Template ~~Meta~~-Programming

https://wg21.link/n4606
http://melpon.org/wandbox
http://webcompiler.cloudapp.net

Recap of Part I

- class templates
- function templates
- variable templates
- alias templates
- template type deduction
- reference collapsing
- full specialization
- partial specialization
- explicit instantiation

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

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

Whoa, what? You mean `tree_iterator<Element>`, right?

```
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 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>; /* ... */ };
```

Whoa, what? You mean `tree_iterator<Element>`, right?

N4606 §14.6.1 [temp.local]/1: Inside the definition of a class template, the bare *template-name* can be used as a *type-name*, in which case it's basically as if you put `<`*`all the template-parameters`*`>` after it.

So this is fine, and helps cut down on repetition. I recommend it.

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`

```
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; };  // S1::A is an object
struct S2 { template<int N> static void A(int) {} };  // S2::A is a function template
struct S3 { template<int N> struct A {}; };  // S3::A is a class template
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; };  // S1::A is an object
struct S2 { template<int N> static void A(int) {} };  // S2::A is a function template
struct S3 { template<int N> struct A {}; };  // S3::A is a class template
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; };  // S1::A is an object
struct S2 { template<int N> static void A(int) {} };  // S2::A is a function template
struct S3 { template<int N> struct A {}; };  // S3::A is a class template
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; };  // S1::A is an object
struct S2 { template<int N> static void A(int) {} };  // S2::A is a function template
struct S3 { template<int N> struct A {}; };  // S3::A is a class template
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*!

So you're getting a linker error...

Let's talk about declarations and definitions.

- Normal entities, like functions and variables.
- `inline` functions and variables.
- Templated entities.

N4606: "Every program shall contain exactly one definition of every non-inline function or variable that is *odr-used* in that program outside of a discarded statement."

"Discarded statement" is C++17ese for "untaken branch of an if-constexpr".

```
int a(int);
extern int b;

int main() {
  return a(b);
}
```

```
int a(int x) {
  return x+2;
}

int b = 2;
```

Templates: what and where

Let's talk about declarations and definitions.

- **Normal entities, like functions and variables.**
- `inline` functions and variables.
- Templated entities.

N4606: "Every program shall contain exactly one definition of every non-inline function or variable that is *odr-used* in that program outside of a discarded statement."

```
int a(int);
extern int b;

int main() {
  return a(b);
}
```

```
int a(int x) {
  return x+2;
}

int b = 2;
```

Templates: what and where

Let's talk about declarations and definitions.

- Normal entities, like functions and variables.
- **`inline` functions and variables.**
- Templated entities.

N4606: "An inline function or variable shall be defined in every translation unit in which it is *odr-used* outside of a discarded statement."

Yes, C++17 finally allows `inline` variables.

```
inline int a(int x)
{ ... }

int main() {
  return a(42);
}
```

Defined in two places!

For `inline` things, this is okay.

```
inline int a(int x)
{ ... }

int foobar() {
  return a(43);
}
```

Templates: what and where

Let's talk about declarations and definitions.

- Normal entities, like functions and variables.
- **`inline` functions and variables.**
- Templated entities.

N4606: "An inline function or variable shall be defined in every translation unit in which it is *odr-used* outside of a discarded statement."

```
inline int a(int);


int main() {
  return a(42);
}
```

OK in practice; officially ill-formed

```
inline int a(int x)
{ ... }

int foobar() {
  return a(43);
}
```


N4606 §3.2 [basic.def.odr] /4

Templates: what and where

Let's talk about declarations and definitions.

- Normal entities, like functions and variables.
- **`inline` functions and variables.**
- Templated entities.

N4606: "An inline function or variable shall be defined in every translation unit in which it is *odr-used* outside of a discarded statement."

```
inline int a(int);


int main() {
  return a(42);
}
```

officially ill-formed, AND gives a linker error in practice

```
inline int a(int x)
{ ... }


// a is unused here
```


N4606 §3.2 [basic.def.odr] /4

Templates: what and where

Let's talk about declarations and definitions.

- Normal entities, like functions and variables.
- `inline` functions and variables.
- **Templated entities.**

```
template<class T>
int a() { ... }

int main() {
  return a<int>();
}
```

```
template<class T>
int a() { ... }

int foobar() {
  return a<int>();
}
```

Templates: what and where

Let's talk about declarations and definitions.

- Normal entities, like functions and variables.
- `inline` functions and variables.
- **Templated entities.**

N4606: "A function template, member function of a class template, variable template, or static data member of a class template shall be defined in every translation unit in which it is implicitly instantiated, unless the corresponding specialization is explicitly instantiated in some translation unit."

