STRING MANIPULATION (IN-BUILT FUNCTION)

Constructor Functions

All constructors are in <string> (alias std::basic_string<char>).

- string()
    - Signature: string() noexcept;
    - Creates an empty string. size() == 0.
    - Complexity: constant.
- string(const string&)
    - Signature: string(const string& other);
    - Copy constructor — constructs a new string with the same contents as other.
    - Complexity: linear in other.size().
- string(const char*)
    - Signature: string(const char* s);
    - Constructs from a null-terminated C-string s. Copies until '\0'.
    - Throws: std::bad_alloc on allocation failure.
    - Complexity: linear in strlen(s).
- string(size_t count, char ch)
    - Signature: string(size_t count, char ch);
    - Constructs a string with count copies of character ch.
    - Complexity: linear in count.
- string(const string& str, size_t pos, size_t len = npos)
    - Signature: string(const string& str, size_t pos, size_t len = npos);
    - Constructs a string by taking up to len characters starting at pos in str.

- Throws: std::out_of_range if pos > str.size() (in some implementations; C++ standard requires pos <= size() or throws).
- Complexity: linear in number of characters copied.

🔹 **Capacity Functions**

- size()
  - Signature: size_type size() const noexcept;
  - Returns number of characters in the string (same as length()).
  - Complexity: constant.
- length()
  - Signature: size_type length() const noexcept;
  - Synonym for size().
- max_size()
  - Signature: size_type max_size() const noexcept;
  - Returns maximum possible length (implementation-dependent). Useful for sanity checks.
  - Complexity: constant.
- capacity()
  - Signature: size_type capacity() const noexcept;
  - Returns number of characters that can be held in the currently allocated storage (without reallocation).
  - Complexity: constant.
- resize(size_t count)
  - Signature: void resize(size_type count);
  - If count < size(), reduces length to count (truncates). If count > size(), appends count - size() default-inserted characters (value-initialized — '\0' for char).
  - Overload: resize(size_type count, char ch) — fills new characters with ch.
  - Throws: std::length_error if count > max_size(), std::bad_alloc on allocation failure.

    - Complexity: linear in max(count, old_size) when growing; constant when shrinking (plus potential destructor work).
- reserve(size_t new_cap = 0)
    - Signature: void reserve(size_type new_cap = 0);
    - Requests capacity at least new_cap. If new_cap <= capacity(), does nothing. If new_cap > max_size(), throws std::length_error.
    - Does not change size().
    - Complexity: amortized; allocation and copy if new_cap > capacity().
- shrink_to_fit()
    - Signature: void shrink_to_fit();
    - Non-binding request to reduce capacity to size. Implementation may or may not reallocate.
    - Complexity: may be linear if reallocation occurs.
- empty()
    - Signature: bool empty() const noexcept;
    - Returns true if size() == 0.
    - Complexity: constant.

🔹 **Element Access**

- operator[]
    - Signature: char& operator[](size_type pos); and const char& operator[](size_type pos) const;
    - Returns reference to character at pos. No bounds checking — accessing pos >= size() is undefined behavior.
    - Complexity: constant.
- at()
    - Signature: char& at(size_type pos); and const char& at(size_type pos) const;
    - Returns reference to character at pos, **with bounds checking** — throws std::out_of_range if pos >= size().

    - Complexity: constant.
- front()
    - Signature: char& front(); and const char& front() const;
    - Returns reference to first character. Undefined behavior if string is empty.
    - Complexity: constant.
- back()
    - Signature: char& back(); and const char& back() const;
    - Returns reference to last character. Undefined behavior if string is empty.
    - Complexity: constant.
- data()
    - Signature (pre-C++17): const char* data() const noexcept;
      Signature (C++17 and later): char* data() noexcept; and const char* data() const noexcept;
    - Returns pointer to the contiguous character array. Since C++11 it points to contiguous storage, and since C++17 non-const data() returns writable pointer and the string is null-terminated.
    - Use caution: modifying data() contents is allowed only for characters within size(); modifying past size() (the implicit null) is UB in older standards.
    - Complexity: constant.
- c_str()
    - Signature: const char* c_str() const noexcept;
    - Returns C-style null-terminated pointer to contents. Guaranteed null-terminated in all modern C++ standards.
    - Complexity: constant.

