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

BY ALEX ON OCTOBER 4TH, 2009 | LAST MODIFIED BY ALEX ON JULY 22ND, 2017

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:

```
string sSource("abcdefg");
cout << sSource[5] << endl;
sSource[5] = 'X';
cout << sSource << endl;
```

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:

```
string sSource("abcdefg");
cout << sSource.at(5) << endl;
sSource.at(5) = 'X';
cout << sSource << endl;
```

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:

```
string sSource("abcdefg");
cout << 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 **not** appended
- The C-style string is owned by the std::string and should not be deleted

Sample code:

```
string sSource("abcdefg");
char *szString = "abcdefg";
// memcmp compares the first n characters of two C-style strings and returns 0 if they are equal
if (memcmp(sSource.data(), szString, sSource.length()) == 0)
    cout << "The strings are equal";
else
    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:

```
string sSource("sphinx of black quartz, judge my vow");

char szBuf[20];
int nLength = sSource.copy(szBuf, 5, 10);
szBuf[nLength]='\0';  // Make sure we terminate the string in the buffer

cout << szBuf << endl;
```

Output:

```
black
```

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