13.3. String Class in Detail

In this section, *string* stands for the corresponding string class: `string` , `wstring` , `u16string` , `u32string` , or any other specialization of class `basic_string<>` . Type *char* stands for the corresponding character type, which is `char` for `string` , `wchar_t` for `wstring` , `char16_t` for `u16string` , or `char32_t` for `u32string` . Other types and values in italic type have definitions that depend on individual definitions of the character type or traits class. The details about traits classes are provided in Section 16.1.4, page 853.

13.3.1. Type Definitions and Static Values

string **:: traits_type**

- The type of the character traits.
- The second template parameter of class `basic_string` .
- For type `string` , it is equivalent to `char_traits<char>` .

string **:: value _ type**

- The type of the characters.
- It is equivalent to `traits_type::char_type` .
- For type `string` , it is equivalent to `char` .

string **:: size _ type**

- The unsigned integral type for size values and indices.
- It is equivalent to `allocator_type::size_type` .
- For type `string` , it is equivalent to `size_t` .

string **:: difference _ type**

- The signed integral type for difference values.
- It is equivalent to `allocator_type::difference_type` .
- For type `string` , it is equivalent to `ptrdiff_t` .

string **:: reference**

- The type of character references.
- It is equivalent to `allocator_type::reference` .
- For type `string` , it is equivalent to `char&` .

string **:: const _ reference**

- The type of constant character references.
- It is equivalent to `allocator_type::const_reference` .
- For type `string` , it is equivalent to `const char&` .

string **:: pointer**

- The type of character pointers.
- It is equivalent to `allocator_type::pointer` .
- For type `string` , it is equivalent to `char*` .

string **:: const _ pointer**

- The type of constant character pointers.
- It is equivalent to `allocator_type::const_pointer` .
- For type `string` , it is equivalent to `const char*` .

string **:: iterator**

- The type of iterators.
- The exact type is implementation defined.
- For type `string`, it is typically `char*`.

string **:: const _ iterator**

- The type of constant iterators.
- The exact type is implementation defined.
- For type `string`, it is typically `const char*`.

string **:: reverse _ iterator**

- The type of reverse iterators.
- It is equivalent to `reverse_iterator<iterator>`.

string **:: const _ reverse _ iterator**

- The type of constant reverse iterators.
- It is equivalent to `reverse_iterator<const_iterator>`.

static const *size_type* ***string***::**npos**

- A special value that indicates either "*not found*"or"*all remaining characters*."
- It is an unsigned integral value that is initialized by `-1`.
- Be careful when you use `npos`. See Section 13.2.12, page 680, for details.

13.3.2. Create, Copy, and Destroy Operations

string **:: string** `()`

- The default constructor.
- Creates an empty string.

string **:: string** `(const` *string* `&` *str* `)`

- The copy constructor.
- Creates a new string as a copy of *str*.

string **:: string** `(` *string* `&&` *str* `)`

- The move constructor.
- Creates a new string initialized with the elements of the existing string *str*.
- The contents of *str* is undefined afterward.
- Available since C++11.

string:: **string** (const *string*& *str*, size_type *str_idx*)

string:: **string** (const *string*& *str*, size_type *str_idx*, size_type *str_num*)

- Create a new string that is initialized by at most the first *str_num* characters of *str*, starting with index *str_idx*.
- If *str_num* is missing, all characters from *str_idx* to the end of *str* are used.
- Throws `out_of_range` if *str _ idx* `>` *str* `.size()`.

string **:: string** `(const` *char* `*` *cstr* `)`

- Creates a string that is initialized by the C-string *cstr*.
- The string is initialized by all characters of *cstr* up to but not including `'\0'`.
- Note that passing a null pointer (`nullptr` or `NULL)` results in undefined behavior.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string **:: string** `(const` *char* `*` *chars* `, size_type` *chars _ len* `)`

- Creates a string that is initialized by *chars_len* characters of the character array *chars*.
- Note that *chars* must have at least *chars _ len* characters. The characters may have arbitrary values. Thus, `'\0'` has no special meaning.
- Throws `length_error` if *chars _ len* is equal to *string* `::npos`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string **:: string** `(size_type` *num* `,` *char c* `)`

- Creates a string that is initialized by *num* occurrences of character *c*.
- Throws `length_error` if *num* is equal to *string* `::npos` .
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

***string* :: string** `(InputIterator` *beg* `, InputIterator` *end* `)`

- Creates a string that is initialized by all characters of the range `[` *beg* `,` *end* `)` .
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

***string* :: string** `(InputIterator` *beg*, `InputIterator` *end* `)`

- Creates a string that is initialized by all characters of the range `[` *beg* `,` *end* `)` .
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

***string* :: string** *(initializer-list)*

- Creates a new string that is initialized by the characters of *initializer-list*.
- Available since C++11.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

***string* :: ~string** `()`

- The destructor.
- Destroys all characters and frees the memory.

Most constructors allow you to pass an allocator as an additional argument (see Section 13.3.13, page 715).