```
template<class T>
int a();

int main() {
  return a<int>();
}
```

OK

```
template<class T>
int a() { ... }

template int
a<int>();
```

Remember this is the syntax for an *explicit instantiation*.

N4606 §14 [temp] /7

Templates: what and where

Let's talk about declarations and definitions.

- Normal entities, like functions and variables.
- `inline` functions and variables.
- **Templated entities.**

N4606: "A function template, member function of a class template, variable template, or static data member of a class template shall be defined in every translation unit in which it is implicitly instantiated, unless the corresponding specialization is explicitly instantiated in some translation unit."

```
template<class T>
int a();

int main() {
  return a<int>();
}
```

OK in practice; officially ill-formed

```
template<class T>
int a() { ... }

int foobar() {
  return a<int>();
}
```

because `a<int>` is merely *implicitly* instantiated here

N4606 §14 [temp] /7

Templates: what and where

Let's talk about declarations and definitions.

- Normal entities, like functions and variables.
- `inline` functions and variables.
- **Templated entities.**

N4606: "A function template, member function of a class template, variable template, or static data member of a class template shall be defined in every translation unit in which it is implicitly instantiated, unless the corresponding specialization is explicitly instantiated in some translation unit."

```
template<class T>
int a();

int main() {
  return a<int>();
}
```

officially ill-formed, AND gives a linker error in practice

```
template<class T>
int a() { ... }

// a<int> is unused
// here
```


N4606 §14 [temp] /7

Templates: what and where

There are at least four ways template stuff can go wrong:

- The compiler can’t figure out the template parameters. Result: Compiler error.
- Problems in instantiating a *declaration*. Result: Compiler error.
- Problems in instantiating a *definition*. Result: Compiler error.
- Instantiating only a *declaration* (or nothing at all),
  when you thought you were getting a *definition*. Result: Linker error.

Q1. When is instantiation needed?

A decent rule of thumb is: ***Never instantiate anything you don’t absolutely 100% have to.***

```
template <typename T>
struct C {
    static_assert(sizeof(T) == 2);
};

C<int> *myvar;  // OK: the definition of C<int> isn’t needed
```


N4606 §14.7.1 [temp.inst] /1

Sidebar: `static_assert`

`static_assert(false)` makes the program ill-formed.

`static_assert(`*`some-falsey-expression-dependent-on-`*`T)` makes the program ill-formed **only** if the template is actually instantiated.

```
template<class T> void f() { static_assert(sizeof(int)==0); }  // ERROR
template<class T> void g() { static_assert(sizeof(T)==0); }    // OK
```

In theory, as of this writing, a sufficiently smart compiler could refuse to compile `g()` as well. In practice, no compiler does, nor would they.

N4606 §7 [dcl.dcl] /6, §14.6 [temp.res] /(8.3)

Q1. When is instantiation needed?

A decent rule of thumb is: ***Never instantiate anything you don’t absolutely 100% have to.***

```
template <typename T>
struct C {
    static constexpr int sdm = T(nullptr);
    static void smf() { static_assert(sizeof(T) == 2); }
    void f() { static_assert(sizeof(T) == 2); }
};

C<int> myvar;  // OK: instantiate declarations, but not definitions
```


N4606 §14.7.1 [temp.inst] /1

Q1. When is instantiation needed?

A decent rule of thumb is: ***Never instantiate anything you don't absolutely 100% have to.***

```
template <typename T>
struct C {
    static int sdm;
    static int smf1();
    static int smf2() { static_assert(sizeof(T) == 2); }
};

int howaboutnow = C<int>::sdm;  // still nope; linker error
int ornow = C<int>::smf1();     // ditto
```


What's the difference between `smf1` and `smf2` here?

Q1. When is instantiation needed?

Remember, template definitions behave basically like they have the `inline` keyword attached to them.

```
inline int ff1();
inline int ff2() { static_assert(sizeof(int) == 2); }

int a = ff1();      // OK, but ff1 better be defined elsewhere
int b = ff2();      // ERROR: static_assert failed
```


N4606 §14.7.1 [temp.inst] /1

Q1. When is instantiation needed?

Remember, template definitions behave basically like they have the `inline` keyword attached to them.

