Template ~~Meta~~-Programming

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

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

```
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

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.

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

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 until slide 21.

Likewise: 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 *can 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.

Template classes are still classes

```
struct S
{
    static int sdm;
};



int main()
{
    return S::sdm;
}
```

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

The resulting “template classes” follow all the same rules as normal classes. In particular, each static data member must be defined somewhere if you want to use it.

```
test.o: In function `main':
test.cc:(.text+0x6): undefined reference to
`S::sdm'
```

Template classes are still classes

```
struct S
{
    static int sdm;
};

int S::sdm = 42;


int main()
{
    return S::sdm;
}
```

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

The resulting “template classes” follow all the same rules as normal classes. In particular, each static data member must be defined somewhere if you want to use it.

Template classes are still classes

```
template<class T>
struct ST
{
    static int sdm;
};



int main()
{
    return ST<char>::sdm;
}
```

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

The resulting “template classes” follow all the same rules as normal classes. In particular, each static data member must be defined somewhere if you want to use it.

```
test.o: In function `main':
test.cc:(.text+0x6): undefined reference to
`ST<char>::sdm'
```

Template classes are still classes

```
template<class T>
struct ST
{
    static int sdm;
};

template<class T>
int ST<T>::sdm = 42;

int main()
{
    return ST<char>::sdm;
}
```

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

The resulting “template classes” follow all the same rules as normal classes. In particular, each static data member must be defined somewhere if you want to use it.

Templates are kind of like `inline`

```
template<class T>
struct ST
{
    static int sdm;
};

template<class T>
int ST<T>::sdm = 42;

int main()
{
    return ST<char>::sdm;
}
```

How come I can define `ST<T>::sdm` in a header file and `#include` it all over the place, whereas I get a multiple-definition linker error if I try the same thing with `S::sdm`?

The same C++ Standard wording that governs `inline` functions and variables also governs the definitions of templates and their members.

N4606 §3.2 [basic.def.odr] /6

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. Here's a class template with a static data member:

```
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. Here's a variable 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

```
typedef std::vector<int> myvec_int;  // C++03 alias syntax

using myvec_double = std::vector<double>;  // C++11 syntax

template<typename T> using myvec = std::vector<T>;  // C++11 syntax

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

void f(const myint& mv);
void g(const myvec<int>& mv);

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

We'll come back to why this is important.

Now you know all the different kinds of templates in C++17!

Kind of template	Year introduced
Function	< 1998
Class	< 1998
Alias	2011
Variable	2014

Okay, they’re the same as they were in C++14.

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`; we'll talk about that deduction process ... now.

Rules of template type deduction

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

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 *]
}
```


N4606 §14.8.2.1 [temp.deduct.call] /2.1

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!

Rules of template type deduction

```
template<typename T, typename U>
void f(T x, U y);

template<typename T>
void g(T x, T y);

int main()
{
    f(1, 2);  // void f(T, U) [T = int, U = int]
    g(1, 2);  // void g(T, T) [T = int]
    g(1, 2u); // error: no matching function for call to g(int, unsigned int)
}
```


N4606 §14.8.2.1 [temp.deduct.call]

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,9>{}, std::array<double,4>{}, 0.0);
}
```

Would it matter if `std::array<int,9>` were implicitly convertible to `std::array<int,8>`?

No, it wouldn't; because parameter x does still contribute to the deduction of template type parameter `T`, so its argument type must match exactly — no implicit conversions allowed!

Puzzle #2

```
template<typename R, typename A>
void foo(R (*fptr)(A))
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    foo(  [](double x) { return int(x); } );  // error
    foo( +[](double x) { return int(x); } );  // compiles
}
```

Captureless lambda types are always implicitly convertible to function pointer type. But being *implicitly convertible to* a thing doesn't mean actually *being* that thing!

Protip: If you absolutely need the function-pointer conversion to happen, add a unary +.

Puzzle #2

```
template<typename R, typename A>
void foo(R (*fptr)(A))
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    foo(  [](double x)
    foo( +[](double x)
}
```

Captureless lambda types are always implicitly convertible to function pointer type. But being *implicitly convertible to* a thing doesn't mean actually *being* that thing!

Protip: If you absolutely need the function-pointer conversion to happen, add a unary +. ***Except on MSVC.***

```
error C2593: 'operator +' is ambiguous
note: could be 'built-in C++ operator+(int (__cdecl *)(double))'
note: or       'built-in C++ operator+(int (__stdcall *)(double))'
note: or       'built-in C++ operator+(int (__fastcall *)(double))'
note: or       'built-in C++ operator+(int (__vectorcall *)(double))'
```

How many people have seen this?