13.3.3. Operations for Size and Capacity

Size Operations

`bool` ***string* :: empty** `() const`

- Returns whether the string is empty (contains no characters).
- It is equivalent to *string* `::size()==0` , but it might be faster.

`size_type` ***string* :: size** `() const`

`size_type` ***string* :: length** `() const`

- Both functions return the current number of characters.
- They are equivalent.
- To check whether the string is empty, you should use `empty()` because it might be faster.

`size_type` ***string* :: max _ size** `() const`

- Returns the maximum number of characters a string could contain.
- Whenever an operation results in a string that has a length greater than `max_size()` , the class throws `length_error` .

Capacity Operations

`size_type` ***string* :: capacity** `() const`

- Returns the number of characters the string could contain without reallocation.

```
void string::reserve ()
void string::reserve (size_type num)
```

- The first form is a nonbinding shrink-to-fit request.
- The second form reserves internal memory for at least *num* characters.
- If *num* is less than the current capacity, the call is taken as a nonbinding request to shrink the capacity.
- If *num* is less than the current number of characters, the call is taken as a nonbinding request to shrink the capacity to fit the current number of characters (equivalent to the first form).
- The capacity is never reduced below the current number of characters.
- This operation might invalidate references, pointers, and iterators to characters. However, it is guaranteed that no reallocation takes place during insertions that happen after a call to `reserve()` until the time when an insertion would make the size greater than `num` . Thus, `reserve()` can increase speed and help to keep references, pointers, and iterators valid (see Section 13.2.5, page 670, for details).

void ***string*** :: **shrink_to_fit** ()

- Reduces the internal memory to fit the current numbers of characters.
- It has the same effect as `reserve(0)` .
- The call is taken as a nonbinding request to allow latitude for implementation-specific optimizations.
- This operation might invalidate references, pointers, and iterators to characters.
- Available since C++11.

13.3.4. Comparisons

```
bool comparison (const string& str1, const string& str2)
bool comparison (const string& str, const char* cstr)
bool comparison (const char* cstr, const string& str)
```

- The first form returns the result of the comparison of two strings.
- The second and third forms return the result of the comparison of a string with a C-string.
- ***comparison*** might be any of the following:

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- The values are compared lexicographically (see Section 13.2.7, page 673).

int ***string*** :: **compare** (const *string* & *str*) const

- Compares the string `*this` with the string *str*.
- Returns
  - `0` if both strings are equal
  - A value `< 0` if `*this` is lexicographically less than *str*
  - A value `> 0` if `*this` is lexicographically greater than *str*
- For the comparison, `traits::compare()` is used (see Section 16.1.4, page 854).
- See Section 13.2.7, page 673, for details.

int ***string*** :: **compare** (size_type *idx* , size_type *len* , const *string* & *str*) const

- Compares at most *len* characters of string `*this` , starting with index *idx* with the string *str*.
- Throws `out_of_range` if *idx* `> size()` .
- The comparison is performed as just described for `compare(` *str* `)` .

```
int string::compare (size_type idx, size_type len,
                     const string& str, size_type str_idx,
                     size_type str_len) const
```

- Compares at most *len* characters of string `*this` , starting with index *idx* with at most *str _ len* characters of string *str*, starting with index *str _ idx*.
- Throws `out_of_range` if *idx* `> size()` .
- Throws `out_of_range` if *str _ idx* > *str* `.size()` .
- The comparison is performed as just described for `compare(` *str* `)` .

int ***string*** :: **compare** (const *char* cstr*) const

- Compares the characters of string `*this` with the characters of the C-string *cstr*.
- The comparison is performed as just described for `compare(` *str* `)` .

int ***string*** :: **compare** (size_type *idx* , size_type *len* , const *char* * *cstr*) const

- Compares at most *len* characters of string `*this` , starting with index *idx* with all characters of the C-string *cstr*.
- The comparison is performed as just described for `compare(` *str* `)` .
- Note that passing a null pointer (`nullptr` or `NULL)` results in undefined behavior.

int **string::compare** (size_type *idx*, size_type *len*,
const *char** *chars*, size_type *chars_len*) const

- Compares at most *len* characters of string `*this` , starting with index *idx* with *chars* _ *len* characters of the character array *chars*.
- The comparison is performed as just described for `compare(` *str* `)` .
- Note that *chars* must have at least *chars* _ *len* characters. The characters may have arbitrary values. Thus, `'\0'` has no special meaning.
- Throws `length_error` if *chars* _ *len* is equal to *string* `::npos` .

