Template Meta-Programming

Template ~~Meta~~-Programming

What is programming?

“The craft of writing useful, maintainable, and extensible source code which can be interpreted or compiled by a computing system to perform a meaningful task.”

—Wikibooks

What is metaprogramming?

“The writing of computer programs that manipulate other programs (or themselves) as if they were data”

—Anders Hejlsberg

Motivation: Generic functions

```c
double abs(double x)
{
    return (x >= 0) ? x : -x;
}


int abs(int x)
{
    return (x >= 0) ? x : -x;
}
```

And then also for long int, long long, float, long double, complex types...
Maybe char types?
Maybe short?
Maybe unsigned types?
Where does it end?

C99 provides:

```
abs (int)
labs (long)
llabs (long long)
imaxabs (intmax_t)
fabsf (float)
fabs (double)
fabsl (long double)
cabsf (_Complex float)
cabs (_Complex double)
cabsl (_Complex long double)
```

Function templates

```
template<typename T>
T abs(T x)
{
    return (x >= 0) ? x : -x;
}
```

Function templates ***are not functions.***
They are *templates* for making functions.

Don't pay for what you don't use:
If nobody calls `abs<int>`, it won't be instantiated by the compiler at all.

Using a function template

```
template<typename T>
T abs(T x)
{
    return (x >= 0) ? x : -x;
}

int main()
{
    double (*foo)(double) = abs<double>;
    printf("%d\n", abs<int>(-42));
}
```

The template `abs` will not be instantiated with any particular type `Foo` until you, the programmer, explicitly mention `abs<Foo>` in your program.

As soon as you mention `abs<Foo>`, the compiler will *have* to go instantiate it, in order to figure out its return type and so on.

Sometimes the compiler can deduce `abs<Foo>` when all you wrote was `abs`; but we'll talk about that deduction process later. Hold that thought.

Motivation: Generic types

```
// We’ve all seen this sort of thing in C, right?

struct my_generic_list
{
    void *data;
    my_generic_list *next;
};

my_generic_list *intlist = ...;
my_generic_list *doublelist = ...;

// Yuck. Type punning. Ugly and error-prone.
```

Slightly better, but more verbose

```
struct mylist_of_int
{
    int data;
    mylist_of_int *next;
};

struct mylist_of_double
{
    double data;
    mylist_of_double *next;
};
```

Class templates *create new types*

```
template<typename T>
struct mylist
{
    T data;
    mylist<T> *next;
};

mylist<int> *intlist = ...;
mylist<double> *doublelist = ...;
```

Class templates ***are not classes.***
They are *templates* for making classes.

Don’t pay for what you don’t use:
If nobody uses `mylist<int>`, it won’t be instantiated by the compiler at all.

Two new kinds of templates

C++11 introduced *alias templates*.

C++14 introduced *variable templates*.

Let's cover variable templates first, because they're a lot like class templates.

Variable templates are syntactic sugar

A variable template is exactly 100% equivalent to a static data member of a class template.

```
template<typename T>
struct is_void {
    static const bool value = (some expression);
};

int main() {
    printf("%d\n", is_void<int>::value);   // 0
    printf("%d\n", is_void<void>::value);  // 1
}
```

Variable templates are syntactic sugar

A variable template is exactly 100% equivalent to a static data member of a class template.

```
template<typename T>
const bool is_void_v = (some expression);




int main() {
    printf("%d\n", is_void_v<int>);   // 0
    printf("%d\n", is_void_v<void>);  // 1
}
```

In the STL: the best of both worlds

```
template<typename T>
struct is_void {
    static constexpr bool value = (some expression);
};

template<typename T>
constexpr bool is_void_v = is_void<T>::value;

int main() {
    printf("%d\n", is_void<int>::value);   // 0
    printf("%d\n", is_void_v<void>);       // 1
}
```

Alias templates ("template typedefs")

```
typedef std::vector<int> myvec_int;

using myvec_double = std::vector<double>;

template<typename T> using myvec = std::vector<T>;

int main()
{
    static_assert(is_same_v<myvec_int, std::vector<int>>);
    static_assert(is_same_v<myvec_double, std::vector<double>>);
    static_assert(is_same_v<myvec<float>, std::vector<float>>);
}
```

Literally the same type

```
using myint = int;
void f(const myint& mv);

template<typename T> using myvec = std::vector<T>;
void g(const myvec<int>& mv);

int main() {
    int i;
    f(i);  // OK
    std::vector<int> v = { 1, 2, 3, 4 };
    g(v);  // OK
}
```

We’ll come back to why this is important.

Type deduction (for function templates)

```
template<typename T>
T abs(T x)
{
    return (x >= 0) ? x : -x;
}

int main()
{
    double (*foo)(double) = abs<double>;
    printf("%d\n", abs<int>(-42));
}
```

Sometimes the compiler can deduce `abs<Foo>` when all you wrote was `abs`; but we'll talk about that deduction process ... now.

