

17.4 — std::string character access and conversion to C-style arrays

BY ALEX ON OCTOBER 4TH, 2009 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Character access

There are two almost identical ways to access characters in a string. The easier to use and faster version is the overloaded operator[]:

char& string::operator[] (size_type nIndex)
const char& string::operator[] (size_type nIndex) const

- Both of these functions return the character with index nIndex
- Passing an invalid index results in undefined behavior
- Using length() as the index is valid for const strings only, and returns the value generated by the string's default constructor. It is not recommended that you do this.
- Because char& is the return type, you can use this to edit characters in the array

Sample code:

```
1 | std::string sSource{ "abcdefg" };
2 | std::cout << sSource[5] << '\n';
3 | sSource[5] = 'X';
4 | std::cout << sSource << '\n';
```

Output:

```
f
abcdeXg
```

There is also a non-operator version. This version is slower since it uses exceptions to check if the nIndex is valid. If you are not sure whether nIndex is valid, you should use this version to access the array:

char& string::at (size_type nIndex)
const char& string::at (size_type nIndex) const

- Both of these functions return the character with index nIndex
- Passing an invalid index results in an out_of_range exception
- Because char& is the return type, you can use this to edit characters in the array

Sample code:

```
1 | std::string sSource{ "abcdefg" };
2 | std::cout << sSource.at(5) << '\n';
3 | sSource.at(5) = 'X';
4 | std::cout << sSource << '\n';
```

Output:

```
f
abcdeXg
```

Conversion to C-style arrays

Many functions (including all C functions) expect strings to be formatted as C-style strings rather than `std::string`. For this reason, `std::string` provides 3 different ways to convert `std::string` to C-style strings.

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

- Returns the contents of the string as a const C-style string
- A null terminator is appended
- The C-style string is owned by the `std::string` and should not be deleted

Sample code:

```
1 | std::string sSource{ "abcdefg" };
2 | std::cout << std::strlen(sSource.c_str());
```

Output:

7

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

- Returns the contents of the string as a const C-style string
- A null terminator is appended. This function performs the same action as `c_str()`
- The C-style string is owned by the `std::string` and should not be deleted

Sample code:

```
1 | std::string sSource{ "abcdefg" };
2 | const char *szString{ "abcdefg" };
3 | // memcmp compares the first n characters of two C-style strings and returns 0 if they are
4 | equal
5 | if (std::memcmp(sSource.data(), szString, sSource.length()) == 0)
6 |     std::cout << "The strings are equal";
7 | else
   |     std::cout << "The strings are not equal";
```

Output:

The strings are equal

`size_type string::copy(char *szBuf, size_type nLength) const`

`size_type string::copy(char *szBuf, size_type nLength, size_type nIndex) const`

- Both flavors copy at most `nLength` characters of the string to `szBuf`, beginning with character `nIndex`
- The number of characters copied is returned
- No null is appended. It is up to the caller to ensure `szBuf` is initialized to NULL or terminate the string using the returned length
- The caller is responsible for not overflowing `szBuf`

Sample code:

```
1 | std::string sSource{ "sphinx of black quartz, judge my vow" };
2 |
3 | char szBuf[20];
4 | int nLength{ static_cast<int>(sSource.copy(szBuf, 5, 10)) };
   |
```

```

5 |   szBuf[nLength] = '\0'; // Make sure we terminate the string in the buffer
6 |
7 |   std::cout << szBuf << '\n';

```

Output:

black

Unless you need every bit of efficiency, `c_str()` is the easiest and safest of the three functions to use.



17.5 -- std::string assignment and swapping



Index



17.3 -- std::string length and capacity

[C++ TUTORIAL](#) | [PRINT THIS POST](#)

19 comments to 17.4 — std::string character access and conversion to C-style arrays



masterOfNothing

November 13, 2019 at 9:58 am · Reply

In this code example from the lesson:

```

1 |   string sSource("abcdefg");
2 |   char *szString = "abcdefg";
3 |   if (memcmp(sSource.data(), szString, sSource.length()) == 0)
4 |       cout << "The strings are equal";
5 |   else
6 |       cout << "The strings are not equal";

```

I cannot initialize like this

```

1 |   char *szString = "abcdefg";

```

throws the following error:

"error: ISO C++ forbids converting a string constant to 'char*' [-Wpedantic]".

Works only this way:

```

1 |   const char *szString = "abcdefg";

```

P.S.

I just realized that if I uncheck a certain warning under compiler flags, the previous problem suddenly isn't a problem anymore. Not so much a compiler error, as some ISO standard which I generally don't understand.