c string

A comprehensive cheat sheet covering essential methods of the std::string class in C++, with practical examples and best practices for efficient string manipulation.

Constructors, Assignment & Capacity

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 string.
	`std::string str = "Hello";`
	`std::string str2(str);`

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

`std::string(const char* s, size_t n)`	Constructs a string from the first `n` characters of a C-style string.
	`std::string str("Example", 3);`
	`// "Exa"`

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

`std::string(std::string::const_iterator first, std::string::const_iterator last)`	Constructs a string from a range specified by iterators.
	`std::string str = "Hello World";`
	`std::string str2(str.begin(), str.begin() + 5);`
	`// "Hello"`

Assignment

`str1 = str2`	Copy assignment. Assigns the contents of `str2` to `str1`.
	`std::string str1 = "One";`
	`std::string str2 = "Two";`
	`str1 = str2; // str1 is now "Two"`

`str = "C-string"`	Assigns a C-style string to `str`.
	`std::string str;`
	`str = "Example";`

`str = 'c'`	Assigns a single character to `str`.
	`std::string str;`
	`str = 'X'; // str is now "X"`

`str.assign(other_str)`	Assigns the content of `other_str` to `str`.
	`std::string str = "Initial";`
	`std::string other_str = "New Value";`
	`str.assign(other_str); // str is now "New Value"`

`str.assign(cstr, n)`	Assigns the first `n` characters from the C-style string `cstr` to `str`.
	`std::string str;`
	`str.assign("Some Text", 4);`
	`// str is now "Some"`

Capacity

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

`max_size()`	Returns the maximum possible size of the string, based on system limitations.
	`std::string str;`
	`size_t max = str.max_size();`

`capacity()`	Returns the number of characters the string has allocated space for. May be greater than `size()`.
	`std::string str = "Hello";`
	`size_t cap = str.capacity();`

`reserve(n)`	Reserves space for at least `n` characters, potentially avoiding reallocations. Does not change `size()`.
	`std::string str;`
	`str.reserve(100); // Reserves space for 100 characters`

`shrink_to_fit()`	Reduces the string's capacity to match its size, freeing up unused memory.
	`std::string str(100, 'A');`
	`str.resize(10);`
	`str.shrink_to_fit(); // Capacity may now be 10`

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

Element Access & Modifiers

Element Access

`str[index]`	Accesses the character at `index` (0-based). No bounds checking.
	`std::string str = "Hello";`
	`char c = str[0]; // c is 'H'`
	`str[1] = 'a';  // str is now "Hallo"`

`str.at(index)`	Accesses the character at `index` with bounds checking. Throws `std::out_of_range` if `index` is invalid.
	`std::string str = "Hello";`
	`char c = str.at(0); // c is 'H'`
	`// str.at(10);  // Throws std::out_of_range`

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

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

Modifiers

`str.append(other_str)`	Appends `other_str` to the end of `str`.
	`std::string str = "Hello";`
	`std::string other_str = " World";`
	`str.append(other_str); // str is now "Hello World"`

`str.append(cstr, n)`	Appends the first `n` characters from the C-style string `cstr` to `str`.
	`std::string str = "Start";`
	`str.append("Some Text", 4); // str is now "StartSome"`

`str.insert(pos, other_str)`	Inserts `other_str` into `str` at position `pos`.
	`std::string str = "Hello";`
	`str.insert(2, "XX"); // str is now "HeXXllo"`

`str.erase(pos, len)`	Erases `len` characters from `str` starting at position `pos`.
	`std::string str = "Hello World";`
	`str.erase(5, 6); // str is now "Hello"`

`str.replace(pos, len, new_str)`	Replaces `len` characters from `str` starting at position `pos` with `new_str`.
	`std::string str = "Hello World";`
	`str.replace(6, 5, "Moon"); // str is now "Hello Moon"`

`str.clear()`	Removes all characters from the string, making it empty.
	`std::string str = "Some Text";`
	`str.clear(); // str is now ""`

`str.push_back(c)`	Appends the character `c` to the end of the string.
	`std::string str = "Hel";`
	`str.push_back('l');`
	`str.push_back('o'); // str is now "Hello"`

`str.pop_back()`	Removes the last character from the string. Undefined behavior if the string is empty.
	`std::string str = "Hello";`
	`str.pop_back(); // str is now "Hell"`

`str.resize(n)`	Resizes the string to length `n`. If `n` is smaller than the current size, the string is truncated. If `n` is larger, the string is padded with null characters (or a specified fill character with the two-argument overload `resize(n, char)`.
	`std::string str = "Hello";`
	`str.resize(3); // str is now "Hel"`
	`str.resize(7, '!'); // str is now "Hel!!!!"`

String Operations & C-Style Conversion

String Searching

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

`str.rfind(sub, pos)`	Finds the last occurrence of substring `sub` in `str` searching backwards from position `pos` (defaults to end of string). Returns the index of the first character of the substring, or `std::string::npos` if not found.
	`std::string str = "Hello Hello World";`
	`size_t found = str.rfind("Hello"); // found is 6`

`str.find_first_of(chars, pos)`	Finds the first occurrence of any character from `chars` in `str` starting from position `pos`. Returns the index or `std::string::npos`.
	`std::string str = "Hello World";`
	`size_t found = str.find_first_of("lo"); // found is 2 ('l')`

