UTF-16, treating invalid sequences as consuming 1 code
↪  unit.
// This is a performance scaffold, not a full policy engine.
std::size_t count_codepoints_utf16(std::u16string_view s) {
    std::size_t count = 0;
    for (std::size_t i = 0; i < s.size(); ) {
        char16_t a = s[i];
        if (is_high_surrogate(a) && (i + 1 < s.size()) && is_low_surrogate(s[i + 1])) {
```

```
            i += 2;
        } else {
            i += 1;
        }
        ++count;
    }
    return count;
}

std::size_t sum_codeunits_utf16(std::u16string_view s) {
    std::size_t sum = 0;
    for (char16_t cu : s) sum += static_cast<std::size_t>(cu);
    return sum;
}

std::size_t sum_codepoints_utf32(std::u32string_view s) {
    std::size_t sum = 0;
    for (char32_t cp : s) sum += static_cast<std::size_t>(cp);
    return sum;
}

template <class F>
auto time_it(F&& f) {
    using clock = std::chrono::steady_clock;
    auto t0 = clock::now();
    auto r  = f();
    auto t1 = clock::now();
    return std::pair{r, std::chrono::duration_cast<std::chrono::microseconds>(t1 - t0)};
}

int main() {
    // Construct a mixed UTF-16 text: BMP + supplementary emoji.
    // Note: literal handling depends on source file encoding; treat this as a scaffold.
```

```
    std::u16string s16 = u"Hello \u03A9 \u4E2D \U0001F600 \U0001F680";
    // UTF-32 direct code point representation:
    std::u32string s32 = U"Hello \u03A9 \u4E2D \U0001F600 \U0001F680";

    auto [sum16, t16] = time_it([&]{ return sum_codeunits_utf16(s16); });
    auto [cnt16, t16c] = time_it([&]{ return count_codepoints_utf16(s16); });
    auto [sum32, t32] = time_it([&]{ return sum_codepoints_utf32(s32); });

    std::cout << "utf16 codeunits sum=" << sum16 << " time(us)=" << t16.count() << "\n";
    std::cout << "utf16 codepoints count=" << cnt16 << " time(us)=" << t16c.count() <<
    ↪  "\n";
    std::cout << "utf32 codepoints sum=" << sum32 << " time(us)=" << t32.count() << "\n";
}
```

How to interpret results

- UTF-32 iteration is computationally simple but touches more memory per scalar value.
- UTF-16 code-point counting adds surrogate checks, but may still be efficient if surrogates are rare.
- The winning representation depends on whether your real workload is compute-bound or bandwidth-bound.

4.4.9 Practical guidance: choosing between UTF-16 and UTF-32 for performance

Choose UTF-16 when

- you are heavily integrated with Windows APIs or other UTF-16 ecosystems and want to minimize conversions,
- your text is mostly BMP and you benefit from 2-byte code units,

- you primarily do streaming scans and want better cache density than UTF-32,
- you can tolerate surrogate-aware iteration for the semantic operations you need.

Choose UTF-32 when

- your algorithms are inherently code-point-indexed (frequent random access by code point),
- you need a simple internal representation for Unicode-property-heavy logic and want to avoid repeated decoding,
- memory is not a primary constraint (or text volumes are modest),
- you accept that many user-facing semantics are still grapheme-based and require additional processing anyway.

Hybrid approach that often performs best

A common high-performance architecture is:

- store and interchange as UTF-8,
- convert to UTF-16 only at Windows API boundaries,
- decode to UTF-32 (or a code-point buffer) only in compute-heavy Unicode algorithm stages,
- keep indices (byte/code-unit offsets) for fast mapping rather than converting repeatedly.

4.4.10 Summary

Performance for UTF-16 and UTF-32 is primarily shaped by memory behavior:

- UTF-32 gives fixed-width, code-point-indexable storage but doubles memory compared to BMP-heavy UTF-16 and often quadruples versus ASCII-heavy UTF-8, reducing cache efficiency.
- UTF-16 reduces memory traffic compared to UTF-32 for BMP-heavy text but remains variable-length at the code-point level due to surrogate pairs, which affects indexing and slicing correctness.
- Streaming scans tend to reward compact encodings; random-access-by-code-point tends to reward fixed-width encodings or precomputed indices.
- Conversion frequency and allocation patterns are often larger performance factors than the raw decoding cost of a single pass.

The correct engineering decision is therefore workload-driven: measure the dominant operations, quantify conversion frequency, and optimize for the memory hierarchy rather than for an abstract ”decoding complexity” narrative.

4.5 Choosing the Right Encoding for the Right Layer

4.5.1 Introduction

Unicode text handling in C++ spans multiple layers: external interfaces, in-memory representation, internal algorithms, and user-facing semantics. No single encoding solves all problems uniformly. Modern C++ practice separates *storage and interchange encodings* from *internal semantic representations*. This section provides a principled strategy for selecting the most appropriate encoding for each layer of text processing, balancing performance, correctness, and interoperability.

4.5.2 The multi-layer text processing model

A robust text system separates concerns across layers:

- **Boundary layer (I/O and interop)**: what encoding is used for files, network protocols, UI frameworks, and OS APIs.
- **Storage layer (in-memory)**: how text is stored within data structures during program execution.
- **Algorithm layer**: how text semantics (code points, normalization, collation, grapheme clusters) are realized in operations.
- **Presentation layer**: how text is rendered or interacted with in user interfaces.

Each layer has different invariants, performance characteristics, and correctness metrics. Choosing a representation suited to each layer avoids cross-cutting anti-patterns.

4.5.3 Boundary layer: UTF-8 as the lingua franca

Interchange formats

UTF-8 is the dominant encoding for files, network protocols, configuration formats (JSON, XML), and logs:

- It is backward compatible with ASCII, facilitating incremental adoption.
- It has no endianness issues and does not require BOM markers.
- It is byte-oriented and integrates with existing I/O and buffer APIs.

Rule At every external boundary where text enters or exits the system, treat UTF-8 as the canonical encoding unless a specific API or protocol dictates otherwise.

OS and environment boundaries

Different platforms have different expectations:

- Many modern Unix-like systems treat file names and environment variables as UTF-8.
- Windows native APIs use UTF-16; therefore, application code must bridge between UTF-8 and UTF-16 at the OS boundary.

Implication Conversion at the boundary should be explicit, localized, and predictable:

```
std::wstring utf8_to_utf16(std::string_view utf8);
std::string  utf16_to_utf8(std::wstring_view utf16);
```

4.5.4 Storage layer: choose based on workload

UTF-8: compact and cache-friendly

Use UTF-8 for storage when:

- text is predominantly ASCII or BMP-heavy,
- memory footprint and cache locality matter,
- you perform sequential scanning or streaming operations,
- the majority of processing can use byte-oriented logic or one-pass decoding.

UTF-8's variable-length nature is only a disadvantage when random access by code point is frequent and unindexed.

UTF-16: platform ABI alignment

Use UTF-16 when:

- you need to interact heavily with APIs that require UTF-16 (notably Windows native APIs),
- you wish to minimize boundary conversions for those APIs,
- text is predominantly BMP and surrogate pairs are rare,
- you are willing to pay surrogate pair handling costs in exchange for medium memory footprint.

UTF-32: fixed-width semantic representation

Use UTF-32 when:

- your algorithms require fast random access by Unicode scalar index,
- you build internal indices (e.g., for search, editing, or transformation),
- memory overhead is acceptable for the text volume,

- you wish to simplify code-point logic at the expense of memory.

UTF-32 excels as an *internal working representation* for code-point-centric algorithms but is rarely used for storage or interchange due to its memory footprint.

4.5.5 Algorithm layer: separate encoding from semantics

Once text is loaded and stored, **semantic operations** (normalization, collation, grapheme segmentation, case folding) should operate on structures that represent code points or grapheme clusters explicitly. Encoding choices influence cost, but algorithms must be agnostic to the underlying storage once decoded.

- Normalize text to a canonical form (NFC/NFD/NFKC/NFKD) before comparison or collation.
- Convert to a sequence of Unicode scalar values (UTF-32 or a code-point iterator) to apply Unicode property tables.
- Perform grapheme cluster segmentation for cursor movement, deletion, and UI metrics.

Guiding principle Algorithmic correctness depends on Unicode semantics, not on the storage encoding. Decode once into a semantic representation, and build indices or iterators as needed.

4.5.6 Presentation layer: user-perceived text

User-interfaces operate at the level of **grapheme clusters**, not code units or code points. For example:

- “Backspace” should remove one grapheme cluster.
- “Character count” in form fields should count grapheme clusters.

- Cursor movement should respect grapheme cluster boundaries.

Consequences Implement grapheme cluster iteration and mapping to storage offsets for UI features. Storage encoding (UTF-8/UTF-16/UTF-32) is irrelevant to user semantics once grapheme boundaries are identified.

4.5.7 General encoding selection guidelines

Guideline 1: Text at system boundaries should be UTF-8

This includes:

- network communication,
- file I/O,
- logs and diagnostics,
- interprocess communication.

UTF-8's compactness and ubiquity make it the most interoperable choice.

Guideline 2: Internal representation should match workload

- Use UTF-8 for compact storage and streaming passes.
- Use UTF-16 when targeting UTF-16 surface APIs to avoid frequent conversion overhead.
- Use UTF-32 when code-point semantics are central and memory cost is acceptable.

Guideline 3: Decode when semantics matter

Treat storage encodings as containers of code units. For operations that require logical characters:

- decode to Unicode scalar values,
- perform normalization and case folding,
- segment into grapheme clusters,
- map back to storage offsets for output or slicing.

4.5.8 Design patterns for real systems

Pattern: boundary-centric encoding

- Input/output: UTF-8
- API interop: UTF-16 (where required)
- Internal storage: UTF-8
- Internal algorithms: decode on demand

Pattern: hybrid representation

- UTF-8 for persistent storage and interchange
- UTF-32 for internal algorithmic passes with heavy Unicode semantics
- Cached indices mapping code-point or grapheme boundaries to storage offsets

Pattern: immutable core + view adapters

Store text immutably in a compact encoding and build zero-copy "views" for algorithmic access:

- `std::u8string` or `std::string` for UTF-8 storage
- custom code-point/grapheme iterators as view adapters
- avoided copies except at well-defined conversion points

4.5.9 Trade-off analysis

Choosing encodings involves balancing:

- **Memory footprint** vs **random access cost**
- **Cache friendliness** vs **decoding complexity**
- **Interchange simplicity** vs **platform API requirements**
- **Internal algorithm simplicity** vs **memory overhead**

4.5.10 Checklist for encoding decisions

Ask:

- Where does the text originate and where will it go?
- Are platform APIs requiring specific encodings?
- Are random code-point accesses common?
- Are user-perceived character operations required?

- Is memory footprint critical?
- Is binary compatibility with legacy systems required?

4.5.11 Summary

There is no single “perfect” encoding for all text processing concerns. Modern C++ systems benefit from:

- using UTF-8 for external boundaries and compact storage,
- using UTF-16 when interoperating with platform APIs that mandate it,
- using UTF-32 as an internal code-point representation when performance and algorithmic simplicity outweigh memory cost,
- decoding to semantic representations (**code points** or **grapheme clusters**) when text meaning matters,
- explicitly documenting encoding contracts at every boundary,
- separating concerns across layers to avoid anti-patterns.

Choosing the right encoding for the right layer is essential for correct, maintainable, and performant Unicode text handling in modern C++20–C++26 applications.

Chapter 5

Unicode Support from C++20 to C++26

5.1 The Introduction of `char8_t` in C++20

5.1.1 Motivation for `char8_t`

Before C++20, UTF-8 text in C++ was commonly stored in `std::string` or `std::basic_string<char>`, but the type `char` was ambiguous: it could represent raw bytes, ASCII text, or UTF-8 code units by convention rather than by language rule. This ambiguity made APIs less expressive, invited bugs, and prevented clear overload resolution based on text encoding.

The introduction of `char8_t` in C++20 provided an **encoding-level type** to represent a UTF-8 code unit explicitly. This change aligns C++ with Unicode's definition of UTF-8 as a sequence of well-defined 8-bit code units, and it improved type safety, documentation, and API design.

5.1.2 Definition and core semantics

`char8_t` is a distinct fundamental type introduced in C++20 to denote an 8-bit code unit for UTF-8 encoded text. Unlike `char`, `char8_t` cannot be implicitly mixed with other character types without explicit conversion.
Key semantic points:

- It has the same underlying width as `unsigned char` (at least 8 bits) but is a separate type at the language level.
- It is intended exclusively for UTF-8 code units; it signals to the reader and compiler that the data represents UTF-8 text.
- It improves overload resolution, because signatures taking `std::u8string_view` or `char8_t*` cannot be confused with `std::string_view` or `char*`.
- It clarifies intent in generic code and template specialization involving text and binary data.

5.1.3 UTF-8 string types and literal support

C++20 introduced literal and container support aligned with `char8_t`:

- The prefix `u8""` denotes a UTF-8 encoded string literal with element type `char8_t`.
- `std::u8string` is an alias for `std::basic_string<char8_t>`.
- `std::u8string_view` is an alias for `std::basic_string_view<char8_t>`.

Example of a UTF-8 literal and string:

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

std::u8string_view greet = u8"Hello UTF-8 world!";
```

Using `char8_t` and its related string types makes encoding explicit in the type system and prevents accidental misuse of text APIs that assume non-UTF-8 encodings.

5.1.4 Type safety and API clarity

`char8_t` improves API clarity in several ways:

- Functions can overload on `std::string_view` (bytes) vs `std::u8string_view` (UTF-8 text), explicitly distinguishing binary data from encoded text.
- Templates and concepts can constrain on text encoding more reliably, enabling better compile-time diagnostics.
- Code reviewers and static analysis tools can reason about encoding based on types rather than documentation or conventions.

Example of improved overload resolution:

```
void process_bytes(std::string_view);
void process_text(std::u8string_view);

process_bytes("raw data");   // calls bytes
process_text(u8"text");      // calls text
```

Without `char8_t`, the same API would need tags, dummy parameters, or naming conventions to communicate encoding.

5.1.5 Interaction with existing APIs and libraries

Because `char8_t` is a distinct type, existing APIs that expect `char*` or `std::string` cannot be passed `char8_t*` without cast or conversion. This both forces clarity and requires migration effort:

- Legacy libraries may require explicit conversion from `char8_t` to `char` when UTF-8 is the desired encoding.
- Modern libraries should provide overloads or templates that accept `std::u8string_view` to support UTF-8 first-class.
- Code generators, serializers, and network protocols that use UTF-8 naturally express intent via `char8_t` without hidden assumptions.

Conversion example:

```
std::string to_string(std::u8string_view u8) {
    return std::string(reinterpret_cast<const char*>(u8.data()), u8.size());
}
```

The explicit cast makes the encoding contract visible rather than implicit.

5.1.6 Impact on generic code and templates

`char8_t` enables more precise template constraints. Prior to C++20, generic text functions often accepted `std::basic_string<T>` where `T == char` might accidentally receive non-UTF-8 data or binary protocols.
With `char8_t`, templates can distinguish between:

- binary or generic byte buffers (`char` or `unsigned char`),
- UTF-8 text (`char8_t`),
- UTF-16 code units (`char16_t`),
- UTF-32 code units (`char32_t`).

Example constraint:

```cpp
template <typename T>
concept Utf8Text = std::same_as<T, char8_t>;

void handle_utf8(std::u8string_view text) requires Utf8Text<decltype(text)::value_type>;
```

This level of specificity was not possible with `char` alone.

5.1.7 Limitations: what `char8_t` does not solve

While `char8_t` makes encoding explicit, it does not by itself provide:

- Unicode normalization (NFC, NFD, etc.),
- grapheme cluster segmentation,
- case folding or language-aware collation,
- bidirectional text layout,
- display width or shaping.

These remain higher-level concerns that require dedicated libraries or explicit implementations. `char8_t` primarily improves API clarity and type safety at the encoding boundary.

5.1.8 Best practices with `char8_t` in modern C++

- Use `std::u8string_view` for functions that consume UTF-8 text without owning storage.
- Use `std::u8string` when ownership of UTF-8 text is required.
- Keep conversions between UTF-8 and other encodings explicit and isolated at boundaries.

- Avoid mixing `char8_t` and `char` without documenting and enforcing encoding contracts.
- Use encoding-aware utility libraries for tasks beyond code-unit handling (e.g., normalization, segmentation).

5.1.9 Evolution beyond C++20: C++23 and C++26 clarifications

While C++20 introduced `char8_t` and UTF-8 literal support, subsequent work in C++23 and C++26 clarified:

- interaction of `char8_t` with string conversions and views,
- stronger overload resolution in generic contexts,
- improved consistency in standard library facilities around basic string types of various character widths.

These clarifications reduce footguns when moving between legacy code and modern Unicode-aware APIs.

5.1.10 Summary

The introduction of `char8_t` in C++20 represents a major step in expressing Unicode encoding intent in the language. By providing a distinct, unambiguous type for UTF-8 code units, C++ enables clearer APIs, safer overloads, and better documentation of encoding contracts. However, encoding clarity is only the foundation: full Unicode support requires explicit decoding, segmentation, normalization, and higher-level algorithms. Using `char8_t` correctly positions C++ code to interoperate with modern Unicode ecosystems while avoiding common pitfalls in legacy text handling.

5.2 Clarifying UTF-8 Intent in the Type System

5.2.1 Motivation: encoding ambiguity in legacy C++

Prior to C++20, the C++ type system lacked a dedicated type for UTF-8 code units. Text encoded as UTF-8 was stored in `std::string` or `std::basic_string<char>`, but the type `char` could just as easily hold binary data, ASCII, legacy code pages, or other byte-level protocols. This ambiguity undermined type-based reasoning:

- function signatures could not distinguish between binary buffers and encoded text,
- APIs that processed text could not enforce encoding contracts at compile time,
- accidental misuse of APIs treating `std::string` as text was common.

The introduction of `char8_t` and related types in C++20 addressed this gap by elevating encoding intent into the type system.

5.2.2 What it means to encode intent in types

Encoding intent in the type system means that the type itself carries information about how the stored values should be interpreted. In the context of Unicode:

- `char` and `std::string` represent *uninterpreted sequences of bytes* by default.
- `char8_t` and `std::u8string` represent *UTF-8 code units*.
- `char16_t`, `char32_t` and their string types represent specific Unicode encoding forms.

By making encoding part of the type, interfaces can document and enforce encoding contracts without relying solely on comments or documentation.

5.2.3 The role of `char8_t` in clarity

C++20 introduced `char8_t` as a distinct fundamental type precisely for UTF-8 code units. This change has several practical benefits:

1. Distinguishing text from binary data `char8_t` signals that a sequence of bytes is intended to represent UTF-8 text. This semantic information is absent if the text is stored in `char`. For example:

```
void process_bytes(std::string_view);      // arbitrary bytes
void process_utf8(std::u8string_view);     // UTF-8 encoded text
```

The two overloads cannot be confused by the compiler, and calls are resolved based on intent rather than convention.

2. Reducing accidental misuse Code that mistakenly treats binary data as text often compiles without errors when all data is stored in `char`. With `char8_t`, misuse often leads to type mismatches or the need for explicit casts, alerting the programmer to the encoding considerations.

3. Improving API expressiveness APIs can now communicate encoding expectations directly through types rather than through naming conventions or documentation. A function taking `std::u8string_view` makes the UTF-8 requirement explicit in its signature.

5.2.4 Literal support and type safety

String literals in C++ have explicit prefixes that yield specific character types:

- `u8"…"` produces an array of `char8_t`.
- `"…"` produces an array of `char` (narrow literal).

- `u"…"` produces an array of `char16_t` (UTF-16 code units).
- `U"…"` produces an array of `char32_t` (UTF-32 code units).

Using `u8""` ensures that the literal's type is unambiguously UTF-8. Prior to C++20, `"…"` in UTF-8 source would produce `char[N]` with implementation-dependent interpretation.

```
auto text_utf8 = u8"Example UTF-8 literal";   // char8_t[N]
std::u8string s = u8"UTF-8 text";             // UTF-8 encoded string
```

5.2.5 Views and range concepts

C++20 also standardizes `std::u8string_view`, enabling non-owning views over UTF-8 sequences. This improves performance and reduces copying when an API does not need ownership of the data.
A typical pattern:

```
void analyze_utf8(std::u8string_view text) {
    // analyze without copying
}

int main() {
    std::u8string data = u8"Unicode data";
    analyze_utf8(data);
}
```

`std::u8string_view` clearly communicates:

- text is UTF-8 encoded,
- ownership is not assumed,
- operations should be encoding-aware.

5.2.6 Interactions with template functions

Encoding-aware types improve generic programming. Templates operating on ranges of characters can enforce encoding constraints with concepts. For example:

```
template <typename CharT>
concept Utf8Character = std::same_as<CharT, char8_t>;

template <Utf8Character CharT>
void process_text(std::basic_string_view<CharT> text) {
    // process text, knowing it's UTF-8
}
```

Without `char8_t`, such constraints would be weaker or rely on tagging and documentation.

5.2.7 Conversion boundaries remain explicit

Although `char8_t` clarifies intent at the type level, conversion between encodings remains explicit. For example, converting from a UTF-8 sequence to UTF-16 (for platform APIs) or to code points for Unicode algorithms is still a deliberate operation:

```
std::wstring utf8_to_utf16(std::u8string_view utf8_text);
std::u32string utf8_to_utf32(std::u8string_view utf8_text);
```

The type system ensures that the programmer is conscious of encoding transitions, reducing accidental misuse.

5.2.8 Limitations of type-level clarity

While `char8_t` improves clarity, it does not automatically provide:

- validation of UTF-8 sequences,
- semantic iteration by code points or grapheme clusters,

- normalization,
- collation or case folding.

These concerns are orthogonal to encoding representation and require algorithmic support beyond the type system. Types communicate intent; algorithms deliver semantics.

5.2.9 Interactions with legacy code and gradual migration

Existing codebases frequently use `std::string` for text storage. Migrating to `char8_t` involves:

- auditing usage of `std::string` to determine where text is UTF-8 vs other binary data,
- updating APIs to accept `std::u8string_view` where appropriate,
- converting between `char*`/`std::string` and `char8_t*`/`std::u8string` explicitly at boundaries,
- ensuring that formatting and I/O code understands the distinction.

The clear encoding contract provided by `char8_t` helps guide such migrations and prevents regressions.

5.2.10 Example: encoding-aware API design

Consider an API layer for processing text:

```
struct TextStats {
    std::size_t code_points;
    std::size_t grapheme_clusters;
};

TextStats analyze(std::u8string_view utf8_text);
```

This signature communicates:

- the function expects UTF-8 encoded text,
- it does not take ownership,
- it produces semantic metrics (code points, graphemes), not raw bytes.

Contrast with:

```
TextStats analyze(std::string_view bytes);
```

The latter could be anything: binary data, ASCII, or UTF-8; the intent is unclear.

