Template Metaprogramming

To template or not to template

Michele Caini

August 24, 2016

1. Introduction
   - To template or not to template
   - Basic concepts
   - Template metaprogramming by examples
     - Factorial
     - `std::enable_if_t` and `std::conditional_t`
     - Traits
     - void_t
   - That's all

What do you think of template metaprogramming?

Bjarne Stroustrup's FAQ about the topic:

- **To template:** Template metaprogramming is a set of powerful programming techniques.
- **Not to template:** Like all powerful techniques they are easily overused.

Metaprogramming

It's the writing of computer programs with the ability to treat programs as their data. - *Wikipedia*

Template metaprogramming

Metaprogramming technique based on the C++ template system.

What do you think of template metaprogramming?

Bjarne Stroustrup's FAQ about the topic:

- **To template:** Template metaprogramming is a set of powerful programming techniques.
- **Not to template:** Like all powerful techniques they are easily overused.

Metaprogramming

It's the writing of computer programs with the ability to treat programs as their data. - *Wikipedia*

Template metaprogramming

Metaprogramming technique based on the C++ template system.

Templates: what else?

Remind that [1]:

- A template defines a family of **classes** or **functions** or an **alias** for a family of types.
- A template declaration can be explicitly specialized (or partially specialized in case of primary classes).
- A template can be implicitly and explicitly instantiated.
- The more specialized, the preferred one.

```
// class template
template<bool B, typename T>
struct S { };

// partial specialization
template<typename T>
struct S<true, T> {
  using type = T;
};

// full specialization
template<>
struct S<true, std::size_t> {
  static constexpr
  std::size_t value = 42;
};

// function template
template<template T, typename F>
void func(F &&f) {
  std::vector<T> vec;
  std::forward<F>(f)(vec);
  // ...
}
```

[1] *Templates [14/1] (current working draft)*

Templates: what else?

Remind that [1]:

- A template defines a family of **classes** or **functions** or an **alias** for a family of types.
- A template declaration can be explicitly specialized (or partially specialized in case of primary classes).
- A template can be implicitly and explicitly instantiated.
- The more specialized, the preferred one.

```
// class template
template<bool B, typename T>
struct S { };

// partial specialization
template<typename T>
struct S<true, T> {
  using type = T;
};

// full specialization
template<>
struct S<true, std::size_t> {
  static constexpr
  std::size_t value = 42;
};

// function template
template<template T, typename F>
void func(F &&f) {
  std::vector<T> vec;
  std::forward<F>(f)(vec);
  // ...
}
```

[1] *Templates [14/1] (current working draft)*

Templates: what else?

Remind that [1]:

- A template defines a family of **classes** or **functions** or an **alias** for a family of types.
- A template declaration can be explicitly specialized (or partially specialized in case of primary classes).
- A template can be implicitly and explicitly instantiated.
- The more specialized, the preferred one.

```
// class template
template<bool B, typename T>
struct S { };

// partial specialization
template<typename T>
struct S<true, T> {
  using type = T;
};

// full specialization
template<>
struct S<true, std::size_t> {
  static constexpr
  std::size_t value = 42;
};

// function template
template<template T, typename F>
void func(F &&f) {
  std::vector<T> vec;
  std::forward<F>(f)(vec);
  // ...
}
```

[1] *Templates [14/1] (current working draft)*

Templates: what else?

Remind that [1]:

- A template defines a family of **classes** or **functions** or an **alias** for a family of types.
- A template declaration can be explicitly specialized (or partially specialized in case of primary classes).
- A template can be implicitly and explicitly instantiated.
- The more specialized, the preferred one.

```cpp
// class template
template<bool B, typename T>
struct S { };

// partial specialization
template<typename T>
struct S<true, T> {
  using type = T;
};

// full specialization
template<>
struct S<true, std::size_t> {
  static constexpr
  std::size_t value = 42;
};

// function template
template<template T, typename F>
void func(F &&f) {
  std::vector<T> vec;
  std::forward<F>(f)(vec);
  // ...
}
```

[1] *Templates [14/1] (current working draft)*

Templates: what else?