13.3.5. Character Access

char& **string :: operator []** (size_type *idx*)
const *char*& **string::operator []** (size_type *idx*) const

- Both forms return the character with the index *idx* (the first character has index `0`).
- `length()` or `size()` is a valid index, and the operator returns the value generated by the default constructor of the character type (for `string` : `'\0'`). Before C++11, `length()` or `size()` was an invalid index value for nonconstant strings.
- Passing an invalid index results in undefined behavior.
- The reference returned for the nonconstant string may become invalidated due to string modifications or reallocations (see Section 13.2.6, page 672, for details).
- If the caller can't ensure that the index is valid, `at()` should be used.

char& **string::at** (size_type *idx*)
const *char*& **string::at** (size_type *idx*) const

- Both forms return the character that has the index *idx* (the first character has index `0`).
- For all strings, an index with `length()` as value is invalid.
- Passing an invalid index — less than `0` or greater than or equal to `length()` or `size()` — throws an `out_of_range` exception.
- The reference returned for the nonconstant string may become invalidated due to string modifications or reallocations (see Section 13.2.6, page 672, for details).
- By ensuring that the index is valid, the caller can use operator `[]` , which is faster.

char & **string::front** ()
const *char*& **string::front** () const

- Both forms return the first character.
- Calling `front()` for an empty string returns the value generated by the default constructor of the character type (for `string` : `'\0'`).
- The reference returned for the nonconstant string may become invalidated due to string modifications or reallocations (see Section 13.2.6, page 672, for details).

char& **string::back** ()
const *char*& **string::back** () const

- Both forms return the last character.
- Calling `back()` for an empty string results in undefined behavior.
- The reference returned for the nonconstant string may become invalidated due to string modifications or reallocations (see Section 13.2.6, page 672, for details).

13.3.6. Generating C-Strings and Character Arrays

const *char** **string::c_str** () const
const *char** **string::data** () const

- Returns the contents of the string as a character array, including a trailing end-of-string character `'\0'` . Thus, this is a valid C-string for `string` s.
- The return value is owned by the string. Thus, the caller must neither modify nor free or delete the return value.

- The return value is valid only as long as the string exists and as long as only constant functions are called for it.
- Before C++11, the return value of `data()` was guaranteed to contain all characters of the string without any trailing `'\0'` character. Thus, the return value of `data()` was *not* a valid C-string.

size_type **string::copy** (*char* buf*, size_type *buf_size*) const
size_type **string::copy** (*char* buf*, size_type *buf_size*, size_type *idx*) const

- Both forms copy at most *buf_size* characters of the string (beginning with index *idx*, if passed) into the character array *buf*.
- They return the number of characters copied.
- No null character is appended. Thus, the contents of *buf* might *not* be a valid C-string after the call.
- The caller must ensure that *buf* has enough memory; otherwise, the call results in undefined behavior.
- Throws `out_of_range` if *idx* > `size()`.

13.3.7. Modifying Operations

Assignments

string& **string::operator=** (const *string*& *str*)
string& **string::assign** (const *string*& *str*)

- Copy assignment operator.
- Both operations assign the value of string *str*.
- They return `*this`.

string& **string::operator=** (*string*&& *str*)
string& **string::assign** (*string*&& *str*)

- Move assignment operator.
- Move the contents of *str* to `*this`.
- The contents of *str* are undefined afterward.
- Return `*this`.
- Available since C++11.

string& **string::assign** (const *string*& *str*, size_type *str_idx*, size_type *str_num*)

- Assigns at most *str_num* characters of *str*, starting with index *str_idx*.
- Returns `*this`.
- Throws `out_of_range` if *str_idx* > *str* `.size()`.

string& **string::operator=** (const *char* cstr*)
string& **string::assign** (const *char* cstr*)

- Both operations assign the characters of the C-string *cstr*.
- They assign all characters of *cstr* up to but not including `'\0'`.
- Both operations return `*this`.
- Note that passing a null pointer (`nullptr` or `NULL)` results in undefined behavior.
- Both operations throw `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::assign** (const *char* chars*, size_type *chars_len*)

- Assigns *chars_len* characters of the character array *chars*.
- Returns `*this`.
- Note that *chars* must have at least *chars_len* characters. The characters may have arbitrary values. Thus, `'\0'` has no special meaning.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::operator=** (*char c*)

- Assigns character *c* as the new value.
- Returns `*this` .
- After this call, `*this` contains only this single character.

string& **string::assign** (size_type *num*, *char c*)

- Assigns *num* occurrences of character *c*.
- Returns `*this` .
- Throws `length_error` if *num* is equal to *string* `::npos` .
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::assign** (InputIterator *beg*, InputIterator *end*)

- Assigns all characters of the range [*beg* , *end*) .
- Returns `*this` .
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::operator=** (*initializer-list*)
string& **string::assign** (*initializer-list*)

- Both operations assign the characters of *initializer-list*.
- Both operations return `*this` .
- Both operations throw `length_error` if the resulting size exceeds the maximum number of characters.
- Available since C++11.

void **string::swap** (*string*& *str*)
void **swap** (*string*& *str1*, *string*& *str2*)

- Both forms swap the value of two strings, either of `*this` and *str* or of *str1* and *str2*.
- You should prefer these functions over copy assignment, if possible, because they are faster. In fact, they are guaranteed to have constant complexity. See Section 13.2.8, page 674, for details.