5.2.11 Best practices for clarifying UTF-8 intent

- Use `char8_t` and `std::u8string` for UTF-8 encoded text storage whenever UTF-8 is the canonical encoding.
- Use `std::u8string_view` for non-owning parameters that accept UTF-8 text.
- Distinguish between binary data and text in APIs by choosing appropriate character types.
- Avoid ambiguous overloads that mix `char` and `char8_t` without clear documentation and conversion paths.
- Document encoding expectations at every module boundary.

5.2.12 Summary

Clarifying UTF-8 intent in the type system is a key advancement in Unicode support in modern C++. By making encoding explicit through `char8_t`, UTF-8 string types, and literal support, C++20 enables APIs to express and enforce encoding contracts. This improves safety,

documentation, and tooling support. However, type clarity is only the first step: semantic text algorithms, normalization, and user-perceived operations require dedicated logic beyond the type system. The combination of clear encoding types and well-structured algorithms forms the foundation for correct and maintainable Unicode text handling from C++20 to C++26.

5.3 Incremental Improvements in C++23

5.3.1 Context: C++20 established the baseline, C++23 tightened the edges

C++20 made the most visible, type-system-level step for UTF-8 by introducing `char8_t` and aligning `u8""` literals and `std::u8string` with that type. C++23 did *not* introduce a comprehensive Unicode text-processing library (normalization, grapheme clustering, collation, bidi, etc.). Instead, the C++23 cycle delivered incremental improvements in three practical dimensions:

1. **Compatibility remediation around** `char8_t`: reducing friction when migrating existing UTF-8 code written against `char`.

2. **Library wording and behavior clarifications** in areas where UTF-8, UTF-16, and UTF-32 types interact with standard facilities.

3. **Better guidance and patterns** for boundary conversions and type-based intent, especially for UTF-8-centric architectures that must still interoperate with legacy APIs.

This section focuses on what materially changed for everyday Unicode-aware C++ programming in a C++23 codebase, and what remained intentionally out of scope.

5.3.2 C++23 and `char8_t`: compatibility and portability tightening

The practical problem C++23 addressed

C++20 intentionally made `char8_t` a *distinct fundamental type* to express UTF-8 code-unit intent and improve overload resolution. The downside was immediate migration pain:

- Existing APIs accepted `char*` / `std::string_view` for UTF-8 text by convention.

- C++20 introduced a new UTF-8 type, so `u8"..."`, `std::u8string`, and `std::u8string_view` stopped being "drop-in" for those APIs.
- Projects were forced into scattered `reinterpret_casts`, ad-hoc bridging helpers, or dual overload sets.

During the C++23 timeframe, the standardization process treated portions of this friction as a defect in the deployability story of `char8_t` and pursued compatibility remediation. The result is not "`char8_t` goes away"; it is that the ecosystem guidance and standard wording matured to support portable migration patterns without undermining the type-safety goal.

What you should do in a C++23 codebase

Rule 1: keep `char8_t` for type intent, but bridge at boundaries In modern code:

- Use `std::u8string_view` in APIs that require UTF-8 text.
- Use `std::string_view` in APIs that accept arbitrary bytes.
- Bridge explicitly when interacting with "legacy UTF-8 via `char`" APIs.

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

using Bytes = std::string_view;
using Utf8  = std::u8string_view;

void consume_bytes(Bytes);
void consume_utf8(Utf8);

// Explicit bridge: UTF-8 code units to bytes.
// This is a byte-level view for interop; it does not validate UTF-8.
inline Bytes as_bytes(Utf8 t) {
    return Bytes(reinterpret_cast<const char*>(t.data()), t.size());
}
```

Rule 2: confine casts to a small, audited interop layer A C++23 project should not sprinkle casts throughout business logic. Centralize them:

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

struct Utf8Interop {
    static std::string_view to_char_view(std::u8string_view s) {
        return { reinterpret_cast<const char*>(s.data()), s.size() };
    }
    static std::string_view to_char_view(std::u8string const& s) {
        return { reinterpret_cast<const char*>(s.data()), s.size() };
    }
};
```

This keeps the encoding contract explicit and reviewable.

5.3.3 C++23 library reality: still code units, not Unicode semantics

C++23 improved many library components, but in Unicode terms the key reminder remains:

- `std::u8string`, `std::u16string`, `std::u32string` are containers of code units.
- Standard string algorithms (`find`, `substr`, iterators, indexing) operate on code units.
- Unicode-aware operations (normalization, grapheme segmentation, collation, case folding) are not provided as standard algorithms in C++23.

C++23 improvements therefore primarily manifest as:

- fewer surprises in how the *types* and *overloads* interact,
- clearer separation between ”bytes” and ”UTF-8 code units”,
- more disciplined patterns around boundary conversions.

5.3.4 Formatting and output: what C++23 did and did not unlock

C++23 includes standardized formatting and printing facilities, but their character-type coverage is conservative:

- The formatting/printing facilities are designed primarily around `char` and `wchar_t`.
- `char8_t`, `char16_t`, and `char32_t` do *not* automatically become first-class formatting domains in C++23.

Engineering implication Even with `char8_t` for UTF-8 correctness, you often still need:

- an explicit interop view as `std::string_view` for output APIs that consume `char`,
- or a conversion to a platform-appropriate wide form when using wide output channels.

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

inline std::string_view as_char_view(std::u8string_view s) {
    return { reinterpret_cast<const char*>(s.data()), s.size() };
}

int main() {
    std::u8string_view u8 = u8"UTF-8 text";
    std::cout << as_char_view(u8) << "\n"; // bytes sent to narrow stream
}
```

This is intentionally explicit: the standard library does not silently ”upgrade” narrow streams into Unicode-aware renderers.

5.3.5 Filesystem interop: continued emphasis on explicitness

The real-world problem

Filesystem APIs are an interop boundary where encodings differ by platform:

- Windows native APIs are UTF-16 oriented.
- Many Unix-like environments treat path byte sequences as UTF-8 by convention, but the OS-level contract can be ”bytes” rather than ”Unicode text”.

C++23 does not change platform reality. Instead, it reinforces that filesystem paths are a *platform boundary type* and you must design your text layer with explicit conversions.

Recommended C++23 design pattern

- Keep your application's *text layer* in UTF-8 (`std::u8string` / `std::u8string_view`) when feasible.
- Convert to platform path representations at the filesystem boundary.
- Do not treat ”path string” extraction as a universal ”valid UTF-8” guarantee on all platforms and all inputs; treat it as boundary data.

5.3.6 Incremental C++23 improvements you can actually feel in code reviews

Even without a full Unicode library, C++23 improves the day-to-day discipline of Unicode handling by making certain mistakes more visible:

1) Stronger overload separation between bytes and UTF-8 text

A C++23 codebase that uses `std::u8string_view` can expose encoding contracts at compile time:

```cpp
#include <string_view>

void parse_protocol_bytes(std::string_view);   // protocol bytes
void parse_user_text(std::u8string_view);      // UTF-8 text

// parse_user_text("...")   // should not compile: wrong type, wrong intent
```

2) Less ”accidental ASCII” in interfaces

When `u8""` yields `char8_t`, passing UTF-8 literals into ”ASCII-by-default” APIs becomes a conscious choice, not an accident. This improves correctness and makes encoding assumptions explicit during review.

3) Better boundary hygiene becomes the norm

C++23-era practice converges on:

- validate and decode at system boundaries,
- store UTF-8 internally (often),
- convert only at specific boundary layers (filesystem, OS APIs, UI toolkits),
- treat semantic operations as Unicode-algorithm work over decoded scalar values or grapheme clusters.

5.3.7 What C++23 still does not provide

To prevent overclaiming, the following remains out of scope in C++23 standard library:

- Unicode normalization (NFC/NFD/NFKC/NFKD).
- Grapheme cluster segmentation (user-perceived characters).

- Unicode collation and locale-correct sorting.
- Unicode case folding and full case mapping.
- Bidirectional algorithm and shaping/layout.

Therefore, ”Unicode support” in C++23 should be described accurately as:

> Improved *type-level intent* and *interop discipline* around UTF encodings, not a complete Unicode text-processing solution.

5.3.8 Practical checklist for C++23 projects

1. Use `std::u8string_view` in APIs that require UTF-8.
2. Use `std::string_view` for bytes; never call it ”text” unless you enforce UTF-8 invariants.
3. Centralize `char8_t` $\leftrightarrow$ `char` bridging in one interop header.
4. Validate UTF-8 at boundaries; pick a strict vs replacement policy and document it.
5. For Windows API calls, convert UTF-8 $\rightarrow$ UTF-16 at the boundary; do not store UTF-16 everywhere just because the OS does.
6. Do not implement UI semantics in code units; use grapheme segmentation where user-visible behavior matters.

5.3.9 Summary

C++23 delivered incremental but meaningful Unicode-related improvements by maturing the `char8_t` ecosystem: it reinforced UTF-8 intent in types, improved portability and deployability expectations, and pushed best practice toward explicit boundary conversions and clearer API

contracts. At the same time, C++23 deliberately did not become a full Unicode text-processing standard library. Correct Unicode behavior still requires explicit validation, decoding, and higher-level Unicode algorithms beyond what the core language and standard containers provide.

5.4 Expected Direction of Text Facilities in C++26

5.4.1 Overview of C++26 development context

The C++26 standard is the next major revision of ISO/IEC 14882 following C++23, currently under development and nearing final approval as of early 2026. While the core language and standard library will receive a broad range of improvements beyond text encoding, Unicode support remains an active area of standardization interest. The fundamental design direction for Unicode and text facilities in C++26 continues the trajectory started in C++20 and refined in C++23: improving expressiveness, type clarity, and library correctness while leaving comprehensive Unicode semantics (normalization, segmentation, collation) to dedicated libraries. :contentReference[oaicite:0]index=0

5.4.2 C++26 goals for text facilities

Where C++20 introduced `char8_t` and C++23 improved interoperability and encoding intent, C++26 is expected to solidify the encoding support foundation and expand text-related utilities in the standard library. These directions include:

- **Removal of deprecated codecvt facets and antiquated Unicode conversion utilities** in favor of clearer, better-specified facilities. Proposals under review target elimination of legacy localecodecvt machinery by making it obsolete or removing it entirely in C++26, while preserving vendor ability to support them outside the core standard. :contentReference[oaicite:1]index=1

- **Stronger, clearer library wording and type relationships** to ensure UTF-8, UTF-16, and UTF-32 string types integrate consistently with I/O, views, and conversion APIs. Clarification around string literal encoding phases, execution character set interpretation, and literal validation continues to be refined. :contentReference[oaicite:2]index=2

- **Continued emphasis on explicit conversions at boundaries** between encodings (UTF-8, UTF-16, UTF-32) and between code-unit sequences and semantic text representations. The standard will likely encourage explicit conversion patterns rather than implicit or ad-hoc interpretations. This preserves the invariant that storage encodings are separate from semantic content models.
- **Better integration of converters and locale awareness** in the core I/O and formatting libraries, improving unified handling of text encodings where formatters can handle different string types with consistent semantics at the point of output. While full Unicode semantics (e.g., collation) is not standardized, better library contracts minimize surprises in cross-platform I/O.
- **Type-safe interactions across character widths**, reducing accidental narrowing or misuse of wide, UTF-16, or UTF-32 text in generic contexts, and clarifying overload resolution rules when multiple character types participate.

5.4.3 Codecvt removal and modernization

The deprecated `std::codecvt` facets, historically part of `<locale>`, have been identified as poorly specified and ineffective for robust Unicode conversion. A standing proposal for C++26 recommends removing these deprecated facilities entirely from the core standard, relying instead on modern conversion utilities and dedicated libraries. This change simplifies the standard library and signals a step toward modern Unicode handling practices. :contentReference[oaicite:3]index=3

5.4.4 Literal encoding and source character set evolution

The C++23 requirement that compilers support UTF-8 source files lays the groundwork for C++26 to further refine how literal encodings are canonicalized during translation phases. Rather than treat literal interpretation as implementation-defined in many contexts, future

wording may tighten expectations so that text literal encodings are consistently mapped from source to execution character set and reflect Unicode scalar values more predictably. Additionally, universal character names (`\uXXXX`, `\UXXXXXXXX`) and named character escapes express Unicode scalar values directly. C++26 work may clarify and formalize edge cases around these forms to reduce ambiguity in practice.

5.4.5 I/O and formatting library enhancements

While C++20 and C++23 extended formatting capabilities (e.g., `std::format`, `std::println`), they do not intrinsically provide full Unicode-aware semantics such as normalization or grapheme clustering during printing. C++26 discussions are likely to focus on:

- Ensuring `std::format` and related utilities support seamless use with the various string types (`std::u8string`, `std::u16string`, `std::u32string`) by defaulting to correct code-unit formatting behavior without requiring ad-hoc casts.
- Clarifying how formatting widths, alignment, and fill characters behave when provided with Unicode text in variable-length encodings.
- Harmonizing formatting semantics across character widths, reducing corner cases when mixing UTF-8 with legacy narrow streams or UTF-16 on platforms like Windows.

Progress in this direction helps developers avoid common pitfalls when outputting text in different encoding contexts.

5.4.6 Refining type safety and generic text API design

C++26 is expected to continue elevating type safety regarding Unicode encoding intent. For example:

- More precise overload sets and concepts that differentiate text interpretation (*UTF-8 vs arbitrary bytes*) to prevent silent misinterpretation.
- Generic algorithms or traits that permit encoding-agnostic traversal when only code units matter, and encoding-aware traversal when semantic processing is required.
- Better support for encoding propagation in template code, allowing algorithms that accept UTF-8 text to participate in overload resolution cleanly while rejecting unintended types.

This direction is consistent with the design principle that encoding should be explicit and interfaces should document and enforce expectations at compile time.

5.4.7 Remaining gaps and the role of external libraries

Despite incremental improvements in the core language and library, C++26 is not expected to provide full Unicode semantics in the standard library. Specifically, *higher-level Unicode algorithms* such as:

- **Normalization (NFC, NFD, NFKC, NFKD)**,
- **Grapheme cluster segmentation**,
- **Locale-aware collation and case folding**,
- **Bidirectional text layout**,
- **Script or language-sensitive text segmentation**,

will remain outside the core standard and are recommended to be supplied by dedicated libraries such as ICU (International Components for Unicode) or modern Unicode utility libraries. C++26's direction is to ensure the foundation (encoding representation, type clarity, I/O interoperability, and conversion primitives) is solid so that such higher-level libraries can interoperate cleanly with standard types and facilities. :contentReference[oaicite:4]index=4

5.4.8 Guiding principles for C++26 text facility expectations

The expected design ethos for C++26 text facilities includes:

- **Explicit encoding contracts**: Types and APIs should communicate and enforce the encoding domain they operate on.
- **Library clarity and modernization**: Deprecated, poorly specified components are eliminated or replaced; new utilities are clearly specified for Unicode-relevant use cases.
- **Interop awareness**: Encoding conversions at system boundaries (e.g., OS APIs, file systems) are made explicit and ergonomic, reducing accidental misuse.
- **Foundation, not semantics**: The standard library provides building blocks for encoding representation, I/O, and conversion, while semantic layers are provided by external libraries.

5.4.9 Summary

C++26's Unicode and text facility evolution builds on C++20 and C++23 by tightening foundational support, modernizing and removing legacy components, clarify encoding intent in types and APIs, and improving interoperability. True higher-level Unicode semantics remain outside the standard library, but the mechanisms that interact with Unicode text—including string types, boundary conversions, and formatting integration—are expected to become more reliable and expressive. By focusing on explicit encoding contracts and clearer library behavior, C++26 aims to make correct text handling easier to integrate into modern C++ applications while avoiding ambiguity and legacy traps.

5.5 Why the Standard Does Not Provide a Unified Unicode String Type

5.5.1 The question developers keep asking

Many developers reasonably expect a single ”Unicode string” type that:

- stores text portably across platforms,
- supports correct length, indexing, slicing, searching,
- handles normalization and case folding,
- supports grapheme clusters and user-perceived characters,
- interoperates smoothly with OS APIs and existing C++ code.

C++ deliberately does not provide such a unified type today. Instead, the standard offers multiple *code-unit* string types (`std::string`, `std::u8string`, `std::u16string`, `std::u32string`, `std::wstring`) and leaves most *Unicode semantics* to external libraries or application-defined layers.
This is not an oversight. It is the result of deep technical, compatibility, and specification constraints.

5.5.2 The core reason: ”text” has multiple incompatible meanings

The root problem is that *”character” is not a single unit in Unicode*. In real Unicode text, you must distinguish:

- **Code units**: storage elements of an encoding (bytes for UTF-8, 16-bit units for UTF-16, 32-bit units for UTF-32).

- **Code points (scalar values)**: abstract values in `U+0000..U+10FFFF` excluding surrogates.
- **Grapheme clusters**: user-perceived characters (what cursor movement and backspace typically operate on).
- **Rendered glyphs**: what a font actually draws, shaped by script, context, and layout engines.

A single ”Unicode string” type would have to pick one of these as the primary indexing and slicing unit. Any choice breaks common expectations:

- If indexing is by **code unit**, it is fast but not ”characters”.
- If indexing is by **code point**, it is not what users perceive as characters.
- If indexing is by **grapheme cluster**, it is expensive, stateful, and locale/script sensitive.

5.5.3 Unicode correctness requires large, evolving data and algorithms

Correct Unicode behavior is not just ”store UTF-8”. It requires:

- large property tables (general categories, scripts, combining classes, case mappings, folding data),
- normalization (NFC/NFD/NFKC/NFKD),
- grapheme cluster segmentation,
- collation (locale-aware sorting),
- bidirectional algorithm and text shaping in UI contexts.

These are not stable forever. Unicode data evolves with each Unicode version, and conformance requires matching data and rules. Standardizing a single unified type would implicitly commit the C++ standard library to:

- ship large and frequently updated data tables,
- define long-term ABI and behavior guarantees across versions,
- specify complex algorithms precisely (including many edge cases),
- coordinate with platform libraries and language locales.

C++ standardization moves on a multi-year cadence and prioritizes long-term ABI stability. Unicode data evolves on a different cadence. This mismatch alone makes a fully unified Unicode string type difficult to standardize and deploy consistently.

5.5.4 Multiple encodings are valid at different boundaries

In real systems, different layers demand different encodings:

- **Interchange and storage**: UTF-8 is dominant for files, protocols, JSON, logs.
- **Windows OS APIs**: UTF-16 is the native wide-string boundary.
- **Some algorithmic pipelines**: UTF-32 is convenient for scalar-value indexing and property lookups.

A single unified type would still need to interact with these boundaries. That means either:

- the unified type must support multiple internal encodings (complexity, branching, size), or
- it must pick one encoding (then conversions become pervasive and costly), or

- it must abstract encoding away (then performance and memory layout become unpredictable).

C++ tends to prefer explicit control over representation and performance characteristics. A single abstracted ”Unicode string” that hides encoding details is at odds with this design culture.

5.5.5 Backwards compatibility and ecosystem inertia

The C++ ecosystem has decades of code using:

- `char*`, `std::string`, and byte-oriented APIs,
- platform APIs (`wchar_t*` on Windows),
- third-party libraries that interpret `std::string` as UTF-8 by convention.

Introducing a unified Unicode string type would not automatically fix old code; it would create a parallel universe:

- two sets of APIs (byte strings vs Unicode strings),
- a conversion problem at every integration point,
- pressure to add implicit conversions (which would reintroduce ambiguity and bugs).

C++20’s introduction of `char8_t` shows this tension clearly: type safety improved, but interop friction increased unless projects adopt disciplined boundary adapters.

5.5.6 What the standard strings intentionally guarantee (and what they do not)

The standard string types guarantee:

- contiguous storage,
- predictable iterator invalidation rules,
- complexity bounds for basic operations,
- value semantics and allocator support.

They intentionally *do not* guarantee:

- that indexing corresponds to user-perceived characters,
- that `size()` means ”characters”,
- that `find()` respects canonical equivalence,
- that slicing preserves grapheme clusters or normalization.

These omissions are not negligence: they avoid specifying Unicode semantics incorrectly. A unified Unicode string type would be expected to provide those semantics, which are far more complex than typical container guarantees.

5.5.7 The hardest part: defining ”length” and ”substring”

A unified Unicode string type must define what these common operations mean:

Length Options:

- code units (fast, but not characters),
- code points (still not characters),
- grapheme clusters (most user-correct, but expensive and context-dependent).

Substring/slicing A substring that splits:

- a UTF-8 multi-byte sequence,
- a UTF-16 surrogate pair,
- a combining sequence within a grapheme cluster,

can produce ill-formed text or user-visible corruption.
So a unified type would need to:

- validate boundaries,
- track segmentation state,
- potentially store auxiliary indices (increasing memory overhead),
- define whether it preserves normalization forms.

These are not ”nice-to-have” details; they are correctness requirements if the type claims to be a Unicode text type rather than a code-unit container.

5.5.8 Why the committee tends to standardize building blocks first

Given these constraints, WG21 has historically focused on incremental, foundational improvements:

- making encoding intent expressible in types (e.g., `char8_t` for UTF-8 code units),
- improving boundary conversion patterns and deprecating legacy, underspecified conversion facilities,
- clarifying library wording so that code-unit containers behave consistently and predictably.

This direction aligns with a practical standard-library philosophy:

> Standardize the universally needed, portable primitives first; leave policy-heavy, data-heavy Unicode semantics to dedicated libraries.

5.5.9 What a ”unified Unicode string” would likely have to look like

To meet user expectations, a true Unicode string type would likely need:

- explicit and validated encoding invariants,
- iterators over code points and grapheme clusters,
- normalization-aware comparison and searching,
- locale-aware collation hooks (or a well-defined non-locale default),
- stable storage plus optional indexing structures for performance,
- careful handling of ill-formed sequences at boundaries (strict vs replacement vs round-trip).

This is closer to what mature Unicode libraries provide than to what `std::basic_string` is designed to be.

5.5.10 A C++20–C++26 design pattern that works in practice

Instead of relying on a single type, modern C++ systems typically use a layered design:

1. **Boundary layer (I/O)**: store interchange text as UTF-8 (`std::u8string` or `std::string` with a strict contract).
2. **Interop layer (OS/UI)**: convert to UTF-16 on Windows boundaries; avoid scattering conversions.

3. **Semantic layer (algorithms)**: decode to scalar values (often `char32_t`) or use iterators that yield code points.

4. **User layer (editing/UI)**: operate on grapheme clusters, not code units.