```
#include <algorithm>

short f();

int main()
{
    int x = 42;
    return std::max(f(), x);  // error: no matching function for...
}
```

Two ways to solve it

```
#include <algorithm>

short f();

int main()
{
    int x = 42;
    return std::max(f(), x);  // error: no matching function for...
}
```

Every parameter that contributes to deduction must match its argument type exactly. So, either make the arguments match exactly... or, make the parameter stop contributing to deduction.

Two ways to solve it

```
// Approach #1
return std::max(static_cast<int>(f()), x);

// Approach #2
return std::max<int>(f(), x);
```

Every parameter that contributes to deduction must match its argument type exactly. So, either make the arguments match exactly... or, make the parameter stop contributing to deduction.

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!

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

...as long as there's no ampersands involved.

Template type deduction: real deal

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

int main()
{
    int i;
    f(i);  // void f(T) [with T = int]
}
```


N4606 §14.8.2.1 [temp.deduct.call]

Template type deduction: real deal

```
template<typename T>
void f(T *t)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    int i;
    f(&i);  // void f(T*) [with T = int]
}
```


N4606 §14.8.2.1 [temp.deduct.call]

Template type deduction: real deal

```
template<typename T>
void f(T& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    int i;
    f(i);  // void f(T&) [with T = int]
}
```

I'm going to assume some knowledge of lvalues and rvalues. In brief, lvalues are objects that would be missed by someone if they were to get their innards stomped on, whereas rvalues are expendable objects.

N4606 §14.8.2.1 [temp.deduct.call]

Template type deduction: real deal

```
template<typename T>
void f(T&& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    int i;
    f(42);  // void f(T&&) [with T = int]
    f(std::move(i));  // ditto
}
```

I'm going to assume some knowledge of lvalues and rvalues. In brief, lvalues are objects that would be missed by someone if they were to get their innards stomped on, whereas rvalues are expendable objects.

N4606 §14.8.2.1 [temp.deduct.call]

Template type deduction: real deal

```
template<typename T>
void f(T&& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    int i;
    f(i);  // void f(T&&) [with T = int&]
}
```

Hmm.
Something interesting is going on here.

N4606 §14.8.2.1 [temp.deduct.call]

Reference collapsing

```
template<typename T>
void f(T&& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    int i;
    f(i);  // [with T=int&]
}
```

`T&&` here is a *forwarding reference*, or what Scott Meyers used to call a “universal reference.”
The rule to remember when dealing with references is that combining two reference types *mins* the number of ampersands between them.

& + & = & (of course)
& + && = &
&& + & = &
&& + && = &&

`i` is `int&`, so we should deduce `T` such that `T&&` is `int&`. Either `T=int` or `T=int&&` would result in `T&&=int&&`, but `T=int&` works!

N4606 §14.8.2.1 [temp.deduct.call] /3

Reference collapsing

```
template<typename T>
void f(T&& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    f(42);  // [with T=int]
}
```

`T&&` here is a *forwarding reference*, or what Scott Meyers used to call a “universal reference.”
The rule to remember when dealing with references is that combining two reference types *mins* the number of ampersands between them.

& + & = & (of course)
& + && = &
&& + & = &
&& + && = &&

42 is `int&&`, so we should deduce `T` such that `T&&` is `int&&`. `T=int` works! (`T=int&&` would also work, but we prefer to deduce fewer ampersands if possible.)

N4606 §14.8.2.1 [temp.deduct.call] /3

A case in which [T=int&&] is deduced

```
template<typename T>
void f(void (*t)(T))
{
    puts(__PRETTY_FUNCTION__);
}

void g(int&&) {}
int main()
{
    f(g);  // [with T=int&&]
}
```

Although we prefer to deduce `T=int` rather than `T=int&&` in the forwarding-reference case, there do exist other cases where `T=int&&` is the only possible deduction.