Rules of template type deduction

```
template<typename T>
void foo(T x)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    foo(4);          // void foo(T) [T = int]
    foo(4.2);        // void foo(T) [T = double]
    foo("hello");    // void foo(T) [T = const char *]
}
```

Type deduction in a nutshell:

- Each function parameter may contribute (or not) to the deduction of each template parameter (or not).
- All deductions are carried out “in parallel”; they don’t cross-talk with each other.
- At the end of this process, the compiler checks to make sure that each template parameter has been deduced at least once (otherwise: “couldn’t infer template argument T”) and that all deductions agree with each other (otherwise: “deduced conflicting types for parameter T”).
- Furthermore, any function parameter that *does* contribute to deduction must match its function argument type *exactly*. No implicit conversions allowed!

Puzzle #1

```
template<typename T, typename U>
void foo(std::array<T, sizeof(U)> x,
         std::array<U, sizeof(T)> y,
         int z)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    foo(std::array<int,8>{}, std::array<double,4>{}, 0.0);
    foo(std::array<int,8>{}, std::array<double,5>{}, 0.0);
}
```

Puzzle #2

```
template<typename T> struct Foo {
    using type = T;
};

template<typename T> using Bar = typename Foo<T>::type;

template<typename T> void f(Foo<T>, Bar<T>) {}

int main()
{
    f(Foo<int>(), 0.0);
}
```

Puzzle #3

```
template<typename T> struct Foo {
    using type = T;
};

template<typename T> using Bar = T;

template<typename T> void f(Foo<T>, T) {}

int main()
{
    f(Foo<int>(), 0.0);
}
```

How to call a specialization explicitly

```
template<typename T>
T abs(T x)
{
    return (x >= 0) ? x : -x;
}

int main()
{
    printf("%d\n", abs<int>('x'));   // [T = int]
    printf("%g\n", abs<double>(3));  // [T = double]
}
```

How to call a specialization explicitly

```
template<typename T, typename U>
void add(T x, U y)
{
    puts(__PRETTY_FUNCTION__);
}

int main() {
    add<int, int>('x', 3.1);  // [T = int, U = int]
    add<int>('x', 3.1);       // [T = int, U = double]
    add<>('x', 3.1);          // [T = char, U = double]
    add('x', 3.1);            // [T = char, U = double]
}
```

Type deduction in a nutshell:

- Any template parameters that were explicitly specified by the caller are fixed as whatever the caller said they were; they don't participate any further in deduction.
- Each function parameter may contribute (or not) to the deduction of each remaining template parameter (or not).
- Deductions are carried out in parallel; they don't cross-talk with each other.
- At the end of this process, the compiler checks to make sure that each template parameter (that wasn't specified by the caller) has been deduced at least once and that all deductions agree with each other.
- Furthermore, any function parameter that *does* contribute to deduction must match its function argument type *exactly*.

Type deduction in a nutshell:

- Any template parameters that were explicitly specified by the caller are fixed as whatever the caller said they were; they don't participate any further in deduction.
- Each function parameter may contribute (or not) to the deduction of each remaining template parameter (or not).
- Deductions are carried out in parallel; they don't cross-talk with each other.
- At the end of this process, the compiler checks to make sure that each template parameter (that wasn't specified by the caller) has been deduced at least once and that all deductions agree with each other.
- Furthermore, any function parameter that *does* contribute to deduction must match its function argument type *exactly*.

Default template parameters

```
template<typename T>
void add()
{
    puts(__PRETTY_FUNCTION__);
}

int main() {
    add<int>();     // [T = int]
    add<>();        // couldn't infer template argument 'T'
    add();          // couldn't infer template argument 'T'
}
```

Default template parameters

```
template<typename T = char *>
void add()
{
    puts(__PRETTY_FUNCTION__);
}

int main() {
    add<int>();     // [T = int]
    add<>();        // [T = char *]
    add();          // [T = char *]
}
```

Type deduction in a nutshell:

- Any template parameters that were explicitly specified by the caller are fixed as whatever the caller said they were; they don’t participate any further in deduction.
- Each function parameter may contribute (or not) to the deduction of each remaining template parameter (or not).
- Deductions are carried out in parallel; they don’t cross-talk with each other.
- If any template parameter (that wasn’t specified by the caller) couldn’t be deduced, but has a default value, then it is fixed as its default value.
- Finally, the compiler checks to make sure that each template parameter (that wasn’t specified by the caller, and wasn’t fixed as its default) has been deduced at least once and that all deductions agree with each other.
- Furthermore, any function parameter that *does* contribute to deduction must match its function argument type *exactly*.

Now you know everything there is to know about template type deduction!

Only function templates do deduction

Kind of template	Year introduced	Type deduction happens?
Function	< 1998	Yes
Class	< 1998	No
Alias	2011	No
Variable	2014	No

```
template<typename T = void> struct foo {};
foo bar;    // error: use of class template 'foo' requires template arguments
foo<> bar;  // OK
```

We'll come back to why this is.