Appending Characters

string& **string::operator +=** (const *string*& *str*)
string& **string::append** (const *string*& *str*)

- Both operations append the characters of *str*.
- They return `*this` .
- Both operations throw `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::append** (const *string*& *str*, size_type *str_idx*, size_type *str_num*)

- Appends at most *str _ num* characters of *str*, starting with index *str _ idx*.
- Returns `*this` .
- Throws `out_of_range` if *str _ idx* > *str* `.size()` .
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::operator +=** (const *char** *cstr*)
string& **string::append** (const *char** *cstr*)

- Both operations append the characters of the C-string *cstr*.
- They return `*this` .
- Note that passing a null pointer (`nullptr` or `NULL)` results in undefined behavior.
- Both operations throw `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::append** (const *char** *chars*, size_type *chars_len*)

- Appends *chars_len* characters of the character array *chars*.
- Returns `*this`.
- Note that *chars* must have at least *chars_len* characters. The characters may have arbitrary values. Thus, `'\0'` has no special meaning.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string & ***string*** :: **append** (`size_type` *num*, *char c*)

- Appends *num* occurrences of character *c*.
- Returns `*this`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::operator +=** (*char c*)
`void` **string::push_back** (*char c*)

- Both operations append character *c*.
- Operator `+=` returns `*this`.
- Both operations throw `length_error` if the resulting size exceeds the maximum number of characters.

string & ***string*** :: **append** (`InputIterator` *beg* , `InputIterator` *end*)

- Appends all characters of the range [*beg* , *end*) .
- Returns `*this`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::operator +=** (*initializer-list*)
`void` **string::append** (*initializer-list*)

- Both operations append all characters of *initializer-list*.
- Both operations return returns `*this`.
- Both operations throw `length_error` if the resulting size exceeds the maximum number of characters.
- Available since C++11.

Inserting Characters

string & ***string*** :: **insert** (`size_type` *idx* , `const` *string*& *str*)

- Inserts the characters of *str* so that the new characters start with index *idx*.
- Returns `*this`.
- Throws `out_of_range` if *idx* `> size()`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::insert** (`size_type` *idx*, `const` *string*& *str*,
`size_type` *str_idx*, `size_type` *str_num*)

- Inserts at most *str_num* characters of *str*, starting with index *str_idx*, so that the new characters start with index *idx*.
- Returns `*this`.
- Throws `out_of_range` if *idx* `> size()`.
- Throws `out_of_range` if *str_idx* > *str* `.size()`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::insert** (`size_type` *idx*, `const` *char* cstr*)

- Inserts the characters of the C-string *cstr* so that the new characters start with index *idx*.
- Returns `*this`.
- Note that passing a null pointer (`nullptr` or `NULL)` results in undefined behavior.
- Throws `out_of_range` if *idx* `> size()`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::insert** (size_type *idx*, const *char** *chars*, size_type *chars_len*)

- Inserts *chars_len* characters of the character array *chars* so that the new characters start with index *idx*.
- Returns `*this`.
- Note that *chars* must have at least *chars_len* characters. The characters may have arbitrary values. Thus, `'\0'` has no special meaning.
- Throws `out_of_range` if *idx* `> size()`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

string& **string::insert** (size_type *idx*, size_type *num*, *char c*)
iterator **string::insert** (const_iterator *pos*, size_type *num*, *char c*)

- Insert *num* occurrences of character *c* at the position specified by *idx* or *pos*, respectively.
- The first form inserts the new characters so that they start with index *idx*.
- The second form inserts the new characters before the character to which iterator *pos* refers.
- The first form returns `*this`.
- The second form returns the position of the first character inserted or *pos* if none was inserted.
- Note that the overloading of these two functions results in a possible ambiguity. If you pass `0` as the first argument, it can be interpreted as an index, which is typically a conversion to `unsigned`, or as an iterator, which is often a conversion to `char*`. In this case, you should pass an index with its the exact type. For example:

```
std::string s;
...
s.insert(0,1,' ');                          // ERROR: ambiguous
s.insert((std::string::size_type)0,1,' ');  // OK
```

- Both forms throw `out_of_range` if *idx* `> size()`.
- Both forms throw `length_error` if the resulting size exceeds the maximum number of characters.
- Before C++11, *pos* had type `iterator`, and the return type of the second form was `void`.

iterator ***string*** **::** **insert** (const_iterator *pos*, *char c*)

- Inserts a copy of character *c* before the character to which iterator *pos* refers.
- Returns the position of the character inserted.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.
- Before C++11, *pos* had type `iterator`.

iterator **string::insert** (const_iterator *pos*,
InputIterator *beg*, InputIterator *end*)

- Inserts all characters of the range `[`*beg*`,` *end*`)` before the character to which iterator *pos* refers.
- Returns the position of the first character inserted or *pos* if none was inserted.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.
- Before C++11, *pos* had type `iterator`, and the return type was `void`.

iterator ***string*** **::** **insert** (const_iterator *pos*, *initializer-list*)

- Inserts all characters of *initializer-list* before the character to which iterator *pos* refers.
- Returns the position of the first character inserted or *pos* if none was inserted.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