Remind that [1]:

- A template defines a family of **classes** or **functions** or an **alias** for a family of types.
- A template declaration can be explicitly specialized (or partially specialized in case of primary classes).
- A template can be implicitly and explicitly instantiated.
- The more specialized, the preferred one.

```
// class template
template<bool B, typename T>
struct S { };

// partial specialization
template<typename T>
struct S<true, T> {
  using type = T;
};

// full specialization
template<>
struct S<true, std::size_t> {
  static constexpr
  std::size_t value = 42;
};

// function template
template<template T, typename F>
void func(F &&f) {
  std::vector<T> vec;
  std::forward<F>(f)(vec);
  // ...
}
```

[1] *Templates [14/1] (current working draft)*

SFINAE

Substitution **F**ailure **I**s **N**ot **A**n **E**rror: a common feature (ab)used in template metaprogramming.
It applies during overload resolution of function templates

[14.8.2/8] - Function template specialization (working draft)

If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed, with a diagnostic required, if written using the substituted arguments. [...] Only invalid types and expressions in the **immediate context** of the function type and its template parameter types can result in a deduction failure.

SFINAE

Substitution **F**ailure **I**s **N**ot **A**n **E**rror: a common feature (ab)used in template metaprogramming.
It applies during overload resolution of function templates

[14.8.2/8] - Function template specialization (working draft)

If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed, with a diagnostic required, if written using the substituted arguments. [...] Only invalid types and expressions in the **immediate context** of the function type and its template parameter types can result in a deduction failure.

SFINAE

This is SFINAE (soft-error)

Only one function of the overload set matches the call:

```
struct S {
  template<typename T>
  std::enable_if_t<std::is_integral<T>::value>
  f(T &&t) { /* ... */ }

  template<typename T>
  std::enable_if_t<not std::is_integral<T>::value>
  f(T &&t) { /* ... */ }
};
```

This is not SFINAE (hard-error)

`T` happens to be outside of the *immediate context of the function type and its template parameter types*:

```
template<typename T>
struct S {
  std::enable_if_t<std::is_integral<T>::value>
  f(const T &t) { /* ... */ }
};
```

SFINAE

This is a typical abuse:

Substitution failure is always an error

`std::enable_if_t` works just fine here:

```
struct S {
  template<typename T>
  std::enable_if_t<std::is_integral<T>::value>
  f(T &&t) { /* ... */ }
};
```

Anyway, it's an *abuse*.
It's what we could call *substitution failure is always an error*.
`static_assert` is good as well, just easier to write and to reason about it:

```
struct S {
  template<typename T>
  void f(T &&t) {
    static_assert(std::is_integral<T>::value, "!");
    /* ... */
  }
};
```

Toolkit

- <type_traits> (`std::enable_if_t`, `std::is_same`, ...)
- <utility> (`std::integer_sequence`, `std::tuple`, ...)
- SFINAE
- `void_t`
- Unevaluated operands
- ...

Unevaluated... what?

The operands of the following operators are expressions that are not evaluated, since those operators only query the compile-time properties of their operands[a]:

`std::declval`, `decltype`, `typeid`, `sizeof`

[a] *Expressions [5/8] (current working draft)*

Toolkit

- `<type_traits>` (`std::enable_if_t`, `std::is_same`, ...)
- `<utility>` (`std::integer_sequence`, `std::tuple`, ...)
- SFINAE
- `void_t`
- Unevaluated operands
- ...

Unevaluated... what?