Wait... back up.

```
template<typename T>
struct is_void {
    static constexpr bool value = (some expression);
};

int main() {
    printf("%d\n", is_void<int>::value);   // 0
    printf("%d\n", is_void<void>::value);  // 1
}
```

We need a way to *specialize* `is_void` for a particular `T`.

Defining a template specialization

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

template<>
struct is_void<void> {
    static constexpr bool value = true;
};

int main() {
    printf("%d\n", is_void<int>::value);   // 0
    printf("%d\n", is_void<void>::value);  // 1
}
```

Defining a specialization in a nutshell

Prefix the definition with `template<>`, and then write the function definition as if you were ***using*** the specialization that you want to write. For function templates, because of their type deduction rules, this *usually* means not needing to write any more angle brackets at all.

But when a type can’t be deduced, you have to write the brackets:

```
template<typename T>
int my_sizeof()  { return sizeof (T); }

template<>
int my_sizeof<void>()  { return 1; }
```

Defining a specialization in a nutshell

Prefix the definition with `template<>`, and then write the function definition as if you were ***using*** the specialization that you want to write. For function templates, because of their type deduction rules, this *usually* means not needing to write any more angle brackets at all.

But when a type can't be deduced **or defaulted**, you have to write the brackets:

```
template<typename T = void>
int my_sizeof()  { return sizeof (T); }

template<>
int my_sizeof()  { return 1; }
```

Defining a template specialization

```
template<typename T>
T abs(T x)
{
    return (x >= 0) ? x : -x;
}

template<>
int abs<int>(int x)
{
    if (x == INT_MIN) throw std::domain_error("oops");
    return (x >= 0) ? x : -x;
}
```

Defining a template specialization

```
template<typename T>
T abs(T x)
{
    return (x >= 0) ? x : -x;
}

template<>
int abs<>(int x)
{
    if (x == INT_MIN) throw std::domain_error("oops");
    return (x >= 0) ? x : -x;
}
```

Defining a template specialization

```
template<typename T>
T abs(T x)
{
    return (x >= 0) ? x : -x;
}

template<>
int abs(int x)  // This is what you'll see most often in practice.
{
    if (x == INT_MIN) throw std::domain_error("oops");
    return (x >= 0) ? x : -x;
}
```

That’s *full specialization*.

Kind of template	Year introduced	Type deduction happens?	Full specialization allowed?
Function	< 1998	Yes	Yes
Class	< 1998	No	Yes
Alias	2011	No	No
Variable	2014	No	Yes

```
template<typename T> using myvec = std::vector<T>;
template<> using myvec<void> = void;
// error: explicit specialization of alias templates is not permitted
```

Alias templates can’t be specialized

```
template<typename T>
using myvec = std::vector<T>;

template<typename T>
void foo(myvec<T>& mv) {
    puts(__PRETTY_FUNCTION__);
}

int main() {
    std::vector<int> v;
    foo(v);  // void foo(myvec<T> &) [T = int]
}
```

We can “propagate `T` through” the definition of `myvec` to find that `foo<T>` takes `std::vector<T>`.

Class templates *can* be specialized

```
template<typename T>
struct myvec { using type = std::vector<T>; };

template<typename T>
void foo(typename myvec<T>::type& mv) {
    puts(__PRETTY_FUNCTION__);
}

int main() {
    std::vector<int> v;
    foo(v);  // couldn't infer template argument 'T'
}
```

Because we don't know what `myvec<T>::type` is until we know what `T` is.

So class templates can’t do deduction

```
template<typename T>
struct myvec {
    explicit myvec(T t);  // constructor
};

int main() {
    myvec v(1);  // error
}
```

Because we don’t know what parameter types `myvec<T>::myvec` might take, until we know what `T` is.

Now you know everything there is to know about *full specialization*!

Partial specialization

```
template<typename T>
constexpr bool is_array = false;

template<typename Tp>
constexpr bool is_array<Tp[]> = true;

int main()
{
    printf("%d\n", is_array<int>);    // 0
    printf("%d\n", is_array<int[]>);  // 1
}
```

A *partial specialization* is any specialization that is, itself, a template. It still requires further “customization” by the user before it can be used.

Partial specialization

```
template<typename T>
constexpr bool is_array = false;

template<typename Tp>
constexpr bool is_array<Tp[]> = true;

template<typename Tp, int N>
constexpr bool is_array<Tp[N]> = true;

template<> // this is a full specialization
constexpr bool is_array<void> = true;
```

A *partial specialization* is any specialization that is, itself, a template. It still requires further “customization” by the user before it can be used.

The user can explicitly specify values for the original template’s template parameters, but not for the partial specialization’s template parameters. So the latter *must be deducible*, or the partial specialization will never be used.

The number of template parameters on the partial specialization is *completely unrelated to* the number of template parameters on the original template.

Which specialization is called?