🔹 **Modifiers**

- clear()
    - Signature: void clear() noexcept;
    - Erases contents; size() becomes 0. Does not necessarily free capacity.

    - Complexity: linear in destructor calls for element types (for char, trivial), but conceptually constant.

- insert()
    - Several overloads:
        - iterator insert(const_iterator pos, char ch);
        - iterator insert(const_iterator pos, size_type count, char ch);
        - string& insert(size_type pos, const string& str);
        - string& insert(size_type pos, const char* s);
        - iterator insert(const_iterator pos, InputIt first, InputIt last);
        - etc.
    - Inserts characters at pos. If pos is an index, throws std::out_of_range for bad positions.
    - Complexity: linear in number of characters inserted plus movement of existing characters (so O(n) overall).
    - Throws: std::bad_alloc, possibly std::out_of_range.
- erase()
    - Overloads:
        - iterator erase(const_iterator pos);
        - iterator erase(const_iterator first, const_iterator last);
        - string& erase(size_type pos = 0, size_type len = npos);
    - Erases specified characters; invalidates iterators pointing to erased characters and beyond (note: specifics depend on implementation — generally iterators to the erased area are invalidated; references before erase remain valid).
    - Complexity: linear in the number of characters moved.
- push_back(char ch)
    - Signature: void push_back(char ch);
    - Appends ch to end. May reallocate.
    - Complexity: amortized constant.
- pop_back()

    - Signature: void pop_back();
    - Removes last character; undefined behavior if empty (C++11 and later require non-empty).
    - Complexity: constant.
- append()
    - Overloads mirror insert and operator+= patterns:
        - string& append(const string& str);
        - string& append(const char* s);
        - string& append(const char* s, size_type n);
        - string& append(size_type count, char ch);
        - template<class InputIt> string& append(InputIt first, InputIt last);
    - Appends content to the end. Throws std::bad_alloc on allocation failure.
    - Complexity: linear in appended length (plus potential reallocation).
- assign()
    - Overloads:
        - string& assign(const string& str);
        - string& assign(const char* s);
        - string& assign(size_type count, char ch);
        - template<class InputIt> string& assign(InputIt first, InputIt last);
    - Replaces contents with provided sequence.
    - Complexity: linear in new length.
- replace()
    - Overloads similar to erase+insert: replace a substring with another sequence.
        - string& replace(size_type pos, size_type len, const string& str);
        - iterator replace(const_iterator first, const_iterator last, const char* s);
        - etc.
    - Complexity: linear in the sizes of removed and inserted parts.

    - Throws: std::out_of_range when position is invalid, std::bad_alloc on allocation failure.

- swap()
    - Signature: void swap(string& other) noexcept; (and a std::swap specialization)
    - Exchanges contents with other in constant time — usually swaps internal pointers and size/capacity fields.
    - Complexity: constant; noexcept.

🔹 **String Operations**

- substr()
    - Signature: string substr(size_type pos = 0, size_type count = npos) const;
    - Returns a new string which is a substring starting at pos of length count (or until end).
    - Throws: std::out_of_range if pos > size().
    - Complexity: linear in returned substring length.
- copy()
    - Signature: size_type copy(char* dest, size_type count, size_type pos = 0) const;
    - Copies up to count characters starting at pos into dest. Does **not** null-terminate dest.
    - Throws: std::out_of_range if pos > size().
    - Returns number of characters copied.
    - Complexity: linear in characters copied.
- compare()
    - Overloads:
        - int compare(const string& str) const noexcept;
        - int compare(size_type pos1, size_type len1, const string& str) const;
        - int compare(const char* s) const;