Erasing Characters

void **string::clear** ()
string& **string::erase** ()

- Both functions delete all characters of the string. Thus, the string is empty after the call.
- `erase()` returns `*this`.

string& **string::erase** (size_type *idx*)

```
string& string::erase (size_type idx, size_type len)
```

- Both forms erase at most *len* characters of `*this`, starting at index *idx*.
- They return `*this`.
- If *len* is missing, all remaining characters are removed.
- Both forms throw `out_of_range` if *idx* `> size()`.

```
iterator string::erase (const_iterator pos)
iterator string::erase (const_iterator beg, const_iterator end)
```

- Both forms erase the single character at iterator position *pos* or all characters of the range `[` *beg* `,` *end* `)`, respectively.
- They return the position of the first character after the last removed character (thus, the second form returns *end*).
- Before C++11, *pos*, *beg*, and *end* had type `iterator`.

`void` ***string*** `::` **pop _ back** `()`

- Erases the last character.
- Calling this for an empty string results in undefined behavior.
- Available since C++11.

Changing the Size

```
void string::resize (size_type num)
void string::resize (size_type num, char c)
```

- Both forms change the number of characters of `*this` to *num*. Thus, if *num* is not equal to `size()`, they append or remove characters at the end according to the new size.
- If the number of characters increases, the new characters are initialized by *c*. If *c* is missing, the characters are initialized by the default constructor of the character type (for `string` : `'\0'`).
- Both forms throw `length_error` if *num* is equal to *string* `::npos`.
- Both forms throw `length_error` if the resulting size exceeds the maximum number of characters.

Replacing Characters

Click here to view code image

```
string& string::replace (size_type idx, size_type len, const string& str)
string& string::replace (begin_iterator beg, begin_iterator end,
                         const string& str)
```

- The first form replaces at most *len* characters of `*this`, starting with index *idx*, with all characters of *str*.
- The second form replaces all characters of the range `[` *beg* `,` *end* `)` with all characters of *str*.
- Both forms return `*this`.
- Both forms throw `out_of_range` if *idx* `> size()`.
- Both forms throw `length_error` if the resulting size exceeds the maximum number of characters.
- Before C++11, *beg* and *end* had type `iterator`.

Click here to view code image

```
string& string::replace (size_type idx, size_type len,
                         const string& str, size_type str_idx, size_type
str_num)
```

- Replaces at most *len* characters of `*this`, starting with index *idx*, with at most *str* _ *num* characters of *str*, starting with index *str* _ *idx*.
- Returns `*this`.
- Throws `out_of_range` if *idx* `> size()`.
- Throws `out_of_range` if *str* _ *idx* > *str* `.size()`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.

Click here to view code image

```
string& string::replace (size_type idx, size_type len, const char* cstr)
string& string::replace (const_iterator beg, const_iterator end,
                         const char* cstr)
```

- Both forms replace at most *len* characters of `*this`, starting with index *idx*, or all characters of the range [*beg*, *end*), respectively, with all characters of the C-string *cstr*.
- Both forms return `*this`.
- Note that passing a null pointer (`nullptr` or `NULL)` results in undefined behavior.
- Both forms throw `out_of_range` if *idx* `> size()`.
- Both forms throw `length_error` if the resulting size exceeds the maximum number of characters.
- Before C++11, *beg* and *end* had type `iterator`.

Click here to view code image

```
string& string::replace (size_type idx, size_type len,
                         const char* chars, size_type chars_len)
string& string::replace (const_iterator beg, const_iterator end,
                         const char* chars, size_type chars_len)
```

- Both forms replace at most *len* characters of `*this`, starting with index *idx*, or all characters of the range [*beg*, *end*), respectively, with *chars_len* characters of the character array *chars*.
- They return `*this`.
- Note that *chars* must have at least *chars_len* characters. The characters may have arbitrary values. Thus, `'\0'` has no special meaning.
- Both forms throw `out_of_range` if *idx* `> size()`.
- Both forms throw `length_error` if the resulting size exceeds the maximum number of characters.
- Before C++11, *beg* and *end* had type `iterator`.

Click here to view code image

```
string& string::replace (size_type idx, size_type len, size_type num, char
c)
string& string::replace (const_iterator beg, const_iterator end,
                        size_type num, char c)
```

- Both forms replace at most *len* characters of `*this`, starting with index *idx*, or all characters of the range [*beg*, *end*), respectively, with *num* occurrences of character *c*.
- They return `*this`.
- Both forms throw `out_of_range` if *idx* `> size()`.
- Both forms throw `length_error` if the resulting size exceeds the maximum number of characters.
- Before C++11, *beg* and *end* had type `iterator`.

Click here to view code image

```
string& string::replace (const_iterator beg, const_iterator end,
                        InputIterator newBeg, InputIterator newEnd)
```

- Replaces all characters of the range [*beg*, *end*) with all characters of the range [*newBeg*,*newEnd*).
- Returns `*this`.
- Throws `length_error` if the resulting size exceeds the maximum number of characters.
- Before C++11, *beg* and *end* had type `iterator`.