```
template<typename T> class A;

template<> class A<void>;

template<typename Tp> class A<Tp*>;

template<typename Tp> class A<Tp**>;

A<int*> a;    // T = int*; from among the base template
              // and its three specializations, A<Tp*> fits best

A<int***> a;  // T = int***; from among the base template
              // and its three specializations, A<Tp**> fits best
```

First, deduce all the template type parameters.
Then, if they exactly match some full specialization, of course we'll use that full specialization.
Otherwise, look for the *best-matching* partial specialization.
If the "best match" is hard to identify (ambiguous), give an error instead.

That’s *partial specialization*.

Kind of template	Year introduced	Type deduction happens?	Full specialization allowed?	Partial specialization allowed?
Function	< 1998	Yes	Yes	No
Class	< 1998	No	Yes	Yes
Alias	2011	No	No	No
Variable	2014	No	Yes	Yes

Function templates can’t be partially specialized

```
template<typename T>
bool is_pointer(T x)
{
    return false;
}

template<typename Tp>
bool is_pointer(Tp *x)
{
    return true;
}
```

Function templates can't be partially specialized

```
template<typename T>
bool is_pointer(T x)
{
    return false;
}

template<typename Tp>
bool is_pointer(Tp *x)
{
    return true;
}
```

Wrong!

This creates a pair of function templates in the same overload set. It *seems* to work in this case, but don't get used to it.

http://www.gotw.ca/publications/mill17.htm

Remember that the syntax for a *full* specialization always starts with `template<>`, and the syntax for a *partial* specialization always contains angle brackets after the template name.

Function templates can’t be partially specialized

```
template<typename T>
void is_pointer(T x)
{
  puts(__PRETTY_FUNCTION__);
}
```

```
template<>
void is_pointer(void *x)
{
  puts(__PRETTY_FUNCTION__);
}
```

```
int main()
{
    void *pv = nullptr;
    is_pointer(pv);
}
```

```
template<typename Tp>
void is_pointer(Tp *x)
{
  puts(__PRETTY_FUNCTION__);
}
```

How to partially specialize a function

Right:

```
template<typename T>
class is_pointer_impl { static bool _() { return false; } };

template<typename Tp>
class is_pointer_impl<Tp*> { static bool _() { return true; } };

template<typename T>
bool is_pointer(T x)
{
    return is_pointer_impl<T>::_();
}
```

If you need partial specialization, then you should delegate all the work to a class template, which can be partially specialized. Use the right tool for the job!

Now you know everything there is to know about

- class templates
- function templates
- variable templates
- alias templates
- template type deduction
- full specialization
- partial specialization

Next: Common template idioms

Some common templatey tasks

- Specify that a particular specialization is defined in another translation unit
- Tag dispatch
- Traits classes
- CRTP

Let’s talk about translation units.

Puzzle: Write a function to reverse the characters of a (possibly multibyte) string in place; so for example "Hello world" should become "dlrow olleH", and "Привет мир" should become "рим тевирП".

The standard (run-time) solution

```c
// The standard ANSI C solution.
void mbreverse(char *str, int n)
{
    char *end = str + n;
    char *p = str;
    while (p != end) {
        int len_this_char = mblen(p, end-p);
        reverse(p, len_this_char);
        p += len_this_char;
    }
    reverse(str, n);
}
```

The standard (run-time) solution

```
void mbreverse(char *str, int n);

int main()
{
    char latin1_buffer[] = "Hello world";
    setlocale(LC_CTYPE, "en_US.iso88591");
    mbreverse(latin1_buffer, 11);

    char utf8_buffer[] = "Привет мир";
    setlocale(LC_CTYPE, "en_US.utf8");
    mbreverse(utf8_buffer, 10);
}
```

Templatize all the things!

```
struct latin1; struct utf8;  // just some dummy types

template<typename Charset>
void mbreverse(char *str, int n);

int main()
{
    char latin1_buffer[] = "Hello world";
    mbreverse<latin1>(latin1_buffer, 11);

    char utf8_buffer[] = "Привет мир";
    mbreverse<utf8>(utf8_buffer, 10);
}
```

Templatize all the things!

```
template<typename Charset>
void mbreverse(char *str, int n)
{
    char *end = str + n;
    char *p = str;
    while (p != end) {
        int len_this_char = mblen<Charset>(p, end-p);
        reverse(p, len_this_char);
        p += len_this_char;
    }
    reverse(str, n);
}
```

Define *full specializations* of mblen()

```
// string_helpers.h
#pragma once

struct latin1; struct utf8;

template<typename Charset> int mblen(const char *, int);

template<> int mblen<latin1>(const char *, int) { return 1; }

template<> int mblen<utf8>(const char *p, int n)
{
    // ...uh-oh. I don’t want this much code in my .h file!
}
```

Define a specialization in another TU

```
// string_helpers.h
#pragma once

struct latin1; struct utf8;

template<typename Charset> int mblen(const char *, int);

template<> int mblen<latin1>(const char *, int) { return 1; }

template<> int mblen<utf8>(const char *p, int n);
```

Define a specialization in another TU

Declarations and definitions work just the way you'd expect them to.
In our .h file, we might have this:

```
template<typename Cs> int mblen(const char *, int);
template<> int mblen<latin1>(const char *, int);
template<> int mblen<utf8>(const char *, int);
template<typename Cs> int mbreverse(const char *, int) { impl }
```

And then in our .cpp file, we might have this:

```
template<> int mblen<latin1>(const char *p, int n) { impl }
template<> int mblen<utf8>(const char *p, int n) { impl }
```

Define a specialization in another TU

Declarations and definitions work just the way you'd expect them to.
In our .h file, we might have this:

```
template<typename Cs> int mblen(const char *, int);
template<> int mblen<latin1>(const char *, int);
template<> int mblen<utf8>(const char *, int);
template<typename Cs> int mbreverse(const char *, int) { impl }
```

*What if we don't want even **this** much code in our .h file?*

And then in our .cpp file, we might have this:

```
template<> int mblen<latin1>(const char *p, int n) { impl }
template<> int mblen<utf8>(const char *p, int n) { impl }
```

Pull mbreverse() out of the .h file too

```
// .h file
template<typename Cs> int mblen(const char *, int);
template<> int mblen<latin1>(const char *, int);
template<> int mblen<utf8>(const char *, int);
template<typename Cs> int mbreverse(const char *, int);
template<> int mbreverse<latin1>(const char *, int);
template<> int mbreverse<utf8>(const char *, int);

// .cpp file
template<> int mblen<latin1>(const char *p, int n) { impl }
template<> int mblen<utf8>(const char *p, int n) { impl }
template<> int mbreverse<latin1>(const char *p, int n) { impl }
template<> int mbreverse<utf8>(const char *p, int n) { impl }
```

Now we have repeated code!

```
// .h file
template<typename Cs> int mblen(const char *, int);
template<> int mblen<latin1>(const char *, int);
template<> int mblen<utf8>(const char *, int);
template<typename Cs> int mbreverse(const char *, int);
template<> int mbreverse<latin1>(const char *, int);
template<> int mbreverse<utf8>(const char *, int);

// .cpp file
template<> int mblen<latin1>(const char *p, int n) { impl }
template<> int mblen<utf8>(const char *p, int n) { impl }
template<> int mbreverse<latin1>(const char *p, int n) { impl }
template<> int mbreverse<utf8>(const char *p, int n) { impl }
```

Explicitly instantiate *without* specializing

```
// .h file
template<typename Cs> int mblen(const char *, int);
template<> int mblen<latin1>(const char *, int);
template<> int mblen<utf8>(const char *, int);
template<typename Cs> int mbreverse(const char *, int);
extern template int mbreverse<latin1>(const char *, int);
extern template int mbreverse<utf8>(const char *, int);

// .cpp file
template<> int mblen<latin1>(const char *p, int n) { impl }
template<> int mblen<utf8>(const char *p, int n) { impl }
template<typename Cs> int mbreverse(const char *, int) { impl }
template int mbreverse<latin1>(const char *p, int n);
template int mbreverse<utf8>(const char *p, int n);
```

Explicitly instantiate *without* specializing

This special syntax means "Please instantiate this template, with the given template parameters, as if it were being used right here." Semantically it is neither a declaration nor a definition; it's not giving the compiler any new information. It's just asking the compiler to instantiate the template.

It looks just like a full specialization without the `<>`.

```
template int abs(int);  // or: abs<>(int)  or: abs<int>(int)

template class vector<int>;

template bool is_void_v<void>;
```

Explicitly instantiate without specializing

This special syntax means "Please instantiate this template, with the given template parameters, as if it were being used right here." Semantically it is neither a declaration nor a definition; it's not giving the compiler any new information. It's just asking the compiler to instantiate the template.

To tell the compiler that you have done this in a different translation unit, and therefore the compiler needn't instantiate this template again in *this* .o file, just add `extern`:

```
extern template class vector<int>;
```

Explicitly instantiate without specializing

```
// .h file
template<typename Cs> int mblen(const char *, int);
template<> int mblen<latin1>(const char *, int);
template<> int mblen<utf8>(const char *, int);
template<typename Cs> int mbreverse(const char *, int);
extern template int mbreverse<latin1>(const char *, int);
extern template int mbreverse<utf8>(const char *, int);

// .cpp file
template<> int mblen<latin1>(const char *p, int n) { impl }
template<> int mblen<utf8>(const char *p, int n) { impl }
template<typename Cs> int mbreverse(const char *, int) { impl }
template int mbreverse<latin1>(const char *p, int n);
template int mbreverse<utf8>(const char *p, int n);
```

Some common templatey tasks

- Specify that a particular specialization is defined in another TU ✔
- Tag dispatch
- Traits classes
- CRTP

Specialize on a complex condition