```cpp
#include <string>
#include <string_view>

// 1) External boundary: UTF-8 code units
using Utf8View = std::u8string_view;

// 2) Interop boundary adapters (sketch; platform-specific implementations omitted)
std::wstring utf8_to_utf16(Utf8View); // Windows boundary
std::u32string utf8_to_utf32(Utf8View); // semantic algorithms

// 3) Semantic operations (operate on code points)
std::size_t count_code_points(std::u32string_view cps);

// 4) UI operations (operate on grapheme clusters; typically requires a Unicode library)
std::size_t count_grapheme_clusters(Utf8View);
```

This approach makes contracts explicit and keeps heavy Unicode semantics where they belong: in a Unicode-aware layer, not in a generic container type.

5.5.11 Summary

The C++ standard does not provide a unified Unicode string type because ”text” is not a single unit, and correct Unicode behavior requires large, evolving data and complex algorithms with policy choices (normalization, segmentation, collation, error handling). A one-size-fits-all type would either:

- be too weak (just another code-unit container), or

- be too heavy and policy-laden (large tables, complex semantics, difficult ABI/versioning), or
- hide representation in ways that conflict with C++'s performance and explicitness goals.

Instead, C++20–C++26 continues to strengthen the *foundation* (type intent, encoding clarity, safer boundaries) while encouraging layered designs and specialized libraries for true Unicode semantics.

5.6 Compatibility, Performance, and Design Constraints

5.6.1 Why Unicode standardization in C++ is unusually constrained

Unicode-aware text facilities are not blocked by lack of interest; they are constrained by three forces that rarely align in a single standard-library feature:

1. **Compatibility constraints**: decades of existing C and C++ interfaces, OS ABIs, file-system conventions, and third-party libraries.
2. **Performance constraints**: predictable latency, cache efficiency, and minimal overhead in tight loops and high-throughput pipelines.
3. **Design constraints**: precise specification, long-term ABI stability, and a standardization cadence that is slower than Unicode's data evolution.

C++20–C++26 therefore focuses on building durable foundations (types and contracts) and removing known-bad legacy facilities, rather than attempting a monolithic "Unicode string" solution.

5.6.2 Compatibility constraints

1) ABI reality: platforms already chose encodings

C++ must interoperate with platform APIs and established ABIs:

- On Windows, the native "wide" API surface is UTF-16 code units (16-bit `wchar_t` and `W`-suffixed functions).
- On many Unix-like systems, file paths and environment variables are byte sequences, with UTF-8 as a common convention but not always a strict invariant.

A standard-library text facility cannot pretend these boundaries do not exist. Any design must either:

- convert at boundaries (explicitly and predictably), or
- leak platform details into the core representation (destroying portability), or
- abstract away representation (risking hidden costs and surprising behavior).

2) The `char8_t` lesson: type safety vs ecosystem friction

C++20 introduced `char8_t` to express UTF-8 code-unit intent in the type system. This improves API correctness and overload resolution, but it also exposed compatibility friction:

- Existing code frequently treats `std::string` / `std::string_view` as "UTF-8 text" by convention.
- Many APIs accept `const char*` and are not immediately callable with `const char8_t*` or `std::u8string_view`.

The standardization response has been to improve migration and portability stories (including remediation guidance) while preserving the core principle: **UTF-8 intent should be explicit in types**. In practice, C++ projects should centralize interop rather than scattering casts.

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

using Utf8View  = std::u8string_view;
using BytesView = std::string_view;

// Boundary adapter: UTF-8 code units -> raw bytes view (no validation).
inline BytesView as_bytes(Utf8View u8) {
    return BytesView(reinterpret_cast<const char*>(u8.data()), u8.size());
```

```
}

// Optional: raw bytes -> UTF-8 view (only if your system guarantees bytes are valid
↪  UTF-8).
inline Utf8View as_utf8(BytesView b) {
    return Utf8View(reinterpret_cast<const char8_t*>(b.data()), b.size());
}
```

This makes the boundary explicit and auditable.

3) Backward compatibility pressures in the standard library

The C++ standard library is expected to be link-compatible across long periods in real deployments. Introducing a ”unified Unicode string” that changes fundamental assumptions about:

- iterator categories,
- `size()` meaning,
- indexing complexity,
- exception/error behavior,
- locale interactions,

would inevitably break large amounts of existing code (or force the type to behave like a code-unit container, undermining the goal).
As a result, the standard library keeps `basic_string`-family types as *code-unit containers* and pushes semantic Unicode operations into dedicated layers.

4) Legacy conversion facilities are underspecified and being removed

Historically, C++ shipped conversion machinery (notably codecvt-related components) that proved difficult to use correctly due to underspecification, weak error handling models, and inconsistent platform behavior. The modern direction through the C++26 cycle is:

- remove deprecated conversion headers/facilities,
- discourage ”locale/codecvt as Unicode” designs,
- prefer explicit, well-specified conversion at boundaries or dedicated Unicode libraries.

5.6.3 Performance constraints

1) Cache efficiency dominates many text workloads

Unicode design cannot focus only on decoding correctness; throughput often depends on memory behavior:

- UTF-8 is compact for ASCII-heavy text and often wins in streaming scans due to reduced bandwidth.
- UTF-16 is moderate in size but requires surrogate-pair-aware iteration for non-BMP characters.
- UTF-32 provides constant-time code-point indexing but doubles memory vs BMP-heavy UTF-16 and often quadruples vs ASCII-heavy UTF-8, increasing cache misses and bandwidth pressure.

If a standard type hides representation or silently upgrades storage, it risks unpredictable performance regressions across workloads.

2) The cost model of ”length” and ”indexing” is not uniform

Developers expect:

- `size()` to be $O(1)$,
- random access by index to be $O(1)$,
- slicing to be cheap.

These expectations conflict with Unicode semantics:

- grapheme clusters require segmentation rules and can span multiple code points,
- code-point indexing in UTF-8/UTF-16 is not naturally $O(1)$ without an index,
- correctness often requires normalization or canonical-equivalence-aware comparison.

A ”Unicode string” that promises user-perceived character indexing would need either:

- persistent indices (extra memory, update complexity),
- lazy computation with hidden costs,
- or relaxed guarantees (surprising behavior).

C++ generally avoids hidden complexity in fundamental types.

3) Validation and error handling must be policy-driven

Text arrives from files, networks, and user input. Not all inputs are well-formed UTF-8/UTF-16. A robust design must decide:

- **Strict**: reject ill-formed sequences.
- **Replacement**: substitute invalid subsequences (e.g., U+FFFD).

- **Round-trip**: preserve original code units even if ill-formed (useful for certain OS boundary cases).

These choices affect:

- security (rejecting malformed sequences can prevent injection-style bugs),
- correctness (replacement may change meaning),
- interop (round-trip may be required to preserve OS path bytes).

A single standard type cannot pick one policy that is always correct for all domains.

4) ”Fast path” requirements for ASCII-heavy data

Large fractions of real-world text are ASCII-heavy (source code, identifiers, JSON keys, protocol tokens). High-performance systems want:

- branch-light scanning,
- vectorized validation,
- minimal allocations and copies,
- a predictable representation.

A standard facility that optimizes only for full Unicode generality (graphemes, normalization) could impose unacceptable overhead on the common fast path. Conversely, a facility that optimizes only for ASCII breaks Unicode correctness. This tension drives layered designs.

5.6.4 Design constraints

1) Unicode versions evolve faster than the C++ standard

Unicode adds characters and updates character properties on a regular cadence. A standard-library facility that bakes in Unicode property tables faces hard problems:

- Which Unicode version is normative for the C++ standard?
- How do implementations update Unicode data without breaking ABI or changing behavior unexpectedly?
- How does a program compiled against an older C++ standard handle text produced under newer Unicode versions?

To preserve stability, C++ tends to standardize *types and interfaces* rather than mandate a constantly evolving Unicode data payload in the core library.

2) Locale and collation are policy-heavy

Sorting and case transformations are not purely technical; they are locale- and language-dependent:

- collation requires tailoring per locale (and often relies on CLDR data),
- case mapping has language-specific exceptions,
- normalization choice affects equivalence and search behavior.

If the standard library provided ”correct collation” it would need:

- a stable data source model,
- explicit locale selection semantics,

- well-defined fallback behavior,
- conformance requirements across implementations.

This is a major specification burden and a major portability risk. It is typically delegated to mature Unicode libraries.

3) The standard library favors explicit contracts over implicit magic

C++ library design generally prefers:

- explicit conversions at boundaries,
- zero-overhead abstractions when possible,
- strong types to communicate intent,
- predictable complexity guarantees.

Unicode ”magic” (automatic normalization, automatic grapheme-aware indexing) violates these principles unless carefully separated into opt-in facilities with explicit policies.

4) Conversions must be composable and testable

A portable Unicode layer in modern C++ typically needs:

- clear encoding contracts (UTF-8 code units vs bytes),
- explicit conversion points (UTF-8 $\leftrightarrow$ UTF-16 for Windows; UTF-8 $\leftrightarrow$ UTF-32 for code-point algorithms),
- validation policy (strict vs replacement vs round-trip),
- stable interfaces that can be unit tested.

A practical design uses small building blocks.

```
#include <string_view>
#include <string>
#include <optional>

enum class UtfPolicy { StrictReject, Replace, RoundTripCodeUnits };

struct Utf8Text {
    std::u8string data; // invariant depends on policy and validation at construction
};

std::optional<Utf8Text> make_utf8(std::string_view bytes, UtfPolicy policy);

std::wstring  to_windows_utf16(std::u8string_view utf8, UtfPolicy policy); // boundary
↪  conversion
std::u32string to_codepoints(std::u8string_view utf8, UtfPolicy policy);   // semantic
↪  algorithms
```

This style makes the cost and behavior explicit.

5.6.5 What C++20–C++26 actually delivers under these constraints

Given the constraints above, the realistic standard trajectory is:

- **Type-level intent**: UTF-8 code units distinguished via `char8_t`, enabling clearer APIs (`std::u8string`, `std::u8string_view`).
- **Cleaner surface area**: moving away from deprecated and underspecified conversion facilities.
- **Better interop discipline**: encouraging explicit conversions at boundaries instead of implicit locale-based transcoding.

- **Foundation over semantics**: leaving normalization, grapheme segmentation, collation, bidi, shaping to specialized libraries or higher layers.

5.6.6 Practical takeaway for system design

A robust C++20–C++26 Unicode architecture is layered:

1. **Interchange layer**: UTF-8 at boundaries (files, network).
2. **Interop layer**: UTF-16 only where mandated (Windows API boundaries).
3. **Semantic layer**: decode to code points (often UTF-32) where Unicode properties are required.
4. **User layer**: grapheme-cluster semantics for editing and UI operations.

This structure respects compatibility realities, preserves performance predictability, and avoids forcing the standard library to standardize policy-heavy Unicode semantics into a single type.

5.6.7 Summary

C++ Unicode support from C++20 to C++26 is shaped more by constraints than by ambition:

- **Compatibility** forces explicit boundaries with platform APIs and legacy ecosystems.
- **Performance** forces predictable representations and avoids hidden indexing/normalization costs in fundamental types.
- **Design** forces stable specifications and ABI expectations in a world where Unicode data and locale behavior evolve continuously.

The result is a disciplined foundation: strong encoding intent in types, explicit conversion boundaries, removal of legacy underspecified components, and an expectation that full Unicode semantics live in dedicated libraries and higher-level layers.

Chapter 6

Professional Unicode Handling in C++ Systems

6.1 Why the Standard Library Is Intentionally Limited

6.1.1 Understanding the philosophical constraints

The C++ Standard Library intentionally provides only a *foundational layer* of Unicode awareness while leaving full Unicode semantics to external libraries. This is not a design flaw; it reflects deliberate trade-offs between generality, performance, compatibility, and the long-term stability expected of a systems programming language.
This section explains the principled reasons behind this intentional limitation.

6.1.2 Constraint 1: Separation of concerns—encoding vs semantics

At its core, the C++ Standard Library distinguishes between:

- **Encoding representations**: sequences of code units (`char`, `char8_t`, `char16_t`,

`char32_t`, `wchar_t`),

- **Text semantics**: what constitutes a ”character,” how text is compared, segmented, normalized, or collated.

Standard containers and character types deal with the representation of text in memory and interoperability with platform boundaries. They do not provide higher-level Unicode semantics because those semantics depend on:

- language definitions,
- locale and cultural conventions,
- normalization forms (NFC, NFD, etc.),
- user-perceived character segmentation (graphemes),
- bidirectional and shaping rules.

Encoding representation is a well-specified, low-semantic domain that the standard library can define reliably and stably. Unicode semantics, by contrast, require large, evolving data and context-sensitive rules, which vary by language and application domain.

6.1.3 Constraint 2: Unicode data evolves faster than the standard

Unicode is a living specification that evolves with each new version. Updates include:

- new scripts and characters,
- updated property tables,
- modified collation and casing rules,
- new grapheme cluster definitions.

Shipping full Unicode semantics as part of the C++ Standard Library would require embedding large, versioned data tables and committing to a Unicode version in the standard. This creates several issues:

- the standard would need frequent updates to keep pace,
- implementers would require mechanisms to update Unicode data independently of the language standard,
- programs compiled under one standard version might behave differently when run against updated Unicode tables unless versioning is explicitly managed.

C++ avoids embedding large, frequently-changing datasets in the core library to preserve:

- ABI stability,
- predictable behavior across deployments,
- portability across compilers and platforms.

6.1.4 Constraint 3: Context and policy decisions in Unicode semantics

Unicode text processing depends on decisions that are outside the scope of a low-level systems library. Examples include:

- when to normalize text (input, storage, comparison),
- whether case folding should be language-aware (`Turkish I/i` rules vs default),
- locale or culture-sensitive collation ordering,
- user interface behaviors such as cursor movement and backspace semantics.

Each of these decisions reflects application policy and user expectations. A one-size-fits-all solution in the standard would risk:

- poor performance for domains that do not need full semantics,
- surprising behavior in contexts where defaults do not match user expectations,
- hidden costs in performance-critical code paths.

Instead, the standard provides primitives that enable libraries to implement these behaviors with deliberate policy control.

6.1.5 Constraint 4: Performance predictability and zero-overhead philosophy

C++ is widely used in systems where predictable performance is essential:

- real-time and embedded systems,
- high-frequency trading and financial systems,
- game engines and graphics,
- large-scale server and database engines.

A unified, high-level Unicode type that automatically handled normalization, segmentation, and collation would impose hidden costs:

- non-constant time operations for length and indexing,
- dynamic memory allocations in semantic processing,
- calls into data tables and branches that cannot be optimized away.

C++ prioritizes *zero-overhead abstractions*: if an abstraction is provided, it should not impose runtime cost unless used. High-level Unicode semantics inherently impose cost by definition, and C++ does not hide that cost.

6.1.6 Constraint 5: Backwards compatibility and ecosystem continuity

Decades of C and C++ code treat `std::string`, `std::wstring`, and various locale facets as text containers. Retrofitting a fully Unicode-aware string type into the standard library would risk:

- breaking existing code that assumes `size()` means code units,
- invalidating portable behavior across compilers and platforms,
- creating dual semantics for existing algorithms and containers.

Preserving backwards compatibility is a core principle of C++ evolution:

- code that compiles today should compile under future standards,
- behavior changes must be deliberate and opt-in, not implicit.

Providing an *additional* Unicode string type might be feasible, but only if its semantics are well-specified, its performance characteristics are clear, and migration paths from legacy types are defined.

6.1.7 Constraint 6: Clear, auditable boundaries between layers

Modern Unicode handling is layered:

- **Boundary layer**: text enters and exits the system (I/O, OS APIs).
- **Representation layer**: text is stored internally (UTF-8, UTF-16, UTF-32).
- **Semantic layer**: logical operations (iteration, normalization, comparison).
- **Presentation layer**: user-perceived characters (grapheme clusters, shaping).

Each layer has different performance and correctness invariants. The standard library is intentionally limited to the representation layer and boundary conversions. Semantic layers must be built on top of these primitives or provided by external libraries.

6.1.8 Examples of limitations in standard containers

Standard string types provide efficient storage and manipulation of code units:

```
std::u8string s = u8"UTF-8 text";
std::size_t n = s.size(); // number of code units, not code points or grapheme clusters
```

But they do not provide:

- `normalize(s)` as a standard facility,
- `next_grapheme(s, i)` to move by user-perceived character,
- `collate(s1, s2, locale)` for locale-aware sorting,
- built-in locale-aware case folding for Unicode.

These omissions are intentional: they keep the standard library foundational and composable.

6.1.9 The role of dedicated Unicode libraries

Professional Unicode handling in C++ systems typically leverages established libraries such as:

- ICU (International Components for Unicode),
- utf8cpp (lightweight UTF-8 utilities),
- Boost.Text (Unicode algorithms and views),
- other platform-specific or domain-specific implementations.

These libraries provide:

- normalization forms (NFC/NFD/NFKC/NFKD),
- grapheme, word, and sentence segmentation,

- locale-aware collation and case folding,
- bidirectional algorithm support,
- optimized implementations for large Unicode tables.

They are able to:

- evolve with Unicode data versions independently of language standards,
- provide policy control to applications,
- deliver richer semantics where needed without forcing all users to pay the cost.

6.1.10 C++20–C++26 standard library improvements that matter

Despite limitations, C++20–C++26 deliver meaningful advances:

- `char8_t` and explicit UTF-8 literal support,
- improved boundary and type intent clarity in APIs,
- explicit conversion patterns at external boundaries,
- deprecation or removal of legacy, underspecified conversion facets.

These improvements make the standard library a solid foundation while leaving semantic text processing where it belongs: in specialized layers or libraries.

6.1.11 Practical guidance for system designers

Given these intentional limitations:

- Use the standard library for representation and boundary encoding contracts.

- Validate and convert at system boundaries explicitly.
- Rely on dedicated Unicode libraries for normalization, segmentation, and collation.
- Document encoding contracts clearly in APIs (`std::u8string_view` vs `std::string_view`).
- Do not assume `size()` or indexing corresponds to ”characters” in user perception.

6.1.12 Summary

The C++ Standard Library is intentionally limited in Unicode semantics to preserve:

- compatibility with existing code and platforms,
- predictable performance and zero-overhead guarantees,
- a clear separation between representation and higher-level semantics,
- composability and long-term stability.

Full Unicode semantics require data and algorithms that are outside the scope of foundational containers. By focusing the standard library on explicit encoding representations, clear type contracts, and explicit conversions at boundaries, C++ provides a stable foundation upon which professional Unicode handling solutions can be built using dedicated libraries and well-structured layers.

6.2 Overview of Established Unicode Libraries

6.2.1 Why you need a Unicode library in professional C++

The C++ Standard Library provides containers and character types that model *code units*. Professional Unicode handling requires *Unicode semantics* that are not provided by `std::basic_string`:

- validation of UTF-8/UTF-16/UTF-32 input under a chosen policy,
- transcoding between encodings,
- normalization (canonical and compatibility forms),
- case mapping and case folding,
- grapheme cluster segmentation (user-perceived characters),
- word and sentence segmentation,
- collation (locale-aware sorting) and locale services,
- bidirectional processing and (sometimes) text layout integration.

Established Unicode libraries exist because these problems require large data tables, many edge-case rules, and continuous maintenance.

6.2.2 A practical taxonomy of Unicode libraries

Established libraries in C++ ecosystems usually fall into one (or more) of these categories:

Category A: Full Unicode platform (”do everything”)

These libraries aim to provide most Unicode algorithms and locale services, including normalization, collation, segmentation, calendars, time zones, transliteration, and more.

Category B: Unicode algorithms and text views (”core text semantics”)

These libraries focus on Unicode text algorithms: iterating code points safely, grapheme segmentation, normalization helpers, and composable views over encoded text.

Category C: High-performance UTF validation/transcoding (”fast code-unit work”)

These libraries focus on extremely fast UTF-8/UTF-16 validation, counting, and transcoding, often using SIMD and careful low-level engineering. They typically do *not* provide normalization or collation.

Category D: Small utilities (”minimal helpers”)

These libraries provide lightweight UTF-8 iteration/validation helpers suitable for small projects, embedded systems, or as building blocks.

6.2.3 ICU (International Components for Unicode)

What it is

ICU is the most widely used, full-featured Unicode and locale services library family in industry. It is commonly treated as the reference-grade solution when you need correctness across:

- normalization,
- collation,

- segmentation (grapheme/word/sentence),
- case mapping and folding,
- locale services and data-driven behavior.

Strengths

- Broad Unicode coverage, including advanced algorithms and locale-sensitive behavior.
- Mature API surface and long history of real-world deployment.
- Designed to track Unicode data updates and provide consistent semantics.

Trade-offs

- Larger dependency footprint than lightweight libraries.
- Requires deliberate integration choices: data packaging, initialization, and build configuration.
- API is powerful but can feel heavyweight if you only need small operations.

Where ICU is the right choice

Choose ICU when you need *semantic correctness* beyond encoding:

- user-visible text editing behavior (graphemes, word breaks),
- locale-aware sorting/search, case folding rules,
- normalization requirements for identifiers, matching, storage, or security,
- multilingual product requirements with stable, well-defined behavior.

Example: normalization workflow (conceptual)

```cpp
#include <string>
#include <vector>

// Conceptual outline only: exact ICU headers/types depend on ICU version/config.
// Goal: normalize UTF-8 input to NFC for stable comparison/storage.

std::string normalize_to_nfc_utf8(std::string_view utf8) {
    // 1) Convert UTF-8 -> ICU UnicodeString (or equivalent internal form)
    // 2) Apply NFC normalization via ICU normalizer
    // 3) Convert back to UTF-8
    // 4) Return normalized UTF-8 bytes (with a documented invariant: valid UTF-8)
    return std::string(utf8); // placeholder for the concept
}
```

6.2.4 Boost.Text (Unicode algorithms and views)

What it is

Boost.Text is a C++ library focused on Unicode text processing primitives and composable text views. The design goal is to provide C++-friendly abstractions for:

- iterating code points correctly over UTF-8/UTF-16/UTF-32,
- grapheme cluster and word break views,
- Unicode-aware transformations and algorithms (scope depends on the library component).

Strengths

- C++-idiomatic, range/view-oriented design (works naturally with modern C++).

- Helps reduce common errors: byte-indexing UTF-8, slicing into surrogate pairs, breaking combining sequences.
- Useful for building your own text layer without adopting a full locale platform.

Trade-offs

- Not a full locale services platform; collation and many locale-heavy features are typically outside its core scope.
- Still requires understanding of what it guarantees (code points vs graphemes) and where policy decisions live.

Where Boost.Text is a strong fit

- You want Unicode-correct iteration and segmentation in a modern C++ style.
- You are building a text-processing pipeline or editor-like behavior and want grapheme awareness.
- You want strong primitives without bringing in a full locale framework.

Example: code point iteration (conceptual)

```
#include <string_view>
#include <cstdint>

// Conceptual outline: iterate UTF-8 as code points using a dedicated library view.
// The purpose is to avoid treating bytes as characters.

void process_utf8_codepoints(std::string_view utf8_bytes) {
    // for (char32_t cp : utf8_as_codepoints(utf8_bytes)) { ... }
}
```

6.2.5 utf8cpp (lightweight UTF-8 utilities)

What it is

utf8cpp is a small, header-centric utility library focused on UTF-8 traversal and conversion helpers. Its typical capabilities include:

- validating UTF-8 sequences,
- iterating code points from UTF-8,
- conversions between UTF-8 and UTF-16/UTF-32 (scope varies by configuration).

Strengths

- Small footprint; easy to embed.
- Practical helpers for projects that primarily use UTF-8 and need safe iteration/validation.
- Good as a building block in a boundary-validation layer.

Trade-offs

- Typically does not provide advanced Unicode semantics (normalization, collation, graphemes).
- Performance is good for many uses but not necessarily the fastest possible for large-scale SIMD-heavy validation/transcoding.

