Gorgo Starting C++

April 11, 2026

Contents

3. Strings **2**
The `std::string` Type . . . 2
String Length . . . 2
Concatenation . . . 3
Comparing Strings . . . 3
Accessing Characters . . . 3
Iterating Through a String . . . 4
Finding and Extracting Substrings . . . 4
Replacing Parts of a String . . . 5
Unicode and UTF-8 . . . 5
String Input . . . 7
Converting Between Strings and Numbers . . . 8
Try It . . . 8
Key Points . . . 9
Exercises . . . 9

3. Strings

In Chapter 2 you met the `char` type for single characters and saw that arrays can store sequences of values. You could, in principle, use a `char` array to store text. But raw `char` arrays are painful: you must track the length yourself, you cannot easily resize them, comparing two of them with `==` compares pointers rather than content, and a single missing null terminator can crash your program. `std::string` solves all of these problems — it manages its own memory, knows its own length, and provides a rich set of operations for searching, slicing, and combining text. In this chapter you will learn how to create strings, manipulate them, and convert between strings and numbers.

The `std::string` Type

To use `std::string`, you need to include the `<string>` header.

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

int main()
{
    std::string song = "MMMBop";
    std::cout << song << std::endl;
    return 0;
}
```

You can create strings in several ways:

```
std::string empty;                  // empty string ""
std::string greeting = "Hola";      // initialized with a string literal
std::string copy = greeting;        // copy of another string
std::string repeat(5, '!');         // "!!!!!" --- 5 copies of '!'
```

The first form creates an empty string, not an uninitialized one. This is different from how numeric types work — an uninitialized `int` contains garbage, but a default-constructed `std::string` is always `""`.

Tip: Always include `<string>` when using `std::string`. Some compilers let you get away without it because `<iostream>` may pull it in, but that is not guaranteed.

String Length

You can find out how many characters are in a string with `.size()` or `.length()`. Their signatures are:

```
size_t size() const;
size_t length() const;
```

They do exactly the same thing.

```
std::string title = "Ice Ice Baby";
std::cout << title.size() << std::endl;    // 12
std::cout << title.length() << std::endl;  // 12
```

An empty string has a size of 0. You can check if a string is empty with `.empty()`, which returns `true` or `false`. Its signature is:

```
bool empty() const;
```

```
std::string nada;
if (nada.empty()) {
```

```
    std::cout << "nothing here" << std::endl;
}
```

Concatenation

You can join strings together with the + operator. Its signature is:

```
std::string operator+(const std::string& lhs, const std::string& rhs);
```

This is called concatenation.

```
std::string first = "Baby";
std::string second = " One More Time";
std::string hit = first + second;
std::cout << hit << std::endl;  // Baby One More Time
```

You can also append to an existing string with +=. Its signature is:

```
std::string& operator+=(const std::string& str);
```

```
std::string lyrics = "Bailamos";
lyrics += ", te quiero";
std::cout << lyrics << std::endl;  // Bailamos, te quiero
```

You can concatenate a `std::string` with a string literal or a single character, but you cannot concatenate two string literals together with +.

```
std::string ok = std::string("ba ") + "da ba";  // works
// std::string bad = "ba " + "da ba";            // ERROR
```

Trap: The expression `"hello" + " world"` does not compile because both sides are string literals (character arrays), not `std::string` objects. At least one side of + must be a `std::string`.

Comparing Strings

You can compare strings using the familiar comparison operators: ==, !=, <, >, <=, >=. Their signatures follow this pattern:

```
bool operator==(const std::string& lhs, const std::string& rhs);
bool operator<(const std::string& lhs, const std::string& rhs);
// similarly for !=, >, <=, >=
```

Comparison is done character by character using the characters' numeric values (their ASCII codes).

```
std::string a = "Hanson";
std::string b = "Vanilla Ice";
if (a < b) {
    std::cout << a << " comes first" << std::endl;
}
```

This prints `Hanson comes first` because `'H'` (72) is less than `'V'` (86).

Be aware that uppercase letters have lower ASCII values than lowercase letters. So `"Zebra"` is less than `"apple"` because `'Z'` (90) is less than `'a'` (97).

Accessing Characters

You can access individual characters in a string using `[]` or `.at()`. Their signatures are:

```
char& operator[](size_t pos);
char& at(size_t pos);
```

Both use zero-based indexing, just like arrays (as we saw in Chapter 2).

```
std::string song = "MMMBop";
std::cout << song[0] << std::endl;     // M
std::cout << song.at(3) << std::endl;  // B
```

The difference is what happens when you go out of bounds. Using `[]` with an index greater than the string's length is undefined behavior — anything could happen. Accessing index `size()` with `[]` returns the null character `'\0'`, but anything beyond that is dangerous. Using `.at()` with an invalid index throws an exception that stops your program with a clear error message.