```
template<class T> int ff1();
template<class T> int ff2() { static_assert(sizeof(T) == 2); }

int a = ff1<int>();  // OK, but ff1<int> better be defined elsewhere
int b = ff2<int>();  // ERROR: static_assert failed
```


N4606 §14.7.1 [temp.inst] /1

Q1. When is instantiation needed?

Recap: An *explicit instantiation* declaration can say “I promise this template instantiation exists elsewhere; please *don’t* instantiate it in this translation unit.”

```
template<class T> int ff1();
template<class T> int ff2() { static_assert(sizeof(T) == 2); }
template<class T> int ff3() { static_assert(sizeof(T) == 2); }
extern template int ff3<int>();  // explicit instantiation

int a = ff1<int>();  // OK, but ff1<int> better be defined elsewhere
int b = ff2<int>();  // ERROR: static_assert failed
int c = ff3<int>();  // OK, but ff3<int> better be defined elsewhere
```


N4606 §14.7.1 [temp.inst] /1

Q1. When is instantiation needed?

Recap: An explicit (full) *specialization* works basically like a plain old function, because you've taken away all the template parameters. There's no templatey stuff left *to* instantiate, in this case.

```
template<class T> int ff1();
template<class T> int ff2() { static_assert(sizeof(T) == 2); }
template<class T> int ff3() { static_assert(sizeof(T) == 2); }
template<> int ff3<int>();  // explicit (full) specialization

int a = ff1<int>();  // OK, but ff1<int> better be defined elsewhere
int b = ff2<int>();  // ERROR: static_assert failed
int c = ff3<int>();  // OK, but ff3<int> better be defined elsewhere
```


N4606 §14.7.1 [temp.inst] /1

Q1. When is instantiation needed?

Variable templates work similarly, but watch out that Clang and GCC currently disagree as to what constitutes a “definition” of a variable template.

N4606 §14.7.1 [temp.inst] /5: Unless a variable template specialization has been explicitly instantiated or explicitly specialized, the variable template specialization is implicitly instantiated when the specialization is used.

```
template <typename T> int vt;
  // GCC: this is a definition just as much as "int v;" would be
  // Clang: this is just a declaration, akin to "extern int v;"

int i = vt<int>;  // GCC/MSVC reserve space for vt<int> here;
                  // Clang merely sets you up for a linker error
```


N4606 §14.7.1 [temp.inst] /5

Now you know everything there is to know about implicit instantiation!

Let’s talk variadic templates!

Hopefully by now (2016) everyone in the audience has seen variadic templates somewhere, and is vaguely familiar with why they’re useful, so I’m going to skip the “motivation” section in the interest of time.

```
template<class T, class... Args>
void mcdonald(void *where, Args&&... args) {
    new (where) T(std::forward<Args>(args)...);
}

void populate(void *here, void *there, int meow) {
    mcdonald<Cat>(here, meow);          // new (here) Cat(meow)
    mcdonald<Cat>(there, meow, meow); // new (there) Cat(meow, meow)
}
```

Variadic template parameter deduction

```
template<class T, class... Us>
void f(Us... us)
{ puts(__PRETTY_FUNCTION__); }   // MSVC: __FUNCSIG__

int main()
{
    f<char>(0,1,2);  // [with T=char, Us=<int,int,int>]
}
```

Variadic template parameter deduction

```
template<class T, class... Us>
void f(Us... us)
{ puts(__PRETTY_FUNCTION__); }   // MSVC: __FUNCSIG__

int main()
{
    f<char>(0,1,2);  // [with T=char, Us=<int,int,int>]
    f<char,char>(0,1,2);  // [with T=char, Us=<char,int,int>]
}
```

Type deduction in a nutshell, v2:

As far as explicitly specified template parameters are concerned, the first pack-expansion (`Ts... ts`) encountered in the *template parameter* list “soaks up” all the remaining explicitly specified template parameters. The type deduction step might wind up *lengthening* `ts`, but will never shorten it.

```
template<class T, class... Us, class V> void f();
f<int,char,int>();  // T=int; Us=<char,int>; V cannot be deduced

template<class... Ts, class U> void g(U);
g<int,char>(3.1);  // Ts=<int,char>; U=double

template<class... Ts> void h(Ts...);
h<int,char>(0, 0, 3.1);  // Ts=<int,char,double>
```

Type deduction in a nutshell, v2:

As far as *deduction* is concerned, a parameter-pack (`Ts... ts`) contributes to deduction only if it comes at the very end of the function parameter list. Otherwise, it does not contribute to deduction.