Where it fits

- Embedded or small projects that need correctness for UTF-8 iteration.
- Input validation and decoding layers where you want minimal dependencies.

Example: validating UTF-8 then decoding

```cpp
#include <string_view>
#include <optional>
#include <u32string>

std::optional<std::u32string> validate_and_decode_utf8(std::string_view bytes) {
    // 1) validate bytes are well-formed UTF-8 (library call)
    // 2) decode to UTF-32 code points (library call)
    // 3) return code points for semantic algorithms
    return std::u32string{}; // concept scaffold
}
```

6.2.6 simdutf (high-performance UTF validation/transcoding)

What it is

simdutf is a performance-oriented library specializing in:

- fast validation of UTF-8/UTF-16,
- fast transcoding between UTF-8, UTF-16, and UTF-32,
- efficient counting/processing operations that are often bandwidth-bound.

Strengths

- Very high throughput for validation and transcoding on modern CPUs.
- Excellent fit for systems processing large volumes of text (logs, ingestion pipelines, network services).
- Focused scope makes it easier to integrate than a full semantic Unicode platform.

Trade-offs

- Not a semantic Unicode library: typically no normalization, grapheme segmentation, or collation.
- You still need policy decisions (strict vs replacement) and separate layers for higher-level semantics.

Where it fits

- boundary validation and transcoding layers,
- large-scale text ingestion and ETL pipelines,
- performance-critical services where correctness and throughput both matter.

Example: boundary validation and transcoding (conceptual)

```
#include <string_view>
#include <string>
#include <vector>

struct TranscodeResult {
    bool ok;
    std::u16string utf16;
};

TranscodeResult validate_utf8_and_to_utf16(std::string_view utf8) {
    // 1) validate utf8 (SIMD)
    // 2) transcode to UTF-16
    // 3) return result
    return {true, std::u16string{}};
}
```

6.2.7 utf8proc (compact Unicode processing for normalization-like needs)

What it is

utf8proc is often used as a compact library for Unicode processing tasks centered on UTF-8, frequently including:

- Unicode character properties,
- normalization-related transforms (depending on build/options),
- code point decoding utilities.

Strengths

- Smaller and simpler than full platforms.
- Useful for normalization-oriented needs where ICU is too heavy.

Trade-offs

- Scope is narrower than ICU; locale-heavy behavior (collation tailoring) is not the typical focus.
- Feature details depend on configuration and the specific integration path.

6.2.8 GNU libunistring (general Unicode string handling primitives)

What it is

GNU libunistring provides Unicode string manipulation primitives and utilities, commonly used in ecosystems where GNU tooling is standard. It often emphasizes:

- Unicode string representations and transformations,

- helper functions for Unicode-aware operations.

Strengths and trade-offs

- Useful in GNU-centric environments and cross-platform C/C++ stacks.
- Typically not positioned as a full locale-services platform like ICU.

6.2.9 Platform libraries: what professionals actually rely on

Even when you use a third-party Unicode library, platform text stacks matter:

Windows

Professional Windows C++ software often:

- stores text internally as UTF-8 (in modern designs),
- converts at the boundary to UTF-16 for Win32 APIs,
- uses Windows-provided shaping/rendering stacks indirectly through UI frameworks.

macOS / iOS

Applications often interoperate with native platform text stacks through Objective-C/Swift layers or bridging APIs, where Unicode semantics (segmentation, rendering) are handled by platform frameworks.

Linux / cross-platform UI stacks

Toolkits and frameworks often impose their own text model and Unicode handling (including shaping, bidi, segmentation). In cross-platform apps, you often use:

- a Unicode library for semantic transformations,
- a toolkit/rendering engine for shaping and user interaction.

6.2.10 How to choose: a decision matrix for real systems

If you need correctness for user-facing semantics

Choose a full semantic library (often ICU) when you need:

- grapheme segmentation for cursor/backspace,
- locale-aware comparisons and sorting,
- stable normalization strategy,
- consistent behavior across OS and UI layers.

If you need Unicode-correct iteration and segmentation primitives in modern C++

Choose a view/algorithm library (e.g., Boost.Text style) when you need:

- safe iteration over UTF-8/UTF-16 without writing your own decoders,
- building a C++-idiomatic text layer,
- grapheme-aware operations without adopting a full locale platform.

If you need maximum throughput for validation/transcoding

Choose a high-performance UTF engine (e.g., simdutf style) when you need:

- to validate huge volumes of text,
- to transcode quickly at boundaries,
- to keep your semantic library calls focused only on text that truly needs them.

If you need small, simple helpers

Choose a lightweight UTF-8 utility (e.g., utf8cpp style) when:

- dependency size matters more than advanced semantics,
- you mainly need correct iteration/validation for UTF-8.

6.2.11 Professional integration pattern (recommended)

A robust Unicode architecture typically layers libraries by responsibility:

1. **Boundary layer**: validate and transcode (often high-performance).
2. **Core text layer**: maintain a clear internal encoding contract (commonly UTF-8) using strong types (`std::u8string_view`) and explicit adapters.
3. **Semantic layer**: normalization, segmentation, collation (often ICU or an equivalent semantic library).
4. **UI layer**: shaping and bidi (often handled by the UI toolkit/rendering engine).

```
#include <string_view>
#include <string>
#include <optional>

enum class UtfPolicy { StrictReject, Replace };

struct Utf8Text {
    std::u8string data; // invariant: valid UTF-8 (by policy)
};

std::optional<Utf8Text> ingest_utf8(std::string_view bytes, UtfPolicy policy) {
    // validate bytes as UTF-8; reject or replace; store as char8_t
```

```
    return Utf8Text{ std::u8string{} };
}

std::u8string normalize_for_compare(std::u8string_view u8) {
    // semantic library call (e.g., ICU) to normalize to a chosen form
    return std::u8string(u8);
}
```

6.2.12 Summary

Established Unicode libraries exist because professional Unicode handling requires:

- stable, well-tested Unicode algorithms,
- large and evolving Unicode data tables,
- careful policy decisions (validation, normalization, locale behavior),
- performance engineering for real workloads.

In practice:

- use a full semantic platform (ICU-style) for correctness in user-facing and locale-aware behavior,
- use modern C++ algorithm/view libraries (Boost.Text-style) for composable iteration and segmentation primitives,
- use high-performance UTF engines (simdutf-style) for validation/transcoding throughput,
- use lightweight helpers (utf8cpp-style) when minimal footprint matters.

The professional approach is layered: pick the right library at the right layer, keep encoding contracts explicit, and isolate conversions and policies at well-defined boundaries.

6.3 ICU: Capabilities and Integration Considerations

6.3.1 What ICU is (and why it dominates in production systems)

ICU (International Components for Unicode) is a mature, widely deployed set of C/C++ libraries (ICU4C) providing Unicode and internationalization services. In professional C++ systems, ICU is often the default choice when you need *semantic* Unicode processing rather than only byte/code-unit storage. ICU is ”data driven”: its behavior is powered by large Unicode and locale datasets (Unicode properties, normalization tables, collation rules, CLDR-derived locale data, time zone data). This data-driven architecture is the primary reason ICU can deliver correct behavior across scripts, languages, and locales, and also the primary reason it is heavier than small UTF-only helper libraries.

6.3.2 ICU’s text model: UTF-16 core with UTF-8 friendly boundaries

The core representation

In ICU4C, many C++ APIs revolve around `icu::UnicodeString`, which stores text as UTF-16 code units (`UChar`, a 16-bit type). This is a pragmatic choice: it matches long-standing industry practice and integrates naturally with many platform stacks, especially Windows and Java-derived ecosystems.

Boundary conversion with UTF-8

Even though the internal C++ string type is UTF-16, ICU supports straightforward UTF-8 interop at system boundaries. Professional systems commonly:

- store interchange text as UTF-8 in application layers,
- convert UTF-8 $\rightarrow$ ICU UTF-16 only when semantic operations are required,

- convert back to UTF-8 for storage, logging, network, or file formats.

```
#include <unicode/unistr.h>      // icu::UnicodeString
#include <unicode/stringpiece.h> // icu::StringPiece
#include <string>

static icu::UnicodeString to_icu_u16(std::string_view utf8) {
    icu::UnicodeString u;
    u = icu::UnicodeString::fromUTF8(icu::StringPiece(utf8.data(),
                                                      static_cast<int32_t>(utf8.size())));
    return u;
}

static std::string to_utf8(const icu::UnicodeString& u16) {
    std::string out;
    u16.toUTF8String(out);
    return out;
}
```

This conversion is explicit and policy-aware (error handling exists in lower-level APIs), which is a key property for professional correctness.

6.3.3 Core Unicode capabilities you typically rely on

ICU is broad. In C++ system design, it helps to group ICU features by the kinds of correctness they provide.

1) Normalization (canonical and compatibility forms)

Normalization is essential for robust comparison, searching, security-sensitive identifier handling, and storage invariants. ICU provides normalization via dedicated APIs (commonly `Normalizer2`) enabling NFC/NFD/NFKC/NFKD workflows and ”quick check” optimizations.

```
#include <unicode/unistr.h>
#include <unicode/normalizer2.h>
#include <unicode/errorcode.h>

static icu::UnicodeString normalize_nfc(const icu::UnicodeString& in) {
    UErrorCode status = U_ZERO_ERROR;

    const icu::Normalizer2* nfc = icu::Normalizer2::getNFCInstance(status);
    if (U_FAILURE(status)) { return in; }

    icu::UnicodeString out;
    nfc->normalize(in, out, status);
    if (U_FAILURE(status)) { return in; }

    return out;
}
```

Professional note If your system compares or stores user-provided text, normalization strategy must be explicit:

- normalize on input (strong invariants, more cost at ingestion),
- normalize on compare (less intrusive, more cost at query time),
- or normalize selectively (e.g., identifiers and security boundaries).

2) Collation (locale-aware sorting and comparison)

Binary comparison of code units does not produce culturally correct ordering. ICU collation provides stable, locale-tailored ordering and comparison, with tunable strength (base letter vs accents vs case vs punctuation).

```
#include <unicode/coll.h>
```

```
#include <unicode/unistr.h>
#include <unicode/locid.h>
#include <unicode/errorcode.h>

static bool less_locale(const icu::UnicodeString& a,
                        const icu::UnicodeString& b,
                        const char* locale_name) {
    UErrorCode status = U_ZERO_ERROR;

    icu::Locale loc(locale_name);
    std::unique_ptr<icu::Collator> col(icu::Collator::createInstance(loc, status));
    if (U_FAILURE(status) || !col) { return a < b; }

    // Example: primary strength ignores accents/case differences.
    col->setStrength(icu::Collator::PRIMARY);

    return col->compare(a, b, status) == UCOL_LESS;
}
```

Professional note Collation is policy-heavy. Your application must define:

- which locale(s) apply,
- whether ordering must be stable across ICU upgrades,
- whether a collation key strategy is needed for indexing/search systems.

3) Text boundary analysis (grapheme, word, sentence, line breaks)

User-facing text editing and UI semantics depend on grapheme cluster boundaries and word breaks, not code points or code units. ICU provides break iterators for:

- grapheme cluster boundaries (cursor/backspace behavior),

- word boundaries (tokenization with Unicode rules),
- sentence boundaries,
- line breaking.

```
#include <unicode/brkiter.h>
#include <unicode/unistr.h>
#include <unicode/locid.h>
#include <unicode/errorcode.h>
#include <vector>

static std::vector<int32_t> grapheme_boundaries(const icu::UnicodeString& s) {
    UErrorCode status = U_ZERO_ERROR;
    std::unique_ptr<icu::BreakIterator> bi(
        icu::BreakIterator::createCharacterInstance(icu::Locale::getRoot(), status)
    );
    std::vector<int32_t> cuts;

    if (U_FAILURE(status) || !bi) { return cuts; }

    bi->setText(s);
    for (int32_t p = bi->first(); p != icu::BreakIterator::DONE; p = bi->next()) {
        cuts.push_back(p); // positions are UTF-16 indices into UnicodeString
    }
    return cuts;
}
```

Important constraint Break iterator operations are not universally ”share one instance everywhere” safe; treat iterator objects as non-thread-safe unless documented otherwise. In multi-threaded code, prefer one iterator per thread or use object pools guarded by synchronization.

4) Case mapping and case folding

Unicode case behavior is not a trivial ASCII transformation. ICU supports:

- locale-aware case mapping (upper/lower/title),
- case folding (for caseless matching),
- special casing rules that depend on language and context.

Professional note For identifier and authentication-like comparisons, the correct operation is often *case folding* combined with normalization, not locale-dependent upper/lower casing.

5) Regular expressions with Unicode properties

ICU provides regex features with Unicode awareness (character classes using Unicode properties and scripts). This is useful for validation and parsing tasks where ASCII-only patterns are incorrect.

6) Transliteration

ICU transliteration supports script-to-script mapping and transformations (e.g., Latinization), which is useful for search indexing, approximate matching, or user experience features. Transliteration is inherently policy-driven and language-sensitive; use it deliberately.

7) IDNA, spoof checking, and security-relevant utilities

Unicode introduces security risks (confusables, mixed-script spoofing, visually similar identifiers). ICU includes security-oriented components (e.g., spoof checking and IDN handling) that help build safer identifier and domain processing pipelines.

6.3.4 Integration considerations that determine success or failure

ICU integration issues in real C++ systems usually fall into a small set of predictable categories.

1) Linking and required libraries

ICU is typically shipped as multiple libraries. For most Unicode semantics you usually need:

- the common Unicode core library,
- the internationalization library (collation, formatting, break iteration, transliteration),
- and the ICU data component.

Practical advice Ensure your build system can:

- locate headers consistently across platforms,
- link the correct set of ICU libraries for your feature set,
- manage debug/release variants and runtime library compatibility on Windows,
- avoid ODR/ABI mismatches by standardizing one ICU distribution per product.

2) ICU data: packaging, discovery, and deployment

ICU is data-driven. Many failures that look like ”ICU is broken” are actually *data discovery* problems. ICU needs access to its compiled data (commonly a `.dat` package or a data shared library), and it locates it via a defined search strategy.
Professional deployment options:

- **Use a packaged data file** deployed next to your binary and configure ICU to find it.

- **Embed ICU common data in memory** and register it at startup.
- **Rely on system ICU** only if your product can accept system version differences and platform packaging policies.

Embedding common data (conceptual initialization)

```
#include <unicode/udata.h>
#include <unicode/uclean.h>

// Conceptual: supply a pointer to the in-memory image of the ICU data file.
// In production you obtain this pointer by embedding/packing the data.
extern const unsigned char* g_icu_data_image;
extern int32_t g_icu_data_size;

static bool init_icu_common_data() {
    UErrorCode status = U_ZERO_ERROR;

    // Provide ICU with a pointer to common data in memory.
    // The pointer must remain valid for the process lifetime.
    udata_setCommonData(g_icu_data_image, &status);
    if (U_FAILURE(status)) { return false; }

    // Optional: later at shutdown, if you dynamically unload ICU:
    // u_cleanup();
    return true;
}
```

Operational rule Treat ICU data initialization as part of your platform layer. Validate early at process startup and fail fast with actionable diagnostics if data cannot be loaded.

3) Versioning and upgrade strategy

ICU is designed with binary compatibility policies, but major upgrades can introduce ABI changes depending on how it is packaged and built in your environment.
Professional upgrade practices:

- Pin ICU to a known version for a product line and upgrade deliberately.
- Run a Unicode regression test suite focused on: normalization, collation ordering, segmentation boundaries, and casing behaviors.
- If deterministic results are required across time (e.g., collation keys used in persistent indexes), version your collation strategy and plan migrations.

4) Thread-safety model

ICU provides many ”service objects” and factories that are safe to use concurrently once ICU is initialized, but individual objects that maintain iteration state (such as break iterators) should be treated as non-thread-safe unless explicitly documented.
Professional patterns:

- Use **thread-local** break iterators or create them on demand in each thread.
- Treat immutable shared objects (like some normalizer instances) as shareable if documented.
- Avoid global mutable ICU objects.

5) Error handling style and failure modes

ICU C APIs commonly use `UErrorCode` out-parameters; ICU C++ APIs often use `UErrorCode` as well, or C++ wrappers.
Professional error-handling rules:

- Always initialize `UErrorCode` to `U_ZERO_ERROR`.
- Always check `U_FAILURE(status)` after each ICU call that can fail.
- Decide and document your boundary policy for ill-formed sequences: strict reject, replacement, or round-trip preservation.

```
#include <unicode/ustring.h>
#include <unicode/utypes.h>
#include <string>
#include <string_view>

static bool utf8_to_utf16_replace(std::string_view utf8, std::u16string& out) {
    UErrorCode status = U_ZERO_ERROR;

    // First pass: compute required length.
    int32_t required = 0;
    u_strFromUTF8WithSub(nullptr, 0, &required,
                         utf8.data(), static_cast<int32_t>(utf8.size()),
                         0xFFFD, nullptr, &status);

    if (status != U_BUFFER_OVERFLOW_ERROR && U_FAILURE(status)) return false;

    status = U_ZERO_ERROR;
    out.resize(static_cast<size_t>(required));
    u_strFromUTF8WithSub(reinterpret_cast<UChar*>(out.data()), required, nullptr,
                         utf8.data(), static_cast<int32_t>(utf8.size()),
                         0xFFFD, nullptr, &status);

    return U_SUCCESS(status);
}
```

This illustrates a *replacement policy* pipeline suitable for robust ingestion where you prefer preserving processing continuity over strict rejection.

6) Performance and memory: where ICU costs come from

ICU is fast enough for most applications, but it is not ”free.” Costs typically come from:

- boundary transcoding (UTF-8 $\leftrightarrow$ UTF-16),
- normalization and case folding (table-driven transformations),
- collation (locale data + algorithmic comparisons; collation keys can help),
- break iteration (stateful scanning, rules-based segmentation),
- data initialization and caching.

Professional performance strategies:

- Convert to ICU types only when semantic processing is needed.
- Use normalization ”quick check” patterns where appropriate.
- Precompute collation keys for indexing and repeated comparisons.
- Separate ”ASCII fast path” from ”full Unicode path” when justified, but keep correctness guarantees explicit.

7) Interop with modern C++ (`char8_t`, views, and boundaries)

Most systems today prefer UTF-8 as the interchange encoding and increasingly express UTF-8 intent using `std::u8string_view`. ICU’s UTF-8 conversion APIs commonly accept byte pointers; bridging must therefore be explicit and auditable.

```
#include <string_view>
#include <unicode/unistr.h>
#include <unicode/stringpiece.h>
```

```
static icu::UnicodeString from_u8(std::u8string_view u8) {
    const char* p = reinterpret_cast<const char*>(u8.data());
    return icu::UnicodeString::fromUTF8(
        icu::StringPiece(p, static_cast<int32_t>(u8.size()))
    );
}
```

Rule Keep casts in a dedicated boundary layer; do not mix `char` and `char8_t` throughout business logic.

6.3.5 When ICU is the right tool (and when it is not)

ICU is the right tool when you need

- correct normalization and canonical equivalence handling,
- locale-aware collation and searching behaviors,
- grapheme/word/sentence boundary segmentation,
- robust casing and folding rules,
- security-relevant Unicode utilities for identifiers/domains.

ICU may be too heavy when you only need

- fast UTF-8 validation and transcoding at scale,
- lightweight code point iteration without locale services,
- minimal dependency footprint (embedded firmware-class constraints).

In such cases, professional architectures often combine ICU (semantic layer) with a high-performance UTF engine (boundary validation/transcoding) and keep ICU usage limited to the parts of the system that actually require Unicode semantics.

6.3.6 Summary

ICU is the industry-grade choice for Unicode semantics in C++ systems because it provides:

- normalization, collation, segmentation, casing/folding, and more,
- data-driven behavior that tracks Unicode and locale evolution,
- robust APIs suitable for production correctness.

Successful ICU integration depends less on calling the right function and more on designing the system correctly:

- define explicit encoding contracts (UTF-8 boundaries, UTF-16 ICU core),
- choose and document validation and error-handling policies,
- package and initialize ICU data reliably,
- manage versioning and regression testing,
- respect thread-safety constraints for stateful iterator objects,
- isolate conversions and avoid spreading casts across the codebase.

This disciplined approach turns ICU from a ”large dependency” into a predictable, professional Unicode foundation for C++20–C++26 systems.

6.4 Lightweight UTF-8 Libraries and Their Trade-offs

6.4.1 Motivation for lightweight libraries

Professional C++ systems commonly use UTF-8 as the canonical external encoding due to its compactness, ASCII compatibility, and dominance in internet protocols and file formats. However, full Unicode semantics (normalization, collation, segmentation) often come from heavyweight libraries. Many real-world use cases only need:

- validated UTF-8 input,
- safe iteration over Unicode scalar values,
- conversion to other encodings,
- basic code-point awareness without building a full semantic text layer.

Lightweight UTF-8 libraries sit between the bare standard library and full Unicode platforms like ICU. They provide essential correctness with minimal footprint and dependency cost.

6.4.2 Categories of lightweight UTF-8 libraries

Lightweight libraries generally fall into the following categories:

UTF-8 traversal and validation helpers

These provide code-point iteration and well-formedness checks for UTF-8 sequences without full semantic layers.

Transcoding and basic conversion utilities

These convert between UTF-8 and UTF-16/UTF-32 or validate and transcode on demand.

Minimal semantic extensions

Some libraries provide small utilities for specific tasks like case folding or normalization to a single form but do not attempt full locale-aware processing.

6.4.3 Representative examples

We discuss several established lightweight UTF-8 libraries and their trade-offs.

utf8cpp: UTF-8 iterator and validation

Overview utf8cpp is a header-only C++ library that provides:

- safe UTF-8 iteration (code-point boundaries),
- validation checks for well-formed UTF-8,
- basic conversion between UTF-8 and UCS-2, UTF-16, UTF-32 where appropriate.

Strengths

- Small footprint and easy to integrate (header-only).
- Improves correctness over naive byte iteration.
- Suitable for projects where full Unicode semantics are not needed.

Trade-offs

- It does not perform normalization or handle grapheme clusters.
- It is not a full Unicode property engine.

```
#include <string_view>
#include "utf8.h" // from utf8cpp

void safe_process_utf8(std::string_view bytes) {
    auto it = utf8::iterator(bytes.begin(), bytes.begin(), bytes.end());
    auto end = utf8::iterator(bytes.end(), bytes.begin(), bytes.end());
    for (; it != end; ++it) {
        char32_t cp = *it; // safe code point
        // process cp
    }
}
```

simdutf (fast validation and transcoding)

Overview simdutf is a high-performance library focused on validation and transcoding across UTF-8, UTF-16, and UTF-32 using optimized algorithms and SIMD instructions where available.

Strengths

- Extremely fast UTF-8 validation and conversion.
- Efficient for large text volumes typical in logging, ingestion, or ETL systems.
- Maintains correctness at the code-point decoding and transcoding boundaries.