N4606 §14.8.2.1 [temp.deduct.call] /3

Reference collapsing in C++03

```
template<typename R>
void f(R r)
{
    R& x = ...;  // int& x
    R&& y = ...;  // int& y (!)
}

int main()
{
    int i;
    f<int&>(i);
}
```

The rule to remember when dealing with references is that combining two reference types *mins* the number of ampersands between them.

& + & = & (of course)
& + && = &
&& + & = &
&& + && = &&

This wasn't standardized until C++11, even though C++03 compilers did have to deal with the issue in practice (for lvalue references, anyway).

http://eli.thegreenplace.net/2014/perfect-forwarding-and-universal-references-in-c/

Reference (and *cv-*) collapsing

```
template<typename T>
void f(T&& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    const int i = 42;
    f(i);  // [with T=const int&]
}
```

`T&&` here is a *forwarding reference*, or what Scott Meyers used to call a "universal reference."
The rule to remember when dealing with references is that combining two reference types *mins* the number of ampersands between them.

& + & = & (of course)
& + && = &
&& + & = &
&& + && = &&

`i` is `const int&`, so we should deduce `T` such that `T&&` is `const int&`. `T=const int&` works!

N4606 §14.8.2.1 [temp.deduct.call] /3

Reference (and *cv-*) collapsing

```
template<typename T>
void f(T&& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main()
{
    const int i = 42;
    f(std::move(i));
}
```

`T&&` here is a *forwarding reference*, or what Scott Meyers used to call a "universal reference."
The rule to remember when dealing with references is that combining two reference types *mins* the number of ampersands between them.

& + & = & (of course)
& + && = &
&& + & = &
&& + && = &&

`i` is `const int&&`, so we should deduce `T` such that `T&&` is `const int&&`. `T=const int` works!

N4606 §14.8.2.1 [temp.deduct.call] /3

Deducing T& (not T&&)

```
template<typename T>
void f(T& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main() {
    int i = 42;
    f(static_cast<int&>(i));              // [with T=int]
    f(static_cast<int&&>(i));             // ERROR (T=int)
    f(static_cast<volatile int&>(i));     // [with T=volatile int]
    f(static_cast<volatile int&&>(i));    // ERROR (T=volatile int)
}
```


N4606 §14.8.2.1 [temp.deduct.call] /3

Deducing T& (not T&&)

```
template<typename T>
void f(T& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main() {
    int i = 42;
    f(static_cast<int&>(i));          // [with T=int]
    f(static_cast<int&&>(i));         // ERROR (T=int)
    f(static_cast<const int&>(i));    // [with T=const int]
    f(static_cast<const int&&>(i));   // [with T=const int] (!)
}
```


N4606 §14.8.2.1 [temp.deduct.call] /3

Deducing T& (not T&&)

```
template<typename T>
void f(T& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main() {
    int i = 42;
    f(static_cast<int&>(i));
    f(static_cast<int&&>(i));
    f(static_cast<const int&>(i));
    f(static_cast<const int&&>(i));   // OK (!)
}
```

What’s going on here? Well, the compiler basically tries to deduce T by stripping all the reference qualifiers (but *not* the *cv*-qualifiers) off of the argument type, and then reintroducing a single-& qualifier if it helps make a match via reference collapsing.

You can’t pass `int&&` to a function expecting `int&`; but you *can* pass `const int&&` to a function expecting `const int&`.

Okay, why is this??

N4606 §13.3.3.1.4 [over.ics.ref] /3

Rvalues are kinda like const lvalues

```
template<typename T>
void f(T& t)
{
    puts(__PRETTY_FUNCTION__);
}

int main() {
    int i = 42;
    f(static_cast<int&>(i));
    f(static_cast<int&&>(i));
    f(static_cast<const int&>(i));
    f(static_cast<const int&&>(i));
}
```

N4606: *"...a standard conversion sequence cannot be formed if it requires binding an lvalue reference other than a reference to a non-volatile const type to an rvalue or binding an rvalue reference to an lvalue..."*

In other words: You *can* bind a parameter of "non-volatile const lvalue reference" type to an argument of "rvalue" type.

However, this rule doesn't interact with the type deduction rules! The compiler won't insert `const` qualifiers into its deduced `T` just to make the binding come out right. That's why `f(static_cast<int&&>(i))` doesn't deduce `T=const int` on your behalf.