Click here to view code image

```
string& string::replace (const_iterator beg, const_iterator end,
                        initializer-list)
```

- Replaces all characters of the range [*beg*, *end*) with all characters of the *initializer-list*.
- Returns `*this`.

- Throws `length_error` if the resulting size exceeds the maximum number of characters.
- Available since C++11.

13.3.8. Searching and Finding

Find a Character

Click here to view code image

```
size_type string::find (char c) const
size_type string::find (char c, size_type idx) const
size_type string::rfind (char c) const
size_type string::rfind (char c, size_type maxIdx) const
```

- These functions search for the first/last character *c* (starting at index *idx/maxIdx*).
- The `find()` functions search forward and return the first substring.
- The `rfind()` functions search backward and return the last substring.
- These functions return the index of the character when successful or *string* `::npos` if they fail.

Find a Substring

Click here to view code image

```
size_type string::find (const string& str) const
size_type string::find (const string& str, size_type idx) const
size_type string::rfind (const string& str) const
size_type string::rfind (const string& str, size_type maxIdx) const
```

- These functions search for the first/last substring *str* (starting at index *idx/maxIdx*).
- The `find()` functions search forward and return the first substring.
- The `rfind()` functions search backward and return the last substring.
- These functions return the index of the first character of the substring when successful or *string* `::npos` if they fail.

Click here to view code image

```
size_type string::find (const char* cstr) const
size_type string::find (const char* cstr, size_type idx) const
size_type string::rfind (const char* cstr) const
size_type string::rfind (const char* cstr, size_type maxIdx) const
```

- These functions search for the first/last substring that is equal to the characters of the C-string *cstr* (starting at index *idx/maxIdx*).
- The `find()` functions search forward and return the first substring.
- The `rfind()` functions search backward and return the last substring.
- These functions return the index of the first character of the substring when successful or *string* `::npos` if they fail.
- Note that passing a null pointer (`nullptr` or `NULL)` results in undefined behavior.

Click here to view code image

```
size_type string::find (const char* chars, size_type idx,
                              size_type chars_len) const
size_type string::rfind (const char* chars, size_type maxIdx,
                               size_type chars_len) const
```

- These functions search for the first/last substring that is equal to *chars* _ *len* characters of the character array *chars*, starting at index *idx/maxIdx*.
- `find()` searches forward and returns the first substring.
- `rfind()` searches backward and returns the last substring.
- These functions return the index of the first character of the substring when successful or *string* `::npos` if they fail.
- Note that *chars* must have at least *chars* _ *len* characters. The characters may have arbitrary values. Thus, `'\0'` has no special meaning.

Find First of Different Characters

Click here to view code image

```
size_type string::find_first_of (const string& str) const
size_type string::find_first_of (const string& str, size_type idx) const
size_type string::find_first_not_of (const string& str) const
size_type string::find_first_not_of (const string& str, size_type idx) const
```

- These functions search for the first character that is or is not also an element of the string *str* (starting at index *idx*).
- These functions return the index of that character or substring when successful or *string* ::npos if they fail.

Click here to view code image

```
size_type string::find_first_of (const char* cstr) const
size_type string::find_first_of (const char* cstr, size_type idx) const
size_type string::find_first_not_of (const char* cstr) const
size_type string::find_first_not_of (const char* cstr, size_type idx) const
```

- These functions search for the first character that is or is not also an element of the C-string *cstr* (starting at index *idx*).
- These functions return the index of that character when successful or *string* ::npos if they fail.
- Note that passing a null pointer (nullptr or NULL) results in undefined behavior.

Click here to view code image

```
size_type string::find_first_of (const char* chars, size_type idx,
                                 size_type chars_len) const
size_type string::find_first_not_of (const char* chars, size_type idx,
                                     size_type chars_len) const
```

- These functions search for the first character that is or is not also an element of the *chars*_*len* characters of the character array *chars*, starting at index *idx*.
- These functions return the index of that character when successful or *string* ::npos if they fail.
- Note that *chars* must have at least *chars*_*len* characters. The characters may have arbitrary values. Thus, '\0' has no special meaning.

Click here to view code image

```
size_type string::find_first_of (char c) const
size_type string::find_first_of (char c, size_type idx) const
size_type string::find_first_not_of (char c) const
size_type string::find_first_not_of (char c, size_type idx) const
```

- These functions search for the first character that has or does not have the value *c* (starting at index *idx*).
- These functions return the index of that character when successful or *string* ::npos if they fail.

Find Last of Different Characters

Click here to view code image

```
size_type string::find_last_of (const string& str) const
size_type string::find_last_of (const string& str, size_type maxIdx) const
size_type string::find_last_not_of (const string& str) const
size_type string::find_last_not_of (const string& str, size_type maxIdx) const
```

- These functions search for the last character that is or is not also an element of the string *str* (searching backward starting at index *maxIdx*).
- These functions return the index of that character or substring when successful or *string* ::npos if they fail.

