# TEMPLATE METAPROGRAMMING ~~IS~~ CAN BE FUN

Sasha Goldshtein

CTO, Sela Group

@goldshtn

github.com/goldshtn

# AGENDA

- Quick reminder on template specialization
- Metafunctions
- Type traits
- ~~Why is any of this useful?~~
- Quick reminder on variadic templates
- Live coding typelists and typelist sort

- This is a “from first principles” talk, it doesn’t try to cover metaprogramming libraries

# CLASS TEMPLATE SPECIALIZATION

- It’s kind of like pattern matching
- The “most matching” pattern is selected by the compiler

```
template <typename T> struct less {
  bool operator()(T a, T b) { return a < b; }
};

template <typename T> struct less<T*> {
  bool operator()(T* a, T* b) { return *a < *b; }
};

template <> struct less<void*> {
  bool operator()(void* a, void* b) { return a < b; }
};
```

# ”VALUE” TEMPLATES

- Templates don’t have to be generic in types; they can be generic in values, too

```
template <size_t A, size_t B> struct power
{
  static constexpr size_t value =
                   A * power<A, B-1>::value;
};

template <size_t B> struct power<0, B> { … };
template <size_t A> struct power<A, 0> { … };
template <>         struct power<0, 0> { … };
```

# FROM FUNCTIONS TO METAFUNCTIONS

- A **function** operates on values, at run-time
  - Take two numbers and return the biggest one
  - Take a string and return the last character
  - Take a string and return true if it's a palindrome
- A **metafunction** operates on types, at compile-time
  - Take two types and return the biggest one
  - Take a list of types and return the last type
  - Take a type and return true if it is a pointer to a function

# METAFUNCTION EXAMPLES

```
template <typename First, typename Second>
struct first
{
  using result = First;
};

template <typename First, typename Second>
struct second
{
  using result = Second;
};

first<int, std::string>  x = 42;
second<int, std::string> s = “Hello”;
```

# METAFUNCTION EXAMPLES

```
template <typename T, typename S>
struct is_same
{
  static constexpr bool value = false;
};

template <typename T>
struct is_same<T, T>
{
  static constexpr bool value = true;
};

bool b = is_same<int, decltype(2+2)>::value;
```

# METAFUNCTION EXAMPLES

```
template <bool Condition,
          typename IfTrue, typename IfFalse>
struct conditional;

template <typename IfTrue, typename IfFalse>
struct conditional<true, IfTrue, IfFalse>
{
  using result = IfTrue;
};

template <typename IfTrue, typename IfFalse>
struct conditional<false, IfTrue, IfFalse>
{
  using result = IfFalse;
};
```

# TESTING METAFUNCTIONS

```
static_assert(is_same<int, decltype(2+2)>::value);

static_assert(is_same<
    conditional<true, int, double>::result,
    int
  >::value);

static_assert(!is_pointer<int>::value);
```

# #INCLUDE <TYPE_TRAITS>

- A massive collection of metafunctions for compile-time testing and type manipulation
  - `is_pointer`, `is_reference`, `is_same`, `is_copy_constructible`, `is_abstract`
  - `remove_reference`, `remove_cv`, `decay`
- Used widely in the standard library itself:
  - `std::distance` and `std::advance` test whether the provided iterators are random-access, and use `-`/`+`
  - `std::copy` tests whether the provided iterators are pointers to trivially copyable types, and uses `memmove`
  - `std::optional::value_or` tests whether the provided argument is a factory function or a fallback value

# VARIADIC TEMPLATES

- Function, class, or variable templates that accept an arbitrary number of type or value parameters

```
template <typename First, typename... Rest>
First& first(std::tuple<First, Rest...>& tup)
{
  return std::get<0>(tup);
}

template <typename First, typename... Rest>
void print(First const& first, Rest const&... rest)
{
  std::cout << first;
  print(rest...);       // TODO: base case!
}
```

# THIS ALSO WORKS WITH VALUES

```
template <size_t...> struct sum;

template <> struct sum<> {
  static constexpr size_t value = 0;
};

template <size_t First, size_t... Rest>
struct sum<First, Rest...>
{
  static constexpr size_t value =
                   First + sum<Rest...>::value;
};

static_assert(sum<1, 2, 3>::value == 6);
```

# TYPELISTS!

- A **typelist** is a list of types (really?)

  ```
  using my_list = typelist<int, char, std::string>;
  ```

- We’ll implement some operations on typelists to practice our metaprogramming skills
- Culminating with `typelist_sort`, which can be used e.g. for tuple layout optimization

# MORE MODERN APPROACHES & LIBRARIES

- `constexpr` functions
- `if constexpr` (C++ 17)

- Boost.MPL
- Boost.Hana
- TinyMPL
- Loki

# MORE EXAMPLES & REFERENCES

- Alexandrescu: Modern C++ Design
- Abrahams, Gurtovoy: C++ Template Metaprogramming

- My GitHub repo with workshop labs and solutions on template metaprogramming
- My blog series on implementing `std::tuple` from scratch

- These slides: https://s.sashag.net/corecpp0418
- The demo code (may slightly differ from what we wrote today): https://s.sashag.net/tmpdemo0418

# THIS WAS FUN.
# WELL, MY IDEA OF FUN.
# QUESTIONS?

Sasha Goldshtein

CTO, Sela Group

@goldshtn

github.com/goldshtn