The operands of the following operators are expressions that are not evaluated, since those operators only query the compile-time properties of their operands[a]:

`std::declval`, `decltype`, `typeid`, `sizeof`

[a] *Expressions [5/8] (current working draft)*

A must: recursion

A couple of basic examples of template mataprogramming:

Function template

```
template<std::size_t N>
constexpr auto factorial() { // recursive definition
    return N * factorial<N-1>();
}

template<>
constexpr auto factorial<0>() { // final step
    return 1;
}
```

Class template

```
template<std::size_t N>
struct factorial { // recursive definition
    static constexpr std::size_t value = N * factorial<N-1>::value;
};

template<>
struct factorial<0> { // final step
    static constexpr std::size_t value = 1;
};
```

Compile time if/then/else

if/then:

```
template<bool cond, typename = void>
struct if_then { }; // default class template

template<typename T>
struct if_then<true, T> { using type = T; }; // partial specialization

template<bool cond, typename T>
using if_then_t = typename if_then<cond, T>::type; // alias
```

if/then/else:

```
template<bool cond, typename, typename F>
struct if_then_else { using type = F; }; // default class template

template<typename T, typename F>
struct if_then_else<true, T, F> { using type = T; }; // partial specialization

template<bool cond, typename T, typename F>
using if_then_else_t = typename if_then_else<cond, T, F>::type; // alias
```

Compile time if/then/else

~~if/then~~ std::enable_if_t:

```
template<bool cond, typename = void>
struct if_then { }; // default class template

template<typename T>
struct if_then<true, T> { using type = T; }; // partial specialization

template<bool cond, typename T>
using if_then_t = typename if_then<cond, T>::type; // alias
```

~~if/then/else~~ std::conditional_t:

```
template<bool cond, typename, typename F>
struct if_then_else { using type = F; }; // default class template

template<typename T, typename F>
struct if_then_else<true, T, F> { using type = T; }; // partial specialization

template<bool cond, typename T, typename F>
using if_then_else_t = typename if_then_else<cond, T, F>::type; // alias
```

Can we do that with functions?

Basic idea:

- **Tag dispatching** (function overloading and properties of a type).
- A bunch of functions (**only declarations** are required).
- `decltype` and `using` declarations.

Tag dispatching

A technique based on function overloading and properties of a type:

```
template<typename T> // internal preferred function
constexpr auto f(int, T &&t) -> decltype(std::decay_t<T>::g()) {
  // do something with those types that has a static member function T::g
}

template<typename T> // internal fallback function
constexpr auto f(char, T &&t) {
  // do something else with all the other types
}

template<typename T> // main function
constexpr auto f(T &&t) {
  return f(0, std::forward<T>(t));
}
```

Can we do that with functions?

Basic idea:

- **Tag dispatching** (function overloading and properties of a type).
- A bunch of functions (**only declarations** are required).
- `decltype` and `using` declarations.

Tag dispatching

A technique based on function overloading and properties of a type:

```
template<typename T> // internal preferred function
constexpr auto f(int, T &&t) -> decltype(std::decay_t<T>::g()) {
  // do something with those types that has a static member function T::g
}

template<typename T> // internal fallback function
constexpr auto f(char, T &&t) {
  // do something else with all the other types
}

template<typename T> // main function
constexpr auto f(T &&t) {
  return f(0, std::forward<T>(t));
}
```

enable_if_t made with functions

Educational purposes only: do not do this in your code!!

```cpp
// the tag (pretty simple indeed)
template<bool> struct tag { };

// the type is chosen only if cond is true
template<typename T> T enable_if(tag<true>);

// tag dispatching
template<bool cond, typename T = void>
auto enable_if() -> decltype(enable_if<T>(tag<cond>{}));

// alias
template<bool cond, typename T>
using enable_if_t = decltype(enable_if<cond, T>());

// SFINAEd function template
template<typename T>
constexpr enable_if_t<std::is_same<T, int>::value, int> f() { return 42; }

// SFINAEd function template
template<typename T>
constexpr enable_if_t<std::is_same<T, double>::value, double> f() { return .42; }

int main() {
  static_assert((42 == f<int>()) && (.42 == f<double>()), "!");
}
```

conditional_t made with functions

Educational purposes only: do not do this in your code!!

```
// the tag (pretty simple indeed)
template<bool>
struct tag { };

// the second type is chosen if cond is false
template<typename, typename F>
F conditional(tag<false>);

// the first type is chosen if cond is true
template<typename T, typename>
T conditional(tag<true>);

// tag dispatching
template<bool cond, typename T, typename F>
auto conditional() -> decltype(conditional<T, F>(tag<cond>{}));

// alias
template<bool cond, typename T, typename F>
using conditional_t = decltype(conditional<cond, T, F>());

int main() {
  static_assert(std::is_same<conditional_t<true, int, double>, int>::value, "!");
  static_assert(std::is_same<conditional_t<false, int, double>, double>::value, "!");
}
```

Traits

Traits are a simple form of template meta-programming. A basic definition:

Traits classes compute a set of properties given a type.