Click here to view code image

```
size_type string::find_last_of (const char* cstr) const
size_type string::find_last_of (const char* cstr, size_type maxIdx) const
size_type string::find_last_not_of (const char* cstr) const
size_type string::find_last_not_of (const char* cstr, size_type maxIdx) const
```

- These functions search for the last character that is or is not also an element of the C-string *cstr* (searching backward starting at index *maxIdx*).
- These functions return the index of that character when successful or *string* ::npos if they fail.
- Note that passing a null pointer (nullptr or NULL) results in undefined behavior.

Click here to view code image

```
size_type string::find_last_of (const char* chars, size_type maxIdx,
                                       size_type chars_len) const
size_type string::find_last_not_of (const char* chars, size_type maxIdx,
                                       size_type chars_len) const
```

- These functions search for the last character that is or is not also an element of the *chars* _ *len* characters of the character array *chars*, searching backward starting at index *maxIdx*.
- These functions return the index of that character when successful or *string* `::npos` if they fail.
- Note that *chars* must have at least *chars* _ *len* characters. The characters may have arbitrary values. Thus, `'\0'` has no special meaning.

Click here to view code image

```
size_type string::find_last_of (char c) const
size_type string::find_last_of (char c, size_type maxIdx) const
size_type string::find_last_not_of (char c) const
size_type string::find_last_not_of (char c, size_type maxIdx) const
```

- These functions search for the last character that has or does not have the value *c* (searching backward starting at index *maxIdx*).
- These functions return the index of that character when successful or *string* `::npos` if they fail.

13.3.9. Substrings and String Concatenation

Click here to view code image

```
string string::substr () const
string string::substr (size_type idx) const
string string::substr (size_type idx, size_type len) const
```

- All forms return a substring of at most *len* characters of the string `*this` (starting with index *idx*).
- If *len* is missing, all remaining characters are used.
- If *idx* and *len* are missing, a copy of the string is returned.
- All forms throw `out_of_range` if *idx* `> size()` .

Click here to view code image

```
string operator + (const string& str1, const string& str2)
string operator + (string&& str1, string&& str2)
string operator + (string&& str1, const string& str2)
string operator + (const string& str1, string&& str2)
string operator + (const string& str, const char* cstr)
string operator + (string&& str, const char* cstr)
string operator + (const char* cstr, const string& str)
string operator + (const char* cstr, string&& str)
string operator + (const string& str, char c)
string operator + (string&& str, char c)
string operator + (char c, const string& str)
string operator + (char c, string&& str)
```

- All forms concatenate all characters of both operands and return the sum string.
- Whenever an argument is an rvalue reference, the move semantics are used, which means that the argument has an undefined value afterward.
- The operands may be any of the following:
  - A string
  - A C-string
  - A single character
- All forms throw `length_error` if the resulting size exceeds the maximum number of characters.

13.3.10. Input/Output Functions

```
ostream& operator << (ostream&& strm, const string& str)
```

- Writes the characters of *str* to the stream *strm*.
- If *strm* `.width()` is greater than `0` , at least `width()` characters are written, and `width()` is set to `0` .
- *ostream* is the ostream type `basic_ostream` *<char* `>` according to the character type (see Section 15.2.1, page 748).

- Before C++11, the stream type was an lvalue reference.

```
istream& operator >> (istream&& strm, string& str)
```

- Reads the characters of the next word from *strm* into the string *str*.
- If the `skipws` flag is set for *strm*, leading whitespaces are ignored.
- Characters are extracted until any of the following happens:
  - – *strm*`.width()` is greater than `0` and `width()` characters are stored
  - – *strm*`.good()` is `false` (which might cause an appropriate exception)
  - – `isspace`(*c*, *strm*`.getloc())` is `true` for the next character *c*
  - – *str*`.max_size()` characters are stored
- The internal memory is reallocated accordingly.
- *istream* is the istream type `basic_istream` <*char*> according to the character type (see Section 15.2.1, page 748).
- Before C++11, the stream type was an lvalue reference.

Click here to view code image

```
istream& getline (istream& strm, string& str)
istream& getline (istream&& strm, string& str)
istream& getline (istream& strm, string& str, char delim)
istream& getline (istream&& strm, string& str, char delim)
```

- Read the characters of the next line from *strm* into the string *str*.
- All characters, including leading whitespaces, are extracted until any of the following happens:
  - – *strm*`.good()` is `false` (which might cause an appropriate exception)
  - – *delim* or *strm*`.widen('\n')` is extracted
  - – *str*`.max_size()` characters are stored
- The line delimiter is extracted but not appended.
- The internal memory is reallocated accordingly.
- *istream* is the istream type `basic_istream` <*char*> according to the character type (see Section 15.2.1, page 748).
- The overloads for rvalue references are available since C++11.