```
template<class Element>
struct tree_iterator {
    // ...
    tree_iterator& operator ++();
};

template<class Element>
struct vector_iterator {
    // ...
    vector_iterator& operator ++();
    vector_iterator operator + (int);
};

template<class Element>
struct vector { using iterator = vector_iterator<Element>; /* ... */ };
template<class Element>
struct set { using iterator = tree_iterator<Element>; /* ... */ };
```

Specialize on a complex condition

```
template<class Iter>
Iter advance(Iter begin, int n)
{
    for (int i=0; i < n; ++i) {
        ++begin;
    }
    return begin;
}
```

The `std::advance` algorithm bumps an iterator by n positions.

For certain kinds of iterator (e.g. our `tree_iterator<E>`), we can’t do any better than this.

For random-access iterators (e.g. our `vector_iterator<E>`), we can do better.

Specialize on a complex condition

```
template<class Iter>
Iter advance(Iter begin, int n)
{
    for (int i=0; i < n; ++i) {
        ++begin;
    }
    return begin;
}

template<class E>
vector_iterator<E> advance(
    vector_iterator<E> begin, int n)
{
    return begin + n;
}
```

The `std::advance` algorithm bumps an iterator by n positions.

For certain kinds of iterator (e.g. our `tree_iterator<E>`), we can't do any better than this.

For random-access iterators (e.g. our `vector_iterator<E>`), we can do better.

Specialize on a complex condition

```
template<class Iter>
Iter advance(Iter begin, int n)
{
    for (int i=0; i < n; ++i) {
        ++begin;
    }
    return begin;
}
```

```
template<class E>
vector_iterator<E> advance(
    vector_iterator<E> begin, int n)
{
    return begin + n;
}
```

The `std::advance` algorithm bumps an iterator by n positions.

For certain kinds of iterator (e.g. our `tree_iterator<E>`), we can’t do any better than this.

For random-access iterators (e.g. our `vector_iterator<E>`), we can do better.

Function templates can’t be partially specialized!!

We control the “input type” `Iter`

```
template<class Element>
struct tree_iterator {
    // ...
    tree_iterator& operator ++();
    static std::false_type supports_plus;
};

template<class Element>
struct vector_iterator {
    // ...
    vector_iterator& operator ++();
    vector_iterator operator + (int);
    static std::true_type supports_plus;
};
```

Overload on `Iter::supports_plus`

```cpp
template<class It>
It advance_impl(It begin, int n, std::false_type /*sp*/) {
    for (int i=0; i < n; ++i) ++begin;
    return begin;
}

template<class It>
It advance_impl(It begin, int n, std::true_type /*sp*/) {
    return begin + n;
}

template<class Iter>
auto advance(Iter begin, int n) {
    return advance_impl(begin, n, Iter::supports_plus);
}
```

In practice it looks more like this

```
template<class Element> struct tree_iterator {
    using supports_plus = std::false_type;  // member typedef
};

template<class Element> struct vector_iterator {
    using supports_plus = std::true_type;  // member typedef
};


template<class It>
It advance_impl(It begin, int n, std::false_type) {
    for (int i=0; i < n; ++i) ++begin;
    return begin;
}

template<class It>
It advance_impl(It begin, int n, std::true_type) {
    return begin + n;
}

template<class Iter>
auto advance(Iter begin, int n) {
    return advance_impl(begin, n, typename Iter::supports_plus{});  // create an object of that type
}
```

In practice it looks more like this

```
template<class Element> struct tree_iterator {
    using supports_plus = std::false_type;  // member typedef
};

template<class Element> struct vector_iterator {
    using supports_plus = std::true_type;  // member typedef
};


template<class It>
It advance_impl(It begin, int n, std::false_type) {
    for (int i=0; i < n; ++i) ++begin;
    return begin;
}

template<class It>
It advance_impl(It begin, int n, std::true_type) {
    return begin + n;
}

template<class Iter>
auto advance(Iter begin, int n) {
    return advance_impl(begin, n, typename Iter::supports_plus{});  // create an object of that type
}
```

Why typename?

Dependent names

C++'s grammar is not context-free. Normally, in order to parse a function definition, you need to know something of the context in which that function is being defined.

```
void foo(int x) {
    A (x);  // if A is a function, this is a function call;
            // if it’s a type, this is a declaration
}
```

So how can we possibly parse ***this*** template definition?

```
template<class T>
void foo(int x) {
    T::A (x);
}

struct S1 { static void A(int); };  ...  foo<S1>(0);
struct S2 { using A = int; };       ...  foo<S2>(0);
```

Dependent names

C++'s grammar is not context-free. Normally, in order to parse a function definition, you need to know something of the context in which that function is being defined.

```
void foo(int x) {
    A (x);  // if A is a function, this is a function call;
            // if it's a type, this is a declaration
}
```

So how can we possibly parse ***this*** template definition?

```
template<class T>
void foo(int x) {
    T::A (x);
}
```

Solution: By default, C++ will *assume* that any name whose lookup is dependent on a template parameter refers to a non-type, non-template, plain old variable/function/object-style entity.