Trade-offs

- It does not provide higher-level Unicode semantics (normalization, graphemes).
- Integration requires linking and appropriate build configuration for performance paths.

```
// conceptual outline: simdutf-style validation
```

```cpp
#include <string_view>
#include <iostream>

bool is_valid_utf8(std::string_view bytes) {
    // library call, optimized via SIMD
    return simdutf::validate_utf8(bytes.data(), bytes.size());
}

int main() {
    std::string text = "some UTF-8 text";
    if (!is_valid_utf8(text)) {
        std::cerr << "Invalid UTF-8";
    }
}
```

utf8proc (compact Unicode utilities)

Overview utf8proc provides:

- Unicode character properties,
- optional normalization and case folding routines,
- validation and transformation utilities.

Strengths

- Offers more semantic capabilities than a pure traversal library.
- Smaller and simpler than full platforms like ICU.

Trade-offs

- Not a full locale service provider.

- May not cover all normalization forms or edge cases without configuration.

6.4.4 General trade-offs for lightweight libraries

1) Footprint vs semantic completeness

Lightweight libraries are often:

- header-only or minimal runtime dependency,
- easy to integrate and portable,
- low memory overhead.

However, this comes at the cost of:

- lack of full Unicode property tables,
- absence of locale-aware collation,
- limited or no normalization policy.

2) Performance vs algorithmic scope

Lightweight libraries often excel at:

- safe iteration over UTF-8 (code points),
- validation and simple transcoding,
- reducing common bugs from naive byte processing.

They do not aim to:

- provide grapheme cluster segmentation by default,
- address locale-dependent semantics,
- offer policy controls for normalization or collation.

3) Explicit boundary roles in layered designs

In professional systems, lightweight libraries typically occupy the *boundary and infrastructure layers*:

- boundary validation of incoming text,
- safe iteration for pipelines that do not need semantics,
- pre-processing before semantic library calls (filtering, quick checks),
- coupling with heavyweight libraries for higher-level semantics.

Example pattern:

```
// 1) boundary read and validate
if (!validate_utf8(bytes)) { /* reject or handle error */ }

// 2) lightweight iteration for basic logic
for (code_point : utf8_codepoints(bytes)) { /* ASCII vs code point logic */ }

// 3) if semantics required, convert and call ICU/semantic library
auto u16 = utf8_to_utf16(bytes);
auto result = ICU_normalize(u16);
```

6.4.5 Choosing the right tool for the layer

When a lightweight library is enough

Use a lightweight UTF-8 helper when:

- your domain only needs validated code-point iteration,
- your text stack does not require locale-aware behavior,

- performance and dependency footprint are primary concerns,
- the only need is boundary validation and safer byte handling.

When to pair with a full Unicode platform

Pair lightweight libraries with heavyweight ones when:

- semantic operations (normalization) affect correctness,
- user-facing text manipulation (graphemes, collation) is required,
- locale-specific rules matter in sorting or comparison,
- policy decisions (case folding vs case mapping) must be explicit and complete.

6.4.6 Best practices when using lightweight UTF-8 libraries

- Validate input explicitly at boundaries rather than assuming correctness.
- Use code-point-aware iterators to avoid bugs from naive byte indexing.
- Isolate transcoding and validation logic in infrastructure layers.
- Combine lightweight utilities with heavyweight Unicode semantics where needed.
- Document the Unicode model and libraries in your system architecture.

6.4.7 Summary

Lightweight UTF-8 libraries provide practical, focused tools for working with UTF-8 in modern C++ systems. They are valuable when:

- dependency size and performance overhead must be minimal,

- core Unicode semantics (normalization, collation) are not required in the current layer,
- code needs safe iteration and validation without the complexity of larger platforms.

Their trade-offs are clear:

- smaller scope and data footprint,
- lack of full semantic support,
- not a replacement for locale-aware or normalization-aware text processing.

Used appropriately in layered architectures, lightweight UTF-8 libraries complement semantic engines to deliver both performance and correctness in professional Unicode handling with modern C++.

6.5 Architectural Patterns for Unicode-Safe Systems

6.5.1 Why architecture matters more than individual functions

Unicode correctness is not achieved by sprinkling ”UTF-8 helpers” across a codebase. It is achieved by designing a system with:

- explicit encoding contracts at every boundary,
- validated and well-defined invariants for stored text,
- isolated conversion points (not scattered conversions),
- correct semantic layers (code points vs graphemes vs locale behavior),
- testable policies for invalid input, normalization, and comparisons.

The main architectural goal is to prevent text data from being interpreted inconsistently in different subsystems.

6.5.2 Pattern 1: The boundary-validation funnel (ingest once, validate once)

Idea

All external text enters through a small number of *ingestion funnels* that:

1. accept raw bytes (because external inputs are bytes),
2. validate and decode according to a declared encoding (typically UTF-8),
3. apply a chosen invalid-sequence policy,
4. optionally apply normalization (if the system requires invariants),
5. produce an internal text type with strong invariants.

Benefits

- Eliminates duplicated validation logic and inconsistent error handling.
- Prevents silent propagation of ill-formed sequences.
- Makes security review feasible: one place controls policies.

```
#include <string_view>
#include <string>
#include <optional>

enum class Utf8Policy { StrictReject, ReplaceWithFFFD };

struct Utf8Text {
    std::u8string data; // invariant: well-formed UTF-8 under chosen policy
};

struct IngestResult {
    bool ok;
    Utf8Text text;
};

bool validate_utf8(std::string_view bytes);              // provided by your UTF layer
std::u8string replace_invalid_utf8(std::string_view b);  // replacement policy
std::u8string normalize_nfc(std::u8string_view u8);      // semantic library (optional)

IngestResult ingest_utf8(std::string_view bytes,
                         Utf8Policy policy,
                         bool normalize_for_storage) {
    if (policy == Utf8Policy::StrictReject) {
        if (!validate_utf8(bytes)) return {false, {}};
        Utf8Text out{ std::u8string(reinterpret_cast<const char8_t*>(bytes.data()),
                                    reinterpret_cast<const char8_t*>(bytes.data()) +
                                    ↪  bytes.size()) };
```

```
        if (normalize_for_storage) out.data = normalize_nfc(out.data);
        return {true, std::move(out)};
    } else {
        Utf8Text out{ replace_invalid_utf8(bytes) };
        if (normalize_for_storage) out.data = normalize_nfc(out.data);
        return {true, std::move(out)};
    }
}
```

6.5.3 Pattern 2: A strong internal text type with explicit intent

Idea

Internally, do not use `std::string` as ”text” unless you enforce an invariant. Prefer a strong type:

- `Utf8Text`: owns a validated UTF-8 sequence (as `std::u8string`).
- `Utf8View`: non-owning view (`std::u8string_view`).

Benefits

- Forces API designers to declare whether data is bytes or UTF-8 text.
- Reduces accidental mixing of binary buffers and text.
- Makes fuzzing and testing easier because invariants are explicit.

```
#include <string_view>

using BytesView = std::string_view;
using Utf8View  = std::u8string_view;

void parse_bytes(BytesView);
void render_text(Utf8View);
```

6.5.4 Pattern 3: The ”one canonical encoding” rule (usually UTF-8)

Idea

Pick one canonical encoding for most internal subsystems. For modern systems this is commonly UTF-8 because it is compact, interoperable, and dominant in protocols and storage formats.

What this does *not* mean

It does not mean you never use UTF-16 or UTF-32. It means:

- UTF-16 is used at Windows API boundaries.
- UTF-32 (code points) is used transiently for semantic algorithms and Unicode property work.

Benefits

- Minimizes conversion churn across modules.
- Simplifies contracts: ”internal text is UTF-8” becomes an invariant.
- Keeps storage and transport efficient.

6.5.5 Pattern 4: Isolated conversion gateways (no scattered transcoding)

Idea

Every conversion between encodings happens in a small number of gateway modules, not in application logic:

- UTF-8 $\rightarrow$ UTF-16 for Windows.

- UTF-8 → UTF-32 for code-point algorithms.
- UTF-16 → UTF-8 on return from Windows.

Benefits

- Prevents inconsistent error handling and partial conversions.
- Allows optimization: caching, pooling, SIMD validation, batch transcoding.
- Eases auditing and security review.

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

enum class TranscodePolicy { StrictReject, ReplaceWithFFFD };

std::u16string utf8_to_utf16(std::u8string_view u8, TranscodePolicy);
std::u8string  utf16_to_utf8(std::u16string_view u16, TranscodePolicy);

struct WindowsPath {
    std::u16string u16; // boundary representation
};

WindowsPath to_windows_path(std::u8string_view u8, TranscodePolicy p) {
    return WindowsPath{ utf8_to_utf16(u8, p) };
}
```

6.5.6 Pattern 5: Separate units for algorithms: code units vs code points vs graphemes

Idea

Professional Unicode-safe systems explicitly choose the right unit for each operation:

Code units (storage-oriented) Use code units for:

- transport, logging, serialization,
- byte-level scanning for ASCII tokens,
- memory-efficient storage.

Code points (semantic but not user-perceived) Use code points for:

- Unicode property checks (categories, scripts),
- identifier rules and security filters,
- normalization steps that conceptually operate on scalar values.

Grapheme clusters (user-perceived characters) Use grapheme clusters for:

- cursor movement, backspace/delete semantics,
- UI selection, truncation displayed to users,
- ”limit to N characters” requirements in UI contexts.

A common bug pattern is doing grapheme work on code units; architecture should prevent this by forcing unit selection.

6.5.7 Pattern 6: Normalization strategy as an explicit system policy

Why normalization is architectural

Normalization is not a local ”helper call.” It affects:

- equality, hashing, and keys,

- database indexing,
- security boundaries,
- deterministic behavior across inputs.

Common normalization strategies

- **Normalize-on-ingest**: store canonical form (often NFC). Pros: stable storage and comparisons. Cons: ingestion cost, may alter original representation.
- **Normalize-on-compare**: store raw input, normalize when comparing/searching. Pros: preserves original bytes. Cons: repeated cost, subtle bugs if not centralized.
- **Hybrid**: normalize identifiers and security-sensitive text; keep other content raw.

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

struct CanonicalKey {
    std::u8string normalized; // e.g., NFC (or NFKC for identifiers), plus optional case
    ↪  folding
};

CanonicalKey make_key_for_lookup(std::u8string_view u8) {
    // 1) normalize (policy-defined)
    // 2) case fold (policy-defined)
    // 3) return for hashing/index
    return CanonicalKey{ std::u8string(u8) };
}
```

6.5.8 Pattern 7: Collation and sorting as a separate service

Idea

Sorting is not "`operator<` on strings" when user-facing correctness matters. Treat collation as a service:

- explicit locale selection,
- explicit collation strength,
- versioning strategy if results must be stable over time.

Implementation

Use a semantic library for collation (often ICU) behind an interface. Do not leak collation logic into general string code.

```
#include <string_view>

struct CollationConfig {
    std::string locale;
    int strength; // abstract policy setting
};

struct CollatorService {
    virtual ~CollatorService() = default;
    virtual bool less(std::u8string_view a, std::u8string_view b,
                      const CollationConfig& cfg) const = 0;
};
```

6.5.9 Pattern 8: Security boundaries and confusable-aware identifiers

Idea

Unicode introduces security risks:

- visually confusable characters,
- mixed-script spoofing,
- normalization and case-folding pitfalls.

Treat identifier handling (usernames, domains, commands, ACL keys) as a security boundary:

- apply strict validation,
- normalize using a security-appropriate form,
- apply script restrictions if required by policy,
- log and audit rejected inputs.

Architecture matters because inconsistent enforcement across modules creates bypasses.

6.5.10 Pattern 9: Testing strategy integrated into architecture

Unicode-safe systems require tests that reflect their policies:

- round-trip tests for transcoding boundaries,
- fuzzing for invalid UTF-8 sequences,
- normalization and case folding regression tests,
- grapheme segmentation tests for UI operations,
- locale/collation tests for user-visible sorting.

Professional practice:

- pin and document the Unicode and locale data versions used by semantic libraries,
- treat upgrades as behavior changes requiring regression validation.

6.5.11 Pattern 10: A layered reference architecture

A robust Unicode-safe architecture can be described as layers:

Layer 0: Raw bytes boundary

- File/network input as `std::string_view` or `std::span<const std::byte>`.
- No interpretation yet.

Layer 1: UTF-8 ingestion and invariant enforcement

- Validate or replace invalid sequences.
- Produce `Utf8Text`.

Layer 2: Conversion gateways

- UTF-8 $\leftrightarrow$ UTF-16 (Windows boundary).
- UTF-8 $\leftrightarrow$ UTF-32 (semantic algorithms).

Layer 3: Semantic services

- Normalization, collation, segmentation, case folding.
- Implemented via ICU or a comparable semantic library.

Layer 4: User-facing semantics

- Grapheme-based editing and display rules.
- UI toolkit shaping/bidi integration.

6.5.12 Summary

Unicode-safe C++ systems are built by architecture, not by ad-hoc utilities. The professional patterns are consistent:

- validate at boundaries and establish invariants,
- use strong UTF-8 intent types internally,
- isolate all transcoding in gateway modules,
- choose the correct unit of processing (code units vs code points vs graphemes),
- make normalization and collation explicit system policies,
- treat identifier handling as a security boundary,
- integrate regression tests and fuzzing around Unicode policies.

These patterns align with C++20–C++26 realities: the standard library provides robust foundations for representation and type intent, while professional correctness comes from well-designed layers and established Unicode libraries for semantic operations.

6.6 Designing APIs That Handle Text Correctly

6.6.1 API design goals for Unicode-safe systems

Correct Unicode handling begins at the API boundary. A well-designed API:

- communicates precisely what form of text it accepts,
- avoids mixing binary data and encoded text,
- distinguishes between storage encodings and semantic operations,
- enforces invariants on text encoding early,
- makes error handling explicit and discoverable.

Poorly designed text APIs are a major source of bugs because they implicitly assume encoding semantics that are not enforced in code or in types.

6.6.2 Express encoding intent in parameter and return types

The first principle in Unicode API design is to make the encoding part of the type system.

- For UTF-8 encoded text, expose types that convey this intent explicitly (`std::u8string`, `std::u8string_view`).
- For byte buffers that may contain arbitrary data, use `std::string_view` or `std::span<const std::byte>`.
- Avoid APIs that take `const char*` or `std::string` without documenting or enforcing the intended encoding.

```
void process_raw_bytes(std::string_view bytes);
void process_utf8(std::u8string_view text);
```

In this example, the difference between ”bytes” and ”UTF-8 text” is explicit in the API signature. This clarity improves compile-time checking and prevents a class of bugs where binary data is inadvertently treated as text.

6.6.3 Design for explicit validation and error handling

Text APIs must define how they react to invalid encoded input. Treat validation as part of the API contract, not an implementation detail.

- Use return types that reflect success/failure (`std::optional`, `std::expected` when available).
- Provide both strict and replacement strategies where relevant.
- Avoid throwing exceptions in low-level loops where performance is critical; instead, return explicit status codes.

```
enum class Utf8Policy { Strict, Replace };

std::optional<std::u8string> normalize_utf8(
    std::u8string_view input,
    Utf8Policy policy
);
```

In this example, the API documents its normalization behavior and error policy explicitly.

6.6.4 Avoid implicit conversions

Implicit conversions between different string encodings can hide expensive operations and lead to undefined behavior if encodings are wrong.

```
std::string to_legacy_api(std::u8string_view utf8);
std::u8string from_legacy_api(std::string_view bytes);
```

By making conversions explicit, the API ensures that:

- all encoding assumptions are visible at the call site,
- performance impact is obvious to the caller,
- potential encoding errors are easier to detect and handle.

6.6.5 Separate encoding concerns from semantic operations

APIs should separate low-level encoding handling from high-level semantic behavior. For example:

- **Encoding layer**: validation, decoding, transcoding.
- **Semantic layer**: normalization, grapheme segmentation, collation.

Keep these layers distinct rather than interleaving them in a single function.

```
std::optional<std::u32string> decode_to_codepoints(
    std::u8string_view utf8,
    Utf8Policy policy
);

std::u8string normalize_nfc(std::u32string_view codepoints);
```

This design provides composable building blocks rather than monolithic text operations with hidden behavior.

6.6.6 Provide clear semantics for length and indexing

API documentation must clarify what `length()`, `operator[]`, and iterators mean:

- **Code unit length**: number of storage elements (bytes or 16-bit, 32-bit units).

- **Code point count**: number of Unicode scalar values after decoding.
- **Grapheme cluster count**: number of user-perceived characters.

Functions that claim to return ”the number of characters” must specify which concept of character they support. For example:

```
std::size_t count_code_units(std::u8string_view utf8);
std::size_t count_code_points(std::u8string_view utf8);
std::size_t count_graphemes(std::u8string_view utf8);
```

Separating these concepts makes the API explicit and less error-prone.

6.6.7 Design for locale and collation as orthogonal policies

When APIs depend on cultural or linguistic rules (sorting, case mapping), they should take a locale or collation config object:

```
struct CollationConfig {
    std::string locale;
    int strength;
};

bool compare_locale(
    std::u8string_view a,
    std::u8string_view b,
    CollationConfig cfg
);
```

By passing a `CollationConfig`, the API consumer explicitly defines which locale rules apply. This avoids hidden dependencies on global locale state or implementation defaults.

6.6.8 Decouple UI and semantic APIs from storage APIs

Text insertion, deletion, and cursor movement semantics depend on grapheme clusters, not code units or code points. UI APIs that operate on text must use grapheme-aware indices:

```
struct GraphemePosition { std::size_t index; };

void insert_at(std::u8string& text,
               GraphemePosition pos,
               std::u8string_view to_insert);

void erase_grapheme(std::u8string& text,
                    GraphemePosition pos);
```

By encoding that the position is a grapheme boundary, the API avoids confusion with raw byte or code-unit indexing.

6.6.9 Design for transcode boundaries explicitly

Many professional systems rely on explicit, named conversion functions for moving between encodings. Do not overload `operator=` or constructors with hidden transcoding.

```
std::u16string utf8_to_utf16(std::u8string_view utf8,
                             Utf8Policy policy);

std::u8string utf16_to_utf8(std::u16string_view utf16,
                            Utf8Policy policy);
```

This design clarifies:

- what conversions are available,
- how errors are handled,
- performance expectations.

6.6.10 Document invariants and policy decisions

Every text API should document:

- which encodings are accepted,
- whether validation is performed, and how errors are reported,
- whether and how normalization is applied,
- whether operations are code-unit, code-point, or grapheme-aware,
- which locale or collation rules are used.

Documentation should be part of the API contract, not an afterthought in comments.

6.6.11 Testing and property-based validation of text APIs

API design should be accompanied by tests that exercise:

- boundary conditions (ill-formed input sequences),
- normalization invariants (round-trip tests),
- indexing and slicing semantics,
- locale/collation variations,
- grapheme segmentation edge cases (combining marks, zero-width joiners).

Property-based testing tools can generate edge cases automatically for thorough coverage.

6.6.12 Versioning and stability guarantees

When APIs expose collation or normalization policies, design for version stability:

- specify which Unicode version the API targets,
- define behavior changes between versions explicitly,
- ensure deterministic behavior across builds and deployments.

6.6.13 Performance considerations in API design

Unicode-aware APIs often involve more work than ASCII-only logic. Design APIs to:

- document cost expectations (e.g., normalization is not constant time),
- provide fast paths for common cases (ASCII or BMP text),
- avoid hidden allocations where possible by offering reserve/append patterns,
- enable bulk operations when processing large text volumes.

```
bool starts_with_utf8(std::u8string_view haystack,
                      std::u8string_view needle);
```

Such APIs should clearly define whether they operate on bytes, code units, or code points and document performance implications.

6.6.14 Summary of API design principles

Designing correct text APIs in modern C++ means:

- making encoding explicit in types,
- separating encoding, decoding, and semantic operations,

- isolating conversions at known gateways,
- communicating normalization and locale policy,
- defining clear semantics for indexing and length,
- documenting invariants and performance costs,
- testing thoroughly across Unicode edge cases.

These principles align with professional Unicode handling requirements and lead to robust, maintainable, and correct text processing in C++20–C++26 systems.

Chapter 7

Best Practices and Design Guidelines

7.1 UTF-8 for Storage, Transport, and Interfaces

7.1.1 Terminology and the non-negotiable baseline

In this booklet, **UTF-8** means the standard Unicode Transformation Format defined for Unicode scalar values, encoded as sequences of 1–4 bytes. Professional systems must adopt two baseline rules:

1. **All external text is bytes until proven otherwise.** Files, sockets, environment variables, and IPC payloads are byte sequences; interpreting them as text is an explicit choice.

2. **If you claim ”UTF-8” you must define validation and error policy.** ”UTF-8” is not ”whatever byte sequence happens to work on my machine”.

7.1.2 Why UTF-8 is the default choice for storage and transport

UTF-8 has properties that make it the practical standard for storage and transport:

- **ASCII compatibility**: bytes `0x00..0x7F` represent ASCII directly. Existing parsers and many protocols already assume ASCII for control tokens and structural characters.
- **Byte order independence**: UTF-8 has no endianness issues; it is a stream of bytes.
- **Compactness for common text**: many real-world texts contain substantial ASCII (protocols, JSON keys, source code, logs), making UTF-8 memory- and bandwidth-efficient.
- **Interoperability**: UTF-8 is the dominant encoding for modern web content, network protocols, and cross-platform data formats.

7.1.3 Core rules for UTF-8 in professional systems

Rule 1: Make UTF-8 intent explicit in types

In modern C++ (C++20 and later), use `char8_t` and `std::u8string_view` to express UTF-8 code-unit intent in APIs that operate on text.

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

using BytesView = std::string_view;      // arbitrary bytes (may be text, may be not)
using Utf8View  = std::u8string_view;    // UTF-8 code units by contract

void write_packet(BytesView payload);
void render_text(Utf8View text);
```

If you must accept `std::string` for ecosystem compatibility, treat it as *bytes* unless your API contract explicitly states ”valid UTF-8” and you enforce it.

Rule 2: Validate at boundaries (never ”somewhere later”)

The safest pattern is: **ingest once, validate once, then carry a validated invariant internally**. Validation should be policy-driven:

- **Strict reject**: invalid UTF-8 is an error.
- **Replacement**: invalid subsequences are replaced with U+FFFD.
- **Round-trip preservation**: preserve original bytes (useful only when your domain requires it; it is not ”UTF-8 text” anymore).