N4606 §13.3.3.1.4 [over.ics.ref] /3

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

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

...as long as there's no variadic templates involved.

We'll worry about those in Part 2.

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

No* — Well, C++17 introduces deduction guides for class constructors. Nobody supports those yet. Come back for Part 2.

```
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
}
```


N4606 §14.7.3 [temp.expl.spec] /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 you don't need 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; }
```


N4606 §14.7.3 [temp.expl.spec] /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 you don't need 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; }
```


N4606 §14.7.3 [temp.expl.spec] /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
```


The Standard calls full specialization "explicit specialization."

Alias templates can’t be specialized

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

template<typename T>
void foo(myvec<T>& mv) {  // void foo(std::vector<T>&)
    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.

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`.

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.

N4606 §14.5.5 [temp.class.spec]

Partial specialization

```
// this is the primary template
template<typename T>
constexpr bool is_array = false;

// these are partial specializations
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.

N4606 §14.5.5 [temp.class.spec]

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 may *seem* 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 (`is_pointer` in this example).

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
- reference collapsing
- full specialization
- 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

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".

Templatize all the things!

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

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

int main()
{
    char ascii_buffer[] = "Hello world";
    reverse<ascii>(ascii_buffer, 11);

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

Templatize all the things!

```
template<typename Charset>
void reverse(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 ascii; struct utf8;

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

template<> int mblen<ascii>(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 ascii; struct utf8;

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

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

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

```
template<> int mblen<ascii>(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<ascii>(const char *, int);
template<> int mblen<utf8>(const char *, int);
template<typename Cs> int reverse(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<ascii>(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<ascii>(const char *, int);
template<> int mblen<utf8>(const char *, int);
template<typename Cs> int reverse(const char *, int);
template<> int reverse<ascii>(const char *, int);
template<> int reverse<utf8>(const char *, int);

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

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

Explicit instantiation definition

This special syntax means "Please instantiate this template, with the given template parameters, as if it were being used right here." Semantically, it’s not giving the compiler any new information. It’s just asking the compiler to instantiate a definition of the template entity *right here*.

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

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

template class vector<int>;

template bool is_void_v<void>;
```


N4606 §14.7.2 [temp.explicit] /2

Explicit instantiation declaration

This special syntax means "Please instantiate this template, with the given template parameters, as if it were being used right here." Semantically, it's not giving the compiler any new information. It's just asking the compiler to instantiate a definition of the template entity *right here*.

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

extern template class vector<int>;

extern template bool is_void_v<void>;
```


N4606 §14.7.2 [temp.explicit] /2

Explicit instantiation of a class template

```
template<class T>
class Foo {
    static int f(T);
    static int g();
};

template class Foo<int>;
// this translation unit will contain definitions
// for both Foo<int>::f(int) and Foo<int>::g()
```

When you explicitly instantiate a template class, that single definition actually counts as an explicit instantiation of *all the members of that template class*, too — both the static members and the non-static members.

N4606 §14.7.2 [temp.explicit] /8

Explicitly instantiate without specializing

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

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

Don’t mix and match, though

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

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

This is NOT guaranteed to work (although it does in practice)

N4606 §14.7.3 [temp.expl.spec] /6; §14.7.2 [temp.explicit] /11

Don’t mix and match, though

Clang/GCC: `__Z1aIiEiT_`

MSVC: `??$a@H@@YAHH@Z`

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

template<> int a(int);


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

OK in practice;
officially
ill-formed

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

template int a(int);
```

because on the left we promised to ***specialize*** `a<int>`, and on the right we have merely ***instantiated*** the ***primary*** template for `a<int>`

N4606 §14.7.3 [temp.expl.spec] /6; §14.7.2 [temp.explicit] /11

Don’t mix and match, though

Clang/GCC: `__Z1aIiEiT_`

MSVC: `??$a@H@@YAHH@Z`

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

template int a(int);


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

Fragile even in practice

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

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

because on the left we instantiate `a<int>`, and on the right we explicitly specialize `a<int>`, so we’ve got two competing definitions here. Which one wins depends on your optimization level.

N4606 §14.7.3 [temp.expl.spec] /6; §14.7.2 [temp.explicit] /11

Now you know everything there is to know about *explicit instantiation*!

This concludes Part I — Questions?

- 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