SPECTRAL:
TECHNOLOGIES

WEEK 4

Templates and basic metaprogramming

Metaprogramming in C++

- macros
- template magic (mostly outdated since c++20)
- compile-time evaluations

Macros

`#ifdef` (or more general `#if`) can be used to include/exclude sections of code based on some condition

```
#if LIB_VERSION <= 5
#include <old_api.h>
#else
#include <new_api.h>
#endif

void f() {
#ifdef NDEBUG // -DNDEBUG
   // fast implementation
#else
   // slow implementation
#endif
}
```

SPECTRAL:
TECHNOLOGIES

Macros

`#define` is still useful for simple codegen or creating pseudo-functions

```
#define ASSERT_EQ(expected, actual) do { \
    auto __expected = expected; \
    auto __actual = actual; \
    if (!(__expected == __actual)) { \
      std::cerr << __FILE__ << ":" << __LINE__ << ": Assertion error\n";\
      std::cerr << "\texpected: " << __expected << " (= " << #expected \
                << ")\n"; \
      std::cerr << "\tgot: " << __actual << " (= " << #actual << ")\n";\
      throw std::runtime_error("Check failed"); \
    } \
} while (false)
```

```
if (...)
  ASSERT_EQ(x, 5); // thats why do while(false)
else
  // something else
```

CRTP

- Not metaprogramming, but a neat trick anyway
- Curiously Recurring Template Pattern

```
template<class T>
class Base {
...
};

class Derived : public Base<Derived>{
...
};
```

- Can be used as a tool for static polymorphism

SPECTRAL:
TECHNOLOGIES

CRTP

```
template<class Derived>
struct Base {
    void interface(){
        static_cast<Derived*>(this)->implementation();
    }
    void implementation(){
        std::cout << "Implementation Base" << std::endl;
    }
};

struct Derived1: Base<Derived1>{
    void implementation(){
        std::cout << "Implementation Derived1" << std::endl;
    }
};
struct Derived2: Base<Derived2>{
    void implementation(){
        std::cout << "Implementation Derived2" << std::endl;
    }
};
struct Derived3: Base<Derived3>{};

template <class T>
void execute(Base<T>& base){
      base.interface();
}
```

CRTP-mixins

```
template <typename T>
struct NumericalFunctions {
    void scale(double multiplicator) {
        T& underlying = static_cast<T&>(*this);
        underlying.setValue(underlying.getValue() * multiplicator);
    }
};

class Value : public NumericalFunctions<Value> {
public:
    double getValue() const;
    void setValue(double value);
};
```

- `std::enable_shared_from_this`

SFINAE

Whats the problem here?

```
template<class T>
class Array {
public:
    explicit Array(size_t n, const T& val = T()) { ... }

    template<class Iterator>
    Array(Iterator first, Iterator last) { ... }
private:
    T *data_;
};
...
Array<int> a(5, 3);
```

SFINAE

Substitution Failure Is Not An Error

```
explicit Array(size_t n, const T& val = T()) { ... }

template<
   class Iterator,
   class Dummy = typename std::enable_if<
      !std::is_arithmetic<Iterator>::value>::type
>
Array(Iterator first, Iterator last) { ... }
```

Traits classes

```
template<class T> struct IsArithmetic {
   static const bool value = false;
};
template<> struct IsArithmetic<int> {
   static const bool value = true;
};
template<> struct IsArithmetic<double> {
   static const bool value = true;
};
// .. unsigned, long long, float, bool ...

template<bool, class T = void> struct EnableIf;

template<class T> struct EnableIf<false, T> { };
template<class T> struct EnableIf<true, T> {
   using type = T;
};
```

std-support

- `#include <type_traits>`
- is_class, is_enum, is_function, is_rvalue_reference, ...
- is_abstract, is_pod, is_trivially_copyable, is_nothrow_move_constructible, is_base_of , ...
- remove_const, add_lvalue_reference, add_rvalue_reference, ...

Current state

- You can build quite sophisticated conditions with sfinae (e.g. https://stackoverflow.com/questions/87372/check-if-a-class-has-a-memberfunction-of-a-given-signature).
- C++20 introduced concepts, which is the proper way to make type-restricted templates.
- Note that C++ doesn't have reflection yet.

Concepts

```
template<typename T>
concept Hashable = requires(T a)
{
{ std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

template<Hashable T>
void f(T) {}
//
// Alternative ways to apply the same constraint:
// template<typename T>
// requires Hashable<T>
// void f(T) {}
//
// template<typename T>
// void f(T) requires Hashable<T> {}
//
// void f(Hashable auto /*parameterName*/) {}
```

SPECTRAL:
TECHNOLOGIES

Our case

```
template<class Iterator>
Array(Iterator first, Iterator last) requires (!std::is_arithmetic_v<Iterator>) {
```

A lot of concepts can be found in `<concepts>`

SPECTRAL:
TECHNOLOGIES

More complex example

```
// std::same_as
template<class T1, class T2>
concept SameAs = std::is_same_v<T1, T2>;

template<class T>
concept MapKey = requires(T a, T b) {
        { a < b } -> SameAs<bool>;
};

template<typename T>
concept Hashable = requires(T a)
{
        { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

template<class T>
concept HashmapKey = requires(T a, T b) {
        { a == b } -> SameAs<bool>;
        requires Hashable<T>;
};

template<class T>
concept Key = MapKey<T> || HashmapKey<T>;
```

constexpr-if

- Another simple way to write compile-time conditions.
- Don't mix with `constexpr`

```
template<class Iterator>
void FastSort(Iterator first, Iterator last) {
    using T = typename std::iterator_traits<Iterator>::value_type;
    if constexpr(std::is_integral_v<T>) {
        RadixSort(first, last);
    } else {
        QuickSort(first, last);
    }
}
```

Compile-time evaluations

- `constexpr` and `consteval` allow to write a functions executed in compile-time
- Programmers used to it with templates before c++11.
- We skip this topic in this course.

SPECTRAL:
TECHNOLOGIES

Additional materials

- https://en.cppreference.com/w/cpp/language/constraints
- About constexpr (cppcon):
  - https://www.youtube.com/watch?v=PJwd4JLYJJY
  - https://www.youtube.com/watch?v=OcyAmlTZfgg
- https://eli.thegreenplace.net/2014/variadic-templates-in-c/