    - Lexicographically compares strings. Returns <0, 0, or >0 for less/equal/greater respectively.
    - Complexity: linear in the compared prefix length.
- find()
    - Overloads: find substring, char, C-string, or using iterators; size_type find(const string& str, size_type pos = 0) const noexcept;
    - Returns the index of the first occurrence at or after pos, or string::npos if not found.
    - Complexity: naive implementations are O(n*m) worst-case; many implementations use efficient algorithms but worst-case is linear times pattern length.
- rfind()
    - Signature: size_type rfind(const string& str, size_type pos = npos) const noexcept;
    - Finds last occurrence (searches backward) before or at pos. Returns npos if not found.
    - Complexity: similar to find, depends on implementation.
- find_first_of()
    - Signature: size_type find_first_of(const string& chars, size_type pos = 0) const noexcept;
    - Finds first position >= pos where any character is in chars. Useful for searching any of several delimiters.
    - Returns npos if none found.
    - Complexity: O(n * k) naive, but for small chars it's effectively linear.
- find_last_of()
    - Signature: size_type find_last_of(const string& chars, size_type pos = npos) const noexcept;
    - Finds last position <= pos where any character is in chars. Returns npos if none found.
    - Complexity: linear in searched area.
- find_first_not_of()

    - Signature: size_type find_first_not_of(const string& chars, size_type pos = 0) const noexcept;
    - Finds first position where character is **not** in chars. Useful for skipping delimiters/whitespace.
    - Returns npos if none found.
- find_last_not_of()
    - Signature: size_type find_last_not_of(const string& chars, size_type pos = npos) const noexcept;
    - Finds last position where character is **not** in chars.
    - Returns npos if none found.

🔹 **Iterators**

All standard iterator semantics apply. std::string stores contiguous characters; iterators are random-access iterators.

- begin() / end()
    - iterator begin() noexcept; const_iterator begin() const noexcept;
    - iterator end() noexcept; const_iterator end() const noexcept;
    - begin() points to first character; end() one past last.
    - Complexity: constant.
- cbegin() / cend()
    - Returns const iterators (same as begin()/end() for const objects).
- rbegin() / rend()
    - Reverse iterator pair, iterates from last to first.
    - Complexity: constant.
- crbegin() / crend()
    - Const reverse iterators.

Notes:

- Because string is contiguous, you can safely pointer-arithmetic from &*begin() to access underlying array (but prefer data()/c_str() for raw pointer access).

- Invalidations: operations that reallocate or modify size/contents may invalidate iterators.

🔹 Numeric Conversions (in <string> / <cstdlib> as overloads)

These parse textual numeric representations; located in <string> (C++11 onwards).

- stoi(const string& str, size_t* idx = 0, int base = 10)
    - Converts string str to int. idx (if non-null) receives index of first unprocessed character.
    - Throws: std::invalid_argument if no conversion could be performed; std::out_of_range if value is outside representable int.
    - Complexity: linear in number of characters parsed.
    - Similar functions: stol, stoll, stoul, stoull (to long, long long, unsigned long, unsigned long long respectively).
    - Floating: stof, stod, stold convert to float, double, long double. Same exception behavior.
- to_string() (free function)
    - Signature: std::string to_string(int value); and overloads for numeric types.
    - Converts numeric values to their decimal string representation.
    - Complexity: depends on magnitude, but effectively linear in number of digits.

Notes:

- Parsing respects leading whitespace and optional sign; base parameter affects integer parsing.
- Use std::from_chars/std::to_chars (C++17 <charconv>) for non-throwing, faster conversions when available.

🔵 Algorithm Functions Used With Strings (<algorithm> / <cctype>)

These are STL algorithms or C character functions that are commonly applied to string contents. Include <algorithm>, <cctype> or <locale> as appropriate.