```
std::string word = "Hola";
// word[99]       --- undefined behavior, might crash, might not
// word.at(99)    --- throws std::out_of_range exception
```

Tip: Use `.at()` when you are not sure the index is valid. The small performance cost is worth the safety.

You can also modify individual characters this way.

```
std::string shout = "hey!";
shout[0] = 'H';
std::cout << shout << std::endl;  // Hey!
```

Iterating Through a String

You can loop through every character in a string using a range-based `for` loop.

```
std::string word = "Iris";
for (char c : word) {
    std::cout << c << ' ';
}
std::cout << std::endl;
// Output: I r i s
```

You can also use a traditional index-based loop.

```
std::string word = "Iris";
for (size_t i = 0; i < word.size(); ++i) {
    std::cout << word[i] << ' ';
}
std::cout << std::endl;
```

Tip: Use `size_t` (or `std::string::size_type`) for the loop variable when comparing against `.size()`. Using `int` can cause a signed/unsigned comparison warning from the compiler.

Finding and Extracting Substrings

The `.find()` method searches for a substring and returns the position where it was found. Its signatures are:

```
size_t find(const std::string& str, size_t pos = 0) const;
size_t find(char c, size_t pos = 0) const;
```

If the substring is not found, it returns `std::string::npos`.

```
std::string line = "Ice Ice Baby";
size_t pos = line.find("Baby");
if (pos != std::string::npos) {
    std::cout << "found at position " << pos << std::endl;  // 8
}
```

You can also search for a single character.

```
size_t space = line.find(' ');  // finds first space at position 3
```

The `.substr()` method extracts a portion of the string. Its signature is:

```
std::string substr(size_t pos = 0, size_t count = npos) const;
```

It takes a starting position and an optional length.

```
std::string song = "Baby One More Time";
std::string part = song.substr(5, 3);   // "One"
std::string rest = song.substr(9);      // "More Time"
```

If you omit the length, `.substr()` returns everything from the starting position to the end of the string.

Replacing Parts of a String

The `.replace()` method replaces a range of characters with new text. Its signature is:

```
std::string& replace(size_t pos, size_t count, const std::string& str);
```

You specify the starting position, the number of characters to replace, and the replacement string.

```
std::string phrase = "doo wop";
phrase.replace(0, 3, "bee bop");
std::cout << phrase << std::endl;  // bee bop wop
```

A common pattern is to combine `.find()` and `.replace()` to find and replace a specific substring.

```
std::string msg = "press play";
size_t pos = msg.find("play");
if (pos != std::string::npos) {
    msg.replace(pos, 4, "stop");
}
std::cout << msg << std::endl;  // press stop
```

Unicode and UTF-8

ASCII covers English letters, digits, and basic punctuation, but the world is much bigger than that. What about Spanish (¿, ñ, á)? Japanese (こんにちは)? Greek (Σωκράτης)? Emojis (🎵)? None of those characters have ASCII codes.

Unicode is the modern standard that gives every character in every writing system — plus a growing pile of emojis, mathematical symbols, and historical scripts — its own number, called a **code point**. Unicode reserves room for over 1.1 million code points, of which roughly 160,000 are currently assigned. A code point is just an integer the same way `'A'` is 65, except now the integers can run all the way up past a million.

That immediately raises a question: how do you store a code point that big in a `char`? A `char` only holds 256 values, so it cannot hold 160,000 — let alone 1,100,000 — directly. The trick is to use *several* `char`s for one Unicode character.

Several encodings exist for doing this; the most common one — and the default for C++ string literals — is **UTF-8**. The full rules of UTF-8 are out of scope for this book, but the part you need to know is simple:

- Every ASCII character (code points 0–127) is exactly **1 byte** in UTF-8, and that byte is identical to the ordinary ASCII byte. Old ASCII text is automatically valid UTF-8.
- Every other Unicode character takes **2, 3, or 4 bytes** in a row.

So a `std::string` can hold any Unicode text you want; under the hood it just stores the UTF-8 bytes and lets the terminal worry about drawing the glyphs.

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

int main()
{
    std::string spanish  = "¡Hola, mundo!";
    std::string japanese = "こんにちは";
    std::string emoji    = "🎵";

    std::cout << spanish  << " has " << spanish.size()  << " bytes\n";
    std::cout << japanese << " has " << japanese.size() << " bytes\n";
    std::cout << emoji    << " has " << emoji.size()    << " bytes\n";
    return 0;
}
```

Output:

```
¡Hola, mundo! has 14 bytes
こんにちは has 15 bytes
🎵 has 4 bytes
```

The Spanish string has 13 visible characters but 14 bytes, because ¡ is a 2-byte UTF-8 character and the rest are 1-byte ASCII. The Japanese string has 5 visible characters but 15 bytes, because each of those characters takes 3 bytes. The single musical note emoji takes 4 bytes all by itself.

