22.3 — std::string length and capacity

Once you've created strings, it's often useful to know how long they are. This is where length and capacity operations come into play. We'll also discuss various ways to convert std::string back into C-style strings, so you can use them with functions that expect strings of type char*.

Length of a string

The length of the string is quite simple -- it's the number of characters in the string. There are two identical functions for determining string length:

size_type string::length() const
size_type string::size() const
: Both of these functions return the current number of characters in the string, excluding the null terminator.

Sample code:

```
std::string s { "012345678" };
std::cout << s.length() << '\n';
```

Output:

```
9
```

Although it's possible to use length() to determine whether a string has any characters or not, it's more efficient to use the empty() function:

bool string::empty() const
: Returns true if the string has no characters, false otherwise.

Sample code:

```
std::string string1 { "Not Empty" };
std::cout << (string1.empty() ? "true" : "false") << '\n';
std::string string2; // empty
std::cout << (string2.empty() ? "true" : "false")  << '\n';
```

Output:

```
false
true
```

There is one more size-related function that you will probably never use, but we'll include it here for completeness:

size_type string::max_size() const

- Returns the maximum number of characters that a string is allowed to have.
- This value will vary depending on operating system and system architecture.

Sample code:

```
std::string s { "MyString" };
std::cout << s.max_size() << '\n';
```

Output:

```
4294967294
```

Capacity of a string

The capacity of a string reflects how much memory the string allocated to hold its contents. This value is measured in string characters, excluding the NULL terminator. For example, a string with capacity 8 could hold 8 characters.

size_type string::capacity() const

Returns the number of characters a string can hold without reallocation.

Sample code:

```
std::string s { "01234567" };
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';
```

Output:

```
Length: 8
Capacity: 15
```

Note that the capacity is higher than the length of the string! Although our string was length 8, the string actually allocated enough memory for 15 characters! Why was this done?

The important thing to recognize here is that if a user wants to put more characters into a string than the string has capacity for, the string has to be reallocated to a larger capacity. For example, if a string had both length and capacity of 8, then adding any characters to the string would force a reallocation. By making the capacity larger than the actual string, this gives the user some buffer room to expand the string before reallocation needs to be done.

As it turns out, reallocation is bad for several reasons:

First, reallocating a string is comparatively expensive. First, new memory has to be allocated. Then each character in the string has to be copied to the new memory. This can take a long time if the string is big. Finally, the old memory has to be deallocated. If you are doing many reallocations, this process can slow your program down significantly.

Second, whenever a string is reallocated, the contents of the string change to a new memory address. This means all references, pointers, and iterators to the string become invalid!

Note that it's not always the case that strings will be allocated with capacity greater than length. Consider the following program:

```
std::string s { "0123456789abcde" };
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';
```

This program outputs:

```
Length: 15
Capacity: 15
```

(Results may vary depending on compiler).

Let's add one character to the string and watch the capacity change:

```
std::string s("0123456789abcde");
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';

// Now add a new character
s += "f";
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';
```

This produces the result:

```
Length: 15
Capacity: 15
Length: 16
Capacity: 31
```

void string::reserve()
void string::reserve(size_type unSize)

- The second flavor of this function sets the capacity of the string to at least unSize (it can be greater). Note that this may require a reallocation to occur.
- If the first flavor of the function is called, or the second flavor is called with unSize less than the current capacity, the function will try to shrink the capacity to match the length. This request to shrink the capacity may be ignored, depending on implementation.

Sample code:

```
std::string s { "01234567" };
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';

s.reserve(200);
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';

s.reserve();
std::cout << "Length: " << s.length() << '\n';
std::cout << "Capacity: " << s.capacity() << '\n';
```

Output:

```
Length: 8
Capacity: 15
Length: 8
Capacity: 207
Length: 8
Capacity: 207
```

This example shows two interesting things. First, although we requested a capacity of 200, we actually got a capacity of 207. The capacity is always guaranteed to be at least as large as your request, but may be larger. We then requested the capacity change to fit the string. This request was ignored, as the capacity did not change.

If you know in advance that you're going to be constructing a large string by doing lots of string operations that will add to the size of the string, you can avoid having the string reallocated multiple times by reserving enough capacity from the outset:

```
#include <iostream>
#include <string>
#include <cstdlib> // for rand() and srand()
#include <ctime> // for time()

int main()
{
    std::srand(std::time(nullptr)); // seed random number generator

    std::string s{}; // length 0
    s.reserve(64); // reserve 64 characters

    // Fill string up with random lower case characters
    for (int count{ 0 }; count < 64; ++count)
        s += 'a' + std::rand() % 26;

    std::cout << s;
}
```

The result of this program will change each time, but here's the output from one execution:

```
wzpzujwuaokbakgijqdawvzjqlgcipiiuuxhyfkdppxpyycvytvyxwqsbtielxpy
```

Rather than having to reallocate s multiple times, we set the capacity once and then fill the string up. This can make a huge difference in performance when constructing large strings via concatenation.