What's a trait exactly?

Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine *policy* or *implementation details*.[a]

[a] Bjarne Stroustrup

A more formal definition

A class used in place of template parameters. As a class, it aggregates useful types and constants; as a template, it provides an avenue for that *extra level of indirection* that solves all software problems.[a]

[a] Nathan C. Myers

Traits

Traits are a simple form of template meta-programming. A basic definition:

Traits classes compute a set of properties given a type.

What's a trait exactly?

Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine *policy* or *implementation details*.[a]

[a] Bjarne Stroustrup

A more formal definition

A class used in place of template parameters. As a class, it aggregates useful types and constants; as a template, it provides an avenue for that *extra level of indirection* that solves all software problems.[a]

[a] Nathan C. Myers

Traits

Traits are a simple form of template meta-programming. A basic definition:

Traits classes compute a set of properties given a type.

What's a trait exactly?

Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine *policy* or *implementation details*.[a]

[a] Bjarne Stroustrup

A more formal definition

A class used in place of template parameters. As a class, it aggregates useful types and constants; as a template, it provides an avenue for that *extra level of indirection* that solves all software problems.[a]

[a] Nathan C. Myers

A real-world example

Traits definition

```cpp
// tag classes
struct IPv4 { }; struct IPv6 { };

// traits class - it is not defined by default
template<typename> struct IpTraits;

// full specialization for IPv4
template<>
struct IpTraits<IPv4> {
  using Type = sockaddr_in;
  using AddrFuncType = int(*)(const char *, int, sockaddr_in *);
  using NameFuncType = int(*)(const sockaddr_in *, char *, std::size_t);
  static constexpr AddrFuncType addrFunc = &uv_ip4_addr;
  static constexpr NameFuncType nameFunc = &uv_ip4_name;
};

// full specialization for IPv6
template<>
struct IpTraits<IPv6> {
  using Type = sockaddr_in6;
  using AddrFuncType = int(*)(const char *, int, sockaddr_in6 *);
  using NameFuncType = int(*)(const sockaddr_in6 *, char *, std::size_t);
  static constexpr AddrFuncType addrFunc = &uv_ip6_addr;
  static constexpr NameFuncType nameFunc = &uv_ip6_name;
};
```

A real-world example

Traits in use

```
// ... a small object whose main purpose is to carry information
// used by an algorithm to determine implementation details ...

template<typename I = IPv4>
void bind(std::string ip, unsigned int port, Flags<Bind> flags = Flags<Bind>{}) {
  // the type of the struct or the function to be used are unknown,
  // they depend on the template parameter I, but still the
  // algorithm can be defined for those are implementation details
  typename details::IpTraits<I>::Type addr;
  details::IpTraits<I>::addrFunc(ip.data(), port, &addr);

  // ...
}

template<typename I>
Addr address(const typename details::IpTraits<I>::Type *aptr, int len) noexcept {
  std::pair<std::string, unsigned int> addr{};
  char name[len];

  // the same applies here: who cares about the function to be used?
  int err = details::IpTraits<I>::nameFunc(aptr, name, len);

  // ...
}
```

void_t

An interesting upcoming feature of C++17 (at least in terms of template metaprogramming) is `void_t`.

A possible implementation

```
template<typename...>
using void_t = void;
```

Trivial indeed: it accepts a variadic number of template parameters and provides an alias for `void`, no matters what are those parameters.

