Fun? with NTTPs…

…maybe

Ben Deane / CppNorth / 2024-07-23

NTTP = Non-type Template Parameter

```
template <auto V> // <- an NTTP!
auto func() { ... }
```

Prior to C++20: integral types etc.

In C++20: structural types!

That's great! But not always possible...

This doesn't work

```
struct S {
private:
  int stuff; // oh no, a private member!
};
```

Not a structural type! Can't pass it as an NTTP.

(No private members allowed; no class invariants for you!)

What to do?

The answer is always lambda?

```
auto func(auto x) {
  constexpr auto X = x(); // yay
};

func([] { return 42; });
```

`func` isn't marked `constexpr`.

The argument isn't `constexpr` (they never are).

But the lambda's call operator is `constexpr` (implicitly)!

This is fine

[expr.const]

"An expression E is a core constant expression unless the evaluation of E, following the rules of the abstract machine (6.9.1), would evaluate one of the following:

- — this (7.5.2), except
    - — in a constexpr function (9.2.6) that is being evaluated as part of E"

Macro it!

```
#define CX_VALUE(...)                                   \
[] {                                                    \
  struct {                                              \
    consteval operator decltype(__VA_ARGS__)() const { \
      return __VA_ARGS__;                               \
    }                                                   \
    using is_cx_value = void;                           \
  } val;                                                \
  return val;                                           \
}()

template <typename T, typename U>
concept compile_time =
  requires { typename T::is_cx_value; }
  and std::convertible_to<T, U>;
```

And now this

You:

```
auto func(compile_time<std::size_t> auto size) {
  return std::array<char, size>{}; // consteval implicit conversion
}

auto arr = func(CX_VALUE(2u));
```

The compiler:

```
<it's all good>
```

Compile-time values wrapped up safely, passed around arbitrarily, and "made constexpr" at point of use.

A boring NTTP, yesterday

```
template <int X>
auto func() { return X; }
```

This is pretty boring. But it turns out…

[temp.param]

"The syntax for *template-parameter*s is:

template-parameter:

type-parameter

parameter-declaration"

parameter-declaration you say?

parameter-declaration is the same grammar production used for "regular" function arguments!

So let's make our NTTPs `const`, why not?

```
template <int const X>
auto func() { return X; }
```

You can only instantiate this template once!

Of course, if it can be `const`...

...you know it can be `volatile`!

```
template <int volatile X>
auto func() { return X; }
```

The only way I can interpret this is that the template is never memoized!

Attributes, anyone?

```
template <[[maybe_unused]] int X>
auto func() { return 42; }
```

This might actually be useful, but...

```
<source>:1:11: error: an attribute list cannot appear here
  1 | template <[[maybe_unused]] int X>
    |           ^~~~~~~~~~~~~~~~
```

Clang, you're wrong. This is grammatical. (And useful!)

Trying other things out

```
class C {
  template <virtual int X>
  auto func() { return X; }
};
```

I don't know what this would mean, but it's grammatical... (did you know `virtual` is a modifier in the grammar for function parameters?)

```
<source>:2:13: error: 'virtual' outside class declaration
    2 |   template <virtual int X>
      |             ^~~~~~~
```

Wrong, GCC. This is clearly inside a class declaration!

Trying other things out

```
class C {
  template <friend auto X>
  auto func() { return X; }
};
```

Whatever `X` is, treat it as a friend within this member function?

```
<source>:2:13: error: friend' used outside of class
    2 |   template <friend auto X>
      |             ^~~~~~~
```

Wrong again...

Trying other things out

```cpp
template <alignas(64) int X>
auto func() { return X; }
```

Speed up the compilation with cache line alignment!

GCC is fine with this.

Maximum strangeness

You know the really weird thing?

This is possible according to grammar productions.

```
template <const mutable alignas(64) auto x =
          co_yield throw throw delete noexcept(false)>
auto func() { return 42; }

auto x = func<0>();
```

And MSVC++ actually compiles this!

Lastly of course, with C++23...

```
class C {
  template <this auto X>
  auto func() { return X; }
};
```

If you want to take yourself as a template argument?

```
<source>:1:1: error: expected template parameter
    2 |   template <this auto X>
      |             ^~~~~~~
```

I don't understand; this is a *template-parameter*!

The importance of play

Try telling your compiler not to memoize templates today!

And spelunking the C++ grammar is a good way to come up with lightning talks...

(Also, C++ has no properly defined grammar.)

[gram.general]

This summary of C++ grammar is intended to be an aid to comprehension. It is not an exact statement of the language.