String literals as non-type template parameters

Document #:	D0424R2
Date:	2017-11-09
Project:	Programming Language C++
Audience:	Evolution Working Group

Reply-to:	Louis Dionne <ldionne.2@gmail.com>
	Hana Dusíková <hanicka@hanicka.net>

1 Revision history

- R0 – Initial draft
- R1 – Rewrite with different UDL form per EWG direction, and update motivation.
- R2 – Incorporate feedback from EWG in Albuquerque:
  - Use array syntax instead of pointer and length.
  - Allow string literals as non-type template arguments.
  - Initial wording attempt.

2 Abstract

We propose allowing string literals as non-type template arguments. A string literal would be passed as a reference to an array of characters:

```
template <auto& str>
void foo();

foo<"hello">(); // creates a ‘constexpr char[6]‘ and passes a reference to it
```

To match this new functionality, we also propose adding a new form of the user-defined literal operator for strings:

```
template <auto& str>
auto operator"" _udl();

"hello"_udl; // equivalent to operator""_udl<"hello">()
```

3 Motivation

Compile-time strings are a sorely missed piece of functionality in C++. Indeed, while we can pass a string as a function argument, there is no way of getting a string as a compile-time entity from within a function. This prevents a function from creating an object whose type depends on the *contents* of the string being passed. This paper proposes solving this problem by allowing string literals as non-type template parameters.

There are many concrete use cases for this functionality, some of which were covered in a previous version of this paper ([P0424R0]). However, some interesting use cases have recently come up, the most notable ones being compile-time JSON parsing and compile-time regular expression parsing. For example, a regular expression engine can be generated at compile-time as follows (example taken from the [CTRE] library):

```
#include "pregexp.hpp"
using namespace sre;

auto regexp = "^(?:[abc]|xyz).+$"_pre;

int main(int argc, char** argv) {
  if (regexp.match(argv[1])) {
    std::cout << "match!" << std::endl;
    return EXIT_SUCCESS;
  } else {
    std::cout << "no match!" << std::endl;
    return EXIT_FAILURE;
  }
}
```

Under the hood, constexpr functions and metaprogramming are used to parse the string literal and generate a type like the following from the string literal:

```
RegExp<
  Begin,
  Select<Char<'a','b','c'>, String<'x','y','z'>>,
  Plus<Anything>,
  End
>
```

Since the regular expression parser is generated at compile-time, it can be better optimized and the resulting code is much faster than `std::regex` (speedups of 3000x have been witnessed).

Similar functionality has traditionally been achieved by using expression templates and template metaprogramming to build the representation of the regular expression instead of simply parsing the string at compile-time. For example, the same regular expression with [Boost.Xpressive] looks like this:

```
auto regexp = bos >> ((set='a','b','c')|(as_xpr('x') >> 'y' >> 'z')) >> +_ >> eos;
```

It is worth noting that the specific use case of parsing regular expressions at compile-time came up at CppCon during a lightning talk, and the room showed a very strong interest in getting a standardized solution to this problem. Today, we must rely on a non-standard extension provided by Clang and GCC, which allows user-defined literal operators of the following form to be considered for string literals:

```
template <typename CharT, CharT ...s>
constexpr auto operator"" _udl();

"foo"_udl // calls operator""_udl<char, 'f', 'o', 'o'>()
```

With this proposal, we could instead write the following:

```
auto regexp = sre::parse<"^(?:[abc]|xyz).+$">();
```

or, for those that prefer user-defined literals:

```
using namespace sre;
auto regexp = "^(?:[abc]|xyz).+$"_pre;
```

4 How would that work?

The idea behind how this would work is that the compiler would generate a constexpr array and pass a reference to that as a template argument:

```
template <auto& str>
void f() {
  // str is a `char const (&)[7]`
}

f<"foobar">();

// should be roughly equivalent to

inline constexpr char __unnamed[] = "foobar";
f<__unnamed>();
```

Calling a function template with such a template-parameter-list works in both Clang and GCC today.

5 Proposed wording

Please note that this wording is very much a strawman aiming to convey the intent of the author. It was put together in a very short amount of time and changes will be made to make it more Core-ready.

This wording is based on the working draft [N4700]. In **[temp.arg.nontype] 17.3.2/2**:

A *template-argument* for a non-type *template-parameter* shall be a converted constant expression of the type of the *template-parameter*. For a non-type *template-parameter* of reference or pointer type, the value of the constant expression shall not refer to (or for a pointer type, shall not be the address of):

- a subobject,
- a temporary object,
- ~~a string literal,~~
- the result of a `typeid` expression, or
- a predefined `__func__` variable.

Remove **[temp.arg.nontype] 17.3.2/4** (TODO: make red):

[*Note:* A string literal is not an acceptable *template-argument*. [*Example:*

```
template<class T, const char* p> class X {
  // ...
};

X<int, "Studebaker"> x1;          // error: string literal as template-argument

const char p[] = "Vivisectionist";
X<int,p> x2;                      // OK
```

— *end example*] — *end note*]

Add after **[temp.arg.nontype] 17.3.2/2** (TODO: make green):

When passed as a *template-argument*, a string literal is a constant expression with external linkage. [*Note:* The intent is that `f<"foobar">()` be roughly equivalent to

```
inline constexpr char __some_mangled_name_including_foobar[] = "foobar";
f<__some_mangled_name_including_foobar>();
```

— *end note*]

6 Discussion on ODR

We have two choices; either we don't mandate that equivalent string literals share the same storage, or we do. If we do *not* mandate that equivalent string literals share the same storage, then template specializations with equivalent string literals would potentially be different template instantiations. This could lead to ODR issues:

```
// in foo.hpp
template <auto& str>
struct foo { };
```

```
inline foo<"hello"> x;

// in a.cpp
#include "foo.hpp"

// in b.cpp
#include "foo.hpp"
```

In the above, there are two possibilities:

1. `x` is defined with a different type in `a.cpp` and `b.cpp` (ODR violation), or
2. `x` has the same type in both translation units (no ODR violation, only one copy of `x` in the resulting program)

We think this should *not* be an ODR violation, and this is therefore what the current wording achieves by giving external linkage to the string literals used as *template-argument*s. Under the covers, an implementation would likely use the contents of the string literal to produce the mangled name of the variable. Based on preliminary discussion with implementers, this does not seem to be a problem.

7 Potential generalization

We could potentially make this applicable to arrays of arbitrary types, with something like the following syntax:

```
template <auto& array> void f();

f<{1, 2, 3}>(); // calls f with an array of type ‘int (&)[3]‘
```

This is an interesting generalization, but the author prefers tackling that as part of a separate proposal, since this proposal only targets string literals and is very useful on its own.

8 References

[P0424R0] Louis Dionne, *Reconsidering literal operator templates for strings*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0424r0.pdf

[Boost.Xpressive] Eric Niebler, *Boost.Xpressive*
http://www.boost.org/doc/libs/release/doc/html/xpressive.html

[CTRE] Hana Dusíková *Compile Time Regular Expression library*
https://github.com/hanickadot/compile-time-regular-expressions

[N4700] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4700.pdf