```
#include <string_view>
#include <optional>
#include <string>
#include <cstddef>

enum class Utf8Policy { StrictReject, ReplaceWithFFFD };

bool validate_utf8(std::string_view bytes);             // provided by your UTF layer
std::u8string replace_invalid_utf8(std::string_view b); // replacement policy

struct Utf8Text {
    std::u8string data; // invariant: valid UTF-8 under the chosen policy
};

std::optional<Utf8Text> ingest_utf8(std::string_view bytes, Utf8Policy policy) {
    if (policy == Utf8Policy::StrictReject) {
        if (!validate_utf8(bytes)) return std::nullopt;
        return Utf8Text{ std::u8string(reinterpret_cast<const char8_t*>(bytes.data()),
                                       reinterpret_cast<const char8_t*>(bytes.data()) +
                                       ↪  bytes.size()) };
    } else {
        return Utf8Text{ replace_invalid_utf8(bytes) };
```

```
    }
}
```

Rule 3: Never treat ”length” as characters

For UTF-8 storage:

- `u8.size()` is **bytes / code units**, not characters.
- ”Characters” might mean **code points** or **grapheme clusters**; neither is the same as bytes.

Therefore:

- do not truncate UTF-8 by byte count unless you ensure you cut on a code-point boundary,
- do not use `operator[]` as ”character access”,
- do not assume ”N bytes = N characters” even for apparently simple scripts.

Rule 4: Avoid UTF-8 BOM unless a specific ecosystem requires it

A UTF-8 BOM (byte order mark used as a signature) is not required for UTF-8 and frequently causes problems in:

- parsers that do not expect it,
- tooling that treats it as data (especially in first-token parsing),
- protocol payloads where a BOM is not allowed.

Unless you have a documented requirement (some Windows-facing tools historically used BOM heuristics), prefer UTF-8 *without* BOM for storage and transport.

Rule 5: Treat normalization as a policy, not an implementation detail

Storing and transporting UTF-8 does not automatically solve equivalence issues. Visually identical text can have multiple valid Unicode representations (different combining sequences). Decide explicitly:

- **Normalize on ingest** (commonly to NFC) for stable storage and consistent comparisons, or
- **Normalize on compare/search** to preserve original bytes at rest, or
- **Hybrid** (normalize identifiers and security boundaries; store user content as-is).

Normalization requires a semantic Unicode library; it is intentionally outside the standard library.

Rule 6: Keep conversions isolated (gateways, not scattered casts)

UTF-8 is ideal for storage and transport, but interfaces may require other encodings:

- **Windows APIs**: typically UTF-16 (wide strings).
- **Some algorithms**: benefit from decoded scalar values (often UTF-32 in memory).

Do not scatter conversions throughout application logic. Centralize them in gateway modules.

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

enum class TranscodePolicy { StrictReject, ReplaceWithFFFD };

std::u16string utf8_to_utf16(std::u8string_view u8, TranscodePolicy); // gateway
std::u8string  utf16_to_utf8(std::u16string_view u16, TranscodePolicy); // gateway
```

```
struct WindowsWide {
    std::u16string u16; // represents UTF-16 code units
};

WindowsWide to_windows_api_arg(std::u8string_view u8) {
    return WindowsWide{ utf8_to_utf16(u8, TranscodePolicy::ReplaceWithFFFD) };
}
```

7.1.4 UTF-8 in storage: files, databases, logs

Files

Best practice for text files:

- store as UTF-8 without BOM unless required,
- document the encoding as part of the format specification,
- validate on read (strict or replacement) and handle errors deterministically.

For configuration formats (JSON/YAML/TOML/INI-like), ensure your parser and serializer both:

- agree on UTF-8 and reject or sanitize invalid sequences,
- preserve round-trip expectations if you promise them (otherwise normalize and re-emit).

Databases

Common professional approaches:

- store UTF-8 in DB fields using a Unicode-capable collation/charset configuration,

- decide whether keys/indexes use normalized/case-folded forms (very important for search and uniqueness constraints),
- treat DB boundaries as ingest points: validate before insertion and validate on extraction if the DB may contain legacy or corrupted data.

Logs and telemetry

Logs are both storage and transport:

- prefer UTF-8 for log messages,
- for safety, use replacement policy (so logging never crashes or throws on invalid input),
- explicitly escape or sanitize control characters where required by your log format.

7.1.5 UTF-8 in transport: network protocols and messaging

Protocol design rules

If you design a protocol:

- declare that text fields are UTF-8,
- specify whether ill-formed sequences are rejected or replaced,
- forbid ambiguous legacy encodings unless you have a strict negotiation mechanism,
- define normalization expectations if comparisons are performed across endpoints.

Streaming and chunking

UTF-8 is a byte stream. When reading from sockets in chunks:

- avoid decoding across chunk boundaries without carrying decoder state,

- if you do incremental decoding, keep a small carry buffer for partial sequences,
- validate incrementally and fail fast (or replace) according to your policy.

```
#include <string>
#include <string_view>
#include <vector>

// Conceptual incremental decoder outline.
// A real implementation must correctly handle partial sequences and validation.
struct Utf8StreamDecoder {
    std::string carry;

    // feed raw bytes; produce complete UTF-8 blocks for higher layers
    std::vector<std::string_view> feed(std::string_view chunk) {
        // 1) append chunk to carry
        // 2) split carry into valid UTF-8 prefix blocks
        // 3) keep trailing partial sequence in carry
        // 4) return views into an owned buffer or copy out completed blocks
        return {};
    }
};
```

7.1.6 UTF-8 at interfaces: APIs, FFI, OS boundaries

Library APIs: prefer views and explicit contracts

Professional C++ APIs should:

- accept `std::u8string_view` for UTF-8 text,
- accept `std::string_view` (or span<byte>) for raw bytes,
- return validated UTF-8 types when they promise text.

Also:

- do not promise ”character indexing” on UTF-8; expose code-point or grapheme APIs separately,
- do not hide transcoding; make conversions explicit and localized.

C and foreign-function interfaces (FFI)

Many external interfaces use `const char*`. In such cases:

- define whether those bytes are UTF-8,
- validate at the boundary,
- avoid passing `char8_t*` directly unless the ABI boundary is explicitly defined for it.

```
#include <string_view>

// C ABI boundary: bytes + length.
extern "C" void c_api_send(const char* p, std::size_t n);

void send_utf8_to_c(std::u8string_view u8) {
    const char* p = reinterpret_cast<const char*>(u8.data());
    c_api_send(p, u8.size());
}
```

Filesystem paths: keep path encoding decisions explicit

Paths are a classic trap. Best practice:

- treat the filesystem boundary as an encoding gateway,
- on Windows, convert UTF-8 $\rightarrow$ UTF-16 at the boundary,

- on POSIX-like systems, remember paths are bytes; UTF-8 is a convention, not a universal law.

In cross-platform code, do not assume that `std::filesystem::path` uses UTF-8 internally everywhere. Instead, define a product policy ("we accept UTF-8 paths from our UI or config") and implement a single, audited conversion gateway per platform.

7.1.7 Anti-patterns (what breaks UTF-8 correctness in storage/transport/interfaces)

- **Storing "UTF-8" as** `std::string` **with no invariant** and later interpreting it differently in different modules.
- **Using** `size()` **as character count** for truncation, UI limits, or indexing.
- **Byte-slicing UTF-8** to create substrings without ensuring boundary alignment.
- **Implicit transcoding** hidden in constructors or overloads (surprising costs, ambiguous error handling).
- **Depending on global locale** for Unicode conversions or assuming "current code page" is UTF-8.
- **Accepting invalid UTF-8 in security-sensitive identifiers** (usernames, ACL keys, commands) without strict policy.

7.1.8 A compact checklist for production systems

1. Choose UTF-8 as canonical for storage/transport unless you have a strict platform requirement.

2. Make UTF-8 intent explicit: `std::u8string_view` for text, `std::string_view` for bytes.
3. Validate at boundaries; define strict vs replacement behavior.
4. Avoid BOM by default; treat it as an exceptional interoperability requirement.
5. Centralize conversions (UTF-8 $\leftrightarrow$ UTF-16/UTF-32) in gateway modules.
6. Decide normalization policy (ingest vs compare vs hybrid) and enforce it consistently.
7. Write tests for invalid sequences, round-trip behavior, and boundary conversions.

7.1.9 Summary

UTF-8 is the professional default for **storage** and **transport** because it is byte-stream friendly, ASCII-compatible, compact for common workloads, and interoperable across ecosystems. In C++20–C++26 systems, correctness comes from disciplined design:

- explicit UTF-8 intent in API types,
- boundary validation with a documented error policy,
- isolated transcoding gateways,
- explicit normalization/collation policies when semantics matter,
- and a strict separation between bytes, code units, code points, and grapheme clusters.

7.2 When to Decode and When to Avoid Decoding

7.2.1 The fundamental trade-off

In Unicode-aware systems, decoding transforms a stored byte sequence (UTF-8 code units) into a semantic representation such as:

- a stream of Unicode scalar values (e.g., UTF-32 code points),
- an iterator that yields scalar values,
- a segmented view of grapheme clusters (user-perceived characters).

Decoding enables semantic text operations (e.g., normalization, case folding, grapheme clustering), but it is not always necessary and can be a performance and complexity cost. The core guideline is:

> Always decode only when the operation semantics require interpreting code units beyond their storage representation; avoid decoding when byte-level processing suffices.

This section formalizes when decoding is appropriate, and when it should be deferred or avoided.

7.2.2 When `no decoding` is sufficient

Certain operations require no decoding of UTF-8 beyond treating it as a byte sequence. For these cases, decoding is unnecessary and can even be harmful if it implicitly assumes encoding correctness:

1) Byte-oriented protocol framing and transport

Protocols often define messages, headers, and fields where:

- delimiters (e.g., CR/LF, NUL) are defined in a fixed byte set,
- control bytes and structural syntax are ASCII,
- text fields are raw bytes until explicitly validated.

In this domain, refrain from decoding during:

- frame boundary detection,
- length prefix interpretation,
- checksums/hashes on raw bytes,
- protocol version negotiation that is agnostic to encoding.

Only when an API explicitly declares that a field is UTF-8 text should you validate or decode it.

2) Binary data scan and filtering

Operations such as scanning for specific byte patterns (e.g., magic headers, binary-format tokens) do not need decoding. Treat the data as `std::string_view` or `span<const std::byte>` and operate on bytes directly:

```
#include <string_view>

bool has_magic(std::string_view bytes,
               std::string_view magic) {
    return bytes.find(magic) != std::string_view::npos;
}
```

Decoding here adds cost without improving correctness for the task.

3) Storage/transmit without semantic transformation

If your system merely stores and re-emits text without needing to inspect or transform logical characters, retain the original encoding and avoid decoding. For example, a logging service that passes UTF-8 through to persistence should validate once and then treat records as opaque:

```
void log_entry(std::string_view bytes) {
    // validate if necessary
    // persist bytes unchanged
}
```

7.2.3 When `decoding` is necessary

Decoding becomes necessary when the intended operation depends on semantics beyond raw bytes:

1) Character semantics—code-point iteration

If the logic must reason about Unicode scalar values (not bytes), decode to code points:

- computing Unicode property categories,
- building frequency counts by code point,
- applying code-point based classification.

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

// conceptual: decode to vector<char32_t>
std::vector<char32_t> decode_utf8_to_codepoints(
    std::u8string_view u8);
```

Use code-point iteration rather than raw bytes whenever the semantics depend on the actual scalar values.

2) Grapheme cluster awareness

For user-visible text editing—cursor movement, deletion, selection—simple code-point iteration is insufficient. Grapheme clusters (user-perceived characters) may consist of multiple code points (base + combining marks). A semantic layer or library that supports grapheme segmentation is required:

```
// conceptual API: iterate graphemes
for (auto cluster : grapheme_clusters(u8_text)) {
    // cluster is a sequence of one or more code points
}
```

Decoding into grapheme clusters is substantially more expensive than byte or code-point iteration, so apply it only where required.

3) Normalization and canonical equivalence

Unicode text can be represented in multiple semantically equivalent forms. If your logic must compare text (e.g., identifiers, keys) in a canonical way, decoding combined with normalization is necessary:

- NFC (Normalization Form C)
- NFD (Normalization Form D)
- NFKC/KD (compatibility forms)

Normalization requires decoding to code points, applying canonical decomposition/recomposition, and re-encoding:

```
std::u8string normalize_to_nfc(std::u8string_view u8);
```

Decode when text equivalence, matching, or canonical storage invariants matter.

4) Collation and locale-aware comparisons

Comparing text under language-specific rules (e.g., dictionary ordering) requires semantic collation, which depends on Unicode collation algorithms and locale data. This cannot be done at the byte level; it requires decoding and the use of a collation engine.

```
bool locale_less(std::u8string_view a,
                 std::u8string_view b,
                 LocaleConfig locale);
```

Here, decoding is part of the collation process.

7.2.4 When to delay decoding

Decoding has a cost. In many pipelines, it is preferable to delay decoding until it is necessary:

1) Lazy evaluation for performance

In high-throughput systems, you may defer decoding until you know the specific text needs to be processed semantically. For example, a filtering stage may test a header and only decode the body when the filter passes:

```
if (header == expected && is_valid_utf8(body_bytes)) {
    auto codepoints = decode_to_codepoints(body_bytes);
    process_semantic_logic(codepoints);
}
```

This avoids decoding text that is never semantically inspected.

2) “Quick check” before full decode

For normalization or Unicode property checks, “quick check” heuristics allow you to determine whether a full decode/normalize cycle is necessary. Many Unicode libraries expose quick-check APIs that detect if text is already in a particular normalized form.

3) Buffer scanning without full decode

When scanning large texts for delimiters or light validation, use partial decoding or stateful iterators that return an error on invalid sequences without producing full scalar sequences in memory.

```
// conceptual: stateful decoder that yields code points on demand
Utf8Decoder decoder(bytes);
while (auto cp = decoder.next()) {
    if (cp == target) break;
}
```

This pattern avoids building an entire decoded buffer when scanning suffices.

7.2.5 Performance considerations

Decoding introduces:

- CPU work to interpret variable-length sequences,
- branching overhead for surrogate detection/validation,
- potential memory allocations for decoded buffers (unless iterators are used).

Therefore:

- avoid decoding in hot loops where bytes suffice,
- prefer streaming or iterator patterns over bulk decode when processing large texts,
- leverage SIMD-optimized validation/transcoding libraries at boundaries when throughput matters.

7.2.6 Security and correctness lenses

Preventing overlong and invalid sequences

When decoding, the decoder must enforce well-formed UTF-8; overlong encodings and invalid code point ranges must be rejected or handled under the chosen policy (strict or replacement). Treat this as part of semantic correctness, not an optional step.

Decode under clear error policies

Design decode APIs to return status or policy-controlled behavior rather than silently ignoring errors:

```
struct DecodeResult {
    std::vector<char32_t> codepoints;
    bool well_formed;
};

DecodeResult decode_utf8_strict(std::u8string_view u8);
```

Systems that treat invalid sequences as code points without explicit policy risk security issues (e.g., bypassing filters or corrupting logic).

7.2.7 Summary guidance

Avoid decoding when

- processing raw protocol framing or transport,
- scanning or filtering at the byte level,
- storing or retrieving text without semantic inspection,
- operating on text where semantics do not affect correctness.

Decode when

- semantics involve code points or grapheme clusters,
- normalization, collation, or case folding is required,
- APIs explicitly promise semantic text behavior,
- data must be compared, validated, or segmented at a logical character level.

Delay or conditionally decode when

- performance matters and text may never be semantically inspected,
- partial scanning or quick checks can short-circuit full decode,
- your pipeline can separate validation from deep processing.

7.2.8 Final rule

> Only decode text when the operation you are performing logically depends on semantic interpretation of the encoded content; otherwise, treat text as validated bytes and delay costly decoding until it is genuinely needed.

This guideline balances correctness, performance, and clarity in modern Unicode-aware C++ systems.

7.3 Separation of Text Representation and Text Semantics

7.3.1 The central principle

Unicode-correct C++ systems succeed when they treat **text representation** and **text semantics** as separate layers with explicit contracts.

- **Representation** answers: *How is text stored and transmitted?* (bytes, UTF-8/UTF-16/UTF-32 code units, allocation, views, lifetime, ABI).
- **Semantics** answers: *What does the text mean to the user and to algorithms?* (code points, grapheme clusters, normalization, collation, casing, segmentation, security rules).

Confusing these layers is the root cause of most Unicode bugs in C++: wrong length, wrong indexing, broken slicing, incorrect comparisons, locale surprises, and security problems.

7.3.2 Why C++ forces you to care about this separation

C++ is intentionally explicit and performance-driven:

- Standard strings are containers of *code units*, not ”characters”.
- `size()` must remain fast and well-defined; Unicode ”character” counting is not.
- Indexing and slicing are representation operations; semantic correctness requires additional rules and often additional data.
- Many semantic operations depend on large, evolving Unicode data and locale tailoring, which are outside the scope of the standard library.

Therefore, professional systems make the separation explicit in architecture and APIs.

7.3.3 Representation layer: what you can safely assume

The representation layer deals with:

- **Code units and encodings**: UTF-8 as bytes, UTF-16 as 16-bit units, UTF-32 as 32-bit units.
- **Storage and performance**: cache behavior, copying vs views, memory ownership, streaming.
- **Boundary contracts**: validating input, transcoding at OS or protocol boundaries.

Representation invariants can be strong and testable:

- ”Internal text is valid UTF-8 code units”.
- ”All incoming external text is validated at ingest”.
- ”All Windows API boundaries are UTF-16 and conversions are centralized”.

Make representation intent explicit in types

Use distinct types for bytes vs UTF-8 text.

```
#include <string_view>

using BytesView = std::string_view;   // arbitrary bytes, no text promise
using Utf8View  = std::u8string_view; // UTF-8 code units by contract

void send_packet(BytesView payload);
void render_label(Utf8View text);
```

If you must accept `std::string` for ecosystem reasons, treat it as bytes unless validated and documented as UTF-8.

7.3.4 Semantic layer: what you cannot assume from storage

The semantic layer is where ”text” becomes meaningful. It includes at least four distinct notions that must not be conflated:

1) Code units (storage elements)

- UTF-8: bytes.
- UTF-16: 16-bit units (with surrogate pairs for non-BMP).
- UTF-32: 32-bit units (typically one scalar value per unit).

Operations on code units are cheap and deterministic, but they are not ”characters”.

2) Code points / Unicode scalar values (semantic atoms)

A Unicode *scalar value* (commonly represented as `char32_t`) is a decoded unit that excludes surrogate code points. Many algorithms that inspect scripts, categories, or do normalization are defined over code points/scalars.

3) Grapheme clusters (user-perceived characters)

What users perceive as one ”character” may consist of:

- base letter + combining marks,
- emoji sequences involving variation selectors and joiners,
- regional indicator pairs,
- complex-script clusters.

UI editing (cursor movement, delete/backspace, truncation-by-characters) must be grapheme-aware.

4) Locale- and language-dependent semantics

Some operations depend on language or locale:

- collation (sorting),
- case mapping rules (language-specific special cases),
- word boundaries and segmentation tailoring.

7.3.5 The key rule: representation operations must not pretend to be semantic operations

Wrong

- ”`u8.size()` is the number of characters”
- ”`u8[i]` is the i-th character”
- ”substring by byte offsets is safe for user-visible truncation”
- ”== on UTF-8 bytes implies user-visible equivalence”

Right

- Use code-unit operations for storage, transport, and ASCII-structured parsing.
- Decode when you need code points (property checks, normalization, security policies).
- Use grapheme segmentation for UI behaviors and ”character limits”.
- Use collation services for user-facing sorting and searching behavior.

7.3.6 A layered design model you can enforce

A professional architecture usually enforces four layers:

Layer A: Boundary ingestion (bytes → validated UTF-8)

- Accept `BytesView`.
- Validate as UTF-8 under a chosen policy (strict reject or replacement).
- Produce a strong internal type: validated `std::u8string`.

Layer B: Representation storage (UTF-8 code units)

- Keep text as UTF-8 for compactness and interoperability.
- Operate on bytes when semantics are not required.

Layer C: Semantic services (decode → normalize/segment/collate)

- Decode to scalar values or library-native form.
- Apply normalization, case folding, segmentation, collation as needed.
- Re-encode to UTF-8 if results must be stored/transmitted.

Layer D: Presentation semantics (UI)

- Cursor movement, selection, truncation: grapheme-aware.
- Rendering/shaping: handled by UI/text stack; do not approximate with code units.

7.3.7 How this separation shapes correct API design

Expose the right unit of abstraction

Prefer separate APIs for separate semantics, rather than one ambiguous ”string API”.

```
#include <string_view>
#include <cstddef>

using Utf8View = std::u8string_view;

std::size_t count_code_units(Utf8View u8);     // bytes
std::size_t count_code_points(Utf8View u8);    // decoded scalars
std::size_t count_graphemes(Utf8View u8);      // user-perceived clusters
```

Use policy objects for semantic operations

Semantic behavior is often policy-driven: normalization form, locale, collation strength, invalid-sequence policy.

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

struct TextPolicy {
    bool normalize_on_ingest = false;  // e.g., NFC
    bool case_fold_for_keys  = false;  // caseless matching
    std::string locale = "root";       // explicit locale selection (if used)
};

std::u8string canonicalize_for_key(std::u8string_view u8, const TextPolicy& p);
```

This makes semantics explicit and avoids hidden global-locale dependencies.

7.3.8 Practical examples of representation vs semantics

Example 1: Safe truncation for storage vs for UI

Storage truncation If you need a maximum byte size (e.g., protocol limit), you may truncate by bytes *only if you cut at a valid UTF-8 boundary*. This is a representation constraint.

UI truncation If you need ”max 20 characters” for display, that usually means grapheme clusters. This is semantic and requires segmentation.

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

// Representation-level: truncate to max_bytes, but ensure the result ends at a code-point
↪  boundary.
std::u8string truncate_utf8_by_bytes(std::u8string_view u8, std::size_t max_bytes);

// Semantic-level: truncate to max grapheme clusters (requires a Unicode segmentation
↪  library).
std::u8string truncate_by_graphemes(std::u8string_view u8, std::size_t max_graphemes);
```

Mixing these two requirements is a common source of broken UIs and corrupted text.

Example 2: Equality and keys

Representation equality Byte-equality is a cheap representation check: identical code units.

Semantic equality User-visible or identifier equality may require:

- normalization (canonical equivalence),
- case folding (caseless comparison),
- optional script restrictions (security policies),

- locale-independent rules for keys (to avoid surprises).