Trap: `std::string::size()` returns the number of *bytes*, not the number of characters. For pure ASCII text the two are the same, but for any string containing non-ASCII characters they differ. There is no built-in way in standard C++ to count "characters" the way a human would — doing it correctly requires a Unicode library. For most string work — passing strings around, writing them to a file, sending them over a network — bytes are exactly what you want, so this is rarely a problem in practice.

Wut: Indexing into a `std::string` with `[]` or `.at()` gives you one *byte*, not one *character*. For an ASCII-only string they are the same thing, but `japanese[0]` from the example above gives you the first byte of こ, which is meaningless on its own. Slicing UTF-8 safely is another job for a Unicode library.

Writing Unicode in Source Code

If your editor cannot type a particular character, or you want to keep a source file pure ASCII, you can write a Unicode character using a hexadecimal escape. C++ has three flavors:

Escape	Meaning
`\xHH`	one byte with the given hex value
`\uHHHH`	Unicode code point, 4 hex digits
`\UHHHHHHHH`	Unicode code point, 8 hex digits

`\u` and `\U` take a *code point* and the compiler emits the right UTF-8 bytes for you. `\x` is lower-level: it places exactly that byte in the string, so you have to know what UTF-8 expects. For characters in the Basic Multilingual Plane (code points up to U+FFFF), `\u` is enough; for anything above that — including most emojis — you need `\U` with 8 digits.

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

int main()
{
    std::string spanish_x = "\xC2\xA1Hola!";   // ¡ as raw UTF-8 bytes
    std::string spanish_u = "\u00A1Hola!";     // ¡ via code point
    std::string note      = "\U0001F3B5";      // 🎵 via 32-bit code point

    std::cout << spanish_x << "\n";
    std::cout << spanish_u << "\n";
    std::cout << note      << "\n";
    return 0;
}
```

Output:

```
¡Hola!
¡Hola!
🎵
```

The first two strings produce identical output because `\xC2\xA1` *is* the UTF-8 encoding of code point U+00A1.

Trap: `\x` keeps eating hex digits as long as it sees them. `"\xC2A1"` is *not* the two bytes `0xC2 0xA1` — it is the single (oversized) hex value `0xC2A1`, which is an error. You can break the run by splitting the literal in two (`"\xC2" "A1"` — C++ glues adjacent string literals together) or just use `\u00A1` instead and skip the manual UTF-8.

String Input

As you saw in Chapter 1, `std::cin >> variable` reads input, but for strings it stops at the first whitespace character (space, tab, or newline).

```
std::string name;
std::cout << "enter your name: ";
std::cin >> name;
// If user types "Vanilla Ice", name is just "Vanilla"
```

To read an entire line of input, use `std::getline()`. Its signatures are:

```
std::istream& getline(std::istream& is, std::string& str);
std::istream& getline(std::istream& is, std::string& str, char delim);
```

```
std::string full_name;
std::cout << "enter your full name: ";
std::getline(std::cin, full_name);
// If user types "Vanilla Ice", full_name is "Vanilla Ice"
```

Trap: If you mix `std::cin >>` and `std::getline()`, you can run into trouble. After `std::cin >>` reads a value, the newline character from pressing Enter is left in the input buffer. The next `std::getline()` sees that newline and returns an empty string. Fix this by adding `std::cin.ignore()` between them. Its signature is:
`std::istream& ignore(std::streamsize count = 1, int_type delim = EOF);`

```
int age;
std::string name;
std::cout << "age: ";
std::cin >> age;
std::cin.ignore();              // discard the leftover newline
std::cout << "name: ";
std::getline(std::cin, name);   // now this works correctly
```

Converting Between Strings and Numbers

Sometimes you have a number stored as text and need to use it as an actual number, or vice versa.

To convert a string to a number, use `std::stoi()` (string to int) or `std::stod()` (string to double). Their signatures are:

```
int stoi(const std::string& str, size_t* pos = nullptr, int base = 10);
double stod(const std::string& str, size_t* pos = nullptr);
```

```
std::string year_str = "1997";
int year = std::stoi(year_str);
std::cout << year + 1 << std::endl;  // 1998

std::string price_str = "9.99";
double price = std::stod(price_str);
```

To convert a number to a string, use `std::to_string()`. Its signatures are:

```
std::string to_string(int value);
std::string to_string(double value);
// also overloaded for long, long long, unsigned, float, etc.
```

```
int track = 7;
std::string label = "Track " + std::to_string(track);
std::cout << label << std::endl;  // Track 7
```

Trap: If the string does not contain a valid number, `std::stoi()` and `std::stod()` throw an exception. For example, `std::stoi("abc")` will crash your program unless you handle the exception.

Try It

Here is a small program to experiment with. Try modifying it to use different string operations.

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

int main()
{
    std::string song = "Bailamos";
    std::cout << song << " has " << song.size()
```

```
              << " characters" << std::endl;

    song += ", mi amor";
    std::cout << song << std::endl;

    size_t pos = song.find("mi");
    if (pos != std::string::npos) {
        std::cout << "found 'mi' at position "
                  << pos << std::endl;
    }

    std::string piece = song.substr(0, 8);
    std::cout << "first word: " << piece << std::endl;

    return 0;
}
```

Key Points

- `std::string` from `<string>` manages text for you — no manual memory management needed.
- Use `.size()` or `.length()` to get the number of characters.
- Concatenate with `+` and `+=`, but at least one operand must be a `std::string`.
- Compare strings with `==`, `<`, `>`, etc. — comparison is character by character using ASCII values.
- Access characters with `[]` (fast, no bounds check) or `.at()` (safe, throws on bad index).
- Use `.find()` to search and `.substr()` to extract portions of a string.
- `std::cin >>` reads one word; `std::getline()` reads a whole line.
- Convert between strings and numbers with `std::stoi()`, `std::stod()`, and `std::to_string()`.
- A `std::string` holds **UTF-8** bytes, so any Unicode text fits, but `.size()` counts bytes (not characters) and indexing returns a single byte.

Exercises

1. What is the difference between `std::cin >> str` and `std::getline(std::cin, str)`? When would you use each one?
2. What does the following code print?

   ```
   std::string a = "Ice";
   std::string b = a + " " + a + " Baby";
   std::cout << b << std::endl;
   std::cout << b.size() << std::endl;
   ```

3. What is `std::string("Hola").at(4)`? What about `std::string("Hola")[4]`?
4. What is the value of `pos` after this code runs?

   ```
   std::string s = "MMMBop ba duba dop";
   size_t pos = s.find("dop");
   ```

5. Where is the bug in this code?

   ```
   std::string greeting = "Hello, " + "world!";
   std::cout << greeting << std::endl;
   ```

6. Where is the bug in this program?

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

```cpp
int main()
{
    int count;
    std::string name;
    std::cout << "how many? ";
    std::cin >> count;
    std::cout << "your name? ";
    std::getline(std::cin, name);
    std::cout << name << ": " << count << std::endl;
    return 0;
}
```

7. What does this code print?

```cpp
std::string s = "Bailamos";
for (char c : s) {
    if (c == 'a') {
        std::cout << '@';
    } else {
        std::cout << c;
    }
}
std::cout << std::endl;
```

8. If `std::stoi("42abc")` returns 42, what do you think `std::stoi("abc42")` does?

9. Write a program that asks the user for their full name using `std::getline()`, then prints:
   - the number of characters in their name
   - their name in reverse (print each character from last to first)

10. **What does this print?**

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

int main()
{
    std::string lyric = "Mmm bop, ba duba dop";
    lyric.replace(0, 3, "Pop");
    std::cout << lyric << "\n";
    return 0;
}
```

11. **What does this print?**

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

int main()
{
    std::string title = "Wannabe";
    std::cout << title.substr(0, 4) << "\n";
    std::cout << title.substr(3) << "\n";
    std::cout << title.substr(3, 100) << "\n";
    return 0;
}
```

The third call passes a length that runs off the end of the string. Does it crash, throw, or do something else?

12. **Think about it:** What does this print?

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

int main()
{
    std::string a = "Wonderwall";
    std::string b = "wonderwall";
    std::cout << (a == b) << "\n";
    std::cout << (a < b) << "\n";
    return 0;
}
```

String comparison is case-sensitive. Why is `a < b` true even though the words are spelled the same? What would you change to make the two strings compare equal regardless of case?

13. **What does this print?**

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

int main()
{
    std::string s = "café";
    std::cout << s << " " << s.size() << "\n";
    return 0;
}
```

`c`, `a`, and `f` are ASCII, but `é` is U+00E9, which UTF-8 encodes as 2 bytes. What does `s.size()` report, and why is it not 4?