```
struct S1 { static void A(int); };   ...   foo<S1>(0);
struct S2 { using A = int; };        ...   foo<S2>(0);
```

```
error: dependent-name 'T:: A' is parsed as a non-type, but instantiation yields a type
```

Dependent names

C++'s grammar is not context-free. Normally, in order to parse a function definition, you need to know something of the context in which that function is being defined.

```
void foo(int x) {
    A (x);  // if A is a function, this is a function call;
            // if it's a type, this is a declaration
}
```

So how can we possibly parse ***this*** template definition?

```
template<class T>
void foo(int x) {
    typename T::A (x);
}
```

Solution: By default, C++ will *assume* that any name whose lookup is dependent on a template parameter refers to a non-type, non-template, plain old variable/function/object-style entity.

```
struct S1 { static void A(int); };   ...   foo<S1>(0);
struct S2 { using A = int; };        ...   foo<S2>(0);
```

```
error: no type named 'A' in 'struct S2'
```

Similarly to refer to a template

```
struct S1 { static constexpr int A = 0; };
struct S2 { template<int N> static void A(int) {} };
struct S3 { template<int N> struct A {}; };
int x;

template<class T>
void foo() {
    T::A < 0 > (x);  // if T::A is an object, this is a pair of comparisons;
                     // if T::A is a typename, this is a syntax error;
                     // if T::A is a function template, this is a function call;
                     // if T::A is a class or alias template, this is a declaration.
}
```

Similarly to refer to a template

```
struct S1 { static constexpr int A = 0; };
struct S2 { template<int N> static void A(int) {} };
struct S3 { template<int N> struct A {}; };
int x;

template<class T>
void foo() {
    T::A < 0 > (x);
}

int main()
{
    foo<S1>();
}
```

Similarly to refer to a template

```
struct S1 { static constexpr int A = 0; };
struct S2 { template<int N> static void A(int) {} };
struct S3 { template<int N> struct A {}; };
int x;

template<class T>
void foo() {
    T::template A < 0 > (x);
}

int main()
{
    foo<S2>();
}
```

Similarly to refer to a template

```
struct S1 { static constexpr int A = 0; };
struct S2 { template<int N> static void A(int) {} };
struct S3 { template<int N> struct A {}; };
int x;

template<class T>
void foo() {
    typename T::template A < 0 > (x);
}

int main()
{
    foo<S3>();
}
```

Revisit our tag dispatch example

```
template<class Element> struct tree_iterator {
    using supports_plus = std::false_type;  // member typedef
};

template<class Element> struct vector_iterator {
    using supports_plus = std::true_type;  // member typedef
};


template<class It>
It advance_impl(It begin, int n, std::false_type) {
    for (int i=0; i < n; ++i) ++begin;
    return begin;
}

template<class It>
It advance_impl(It begin, int n, std::true_type) {
    return begin + n;
}

template<class Iter>
auto advance(Iter begin, int n) {
    return advance_impl(begin, n, typename Iter::supports_plus{});  // create an object of that type
}
```

Now you know everything there is to know about *tag dispatch*!

But what if we don’t control `Iter`?

```
template<class Element>
struct tree_iterator {
    // ...
    tree_iterator& operator ++();
    using supports_plus = std::false_type;
};

int main()
{
    int buf[10];
    int *begin = buf;
    int *fourth = advance(buf, 4);  // [Iter = int *]
}
```

We have no class (such as `tree_iterator`) off of which to hang our `supports_plus` member typedef. What do we do in this situation?

But what if we don’t control `Iter`?

```
template<class /*Iter*/>
struct iter_traits {
    using supports_plus = std::false_type;
};

template<class T>
struct iter_traits<T *> {
    using supports_plus = std::true_type;
};

template<class T>
struct iter_traits<vector_iterator<T>> {
    using supports_plus = std::true_type;
};
```

We have no class (such as `tree_iterator`) off of which to hang our `supports_plus` member typedef. What do we do in this situation?

We **create** a class off of which to hang our member typedef!

Overload on supports_plus again

```
template<class Iter>
auto advance(Iter begin, int n)
{
    return advance_impl(
        begin, n, typename iter_traits<Iter>::supports_plus{}
    );
}
```

This will be `false_type` for any type `Iter` at all, except those types for which we've specialized the `iter_traits` class template.

STL best practice: _t synonyms

```
template<class Iter>
using iter_supports_plus_t =
    typename iter_traits<Iter>::supports_plus;

template<class Iter>
auto advance(Iter begin, int n)
{
    return advance_impl(
        begin, n, iter_supports_plus_t<Iter>{}
    );
}
```

Now you know everything there is to know about *traits classes*!

Some common templatey tasks

- Specify that a particular specialization is defined in another TU ✔
- Tag dispatch ✔
- Traits classes ✔
- CRTP

Add common functionality to a class