Character Handling (<cctype>)

All take/return int and expect unsigned char cast or EOF to avoid UB.

- toupper(int ch) / tolower(int ch)
    - Convert character to uppercase/lowercase according to the C locale. Use std::toupper(static_cast<unsigned char>(ch)).
    - For locale-aware conversions, use <locale>'s std::toupper with std::locale.
- isalpha(int ch), isdigit(int ch), isalnum(int ch), isspace(int ch)
    - Test character classes. Return non-zero if true.
    - **Important:** Always cast to unsigned char before calling to avoid undefined behavior for negative char.

String Manipulation Algorithms (<algorithm>)

These operate on iterators (so they work nicely with string::begin()/end()).

- reverse(begin, end)
    - Reverses order of elements in range [begin, end). Complexity linear.
- sort(begin, end)
    - Sorts characters in range. Complexity O(n log n) average/worst depending on implementation.
- unique(begin, end)
    - Moves duplicates to the end, keeps only unique *consecutive* values. Returns iterator to new logical end; usually used with erase to remove duplicates: s.erase(std::unique(s.begin(), s.end()), s.end());.
    - Complexity linear.
- transform(begin, end, out, unary_op)
    - Applies operation to each element; commonly used for toupper/tolower conversions: std::transform(s.begin(), s.end(), s.begin(), [](unsigned char c){ return std::toupper(c); });.
    - Complexity linear.
- count(begin, end, value)
    - Counts occurrences of value. Complexity linear.
- find(begin, end, value) (iterator version from <algorithm>)

    - Finds first element equal to value. Complexity linear.
- equal(begin1, end1, begin2) / lexicographical_compare(...)
    - equal checks elementwise equality; lexicographical_compare compares two ranges lexicographically (like compare()).

Notes:

- For character operations, be mindful of char signedness; use unsigned char casts.

🔵 Memory / Utility Functions (C headers)

From C standard headers — sometimes used for raw C-string operations or low-level memory ops; include <cstring>, <cstdlib>, <cstring>.

- memset(void* s, int c, size_t n)
    - Fills memory with byte value c for n bytes. Useful for raw buffers; not for std::string internals unless using data() carefully.
    - Complexity linear.
- memcmp(const void* s1, const void* s2, size_t n)
    - Compares n bytes; returns <0, 0, >0 per lexicographic comparison.
- memcpy(void* dest, const void* src, size_t n)
    - Copies n bytes from src to dest. Undefined if ranges overlap (use memmove when overlap possible).
- strlen(const char* s)
    - Returns length of null-terminated C-string (number of chars before '\0').
- strcpy(char* dest, const char* src)
    - Copies C-string including terminating null. Unsafe wrt buffer overflow — prefer strncpy or strcpy_s.
- strncpy(char* dest, const char* src, size_t n)
    - Copies at most n characters; may not null-terminate if src length >= n.
- strcat(char* dest, const char* src)
    - Appends C-string src to dest (dest must have enough space).
- strcmp(const char* s1, const char* s2) / strncmp(...)

    - Lexicographical comparison of C-strings.

Notes:

- Prefer std::string operations or std::copy and std::string::data() for safety and clarity. Use C functions only for interop with C APIs.

🔵 **Regex Related (<regex>)**

- std::regex / std::smatch etc. (in <regex>)
- regex_match(const string& s, smatch& m, const regex& re)
    - Attempts to match the entire s against re. Returns true if the entire sequence matches.
    - Complexity: can be expensive depending on the regex (potential exponential backtracking in some patterns).
    - Throws: std::regex_error for invalid regex.
- regex_search(const string& s, smatch& m, const regex& re)
    - Searches for any substring matching re; fills smatch with matched subexpressions.
    - Complexity: depends on regex.
- regex_replace(const string& s, const regex& re, const string& fmt)
    - Returns a new string where matches are replaced by fmt.
    - Complexity: linear times cost of regex matching for each match.