```
template<class... Ts, class U> void f(U, Ts...);
f('x', 1, 2);  // U=char; Ts=<int,int>

template<class T, class... Us> void g(Us... us, T);
g('x', 1, 2);  // us doesn’t contribute to deduction, so this fails

template<class T, class... Us> void h(Us... us, T);
h<int,int,int>('x', 1, 2);  // us doesn’t contribute to deduction,
// but we explicitly stated T=int, Us=<int,int>, which happens to work!
```

Type deduction in a nutshell, v2:

As far as *deduction* is concerned, a parameter-pack (`Ts... ts`) contributes to deduction only if it comes at the very end of the function parameter list. Otherwise, it does not contribute to deduction.

```
template<class... Ts, class... Us> void f(Ts..., Us...);
f<int,int>(1, 2, 3);  // Ts must start with <int,int>; yet ts
                      // doesn't contribute to deduction
```

Clang: claims `Ts` can't be deduced; *compiler error*

MSVC/GCC: assumes `Ts` won't be lengthened; *`Ts=<int,int> Us=<double>`*

Now you know everything there is to know about template type deduction, even for variadics!

Two patterns to know:
The CRTP
The Mixin Pattern

CRTP: add common functionality to classes

```
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(); c.speaktwice();
    Dog d; d.speak(); d.speaktwice();
}
```

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`.

CRTP: add common functionality to classes

```
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(); c.speaktwice();
    Dog d; d.speak(); d.speaktwice();
}
```

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()
```

Mixin: Extend an existing polymorphic class

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

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

struct VerboseCat : public Cat {
    virtual void speaktwice() { speak(); speak(); }
};

struct VerboseDog : public Dog {
    virtual void speaktwice() { speak(); speak(); }
};
```

For any Objective-C programmers in the audience: we're kind of talking about *categories* here.

Classes `Cat` and `Dog` are provided by our existing polymorphic hierarchy. We want to extend them to make verbose `Cats` and `Dogs` who can `speaktwice()` in addition to all their normal behaviors.

We'll need a way to create our `VerboseCats`; i.e., we'll probably make a `VerboseCatFactory` to sit alongside our (now-deprecated) `CatFactory`.

The mixin-derived class still IS-A base

```cpp
// mp_throw.hpp

struct Trace {
  Trace(int line);
  std::string get_stack_trace() const;
};

template<class E>
struct Traceable : public E, public Trace {
  Traceable(E e, int line)
    : E(std::move(e)), Trace(line) {}
};

template<class E>
auto make_traceable(E e, int line) {
  return Traceable<E>(std::move(e), line);
}

#define MP_THROW(e) \
  throw make_traceable((e), __LINE__)
```

```cpp
// client code

using namespace std;

void foo()
{
  MP_THROW(logic_error("oops"));
}

int main()
{
  try {
    foo();
  } catch (const exception& e) {
    if (auto t = dynamic_cast<const Trace*>(&e))
    {
      cout << t->get_stack_trace();
    }
  }
}
```

What’s new in C++17? (Oulu update)

- `inline` variables
- `template<auto>`
- Template parameter deduction for class template constructors
- Explicit deduction guides for class template constructors

`template<auto>`

I haven’t talked much about template *non-type* parameters at all, have I? Well, they work how you’d expect by now.

```
template<class T, unsigned long N>
class array { /* ... */ };
```

They are ***mostly*** useful for metaprogramming, though.

```
template<size_t... Ns> struct index_sequence { /* ... */ };
using MySequence = index_sequence<0,1,2,3,4>;
```


N4606 §14.3.2 [temp.arg.nontype] /1

template<auto>

In metaprogramming you end up with stuff like this:

```
template<class Ty, Ty Value>
struct integral_constant {
    static constexpr auto value = Value;
};

using fortytwo_type = integral_constant<int, 42>;
using true_type = integral_constant<bool, true>;
```


N4606 §14.3.2 [temp.arg.nontype] /1

template<auto>

Here’s what we’d like to do — and what C++17 actually lets us do:

```
template<auto Value>
struct new_integral_constant {
    static constexpr auto value = Value;
};

using fortytwo_type = new_integral_constant<42>;
using true_type = new_integral_constant<true>;
```


N4606 §14.3.2 [temp.arg.nontype] /1

Remember when I said this in Part 1?

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.

Forward works: If `T` is `int`, we know that `myvec<T>`'s constructor takes an `int` parameter.
But what we need here is to go *backward*: If `myvec<U>`'s constructor takes an `int` parameter, determine the value of `U`.

Class template deduction

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

int main() {
    myvec v(1);  // OK in C++17
}
```

Construct a fictitious set of function overloads matching all the constructors of the `myvec` class template. (This doesn't involve instantiating anything!) In our case, there's only one:

```
template<class T>
auto make_myvec(T t);
```

Now use overload resolution to resolve `make_myvec(1)`. Deduce `T=int`. Ta-da!