13.3.11. Numeric Conversions

Click here to view code image

```
int stoi (const string& str, size_t* idxRet=nullptr, int base=10)
long stol (const string& str, size_t* idxRet=nullptr, int base=10)
unsigned long stoul (const string& str, size_t* idxRet=nullptr, int base=10)
long long stoll (const string& str, size_t* idxRet=nullptr, int base=10)
unsigned long long stoull (const string& str, size_t* idxRet=nullptr, int
base=10)
int stof (const string& str, size_t* idxRet=nullptr)
int stod (const string& str, size_t* idxRet=nullptr)
int stold (const string& str, size_t* idxRet=nullptr)
```

- Convert *str* to the corresponding return type.
- *str* might be a string of type `string` or `wstring`.
- Skip leading whitespace.
- If *idxRet* `!=nullptr`, it returns the index of the first character not processed for the conversion.
- `base` allows you to specify a base number.
- Might throw `std::invalid_argument` if no conversion is possible and `std::out_of_range` if the converted value is outside the range of representable values for the return type.

```
string to_string (Type val)
wstring to_wstring (Type val)
```

- Converts *val* to a `string` or `wstring`.
- Valid types for *val* are `int`, `unsigned int`, `long`, `unsigned long`, `long long`,

`unsigned long long` , `float` , `double` , or `long double` .

13.3.12. Generating Iterators

Click here to view code image

```
iterator string::begin ()
const_iterator string::begin () const
const_iterator string::cbegin ()
```

- All forms return a random-access iterator for the beginning of the string (the position of the first character).
- If the string is empty, the call is equivalent to `end()` or `cend()` .

Click here to view code image

```
iterator string::end ()
const_iterator string::end () const
const_iterator string::cend ()
```

- All forms return a random-access iterator for the end of the string (the position after the last character).
- Note that the character at the end is not defined. Thus, `*` *s* `.end()` and `*` *s* `.cend()` result in undefined behavior.
- If the string is empty, the call is equivalent to `begin()` or `cbegin()` .

Click here to view code image

```
reverse_iterator string::rbegin ()
const_reverse_iterator string::rbegin () const
const_reverse_iterator string::crbegin ()
```

- All forms return a random-access iterator for the beginning of a reverse iteration over the string (the position of the last character).
- If the string is empty, the call is equivalent to `rend()` or `crend()` .
- For details about reverse iterators, see Section 9.4.1, page 448.

Click here to view code image

```
reverse_iterator string::rend ()
const_reverse_iterator string::rend () const
const_reverse_iterator string::crend ()
```

- All forms return a random-access iterator for the end of the reverse iteration over the string (the position before the first character).
- Note that the character at the reverse end is not defined. Thus, `*` *s* `.rend()` and `*` *s* `.crend()` result in undefined behavior.
- If the string is empty, the call is equivalent to `rbegin()` or `crbegin()` .
- For details about reverse iterators, see Section 9.4.1, page 448.

13.3.13. Allocator Support

Strings provide the usual members of classes with allocator support.

```
string::allocator_type
```

- The type of the allocator.
- Third template parameter of class `basic_string<>` .
- For type `string` , it is equivalent to `allocator<char>` .

```
allocator_type string::get_allocator () const
```

- Returns the memory model of the string.

Strings also provide all constructors with optional allocator arguments. The following are all the string constructors, including their optional allocator arguments, according to the standard:[11]

Click here to view code image

```
namespace std {
```

[11] The copy constructor with allocator, the move constructors, and the constructor for initializer list are available since C++11.

```
    template <typename charT,
              typename traits = char_traits<charT>,
              typename Allocator = allocator<charT> >
    class basic_string {
      public:
        // default constructor
        explicit basic_string(const Allocator& a = Allocator());

        // copy and move constructor (with allocator)
        basic_string(const basic_string& str);
        basic_string(basic_string&& str);
        basic_string(const basic_string& str, const Allocator&);
        basic_string(basic_string&& str, const Allocator&);

        // constructor for substrings
        basic_string(const basic_string& str,
                     size_type str_idx = 0,
                     size_type str_num = npos,
                     const Allocator& a = Allocator());

        // constructor for C-strings
        basic_string(const charT* cstr,
                     const Allocator& a = Allocator());

        // constructor for character arrays
        basic_string(const charT* chars, size_type chars_len,
                     const Allocator& a = Allocator());

        // constructor for num occurrences of a character
        basic_string(size_type num, charT c,
                     const Allocator& a = Allocator());

        // constructor for a range of characters
        template <typename InputIterator>
        basic_string(InputIterator beg, InputIterator end,
                     const Allocator& a = Allocator());

        // constructor for an initializer list
        basic_string(initializer_list<charT>,
                     const Allocator& a = Allocator());
        ...
    };
}
```

These constructors behave as described in Section 13.3.2, page 694, with the additional ability that you can pass your own memory model object. If the string is initialized by another string, the allocator also gets copied.[12] See Chapter 19 for more details about allocators.

[12] The original standard states that the default allocator is used when a string gets copied. However, this does not make much sense, so this is the proposed resolution to fix this behavior.