Notes:

- Use std::regex only when necessary. For speed-sensitive code, many prefer third-party regex engines (RE2) or manual parsing.

🔵 **String Stream (from <sstream>)**

- std::stringstream, std::istringstream, std::ostringstream
    - Provide stream-based parsing/formatting to/from strings.
    - istringstream iss(s); iss >> x; parses formatted input.
    - ostringstream oss; oss << x; string r = oss.str(); builds string from formatted output.

    - Complexity: streams have overhead; std::from_chars/std::to_chars or to_string may be faster for simple conversions.
- getline(istream&, string&)
    - Reads a line from input stream into a string (stops at newline, removes delimiter). There's also std::getline that reads from any stream into std::string.
    - Signature: std::getline(std::istream& is, std::string& str);

🔵 **Other Useful Utilities**

- std::getline() (free function)
    - Already explained: reads a line from std::istream into std::string.
    - Variant: std::getline(is, s, delim) to use custom delimiter.
- std::wstring, std::u16string, std::u32string
    - Wide-character and UTF-16/UTF-32 basic_string specializations.
    - std::wstring uses wchar_t and is platform-dependent width; std::u16string uses char16_t, std::u32string uses char32_t. Useful for non-UTF-8 encodings or when interfacing with platform APIs expecting those encodings.
    - Operations mirror std::string but encoding/character-width semantics differ. For Unicode-aware operations, use proper libraries (ICU, std::wstring_convert deprecated in C++17) or work at the char/UTF-8 code unit level carefully.

Practical Notes, Pitfalls & Performance Tips

- **Contiguity guarantee**: Since C++11 std::string stores characters contiguously. Use data() or &s[0] (C++11 onwards) to access raw buffer. Since C++17 data() returns non-const char* for modification.
- **Null-termination**: c_str() and data() guarantee null-termination in modern standards (C++11 onwards c_str() always does; C++17 made data() null-terminated as well).
- **Exceptions**: Many operations may throw std::length_error (if request exceeds max_size()), std::out_of_range (e.g., .at() / substr pos), and std::bad_alloc. Use careful checks for user-provided indices/lengths.

- **Iterator invalidation**:
    - Any operation that changes the capacity or does reallocation invalidates pointers/iterators/references to characters.
    - Operations that only change characters in-place (like operator[] =) do not invalidate iterators.
    - insert, erase, append, replace, reserve (when reallocation happens) can invalidate iterators.
- **Performance**:
    - Avoid repeated operator+ building in a loop; prefer reserve() then append() or use ostringstream with care.
    - For parsing numeric values in hot paths, prefer std::from_chars (C++17) which is non-throwing and faster.
    - Regular expressions can be heavy; for simple patterns find/find_first_of/manual parsing can be faster and clearer.
- **Character signedness**:
    - Standard functions in <cctype> (like isalpha or toupper) expect unsigned char or EOF. Always use static_cast<unsigned char>(ch) to avoid UB when char is signed.
- **Locale**:
    - std::toupper/std::tolower from <cctype> are C locale-based. For locale-specific behavior use std::use_facet<std::ctype<char>>(loc).toupper(ch) from <locale> or std::wstring + wide locale facilities.
- **Unicode**:
    - std::string itself is just a sequence of bytes and knows nothing about character boundaries in UTF-8. For true Unicode-aware operations (case folding, grapheme clusters, normalization), use a Unicode library (ICU, boost::locale, or dedicated Unicode tooling).

Quick Reference: Common Exceptions and npos

- string::npos is static constexpr size_type npos = -1; used to indicate “not found” in find/rfind etc.
- std::invalid_argument / std::out_of_range: from stoi/stod family.

- std::out_of_range: from .at() and some constructor overloads if invalid position.
- std::length_error: from reserve()/resize()/append() trying to exceed max_size().