c string

A comprehensive guide to the std::string class in C++, covering constructors, modifiers, search functions, and best practices for efficient string manipulation.

Construction and Assignment

Constructors

`std::string()`	Default constructor. Creates an empty string. `std::string str;`

`std::string(const std::string& other)`	Copy constructor. Creates a string as a copy of another.
	`std::string str1 = "Hello";`
	`std::string str2 = str1;`

`std::string(const char* s)`	Constructs from a C-style string. `std::string str = "World";`

`std::string(const std::string& other, size_t pos, size_t len = npos)`	Constructs a substring from another string.
	`std::string str1 = "Example string";`
	`std::string str2 = str1.substr(0, 7); // "Example"`

`std::string(size_t n, char c)`	Constructs a string with `n` copies of character `c`. `std::string str(5, 'A'); // "AAAAA"`

`std::string(InputIterator first, InputIterator last)`	Constructs from a range specified by iterators.
	`std::vector<char> chars = {'H', 'i'};`
	`std::string str(chars.begin(), chars.end()); // "Hi"`

Assignment Operators

`operator=`	Assigns a new value to the string. Supports assignment from another `std::string`, a C-style string, or a single character.
	`std::string str1 = "Initial";`
	`str1 = "New value";`
	`str1 = 'X';`

`operator=` move assignment	Move assigns a new value to the string.
	`std::string str1 = "Initial";`
	`std::string str2 = std::move(str1);`

Capacity and Size

Size and Length

`size()` / `length()`	Returns the number of characters in the string. `size()` and `length()` are equivalent.
	`std::string str = "Hello";`
	`size_t len = str.size(); // len == 5`

`max_size()`	Returns the maximum possible number of characters a `std::string` can hold. This is system-dependent and generally very large. `size_t max = str.max_size();`

`empty()`	Returns `true` if the string is empty (i.e., `size() == 0`), `false` otherwise.
	`std::string str;`
	`bool isEmpty = str.empty(); // isEmpty == true`

Capacity Management

`capacity()`	Returns the number of characters the string has allocated space for. This can be greater than `size()` to allow for future growth without reallocation.
	`std::string str = "Hello";`
	`size_t cap = str.capacity();`

`reserve(size_t n)`	Reserves storage for at least `n` characters. This can prevent reallocations if you know the string will grow.
	`std::string str;`
	`str.reserve(100); // Reserves space for 100 characters`

`shrink_to_fit()`	Reduces the string's capacity to match its size, releasing any excess memory. This is a non-binding request, and the implementation is free to ignore it.
	`std::string str(100, 'A');`
	`str.resize(10);`
	`str.shrink_to_fit();`

Element Access

Character Access

`operator[]`	Provides direct access to the character at the specified index. No bounds checking is performed. Undefined behavior if the index is out of range.
	`std::string str = "Hello";`
	`char c = str[0]; // c == 'H'`
	`str[0] = 'J'; // str == "Jello"`

`at(size_t pos)`	Provides access to the character at the specified index with bounds checking. Throws `std::out_of_range` exception if the index is out of range.
	`std::string str = "Hello";`
	`char c = str.at(1); // c == 'e'`
	`try {`
	`  char d = str.at(100); // Throws std::out_of_range`
	`} catch (const std::out_of_range& e) {`
	`  std::cerr << "Out of range access" << std::endl;`
	`}`

`front()`	Returns a reference to the first character of the string. Undefined behavior if the string is empty.
	`std::string str = "Hello";`
	`char first = str.front(); // first == 'H'`

`back()`	Returns a reference to the last character of the string. Undefined behavior if the string is empty.
	`std::string str = "Hello";`
	`char last = str.back(); // last == 'o'`

String Operations and Search

Finding Substrings

`find(const std::string& str, size_t pos = 0)`	Finds the first occurrence of the substring `str` starting from position `pos`. Returns the index of the first character of the found substring, or `std::string::npos` if not found.
	`std::string str = "Hello World";`
	`size_t pos = str.find("World"); // pos == 6`

`rfind(const std::string& str, size_t pos = npos)`	Finds the last occurrence of the substring `str` searching backward from position `pos`. Returns the index of the first character of the found substring, or `std::string::npos` if not found.
	`std::string str = "Hello World World";`
	`size_t pos = str.rfind("World"); // pos == 12`

`find_first_of(const std::string& str, size_t pos = 0)`	Finds the first occurrence of any character from `str` starting from position `pos`. Returns the index of the found character, or `std::string::npos` if not found.
	`std::string str = "Hello World";`
	`size_t pos = str.find_first_of("eo"); // pos == 1 ('e')`

`find_last_of(const std::string& str, size_t pos = npos)`	Finds the last occurrence of any character from `str` searching backward from position `pos`. Returns the index of the found character, or `std::string::npos` if not found.
	`std::string str = "Hello World";`
	`size_t pos = str.find_last_of("eo"); // pos == 7 ('o')`

`find_first_not_of(const std::string& str, size_t pos = 0)`	Finds the first character that is *not* in `str` starting from position `pos`. Returns the index of the found character, or `std::string::npos` if not found.
	`std::string str = "123abc456";`
	`size_t pos = str.find_first_not_of("123"); // pos == 3 ('a')`

`find_last_not_of(const std::string& str, size_t pos = npos)`	Finds the last character that is *not* in `str` searching backward from position `pos`. Returns the index of the found character, or `std::string::npos` if not found.
	`std::string str = "123abc456";`
	`size_t pos = str.find_last_not_of("456"); // pos == 5 ('c')`

Substring and Comparison

`substr(size_t pos = 0, size_t len = npos)`	Returns a new string containing a substring of the original string, starting at position `pos` and with length `len`.
	`std::string str = "Hello World";`
	`std::string sub = str.substr(6, 5); // sub == "World"`

`compare(const std::string& str)`	Compares the string to another string lexicographically. Returns:
	• 0 if the strings are equal.
	• A negative value if the string is less than `str`.
	• A positive value if the string is greater than `str`.
	`std::string str1 = "apple";`
	`std::string str2 = "banana";`
	`int result = str1.compare(str2); // result < 0`