Member detector

A classical use of `void_t`, good enough for this talk:

- The template class:

```
template<typename T, typename = void_t<>>
struct detector: std::false_type { };
```

- The partial specialization:

```
template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

- And that's all, it works:

```
struct S { void foobar(); };
struct U { };
// ...
static_assert(detector<S>::value, "!");
static_assert(not detector<U>::value, "!");
```

Member detector

A classical use of `void_t`, good enough for this talk:

- The template class:

```
template<typename T, typename = void_t<>>
struct detector: std::false_type { };
```

- The partial specialization:

```
template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

- And that's all, it works:

```
struct S { void foobar(); };
struct U { };
// ...
static_assert(detector<S>::value, "!");
static_assert(not detector<U>::value, "!");
```

Member detector

A classical use of `void_t`, good enough for this talk:

- The template class:

```
template<typename T, typename = void_t<>>
struct detector: std::false_type { };
```

- The partial specialization:

```
template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

- And that's all, it works:

```
struct S { void foobar(); };
struct U { };
// ...
static_assert(detector<S>::value, "!");
static_assert(not detector<U>::value, "!");
```

Member detector

A classical use of `void_t`, good enough for this talk:

- The template class:

```
template<typename T, typename = void_t<>>
struct detector: std::false_type { };
```

- The partial specialization:

```
template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

- And that's all, it works:

```
struct S { void foobar(); };
struct U { };
// ...
static_assert(detector<S>::value, "!");
static_assert(not detector<U>::value, "!");
```

Member detector

Why does it work? A few hints:

- void_t<> and void_t<void> are both *aliases* for void.
- The primary declaration is always valid (default to void_t<>).
- The partial specialization can be either valid or invalid:
  - If it's valid, it's a **more specialized declaration**.
  - If it's invalid, well: **SFINAE**.
- Everybody knows about std::declval and decltype, right?

```
template<typename...>
using void_t = void;

template<typename T, typename = void_t<>>
struct detector: std::false_type { };

template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

Member detector

Why does it work? A few hints:

- `void_t<>` and `void_t<void>` are both *aliases* for `void`.
- The primary declaration is always valid (default to `void_t<>`).
- The partial specialization can be either valid or invalid:
  - If it's valid, it's a **more specialized declaration**.
  - If it's invalid, well: **SFINAE**.
- Everybody knows about `std::declval` and `decltype`, right?

```
template<typename...>
using void_t = void;

template<typename T, typename = void_t<>>
struct detector: std::false_type { };

template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

Member detector

Why does it work? A few hints:

- `void_t<>` and `void_t<void>` are both *aliases* for `void`.
- The primary declaration is always valid (default to `void_t<>`).
- The partial specialization can be either valid or invalid:
  - If it's valid, it's a **more specialized declaration**.
  - If it's invalid, well: **SFINAE**.
- Everybody knows about `std::declval` and `decltype`, right?

```
template<typename...>
using void_t = void;

template<typename T, typename = void_t<>>
struct detector: std::false_type { };

template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

Member detector

Why does it work? A few hints:

- `void_t<>` and `void_t<void>` are both *aliases* for `void`.
- The primary declaration is always valid (default to `void_t<>`).
- The partial specialization can be either valid or invalid:
  - If it's valid, it's a **more specialized declaration**.
  - If it's invalid, well: **SFINAE**.
- Everybody knows about `std::declval` and `decltype`, right?

```
template<typename...>
using void_t = void;

template<typename T, typename = void_t<>>
struct detector: std::false_type { };

template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

Member detector

Why does it work? A few hints:

- `void_t<>` and `void_t<void>` are both *aliases* for `void`.
- The primary declaration is always valid (default to `void_t<>`).
- The partial specialization can be either valid or invalid:
  - If it's valid, it's a **more specialized declaration**.
  - If it's invalid, well: **SFINAE**.
- Everybody knows about `std::declval` and `decltype`, right?

```
template<typename...>
using void_t = void;

template<typename T, typename = void_t<>>
struct detector: std::false_type { };

template<typename T>
struct detector<T, void_t<decltype(std::declval<T>().foobar())>>: std::true_type { };
```

That's all

Remember that
template metaprogramming is a set of powerful programming techniques.
On the other side,
like all powerful techniques they are easily overused.

Enjoy template metaprogramming.