`str.find_last_of(chars, pos)`	Finds the last occurrence of any character from `chars` in `str` searching backwards from `pos`. Returns the index or `std::string::npos`.
	`std::string str = "Hello World";`
	`size_t found = str.find_last_of("lo"); // found is 9 ('l')`

`str.find_first_not_of(chars, pos)`	Finds the first character in `str` (from `pos`) that is *not* in `chars`. Returns the index or `std::string::npos`.
	`std::string str = "Hello World";`
	`size_t found = str.find_first_not_of("Helo "); // found is 5 ('W')`

`str.find_last_not_of(chars, pos)`	Finds the last character in `str` (searching backwards from `pos`) that is *not* in `chars`. Returns the index or `std::string::npos`.
	`std::string str = "Hello World";`
	`size_t found = str.find_last_not_of("dlroW "); // found is 0 ('H')`

String Operations

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

`str.compare(other_str)`	Compares `str` lexicographically with `other_str`. Returns:
	• A negative value if `str < other_str`
	• 0 if `str == other_str`
	• A positive value if `str > other_str`
	`std::string str1 = "apple";`
	`std::string str2 = "banana";`
	`int result = str1.compare(str2); // result is negative`

`str.compare(pos, len, other_str)`	Compares a substring of `str` (starting at `pos` with length `len`) with `other_str`.
	`std::string str = "Hello World";`
	`std::string other_str = "World";`
	`int result = str.compare(6, 5, other_str); // result is 0`

C-Style String Conversion

`str.c_str()`	Returns a pointer to a null-terminated C-style string representing the contents of `str`. The pointer is valid as long as `str` exists and is not modified. **Important:** The returned pointer is invalidated if the `std::string` object is modified.
	`#include <iostream>`
	`#include <cstring>`
	`int main() {`
	`  std::string str = "Hello";`
	`  const char* cstr = str.c_str();`
	`  std::cout << std::strlen(cstr) << std::endl; // Output: 5`
	`  return 0;`
	`}`

`str.data()`	Returns a pointer to the underlying character array of the string. The returned pointer is *not* guaranteed to be null-terminated before C++17. From C++17 onward, it *is* guaranteed to be null-terminated, but modifying the string in any way invalidates the pointer. Use with caution.
	`std::string str = "Hello";`
	`const char* data = str.data();`

`std::string_view`	A non-owning view of a string-like object (including `std::string` and C-style strings). Provides read-only access without copying. Can be constructed from `str.data()` and `str.size()`.
	`#include <string_view>`
	`std::string str = "Hello World";`
	`std::string_view view(str.data(), 5); // "Hello"`

Iterators and Non-Member Functions

Iterators

`str.begin()`	Returns an iterator to the beginning of the string. `std::string str = "Hello"; for (auto it = str.begin(); it != str.end(); ++it) { std::cout << *it; } // Output: Hello`
`str.end()`	Returns an iterator to the end of the string. `std::string str = "Hello"; auto it = str.end(); --it; // Points to the last character 'o' std::cout << *it; // Output: o`
`str.rbegin()`	Returns a reverse iterator to the beginning of the reversed string (i.e., the last character). `std::string str = "Hello"; for (auto it = str.rbegin(); it != str.rend(); ++it) { std::cout << *it; } // Output: olleH`
`str.rend()`	Returns a reverse iterator to the end of the reversed string (i.e., one position before the first character). `std::string str = "Hello"; auto it = str.rend(); // Reverse end iterator`
`str.cbegin()`, `str.cend()`, `str.crbegin()`, `str.crend()`	Return const iterators, preventing modification of the string's characters through the iterator. `std::string str = "Hello"; for (auto it = str.cbegin(); it != str.cend(); ++it) { std::cout << *it; // Read-only access }`

Non-Member Functions / Operators

`str1 + str2`	String concatenation. Returns a new string that is the concatenation of `str1` and `str2`. `std::string str1 = "Hello"; std::string str2 = "World"; std::string result = str1 + str2; // result is "Hello World"`
`str + "C-string"`	Concatenation with a C-style string. `std::string str = "Value: "; std::string result = str + "123"; // result is "Value: 123"`
`str == other_str`, `str != other_str`, `str < other_str`, `str <= other_str`, `str > other_str`, `str >= other_str`	Comparison operators. Perform lexicographical comparison between strings. Return `true` or `false`. `std::string str1 = "apple"; std::string str2 = "banana"; bool isEqual = (str1 == str2); // false bool isLess = (str1 < str2);  // true`
`std::getline(istream, str)`	Reads a line from the input stream `istream` into the string `str`, up to the next newline character. The newline character is extracted but not stored in `str`. `#include <iostream> #include <string> int main() { std::string line; std::cout << "Enter a line: "; std::getline(std::cin, line); std::cout << "You entered: " << line << std::endl; return 0; }`

Best Practices and Common Pitfalls

- **Pre-allocate Memory:** Use `reserve()` if you know the approximate size of the string to avoid reallocations.
- **Avoid Unnecessary Copies:** Use `std::string_view` for read-only access to substrings to avoid creating copies.
- **Use `at()` for Safety:** Use `at()` when you need bounds checking to prevent potential crashes.
- **Be Careful with `c_str()`:** The pointer returned by `c_str()` becomes invalid if the `std::string` object is modified. Copy the string if you need to keep it around.
- **Consider SSO (Short String Optimization):** Small strings (typically up to 15-22 characters, depending on the implementation) are often stored directly within the `std::string` object itself, avoiding dynamic allocation. Be aware that this optimization exists and that memory allocation patterns might change as string size increases.