Steve Lorimer
Yammerings on C++, Linux and other random topics that interest me

C++17 string_view

Introduced under proposal n3921, **`string_view`** gives us the ability to refer to an existing string in a non-owning way.

Rationale

`std::string` is the standard way of working with strings in C++, and offers a lot of convenient functionality for working with strings, such as comparison, searching for substrings, concatenating, slicing, etc.

There is a cost to working with strings though, and that is that they *own* the underlying buffer in which the string of characters is stored. In order to own this buffer, they often require dynamic memory. (Note if the string is small enough it won't, due to the "small string optimization" or SSO. You can read more here if you are interested.) However, to all intents and purposes, and particularly in generic code, unless you are certain of the length of the input and the length of string your implementation can handle *without* a dynamic allocation, I would suggest it is good practice to assume the creation of a string will result in a dynamic allocation.

It is important to understand when dynamic memory allocations occur, and whether they are necessary or not. You should always strive not to be "wasteful" with dynamic allocations, and prefer stack allocations when possible. This holds true in particular for performance critical code, as dynamic memory allocations are orders of magnitude slower than stack allocations.

So if `std::string` is the standard way of working with strings in C++, then why does this matter?

The reason is that there are often times when we want to work with *string-like* data, and we don't necessarily want to transform it into a standalone `std::string`.

For example, the arguments to a C++ application are passed to your `main` function as an array of c-style character arrays.

```
int main(int argc, char* argv[])
{
    // argv is an array of char*
}
```

Perhaps we want to write a function which takes two strings and compares them

```
bool compare(const std::string& s1, const std::string& s2)
{
    // do some comparisons between s1 and s2
}
```

What happens if we want to check a string we currently have, `str`, against a number of string literals?

```
bool r1 = compare(str, "this is the first test string");
bool r2 = compare(str, "this is the second test string");
bool r3 = compare(str, "this is the third test string");
```

For each of the calls to `compare` above, a `std::string` will be created, a buffer sufficiently large to hold the data will be created in dynamic memory, and the string literal copied into it.

Here we can see that dynamic allocations are occurring when we call `compare` with a string-literal by overloading the global `operator new`

```
#include <iostream>

void* operator new(std::size_t n)
{
    std::cout << "[allocating " << n << " bytes]\n";
    return malloc(n);
}

bool compare(const std::string& s1, const std::string& s2)
```

```
{
    if (s1 == s2)
        return true;
    std::cout << '\"' << s1 << "\" does not match \"" << s2 <<
    return false;
}

int main()
{
    std::string str = "this is my input string";

    compare(str, "this is the first test string");
    compare(str, "this is the second test string");
    compare(str, "this is the third test string");

    return 0;
}
```

Build and run:

```
$ g++ -std=c++11 -Wall -Wextra -Werror main.cpp
$ ./a.out
[allocating 24 bytes]
[allocating 30 bytes]
"this is my input string" does not match "this is the first te
[allocating 31 bytes]
"this is my input string" does not match "this is the second t
[allocating 30 bytes]
"this is my input string" does not match "this is the third te
```

All of these allocations, just to compare a string?

Of course we could create a second overload which takes C-style strings, but then we lose the benefit of having an O(1) `size` function.

```
bool compare(const std::string& s1, const char* s2)
{
    size_t s2_len = strlen(s2); // O(N) complexity
}
```

Another downside to this is that we now have to manage multiple overloads which ostensibly do the same thing.

What happens if we have another string type, such as Qt's `QString`. Do we create a third overload which compares against `QStrings`?

What happens if we want the first argument to be a C-style string or a `QString`, now we need multiple overloads for those too.

```
bool compare(const std::string& s1, const std::string& s2)
bool compare(const std::string& s1, const char* s2)
bool compare(const std::string& s1, const QString& s2)
bool compare(const char* s1, const std::string& s2)
bool compare(const char* s1, const char* s2)
bool compare(const char* s1, const QString& s2)
bool compare(const Qstring& s1, const std::string& s2)
bool compare(const Qstring& s1, const char* s2)
bool compare(const Qstring& s1, const QString& s2)
```

Clearly the number of overloads could quickly balloon if we decide to go down this path.

String views

Enter `string_view`, a way to *wrap* an existing string in a *non-owning* way.

The likely implementation will consist of just two data members, a pointer to the start of the string and a length.

They are cheap to construct and cheap to copy.

Example:

```
#include <iostream>
#include <experimental/string_view>

void* operator new(std::size_t n)
{
    std::cout << "[allocating " << n << " bytes]\n";
```

```
    return malloc(n);
}

bool compare(std::experimental::string_view s1, std::experimen
{
    if (s1 == s2)
        return true;
    std::cout << '\"' << s1 << "\" does not match \"" << s2 <<
    return false;
}

int main()
{
    std::string str = "this is my input string";

    compare(str, "this is the first test string");
    compare(str, "this is the second test string");
    compare(str, "this is the third test string");

    return 0;
}
```

Build and run:

```
$ g++ -std=c++1z -Wall -Wextra -Werror main.cpp
$ ./a.out
[allocating 24 bytes]
"this is my input string" does not match "this is the first te
"this is my input string" does not match "this is the second t
"this is my input string" does not match "this is the third te
```

You can see there is only a single allocation, when we create our `str` string. The creation of `string_vew` from the literals does not require a dynamic allocation.

Note: You can see I'm using `string_view` in the `experimental` namespace, as the version of gcc I'm using hasn't yet moved `string_view` into it's C++17 location (ie: out of experimental), which is where it will be in a C++17 compliant compiler.

Additional benefits

There are additional benefits to `string_view`, such as creating a `string_view` from a substring in an existing `string`. `std::string::substr` returns a new string, potentially involving a dynamic allocation. However, we can construct a `string_view` from the address of a position in our string, and that won't involve a dynamic allocation.

Example:

```
#include <iostream>
#include <experimental/string_view>

void* operator new(std::size_t n)
{
    std::cout << "[allocating " << n << " bytes]\n";
    return malloc(n);
}

bool compare(std::experimental::string_view s1, std::experimen
{
    if (s1 == s2)
        return true;
    std::cout << '\"' << s1 << "\" does not match \"" << s2 <<
    return false;
}

int main()
{
    std::string str = "this is my input string";

    std::experimental::string_view sv(&str.at(str.find_first_o

    compare(str, sv);

    return 0;
}
```

Build and run:

```
$ g++ -std=c++1z -Wall -Wextra -Werror main.cpp
$ ./a.out
[allocating 24 bytes]
"this is my input string" does not match "my input string"
```

Written on September 16, 2016