```cpp
#include <string_view>

bool bytes_equal(std::u8string_view a, std::u8string_view b) { return a == b; }

// Semantic equality is policy-driven and may not be stable unless you define the policy
↪  and data version.
bool semantic_equal_for_keys(std::u8string_view a, std::u8string_view b);
```

Example 3: Searching

Byte search Searching for an ASCII delimiter or token in UTF-8 bytes (e.g., `' '`, `','`, `\n`) is representation-safe and fast.

Semantic search Searching for "the same letter" or doing case-insensitive search across languages is semantic:

- case folding vs upper/lower,
- normalization impacts matches,
- collation-tailored comparisons may be required for user expectations.

7.3.9 What to centralize to protect the separation

The separation becomes robust when these are centralized:

1) Boundary validation

All external text must pass through a small ingestion funnel.

2) Transcoding gateways

All UTF-8 ↔ UTF-16/UTF-32 conversions live in one module.

3) Semantic services

Normalization, collation, segmentation live behind clear interfaces (often backed by ICU or similar libraries).

4) Policy definitions

One place defines:

- invalid UTF-8 policy (strict vs replacement),
- normalization strategy (on ingest vs on compare),
- key canonicalization rules (case folding, normalization form),
- locale selection rules (avoid hidden global state).

7.3.10 A minimal reference design you can apply immediately

```
#include <string_view>
#include <string>
#include <optional>

enum class Utf8Policy { StrictReject, ReplaceWithFFFD };

struct Utf8Text {
    std::u8string data; // invariant: valid UTF-8 (under the chosen policy)
};

struct TextSemantics {
```

```
    // semantic ops (backed by a Unicode library)
    std::u8string normalize_nfc(std::u8string_view u8) const;
    std::u8string case_fold(std::u8string_view u8) const;
    std::size_t   grapheme_count(std::u8string_view u8) const;
};

bool validate_utf8(std::string_view bytes);
std::u8string replace_invalid_utf8(std::string_view bytes);

std::optional<Utf8Text> ingest_text(std::string_view bytes, Utf8Policy p) {
    if (p == Utf8Policy::StrictReject) {
        if (!validate_utf8(bytes)) return std::nullopt;
        return Utf8Text{ std::u8string(reinterpret_cast<const char8_t*>(bytes.data()),
                                       reinterpret_cast<const char8_t*>(bytes.data()) +
                                       ↪  bytes.size()) };
    } else {
        return Utf8Text{ replace_invalid_utf8(bytes) };
    }
}

// Key canonicalization is semantic (policy-driven), not a property of the stored
↪  representation.
std::u8string make_lookup_key(std::u8string_view u8, const TextSemantics& sem) {
    auto n = sem.normalize_nfc(u8);
    return sem.case_fold(n);
}
```

This design keeps representation invariants in `Utf8Text` and pushes meaning-bearing transformations into `TextSemantics`.

7.3.11 Summary

Separating text representation from text semantics is the most important best practice for Unicode-safe C++ systems:

- Representation is about **code units, storage, transport, and explicit boundaries**.
- Semantics is about **code points, grapheme clusters, normalization, collation, casing, and locale**.
- Standard strings and `size()/operator[]` are representation tools; they cannot be treated as semantic tools.
- Correct systems enforce this separation through strong types, centralized gateways, explicit policies, and dedicated semantic services.

Once the separation is architectural, Unicode correctness becomes predictable, testable, and maintainable across C++20–C++26 codebases.

7.4 Testing, Validation, and Long-Term Maintenance

7.4.1 Why Unicode quality is a maintenance problem, not a one-time fix

Unicode correctness is not achieved once and then ”done.” It is a long-term maintenance discipline because:

- Unicode algorithms depend on evolving data (new scripts, updated properties, revised segmentation rules).
- Locale behavior changes with data updates (collation tailoring, case rules, CLDR-derived behavior).
- Your dependencies (ICU, platform libraries, compilers, standard library) evolve independently.
- Bugs are often input-dependent and appear only under specific languages, combining sequences, emoji clusters, or ill-formed byte patterns.

Therefore, professional Unicode handling requires a deliberate test strategy, explicit validation policies, and upgrade procedures that treat Unicode behavior as a compatibility surface.

7.4.2 Define invariants and enforce them with tests

Before writing tests, define *invariants* your system guarantees. Common production invariants:

Encoding invariants

- ”All internal text objects are well-formed UTF-8.”
- ”All platform boundaries are transcoded explicitly in one gateway module.”
- ”All API parameters typed as UTF-8 views are validated at ingest.”

Semantic invariants

- ”All keys used for lookup are canonicalized by (policy) normalization and (policy) case folding.”
- ”UI cursor movement and deletion operate on grapheme clusters, not bytes.”
- ”User-facing sorting uses locale-aware collation, not byte ordering.”

Once these invariants are written down, they become testable contracts.

7.4.3 Validation strategy: strict vs replacement and how to test it

Pick one policy per boundary

Every ingestion boundary (files, sockets, IPC, user input) must specify one of:

- **Strict reject**: invalid UTF-8 is an error.
- **Replacement**: invalid subsequences are replaced with U+FFFD.
- **Preserve bytes**: treat as bytes, not text, and do not call it UTF-8 text.

Tests for boundary validation

Create a corpus of invalid UTF-8 patterns and ensure behavior is consistent across builds:

- isolated continuation bytes,
- truncated multi-byte sequences at end-of-buffer,
- overlong sequences,
- encoded surrogate code points (invalid scalar values),

- code points beyond Unicode range,
- mixed valid and invalid segments.

```
#include <string_view>
#include <vector>
#include <cassert>

bool validate_utf8(std::string_view bytes); // your validated UTF layer

static void test_invalid_utf8_rejected() {
    const std::vector<std::string> samples = {
        "\x80",                 // lone continuation
        "\xC2",                 // truncated 2-byte sequence
        "\xE2\x82",             // truncated 3-byte sequence
        "\xF0\x9F\x92",         // truncated 4-byte sequence
        "\xC0\xAF",             // overlong encoding (example pattern)
        "\xED\xA0\x80",         // UTF-8 encoding of surrogate (invalid scalar)
        "\xF4\x90\x80\x80"      // above max Unicode range
    };

    for (const auto& s : samples) {
        assert(!validate_utf8(std::string_view{s.data(), s.size()}));
    }
}
```

Important testing rule Do not assume that a platform library or string function will reject invalid UTF-8 for you. Validation must be explicit, and tests must verify *your* validation layer.

7.4.4 Golden tests for transcoding gateways

Transcoding (UTF-8 $\leftrightarrow$ UTF-16/UTF-32) is a correctness hotspot. Professional practice is to treat conversion functions as stable, heavily tested infrastructure.

Golden input/output pairs

Build a test set containing:

- ASCII-only samples,
- characters above U+007F but within BMP,
- characters outside BMP (requiring surrogate pairs in UTF-16),
- mixed sequences (ASCII + combining marks + emoji),
- edge cases near boundary values.

```
#include <string>
#include <cassert>

std::u16string utf8_to_utf16(std::u8string_view u8); // gateway
std::u8string  utf16_to_utf8(std::u16string_view u16); // gateway

static void test_roundtrip_utf8_utf16() {
    const std::u8string sample =
        u8"ASCII +          +         +        +    + e\u0301";

    auto u16 = utf8_to_utf16(sample);
    auto back = utf16_to_utf8(u16);

    // Round-trip should preserve well-formed UTF-8 content exactly if policy is strict and
    ↪  lossless.
    assert(back == sample);
}
```

Policy note Round-trip equality is expected only when:

- input is well-formed,

- transcoding is lossless,
- no normalization or replacement is applied.

If replacement or normalization is part of your pipeline, then your golden tests must reflect that policy instead.

7.4.5 Semantic test categories that catch real bugs

1) Normalization tests

Normalization changes representation while preserving intended equivalence. Tests should cover:

- canonical equivalence pairs,
- combining marks sequences,
- ensuring your chosen normalization form is applied consistently at the chosen stage (ingest vs compare).

```
#include <string>
#include <cassert>

std::u8string normalize_nfc(std::u8string_view u8);

static void test_normalization_invariant() {
    // "e" + combining acute accent
    const std::u8string decomposed = u8"e\u0301";
    // precomposed "é"
    const std::u8string composed   = u8"\u00E9";

    assert(normalize_nfc(decomposed) == normalize_nfc(composed));
}
```

2) Grapheme segmentation tests (UI correctness)

UI bugs frequently come from treating code points as characters. Test:

- combining sequences (base + marks),
- emoji ZWJ sequences,
- flags (regional indicator pairs),
- skin-tone modifiers and variation selectors.

```
#include <cassert>
#include <cstddef>

std::size_t grapheme_count(std::u8string_view u8); // semantic service

static void test_grapheme_clusters() {
    // "e" + combining acute is one user-perceived character.
    assert(grapheme_count(u8"e\u0301") == 1);

    // Many emoji sequences are one grapheme cluster despite multiple code points.
    // The exact examples depend on the Unicode version and segmentation rules your library
    ↪  implements.
}
```

Maintenance note Grapheme boundary rules can evolve with Unicode versions. Your tests should be aligned with the Unicode data version shipped by your semantic library (e.g., ICU). See Section 7.4.8.

3) Collation and locale tests

Sorting and comparison must be tested with:

- a fixed locale configuration,
- a defined collation strength,
- stable expectations if ordering is used in user-visible behavior.

Do not use the process global locale as a hidden dependency; tests must construct the locale policy explicitly.

4) Security tests for identifiers

If you handle identifiers (usernames, domains, keys), build tests for:

- mixed-script inputs,
- confusable characters,
- normalization + case folding applied as documented,
- rejection behavior for invalid or policy-violating strings.

7.4.6 Fuzzing and property-based testing

Unicode correctness benefits greatly from automated generation of edge cases.

Fuzz targets

Good fuzz targets include:

- UTF-8 validation function,
- UTF-8 decoder/iterator,
- transcoding gateways,

- normalization and case folding wrappers,
- tokenizers or parsers that accept text input.

Properties to assert

- **No crashes**: ill-formed input must never crash.
- **Determinism**: same input and policy produce same output.
- **Round-trip**: for valid inputs, encode(decode(x)) $= x$ (when no normalization is applied).
- **Boundary stability**: partial chunks plus carry buffer equals full input behavior for streaming decoders.

```
// Conceptual property check outline
// For fuzzing frameworks, the fuzzer provides arbitrary bytes.
void fuzz_target(std::string_view bytes) {
    // 1) validate under strict policy
    bool ok = validate_utf8(bytes);

    // 2) ensure strict decode fails only if ok == false (policy-specific)
    // 3) ensure replacement policy always returns well-formed UTF-8
}
```

7.4.7 Regression testing across upgrades

Unicode behavior is a compatibility surface. When you upgrade:

- ICU or another Unicode library,
- the standard library implementation,

- compiler or platform runtime,

you must expect potential changes in:

- collation order,
- segmentation boundaries,
- casing behavior,
- normalization corner cases and performance.

7.4.8 Version pinning and behavioral contracts

Professional systems should pin and record:

- the semantic library version (e.g., ICU release used),
- the Unicode data version embedded or packaged with it,
- locale data sources (commonly CLDR-derived datasets in ICU contexts).

Then define upgrade policies:

- **Permissive**: behavior may change; UI and tests adapt.
- **Strict**: behavior must remain stable; upgrades require migration of stored keys/collation keys.

If you persist derived artifacts (collation keys, canonicalized identifiers), treat them as versioned data. Upgrades may require rebuilding or migration.

7.4.9 Long-term maintenance patterns

1) Centralize text policies

Keep one module that defines:

- invalid-sequence policy,
- normalization strategy,
- key canonicalization rules (normalization + case folding),
- locale selection and collation configuration,
- allowable scripts/characters for identifiers.

This ensures consistent behavior across the codebase.

2) Keep conversion and semantic dependencies behind interfaces

Wrap ICU or any semantic library behind internal interfaces so that:

- the rest of the codebase does not depend on library-specific types,
- you can swap implementations or upgrade versions with localized changes,
- testing can use mocks or reference implementations for certain behaviors.

3) Maintain a curated Unicode test corpus

In addition to fuzzing, keep a curated corpus reflecting:

- languages relevant to your user base,
- scripts you support,

- known edge-case sequences that have caused bugs,
- security-relevant confusable examples (policy-appropriate).

The corpus should be versioned with the product and used for regression tests.

4) Monitor performance regressions

Unicode operations can be expensive. Maintain benchmarks for:

- validation throughput,
- transcoding throughput,
- normalization throughput,
- collation comparisons (and collation key generation),
- segmentation speed in UI-heavy paths.

Treat benchmark changes across upgrades as part of your acceptance criteria.

7.4.10 A minimal Unicode test harness structure

```
#include <string>
#include <vector>
#include <cassert>

struct UnicodePolicy {
    bool normalize_on_ingest = false;
    bool case_fold_for_keys  = false;
};

bool validate_utf8(std::string_view bytes);
```

```
std::u8string canonicalize_key(std::u8string_view u8, const UnicodePolicy& p);
std::size_t grapheme_count(std::u8string_view u8);

static void run_unicode_smoke_tests() {
    // 1) invalid UTF-8 must be rejected (strict boundary)
    assert(!validate_utf8("\x80"));

    // 2) key canonicalization must be stable under policy
    UnicodePolicy p;
    p.normalize_on_ingest = true;
    p.case_fold_for_keys  = true;

    auto k1 = canonicalize_key(u8"e\u0301", p);
    auto k2 = canonicalize_key(u8"\u00E9", p);
    assert(k1 == k2);

    // 3) grapheme behavior must match UI expectations for chosen library version
    assert(grapheme_count(u8"e\u0301") == 1);
}
```

This harness does not replace deep correctness suites, but it provides a practical, enforced baseline for continuous integration.

7.4.11 Summary

Professional Unicode handling requires a test and maintenance discipline:

- Define explicit invariants for encoding and semantics.
- Validate at boundaries under a documented policy (strict vs replacement).
- Treat transcoding gateways as critical infrastructure with golden tests.
- Test semantic behaviors (normalization, graphemes, collation) using curated edge cases.

- Use fuzzing and property-based testing to expose input-dependent failures.
- Pin and record library/data versions and treat upgrades as behavior changes.
- Centralize text policies and wrap semantic libraries behind interfaces.
- Monitor performance regressions with benchmarks.

With these practices, Unicode correctness becomes sustainable across C++20–C++26 deployments, library upgrades, and the ongoing evolution of Unicode itself.

Conclusion

Unicode in C++ as an Engineering Responsibility, Not a Language Defect

Reframing the problem

Unicode difficulties in C++ are frequently mischaracterized as a failure of the language itself. This conclusion is inaccurate. The challenges arise from a mismatch between:

- the complexity and evolving nature of Unicode text semantics, and
- the design goals of C++ as a systems programming language that prioritizes explicitness, performance predictability, and long-term stability.

C++ does not attempt to hide complexity behind opaque abstractions. Instead, it exposes text representation clearly and requires engineers to make deliberate, explicit decisions about encoding, validation, semantics, and policy.

What C++ deliberately guarantees

From C++20 through C++26, the language and standard library provide:

- precise control over text *representation* (code units and storage),

- explicit types to express encoding intent (notably `char8_t` and UTF-8 string/view types),
- zero-overhead abstractions that do not impose semantic costs unless explicitly requested,
- strong compatibility guarantees across decades of existing code.

These guarantees are essential for operating systems, databases, compilers, game engines, embedded systems, and performance-critical infrastructure. None of them are compatible with a hidden, implicit, or globally enforced Unicode semantic model.

What C++ intentionally does not guarantee

Equally important is what C++ does *not* promise:

- it does not equate ”string” with ”text”,
- it does not define a universal ”character” abstraction,
- it does not silently normalize, case-fold, collate, or segment text,
- it does not impose locale or language rules implicitly.

These omissions are not defects. They are necessary constraints that preserve correctness, predictability, and performance across vastly different domains and platforms.

Unicode complexity is real and unavoidable

Unicode text is inherently complex:

- variable-length encodings,
- canonical equivalence and normalization,
- combining marks and grapheme clusters,

- locale- and language-specific rules,
- security concerns such as confusables and mixed-script identifiers,
- continuous evolution of the standard and its data.

No programming language can eliminate this complexity without either:

- sacrificing correctness in edge cases, or
- hiding costs and policies in ways that eventually surprise engineers.

C++ chooses transparency over illusion.

Engineering responsibility: the correct mental model

Correct Unicode handling in C++ must be understood as an **engineering responsibility**. This responsibility includes:

- choosing a canonical encoding strategy (commonly UTF-8),
- validating text at system boundaries under a documented policy,
- separating representation from semantics in architecture and APIs,
- centralizing transcoding and semantic services,
- selecting appropriate libraries for normalization, segmentation, and collation,
- designing APIs that make encoding and semantic intent explicit,
- testing Unicode behavior as a long-term maintenance concern.

These responsibilities exist regardless of language choice. Languages that appear to ”solve” Unicode often do so by:

- embedding heavyweight runtime systems,
- enforcing implicit normalization or locale behavior,
- accepting hidden performance and memory costs.

C++ simply requires these trade-offs to be made consciously.

Modern C++: clearer intent, fewer footguns

The evolution from C++20 to C++26 significantly improves Unicode correctness by making intent explicit:

- `char8_t` removes ambiguity between bytes and UTF-8 code units,
- UTF-8 string and view types enable precise API contracts,
- legacy, underspecified facilities are de-emphasized,
- the ecosystem increasingly converges on UTF-8 as the interchange format.

These changes do not add hidden semantics; they reduce accidental misuse.

A concise engineering checklist

A Unicode-safe C++ system consistently follows these rules:

- bytes are not text until validated,
- encoding intent is expressed in types,
- UTF-8 is used for storage and transport by default,
- conversions are isolated in gateway modules,

- semantic operations are explicit and policy-driven,
- UI behavior is grapheme-aware,
- identifiers and security boundaries are treated specially,
- Unicode behavior is tested, versioned, and maintained.

None of these rules require changes to the C++ language. They require disciplined engineering.

Why blaming the language is counterproductive

Attributing Unicode difficulties to C++ obscures the real issues:

- unclear architectural boundaries,
- implicit assumptions about ”characters”,
- misuse of representation APIs for semantic tasks,
- lack of testing and policy definition.

Blame shifts attention away from the necessary design decisions and leads to fragile systems regardless of language choice.

The correct conclusion

Unicode support in C++ is not ”missing.” It is *deliberately layered*:

- the language and standard library provide explicit, efficient representation tools,
- semantic correctness is achieved through architecture, policy, and appropriate libraries,
- engineers retain full control over performance, behavior, and compatibility.

This model aligns with C++’s core philosophy: **do not pay for what you do not use, and never hide costs or semantics from the programmer**.

Final statement

Unicode correctness in C++ is a test of engineering maturity, not a measure of language deficiency. When representation and semantics are cleanly separated, policies are explicit, and boundaries are well-defined, C++ is fully capable of handling modern Unicode text correctly, efficiently, and sustainably.

The responsibility is not on the language to guess intent—but on the engineer to design it.

Appendices

Appendix A — Unicode Terminology Reference

This reference defines key terms used throughout this booklet. Precision in terminology is essential for understanding Unicode's complexity and for communicating encoding and semantic requirements correctly in C++ systems.

Code Unit

A **code unit** is the minimal unit of storage in a given encoding form. In UTF-8 a code unit is an 8-bit byte. In UTF-16 a code unit is a 16-bit value. In UTF-32 a code unit is a 32-bit value. Code units are not inherently characters; they are representation elements that must be decoded to derive semantic meaning.

Code Point

A **code point** (Unicode scalar value) is an abstract numerical value in the range `U+0000` to `U+10FFFF` assigned by the Unicode standard to a discrete character-like entity. Code points exclude the surrogate code point range. Code points are semantic atoms: they represent the conceptual unit of Unicode text before grapheme or normalization treatment.

Scalar Value

A **scalar value** is a code point that is not a surrogate. The Unicode standard defines scalar values precisely, and many APIs use this term to distinguish from surrogate code points that exist only in UTF-16 encoding.

Encoding

An **encoding** is a mapping between abstract code points and concrete sequences of code units. Common encodings include UTF-8, UTF-16, and UTF-32. Encoding is a representation concern; decoding is the process of mapping code units back to code points (or reporting errors under a chosen policy).

UTF-8

UTF-8 (Unicode Transformation Format, 8-bit) is a variable-length encoding form in which code points are encoded as sequences of one to four 8-bit code units. It is backward-compatible with ASCII and is the dominant interchange and storage format in modern text systems.

UTF-16

UTF-16 is a variable-length encoding form in which most common code points are represented as a single 16-bit code unit, while supplementary characters (above `U+FFFF`) are represented as surrogate pairs (two 16-bit code units). UTF-16 is common in some platform ABIs and legacy APIs.

UTF-32

UTF-32 is a fixed-length encoding where each code point is represented by a single 32-bit code unit. It simplifies code point indexing at the cost of increased memory footprint compared to

UTF-8 or UTF-16.

Well-Formed and Ill-Formed Sequences

A sequence of code units is **well-formed** if it conforms to the syntax rules of the encoding (e.g., proper UTF-8 byte sequences with no overlong encodings or illegal scalar values). An **ill-formed** sequence violates these rules. Validation is the process of detecting ill-formed sequences and applying a policy (strict reject, replacement, or round-trip preservation).

Normalization

Normalization is the process of transforming text into a canonical form where semantically equivalent code point sequences are represented consistently. Common normalization forms include NFC (Normalization Form C), NFD (Normalization Form D), NFKC, and NFKD. Normalization is a semantic operation that applies after decoding.

Combining Character

A **combining character** is a code point that attaches to a base character to modify its appearance or semantics. Combining marks do not stand alone as user-visible characters; they combine with a preceding base character. Grapheme cluster algorithms group combining sequences appropriately.

Grapheme Cluster

A **grapheme cluster** is the smallest sequence of code points that a user perceives as a single character. Grapheme clusters can consist of a base code point plus any number of combining marks or multi-code-point constructs (e.g., emoji sequences with zero-width joiners). Grapheme clusters are the unit for cursor movement, deletion, and other user-visible operations.

Locale

A **locale** defines cultural and linguistic conventions for text operations such as collation, number and date formatting, and localized casing rules. Locale does not affect encoding; it affects interpretation and ordering of decoded text.

Collation

Collation refers to locale-aware comparison and sorting of text. Collation algorithms consider language-specific rules, accent weights, case weights, and other tailoring options to produce orderings that match human expectations in a given locale.

Case Folding

Case folding is the process of mapping strings to a form that supports case-insensitive comparison. Unlike simple upper/lower case conversion, case folding is defined by Unicode with rules that account for language and script-specific variants. Case folding is a semantic operation on decoded code points.

Bidirectional (Bidi) Text

Bidirectional text contains both left-to-right and right-to-left scripts (e.g., Latin and Arabic). The Unicode bidirectional algorithm defines how to reorder text for display while preserving logical order. Bidi processing is a semantic and presentation concern.

Code Unit Iterator

A **code unit iterator** traverses code units in memory without decoding them. It is suitable for storage- or transport-oriented operations but not for semantics that require decoding.

Code Point Iterator

A **code point iterator** decodes a sequence of code units into Unicode scalar values on the fly. It provides the semantic atoms needed for many Unicode algorithms while avoiding the cost of materializing a full decoded buffer.

Semantic Operation

A **semantic operation** on text is any computation that depends on the meaning of code points or sequences beyond raw code units. Examples include normalization, collation, grapheme segmentation, case-insensitive comparison, and locale-dependent behavior.

Representation Operation

A **representation operation** manipulates code units without decoding them into semantic units. Examples include copying, slicing by byte offsets (when boundaries are known), byte-level scanning for ASCII tokens, and buffer management.

Transcoding

Transcoding is converting between different encoding forms (e.g., UTF-8 to UTF-16 or vice versa). Transcoding requires decoding from the source encoding and re-encoding to the target encoding, typically under an error policy.

Error Handling Policy

An **error handling policy** specifies how invalid or ill-formed sequences are handled during decoding or transcoding. Common policies include:

- **Strict reject**: invalid sequences cause an error.

- **Replacement**: invalid subsequences are replaced with a designated replacement character (commonly U+FFFD).
- **Round-trip preservation**: original invalid sequences are treated as raw bytes and preserved for output without interpreting them as text.

Canonical Equivalence

Canonical equivalence refers to the Unicode property that different sequences of code points can represent the same semantic text (e.g., precomposed vs decomposed forms). Normalization ensures that canonically equivalent sequences are represented consistently for comparison and indexing.

Compatibility Equivalence

Compatibility equivalence is a broader equivalence relation that treats certain characters with compatibility variants as if they were similar (e.g., superscripts, subscripts) for certain applications. Compatibility normalization forms (NFKC, NFKD) reflect this broader equivalence.

Script

A **script** is a set of writing symbols used for a particular language or group of languages (e.g., Latin, Cyrillic, Arabic). Scripts have properties defined in Unicode that affect segmentation and rendering.

Shaping

Shaping is the process of selecting glyph forms for display based on context (common in complex scripts such as Arabic or Indic scripts). Shaping is part of text layout and rendering,

which is beyond core encoding and decoding layers.

Text Boundary

A **text boundary** is a position between text units defined by segmentation rules. Boundaries can be defined at the level of grapheme clusters, words, sentences, or lines. Unicode boundary rules are specified in UAX #29 (Unicode Text Segmentation).

Unicode Version

The **Unicode Version** refers to a specific release of the Unicode Standard that defines code points, properties, normalization data, collation defaults, and segmentation rules. Libraries that implement Unicode often specify which Unicode version they conform to.

UAX and UTS

Unicode Standard Annexes (UAX) and **Unicode Technical Standards (UTS)** are supplementary documents to the core Unicode Standard defining algorithms and data formats. Examples include UAX #9 (Bidirectional Algorithm), UAX #29 (Text Segmentation), and UTS #46 (IDNA).

Summary of term relationships

- **Representation** covers code units, encoding, and storage.
- **Decoding** maps representation to semantic units (code points).
- **Semantic operations** act on decoded content under policy.
- **Errors** and policies define how ill-formed sequences are handled.

- **Normalization and collation** define how semantically equivalent text is treated in comparison and indexing.
- **Grapheme clusters** and boundary rules define user-perceived text units.

This terminology reference supports precise discussion and engineering of Unicode-aware C++ systems from C++20 to C++26.

Appendix B — Encoding Comparison Summary

This appendix summarizes the practical engineering differences between common Unicode encoding forms in C++ systems. The purpose is not to promote a single encoding for all layers, but to clarify *what each encoding optimizes for*, where it fits best, and which pitfalls recur in real implementations.

High-level comparison (engineering view)

- **UTF-8**: Best default for storage, transport, file formats, protocols, and most application-layer text. Variable-length bytes, ASCII compatible.
- **UTF-16**: Common at Windows API boundaries and some legacy stacks. Variable-length 16-bit units; surrogate pairs for non-BMP.
- **UTF-32**: Useful for internal scalar-value processing and certain algorithms that benefit from direct code point indexing. Fixed-width 32-bit units; high memory cost.

Core properties table

Property	UTF-8	UTF-16	UTF-32
Code unit size	8-bit (byte)	16-bit	32-bit
Length form	Variable (1–4 bytes per scalar)	Variable (1 or 2 code units per scalar)	Fixed (1 code unit per scalar)
ASCII compatibility	Yes: ASCII maps directly to single bytes	Not byte-compatible; ASCII becomes 16-bit units	Not byte-compatible; ASCII becomes 32-bit units

Property	UTF-8	UTF-16	UTF-32
Endianness concerns	None	Yes (UTF-16LE/BE in some contexts)	Yes (UTF-32LE/BE in some contexts)
Non-BMP characters	Encoded directly using 4-byte sequences	Requires surrogate pairs (two 16-bit units)	Single 32-bit unit per scalar
Random access by “character”	Not constant-time (must decode)	Not constant-time in general (surrogates)	Constant-time by code point (not grapheme)
Indexing unit meaning	Byte index (code units)	16-bit unit index (code units)	32-bit unit index (code points/scalars)
Typical memory for ASCII-heavy text	Smallest	∼2x UTF-8	∼4x UTF-8
Typical memory for non-Latin scripts	Often efficient but depends on script distribution	Often efficient for BMP-heavy text	Largest
Best suited layers	Storage, transport, interfaces, logs, JSON, web	Windows ABI boundaries, legacy APIs	Algorithmic scalar processing, property checks
Common failure mode	Treating bytes as characters; slicing by bytes	Forgetting surrogate pairs; treating 16-bit units as characters	Assuming “one code point = one user character”; memory blowup

What the standard C++ string types actually represent

C++ string types are containers of code units:

- `std::string`: bytes (`char`). Encoding is not guaranteed.
- `std::u8string`: UTF-8 code units (`char8_t`) by intent.
- `std::u16string`: UTF-16 code units (`char16_t`).
- `std::u32string`: UTF-32 code units (`char32_t`).
- `std::wstring`: `wchar_t` code units; width/platform-dependent.

Engineering implication Even `std::u32string` does not automatically give ”characters” in the user-perceived sense. It gives scalar values; grapheme clusters can still be multi-code-point sequences.

Operational complexity summary

UTF-8 operational profile

- **Strengths**:
  - best interoperability and compact storage for many workloads,
  - stable across endianness and streaming,
  - ASCII-fast-path friendliness for parsers and protocols.
- **Costs**:
  - decoding required for code-point iteration,
  - substring operations require boundary awareness,
  - `size()` is bytes, not characters.

UTF-16 operational profile

- **Strengths**:
  - practical ABI match for Windows wide APIs,
  - can be relatively compact for BMP-heavy text.
- **Costs**:
  - surrogate pairs complicate indexing and iteration,
  - easy to mis-handle non-BMP (emoji and many historic scripts),
  - endianness matters in serialized forms.

UTF-32 operational profile

- **Strengths**:
  - simple iteration: each code unit corresponds to one scalar value,
  - direct indexing by code point (constant-time).
- **Costs**:
  - large memory footprint and cache pressure,
  - often slower in practice due to bandwidth and cache misses,
  - still not grapheme-aware, so UI semantics still require segmentation.

Cache behavior and throughput: a practical view

In real systems, performance is often dominated by memory bandwidth and cache locality:

- UTF-8 tends to be cache-friendly because it stores ASCII and common text compactly, improving throughput for scanning and tokenization.

- UTF-16 doubles code unit width; for ASCII-heavy text this increases memory traffic without semantic benefit.
- UTF-32 quadruples code unit width and commonly increases cache misses; it can make simple loops slower despite simpler indexing.

Key lesson Choose encoding by *layer*:

- storage/transport: compactness and interoperability dominate (UTF-8),
- OS boundaries: ABI dictates (UTF-16 on Windows),
- semantic algorithms: decoding to scalars may help locally (UTF-32 transiently).

Error handling and validation implications

All encodings can be ill-formed:

- UTF-8: invalid lead/continuation patterns, overlong sequences, invalid scalars.
- UTF-16: unpaired surrogates, invalid sequences.
- UTF-32: values outside Unicode range or surrogates (invalid scalars).

Best practice is consistent across encodings:

- validate at boundaries,
- define strict vs replacement policy,
- isolate transcoding in gateway modules.

Typical best-practice mapping by layer

System Layer	Recommended Encoding Strategy
Network protocols / APIs	UTF-8 for text fields; validate on ingest; avoid BOM; define error policy
File formats / persistence	UTF-8 (without BOM unless required); normalization policy documented (ingest vs compare)
Application core / business logic	Validated UTF-8 as canonical representation; decode on demand for semantics
Windows platform boundaries	Convert UTF-8 $\leftrightarrow$ UTF-16 in a centralized gateway
Indexing / search keys	Canonicalize using normalization + case folding (policy-driven); store derived keys versioned
UI editing and cursor movement	Operate on grapheme clusters using a segmentation engine; never on bytes

Minimal C++ examples: safe intent and explicit boundaries

Declare intent with UTF-8 views

