Non-type template parameters

AKA NTTP

Tony Thompson

tony.thompson@sig.com

C++@SIG

9/5/2019

SIG
SUSQUEHANNA

About Me

- C++ since 2013, industry since 2016, SIG since 2017
  - Physics data acquisition/analysis, radar systems, distributed monitoring systems, library design
- Software Developer on SMI team in Enterprise Technology
  - LibSMI (Cross-platform C++ messaging middleware library with C# and Python bindings)
  - SMI Agent (running on ~9900 SIG servers/desktops)
  - SMI AlertEngine, SMI Ganglia Adapter, SMI Router, SMI JsonAdapter, etc.
  - Maintainer/user of Jolt (STL/boost extensions for SIG), UnitTest libraries in plus toolchains
- Resident elasticsearch/Kibana power user

Type template parameters

```
template <class T> // or typename T
T add(T _lhs, T _rhs);

// ...
auto ints    = add(5, 2);     // T is int
auto doubles = add(2.2, 5.2); // T is double
auto ints2   = add(3, 12);    // T is int
// etc...
```

Telling different instantiations apart

```
template <class T>
T add(T _lhs, T _rhs)
{
   static int called(0);
   ++called;
   std::cout << called << std::endl;
   return _lhs + _rhs
}

// ...
auto ints    = add(5, 2);     // 1 is printed
auto doubles = add(2.2, 5.2); // 1 is printed
auto ints2   = add(3, 12);    // 2 is printed
// etc...
```

Non-type template parameters

```
template <int N>
std::string to_string(char const (& _text)[N])
{
   static int called(0);
   ++called;
   std::cout << called << std::endl;
   return std::string(_text, N - 1);
}

auto text  = to_string("Convert me pls");  // 1 is printed
auto text2 = to_string("Convert me pls2"); // 1 is printed
auto text3 = to_string("Convert me pls3"); // 2 is printed
```

Non-type template parameters

```
template<class T, T v>
struct integral_constant
{
   static T constexpr value = v;
};

using true_type  = std::integral_constant<bool, true>;
using false_type = std::integral_constant<bool, false>;

std::array<int, 5> my_array = { 5, 3, 2, 1, 5 };
```

Allowable non-type template parameters

- Allowed*
  - lvalue reference
  - Integral types (int, bool, char, etc.)
  - Pointer types (object or function)
  - Pointer to member types (to member object or member function)
  - Enumeration types
  - std::nullptr_t
  - Array and function types (automatically replaced with pointer to object/function)
- Notably disallowed
  - Floating point types
  - Anything else

* C++20 recently expanded this, more on this later

The curious case of string literals

```
template <char const * _value, class U>
struct pair;

/* String literal to type map with find_t member metafunction*/
template <class... Pairs>
struct map;
/* implementation left as an exercise for the reader */

using my_type_map = map
<
   pair<"int", int>
   , pair<"double", double>
   , pair<"char", char>
>;

using my_type = typename my_type_map::find_t<"double">;
static_assert(std::is_same_v<my_type, double>);
```

Are these different template instantiations?

```
template <char const * _value, class U>
struct pair; /* some implementation */

pair<"double", double>
pair<"double", double>
```

Maybe!

Compilers and string literals

- Optimizing compilers will store identical literals at the same address
- String literals have internal linkage, so different translation units would have different addresses for the same literals, although the compiler could also optimize this
- For these reasons, string literals are not allowed to be used as non-type template parameters
- This would even be the case if instead of a char pointer we used the actual string literal type as below, as it readily decays to a pointer to compare

```
template <int N, char const (& _value)[N], class U>
struct pair; /* some implementation */
```

But wait!

```
template <char const * Value>
void
print()
{
   std::cout << Value << std::endl;
}

char text[] = "my external entity";

int
main(int argc, char * argv[])
{
   print<text>();
}
```

Hacks for using string literals as NTTPs

- PEGTL example:

```
template <char...>
struct string;

TAO_PEGTL_STRING("literal") -> string<'l', 'i', 't', 'e', 'r', 'a', 'l'>
```

- Current compile time regex implementations rely on similar methods, or require the regular expression to have external linkage

TAO_PEGTL_STRING source

C++20 and beyond

- Auto syntax for NTTPs

```
template<auto n>
struct B { /* ... */ };

B<5> b1;   // OK: non-type template parameter type is int
B<'a'> b2; // OK: non-type template parameter type is char
B<2.5> b3; // error: non-type template parameter type cannot be double
```

- User-defined types (UDNTTP?)
  - Must satisfy *LiteralType* and have strong structural equality
  - Some dissenting opinions: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1837r0.html
  - Officially voted in to C++20

std::basic_fixed_string

C++20 map example

```
template <std::basic_fixed_string Value, class T>
struct pair; /* some implementation */

/* String literal to type map with find_t member metafunction*/
template <class... Pairs>
struct map;
/* implementation left as an exercise for the reader */

//...
using my_type_map = map
<
   pair<"int", int>
   , pair<"double", double>
   , pair<"char", char>
>;

using my_type = typename my_type_map::find_t<"double">;
static_assert(std::is_same_v<my_type, double>);
```

Questions?

SIG
SUSQUEHANNA