```
struct Cat {
    void speak() { puts("meow"); }
    void speaktwice() { speak(); speak(); }
};

struct Dog {
    void speak() { puts("woof"); }
    void speaktwice() { speak(); speak(); }
};

int main() {
    Cat c; c.speak(); speaktwice(c);
    Dog d; d.speak(); speaktwice(d);
}
```

This implementation falls afoul of “DRY”: Don’t Repeat Yourself.

We really want to factor out the repeated `speaktwice()` code into a common base class.

Let’s call that common base class `DoubleSpeaker`.

Add common functionality to a class

```
struct DoubleSpeaker {
    void speaktwice() { speak(); speak(); }
};

struct Cat : public DoubleSpeaker {
    void speak() { puts("meow"); }
};

struct Dog : public DoubleSpeaker {
    void speak() { puts("woof"); }
};

int main() {
    Cat c; c.speak(); speaktwice(c);
    Dog d; d.speak(); speaktwice(d);
}
```

Unfortunately, this doesn't work. DoubleSpeaker can't call `speak()` because `speak()` isn't defined in this scope.

`speak()` is defined only for Cats and Dogs, so if we're going to use `speak()` here, we need to get our hands on a Cat or a Dog.

(Or we could make `speak()` a virtual member function. See the next slide for why that's not always a good idea.)

We could make everything virtual

```
struct VirtualDoubleSpeaker {
    virtual void speak() = 0;
    void speaktwice() { speak(); speak(); }
};

struct VirtualCat : public VirtualDoubleSpeaker {
    void speak() { puts("meow"); }
};

struct VirtualDog : public VirtualDoubleSpeaker {
    void speak() { puts("woof"); }
};
```

```
clang++ test.cc -S -O3 -fomit-frame-pointer

__ZN20VirtualDoubleSpeaker10speaktwiceEv:
  pushq %rbx
  movq  %rdi, %rbx
  movq  (%rbx), %rax
  callq *(%rax)   # indirect call to speak()
  movq  (%rbx), %rax
  movq  %rbx, %rdi
  popq  %rbx
  jmpq  *(%rax)   # indirect tailcall to speak()
```

Two virtual method calls, plus the original virtual method call to `speaktwice`? That's pretty costly. We'd like `Cat::speaktwice` to just do the right thing, statically.
Plus, polymorphism is *viral*.

Or we could use the CRTP

```
template<typename CD>
struct DoubleSpeaker {
    void speaktwice() {
        CD *cat_or_dog = static_cast<CD *>(this);
        cat_or_dog->speak();
        cat_or_dog->speak();
    }
};

struct Cat : public DoubleSpeaker<Cat> {
    void speak() { puts("meow"); }
};

struct Dog : public DoubleSpeaker<Dog> {
    void speak() { puts("woof"); }
};
```

Here’s our next attempt.

`DoubleSpeaker` is now a class template. To inherit from it, the user has to pass in a parameter that tells `DoubleSpeaker` what kind of animal it is, so that `DoubleSpeaker` knows how to make that animal speak.

Notice that even though we’re using the name `CD *cat_or_dog`, `CD` could actually be any type `T` at all, as long as `T` has a `.speak()` member function and inherits from `DoubleSpeaker<T>`.

CRTP vs. virtual

```
template<typename D>
struct DoubleSpeaker {
    void speaktwice() {
        D *derived = static_cast<D*>(this);
        derived->speak();
        derived->speak();
    }
};
struct Cat : public DoubleSpeaker<Cat> {
    void speak() { puts("meow"); }
};

struct VirtualDoubleSpeaker {
    virtual void speak() = 0;
    void speaktwice() { speak(); speak(); }
};
struct VirtualCat : public VirtualDoubleSpeaker {
    void speak() { puts("meow"); }
};
```

```
clang++ test.cc -S -O3 -fomit-frame-pointer

__ZN13DoubleSpeakerI3CatE10speaktwiceEv:
  pushq %rbx
  leaq  L_.str(%rip), %rbx   # "meow"
  movq  %rbx, %rdi
  callq __Z4putsPKc
  movq  %rbx, %rdi
  popq  %rbx
  jmp   __Z4putsPKc

__ZN20VirtualDoubleSpeaker10speaktwiceEv:
  pushq %rbx
  movq  %rdi, %rbx
  movq  (%rbx), %rax
  callq *(%rax)   # indirect call to speak()
  movq  (%rbx), %rax
  movq  %rbx, %rdi
  popq  %rbx
  jmpq  *(%rax)   # indirect tailcall to speak()
```

Questions?

- Specify that a particular specialization is defined in another TU ✔
- CRTP ✔
- Tag dispatch ✔
- Traits classes ✔

Kind of template	Year introduced	Type deduction happens?	Full specialization allowed?	Partial specialization allowed?
Function	< 1998	Yes	Yes	No
Class	< 1998	No	Yes	Yes
Alias	2011	No	No	No
Variable	2014	No	Yes	Yes