```
#include <string_view>

using Utf8View  = std::u8string_view;
using BytesView = std::string_view;

void render_label(Utf8View text);
void send_payload(BytesView bytes);
```

Centralize transcoding

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

enum class TranscodePolicy { StrictReject, ReplaceWithFFFD };

std::u16string utf8_to_utf16(std::u8string_view u8, TranscodePolicy);
std::u8string  utf16_to_utf8(std::u16string_view u16, TranscodePolicy);
```

Common misconceptions addressed

- **Misconception:** UTF-32 solves ”Unicode correctness.”
  Reality: it only simplifies code point handling; grapheme clusters and locale semantics still exist.

- **Misconception:** UTF-16 is ”fixed width.”
  Reality: surrogate pairs make it variable length for non-BMP text.

- **Misconception:** UTF-8 is hard because it is variable length, so it should be avoided.
  Reality: variable length is manageable with correct iterators and boundary rules; UTF-8 is often the best engineering trade-off for storage and transport.

Summary

Encoding selection is a design decision that must be made per layer:

- **UTF-8** is the practical default for storage, transport, and interfaces due to compactness and interoperability.

- **UTF-16** is a boundary encoding primarily driven by platform ABIs, especially Windows.

- **UTF-32** is valuable for local semantic algorithms but expensive as a general storage format.

Correctness comes from explicit contracts, validation, and semantic processing where required, not from assuming any single encoding is a universal solution.

Appendix C — Practical Unicode Pitfalls Checklist

This appendix provides a comprehensive checklist of common pitfalls in Unicode handling and their corresponding engineering safeguards. It is designed for C++ engineers building robust text systems from C++20 through C++26.

Terminology sanity checks

Ensure your team agrees on definitions:

- **Code unit vs code point vs grapheme cluster:** Confirm which unit each API uses.
- **UTF-8 vs bytes:** Do not conflate a `std::string` with UTF-8 text unless validated.
- **Normalization:** Decide whether your system uses NFC, NFD, NFKC, or NFKD for comparison and storage.

Validation and encoding handling

- **Pitfall:** Accepting raw bytes as text without validation.
  Safeguard: Validate at the boundary under a documented policy (strict reject or replacement).
- **Pitfall:** Mixing unvalidated and validated text in the same code paths.
  Safeguard: Use strong types (`std::u8string`, `std::u8string_view`) to enforce invariants.
- **Pitfall:** Assuming every `char` is UTF-8.
  Safeguard: Treat `std::string` as bytes unless the contract declares UTF-8 and validation is applied.

Iteration and indexing

- **Pitfall:** Using byte indexing for character semantics.
  Safeguard: Decode to code points or use grapheme-aware iterators for semantic operations.

- **Pitfall:** Using `size()` as a count of user-visible characters.
  Safeguard: Clarify in API documentation the distinction between bytes, code points, and graphemes.

- **Pitfall:** Storing interim state as code units when semantic units are required later.
  Safeguard: Convert and store in the required semantic domain early if required repeatedly.

Slicing and substrings

- **Pitfall:** Slicing by arbitrary byte offsets.
  Safeguard: Clip at code-unit boundaries that correspond to valid sequences; use decoders to find boundary indices.

- **Pitfall:** Returning substrings that break surrogate pairs or multi-byte sequences.
  Safeguard: Validate boundaries before returning views or copies.

Transcoding pitfalls

- **Pitfall:** Implicitly converting between encodings in library overloads.
  Safeguard: Centralize transcoding in gateway modules with explicit overloads and documented error behavior.

- **Pitfall:** Assuming a Windows API will accept UTF-8.
  Safeguard: Convert to UTF-16 explicitly in a well-reviewed boundary function.

- **Pitfall:** Ignoring endianness when encoding to/from UTF-16 or UTF-32 on disk.
  Safeguard: Always define and document endianness for on-disk formats and handle it explicitly.

Normalization and equivalence

- **Pitfall:** Comparing text with different canonical representations.
  Safeguard: Normalize text to an agreed form (e.g., NFC) before comparison, indexing, or storage.

- **Pitfall:** Normalizing at inconsistent points (input vs compare vs output).
  Safeguard: Define a normalization policy and apply it uniformly or document exceptions.

- **Pitfall:** Forgetting canonical equivalence when generating collation keys.
  Safeguard: Integrate normalization into key generation where semantic comparison matters.

Collation and locale

- **Pitfall:** Using byte ordering for sorting user-visible text.
  Safeguard: Use locale-aware collation libraries and clear locale configuration objects.

- **Pitfall:** Relying on global locale state.
  Safeguard: Pass locale or collation policy explicitly to text comparison functions.

- **Pitfall:** Collation results that change after library upgrades without policy versioning.
  Safeguard: Pin Unicode/locale data versions or define migration practices.

Case mapping and folding

- **Pitfall:** Using simple ASCII case conversion for Unicode text.
  Safeguard: Use library support for Unicode case folding and language-sensitive casing rules.

- **Pitfall:** Confusing case mapping and case folding.
  Safeguard: Apply case folding when building language-independent keys; reserve case mapping for localized display transformations.

Grapheme and UI semantics

- **Pitfall:** Treating each scalar value as a user character.
  Safeguard: Use grapheme cluster segmentation for cursor movement, deletion, and user-visible truncation.

- **Pitfall:** UI truncation based on `size()` instead of graphemes.
  Safeguard: Provide grapheme-aware metrics and clipping functions.

Boundary interface design

- **Pitfall:** Exposing APIs that accept `const char*` without documented encoding.
  Safeguard: Use explicit types and document encoding expectations in API contracts.

- **Pitfall:** Encoding assumptions hidden in overloaded functions.
  Safeguard: Avoid overloads that mix byte buffers and text without explicit encoding parameters.

Security and identifiers

- **Pitfall:** Accepting lookalike characters in identifiers without policy controls.

Safeguard: Use confusable detection and script restrictions where appropriate for authentication or ACLs.

- **Pitfall:** Inconsistent normalization/case folding in security logic.
  Safeguard: Apply canonicalization in a consistent pipeline for keys and credentials.

Filesystem and OS integration

- **Pitfall:** Assuming filesystem paths are UTF-8 on all platforms.
  Safeguard: Explicitly transcode and validate at the OS boundary; treat POSIX paths as byte sequences with semantics defined by policy.

- **Pitfall:** Ignoring platform-specific encodings for registry, environment, and config APIs.
  Safeguard: Centralize OS boundary handling with documented conversion policies.

Testing and maintenance

- **Pitfall:** Limited test coverage against edge cases in Unicode data.
  Safeguard: Maintain a curated test corpus and employ property-based/fuzz testing for encodings, segmentation, and normalization.

- **Pitfall:** Ignoring Unicode version changes in data or libraries.
  Safeguard: Record and pin Unicode and locale data versions; test behavior on upgrades.

- **Pitfall:** Untested implicit dependencies on library behavior.
  Safeguard: Treat each semantic operation contract as testable, including boundary, normalization, and collation behavior.

Performance and scalability

- **Pitfall:** Blind decoding of text in hot loops.

Safeguard: Delay decoding until strictly necessary and use iterators or streaming patterns.

- **Pitfall:** Unbounded allocations due to naive grapheme segmentation on large texts.
  Safeguard: Use incremental and iterator-based segmentation where possible.
- **Pitfall:** Treating semantic operations as constant-time.
  Safeguard: Document and benchmark costs for normalization, collation, and segmentation; expose performance implications in API docs.

Documentation and developer education

- **Pitfall:** Unicode constraints assumed rather than documented.
  Safeguard: Document encoding and semantic policies in architecture docs and API contracts.
- **Pitfall:** Onboarding engineers on implicit encoding assumptions.
  Safeguard: Educate teams on code units vs code points vs graphemes; include a terminology reference in onboarding materials.

Summary checklist (quick reference)

1. Validate all external text at the ingress boundary.
2. Use explicit encoding types in APIs.
3. Separate representation from semantic logic.
4. Centralize transcoding gateways and avoid scattered casts.
5. Define and enforce normalization and collation policies.
6. Use grapheme-aware logic for UI semantics.

7. Test encoding, decoding, normalization, and segmentation exhaustively.
8. Pin and version Unicode/locale data to avoid unwanted behavioral changes.
9. Document encoding contracts, semantic expectations, and performance costs.
10. Educate developers on Unicode concepts and pitfalls.

This checklist mitigates the majority of real-world Unicode defects in professional C++ systems and forms the basis for long-term text correctness and maintainability.

References

This booklet intentionally distinguishes between **authoritative specifications** (normative standards, RFCs, and Unicode technical standards) and **engineering documentation** (compiler, standard library, and ICU implementation docs). Unicode-correct C++ work depends on reading the specification *and* understanding real implementation behavior.

ISO/IEC C++ Standards (C++11–C++26 drafts and papers)

1. **ISO/IEC 14882:2011** — *Information technology — Programming languages — C++ (C++11)*.
   Foundational for modern C++ memory model, `char16_t`/`char32_t`, and library baselines.

2. **ISO/IEC 14882:2014** — *C++14*.
   Incremental evolution; relevant when reading legacy code and migration constraints.

3. **ISO/IEC 14882:2017** — *C++17*.
   Widely deployed baseline; important for filesystem and core library evolution.

4. **ISO/IEC 14882:2020** — *C++20*.
   Introduces `char8_t` and clarifies UTF-8 intent in the type system; establishes key library and language facilities used in modern text pipelines.

5. **ISO/IEC 14882:2023** — *C++23*.
   Incremental library improvements; relevant for portability and evolving interfaces.

6. **ISO/IEC 14882 (C++26) Working Drafts and Committee Papers**.
   Use the WG21 Working Draft (document number `Nxxxx`) corresponding to the time of writing, and track feature papers (`PxxxxRyy`) for normative wording and design rationale.
   Engineering note: Always record the exact draft identifier used when claiming behavior, because wording and defect resolutions can change across drafts.

7. **WG21 (ISO C++ Committee) Mailings, Minutes, and Papers**.
   The committee document stream (`N`-papers and `P`-papers) is the authoritative public record for: rationale, adopted wording, defect reports, and post-adoption clarifications.
   Examples of paper classes used in this booklet: papers that introduce or revise text-related core language/library wording, papers that clarify encoding intent, and papers that standardize boundary behaviors.

Unicode Standard and Technical Reports

1. **The Unicode Standard (latest released version at time of writing)**.
   The core specification defining code points, scalar values, conformance, and the stability policies that govern evolution of the standard.

2. **Unicode Character Database (UCD)**.
   The authoritative data files backing Unicode properties (General Category, Script, Case mappings, normalization data, etc.). Many semantic operations are defined by algorithms that reference UCD data.

3. **UAX #9 — Unicode Bidirectional Algorithm**.
   Defines logical-to-visual ordering for mixed-direction text; essential for display pipelines and for avoiding incorrect assumptions about byte or code-point order in rendered text.

4. **UAX #15 — Unicode Normalization Forms**.
   Defines NFC/NFD and related forms, canonical equivalence, and the algorithmic basis for normalization and stable comparisons.

5. **UAX #29 — Unicode Text Segmentation**.
   Defines grapheme cluster, word, and sentence boundary rules; critical for UI correctness (cursor movement, deletion, selection, and user-perceived length).

6. **UTS #10 — Unicode Collation Algorithm (UCA)**.
   Defines default collation behavior and the framework for locale tailoring; necessary for correct user-facing sorting beyond byte order.

7. **UTS #39 — Unicode Security Mechanisms**.
   Defines confusables and guidance for identifier security policies; relevant for authentication, authorization keys, and user identifiers.

8. **UTS #46 — Unicode IDNA Compatibility Processing**.
   Defines processing relevant to Internationalized Domain Names; important when interfacing with DNS/IDNA ecosystems.

9. **ISO/IEC 10646 (Universal Coded Character Set)**.
   The ISO/IEC standard aligned with Unicode repertoire; frequently referenced by encoding specifications and RFCs.

ICU Documentation

1. **ICU (International Components for Unicode) Project Documentation**.
   The primary cross-platform implementation reference for Unicode algorithms and locale data integration in C/C++ and Java ecosystems.

2. **ICU User Guide / API References**.

Implementation details for:

- UTF-8/UTF-16 conversion and error policies,
- normalization APIs (NFC/NFD/NFKC/NFKD),
- collation (UCA-based with locale tailoring),
- break iteration (grapheme/word/sentence boundaries),
- case mapping and case folding,
- locale data behavior and versioning.

3. **ICU Release Notes and Versioning Policy**.
   Required for long-term maintenance: ICU version upgrades can change segmentation, collation, casing, and data-driven behavior due to Unicode/UCD and locale data updates. Record ICU and Unicode data versions used for reproducible behavior.

UTF-8 and UTF-16 Encoding Specifications

1. **RFC 3629** — *UTF-8, a transformation format of ISO 10646*.
   Defines UTF-8 encoding form, well-formedness constraints, and interoperability framing commonly used by network and storage specifications.

2. **RFC 2781** — *UTF-16, an encoding of ISO 10646*.
   Defines UTF-16 encoding form, surrogate pair structure, and serialization considerations (including endianness variants).

3. **RFC 5198** — *Unicode Format for Network Interchange*.
   Provides guidance for Unicode use in Internet protocols; helpful for designing UTF-8-based interfaces and for boundary validation expectations.

Compiler and Standard Library Documentation

1. **Clang / LLVM Documentation**.
   Source character set handling, execution character set, `char8_t` behavior, `u8` literal typing rules, diagnostics, and flags affecting encoding assumptions.

2. **GCC Documentation**.
   Character set and literal encoding options, warnings relevant to signedness and narrow/wide conversions, and behavior of library facets and locale-related text utilities.

3. **MSVC (Visual C++) Documentation**.
   Source file encoding expectations, compiler switches for UTF-8 source, wide/narrow literal behavior, and Windows-facing integration considerations.

4. **libstdc++ Documentation**.
   Implementation details of standard library strings, locales, and the state of deprecated or legacy conversion facilities.

5. **libc++ Documentation**.
   Library conformance notes and implementation-defined behaviors relevant to text handling and locales.

6. **Microsoft STL Documentation**.
   Conformance notes for C++20+ text-related types (`char8_t`, UTF string/view types), locale behavior, and platform-specific considerations.

Selected Academic and Industry References

1. **ISO C++ Core Guidelines (relevant sections)**.
   Guidance on type safety, boundary validation, and explicitness principles that apply directly to text (bytes vs text invariants, avoiding implicit conversions).

2. **Unicode Consortium Technical Notes, Reports, and Stability Policies**.
   For understanding what is stable across versions and what may change (important for long-term maintenance commitments).

3. **Industry-grade text processing practices (security and interoperability)**.
   Materials focusing on:

   - avoiding ambiguous encodings at interfaces,
   - validating and normalizing identifiers consistently,
   - grapheme-accurate UI behavior,
   - versioning of collation/normalization results when persisted.

4. **Reference implementations and test data**.
   When validating behavior, rely on:

   - Unicode-provided conformance test data (where applicable),
   - ICU test suites and behavioral documentation,
   - compiler/library conformance documentation for the specific toolchains used.

How to use these resources effectively (practical reading order)

1. Start with **Unicode terminology and invariants**: code units, code points, scalar values, grapheme clusters, and well-formedness.
2. Read the **encoding RFCs** (UTF-8/UTF-16) to internalize what is and is not valid input.
3. Use **UAX #15 and UAX #29** to understand why "characters" are not bytes or code points.
4. Use **ICU documentation** for real engineering APIs implementing normalization, segmentation, and collation.

5. Use **C++ standard + WG21 papers** for precise language/library guarantees and for explaining why the standard library is intentionally limited.

6. Use **compiler and standard library docs** to pin down toolchain behavior and portability constraints.