Result: `myvec(T) [with T=int]`

N4606 §13.3.1.8 [over.match.class.deduct]

Class template deduction

```
template<typename T>
struct myvec {
    myvec(T t);
    myvec(T *p);
};

int main() {
    myvec v(1);  // OK in C++17
}
```

Construct a fictitious set of function overloads:

```
template<class T>
auto make_myvec(T t);

template<class T>
auto make_myvec(T *p);
```

Now use overload resolution to resolve `make_myvec(1)`.

Result: `myvec(T) [with T=int]`

N4606 §13.3.1.8 [over.match.class.deduct]

Class template deduction

```
template<typename T>
struct myvec {
    myvec(T t);
    myvec(T *p);
};

int main() {
    int i;
    myvec v(&i);  // OK in C++17
}
```

Construct a fictitious set of function overloads:

```
template<class T>
auto make_myvec(T t);

template<class T>
auto make_myvec(T *p);
```

Now use overload resolution to resolve `make_myvec(&i)`.

Result: `myvec(T*) [with T=int]`

N4606 §13.3.1.8 [over.match.class.deduct]

Class template deduction

```
template<typename T>
struct myvec {
    myvec(T t);
    myvec(double d);
};

int main() {
    myvec v(1.0);  // OK in C++17
}
```

Construct a fictitious set of function overloads:

```
template<class T>
auto make_myvec(T t);

template<class T>
auto make_myvec(double d);
```

Now use overload resolution to resolve `make_myvec(1.0)`. The second overload doesn't participate because T cannot be deduced.
Result: `myvec(T) [with T=double]`

N4606 §13.3.1.8 [over.match.class.deduct]

Class template deduction

```
template<typename T>
struct myvec {
    myvec(T t);
};
template<typename T>
struct myvec<T*> {
    myvec();
};

int main() {
    int i;
    myvec v(&i);  // OK in C++17
}
```

Construct a fictitious set of function overloads:

```
template<class T>
auto make_myvec(T t);
```

Now use overload resolution to resolve `make_myvec(&i)`.

Result: `myvec(T) [with T=int*]`

But we have a partial specialization for `myvec<int*>`! So, hard error... I think.

N4606 §13.3.1.8 [over.match.class.deduct]

What’s the difference between these two classes?

```
template<typename T>
struct myvec {
    struct iter { using type = T; };
    myvec(T t);
};

template<typename T>
struct myvec {
    struct iter { using type = T; };
    myvec(iter::type t);
};
```

What’s the difference between these two classes?

```
template<typename T>
struct myvec {
    struct iter { using type = T; };
    myvec(T t);
};
```

```
template<typename T>
struct myvec {
    struct iter { using type = T; };
    myvec(iter::type t);
};
```

Construct a fictitious set of function overloads:

```
template<class T>
auto make_myvec(T t);
```

```
template<class T>
auto make_myvec(
    myvec<T>::iter::type t
);
```

Now use overload resolution to resolve `make_myvec(1)`.
In the blue case, deduction fails.

http://wg21.link/p0091r3

Explicit deduction guides

```
template<typename T>
struct myvec {
    myvec(T t);
    myvec(double d);
};



int main() {
    myvec v(1.0);  // Let’s say we really want this to resolve
}                  // to myvec<int> instead of myvec<double>...
```


N4606 §14.9 [temp.deduct.guide]

Explicit deduction guides

```
template<typename T>
struct myvec {
    myvec(T t);
    myvec(double d);
};

myvec(double) -> myvec<int>;  // this is a deduction guide

int main() {
    myvec v(1.0);  // Let's say we really want this to resolve
}                  // to myvec<int> instead of myvec<double>...
```


N4606 §14.9 [temp.deduct.guide]

Explicit deduction guides

```
template<typename T>
struct myvec {
    myvec(T t);
    myvec(double d);
};

myvec(double) -> myvec<int>;

int main() {
    myvec v(1.0);
}
```

Construct a fictitious set of function overloads:

```
template<class T>
auto make_myvec(T t);

template<class T>
auto make_myvec(double d);

auto make_myvec(double);
```

Now the third one is clearly the best match.
Also: Deduction guides can themselves be templates.

N4606 §14.9 [temp.deduct.guide]

Now you know all the crazy stuff coming in C++17!

Questions?