

# Introduction to C++ templates

**Jonas Rylund Glesaaen**  
jonas@glesaaen.com

**SA<sup>2</sup>C Seminar Series**  
**February 9th 2018**

**Introduction introduction**

# What is a template

**Exactly what it says on the box.**

**A template is a template for constructing the thing it templates**

# What is a template

**Say you asked a colleague to implement a function that checks whether an integer is even or not**

**and then you get this...**

# What is a template

```
bool is_even(unsigned i)
{
    if (i == 0)
        return true;
    else if (i == 1)
        return false;
    else if (i == 2)
        return true;
    else if (i == 3)
        return false;
    /* ... */
    else if (i == 18446744073709551614)
        return true;
    else
        return false;
}
```

# What is a template

**Similar to how control structures and functions help us not repeat ourselves in code, templates does this on the type level**

**You can apply all the same techniques**  
(more on this in the next talk)

# What is a template

```
double add(double a, double b)
{
    return a + b;
}
```

```
double add(double a, int b)
{
    return a + b;
}
```

```
double add(int a, double b)
{
    return a + b;
}
```

```
int add(int a, int b)
{
    return a + b;
}
```

# What is a template

```
template <typename T, typename U>  
auto add(T a, U b)  
{  
    return a + b;  
}
```

Usable by any type **T** and **U** pair that has a valid  
**operator+**



# Types of templates

# Types of templates : functions

## Function templates are not functions

```
template <typename Type>
bool not_empty(Type const& x)
{
    return ! x.empty();
}
```

## If a specific instance isn't used it is not created

(can you identify all the requirements on `Type`?)

# Types of templates : classes

## Class templates are not classes

```
template <typename Type>
class Linked_List
{
public:
    class Node
    {
    private:
        Type val;
        std::unique_ptr<Node> next;
    };

private:
    std::unique_ptr<Node> first_node;
};
```

# Types of templates : aliases

First thought when you see the **using** keyword

```
typedef std::vector<double> Vector;
```

↖ C++03 Syntax

```
using Vector = std::vector<double>;
```

↖ C++11 Syntax

# Types of templates : aliases

## Also introduces alias templates

...they are actually aliases

```
template <Typename T>  
using String_Map = std::map<std::string, T>;
```

## These are the exact same types

```
std::is_same<  
    String_Map<double>,  
    std::map<std::string, double>  
>::value <----- true
```

# Types of templates : variables

## Template variables are not variables

```
template <typename T>  
constexpr bool is_double =  
    std::is_same<T, double>::value;
```

```
is_double<int>; ← false  
is_double<double>; ← true
```

# Types of templates : variables

We used to do this

```
template <typename T>
struct is_double
{
    static constexpr bool value =
        std::is_same<T, double>::value
};
```

```
is_double<int>::value; ← false
is_double<double>::value; ← true
```

# Fundamental theorem of software engineering

**We can solve any problem by introducing an extra level of indirection.**

David J. Wheeler



# Fundamental theorem of templates

**Any finite problem can be solved with a finite number of structs.**

Jonas R. Glesaaen

# Template instantiation

# Template instantiation

**Templates are instantiated when they are used**

(which is why linking can be counterintuitive)

**Can also instantiate them explicitly**

```
template bool is_empty(std::vector<double>);  
template class Linked_List<int>;  
template bool is_double<char>;
```

# Template instantiation

**C++ will never instantiate anything it doesn't 100% need** (more or less true)

```
template <typename T>
struct invalid
{
    static_assert(sizeof(T) == 0);
};
```

```
invalid<int> *inv_ptr; ← Compiles
```

# Template instantiation

**C++ will never instansiate anything it doesn't 100% need** (more or less true)

```
template <typename T>
struct invalid
{
    static_assert(sizeof(T) == 0);
};
```

```
invalid<int> inv_val; ← Compile error
```

# Template instantiation

## Does it compile?

```
template <typename T>
struct invalid
{
    void foo()
    {
        static_assert(sizeof(T) == 0);
    }
};

invalid<int> inv_var;
```

# Template instantiation

## Does it compile?

```
template <typename T>
struct invalid
{
    void foo()
    {
        static_assert(sizeof(T) == 0);
    }
};
```

```
invalid<int> inv_var;
```

**Yes, it does**

# Template instantiation

## Does it compile?

```
template <typename T>
struct invalid
{
    void foo()
    {
        static_assert(sizeof(char) == 0);
    }
};

invalid<int> inv_var;
```



# Template instantiation

## Does it compile?

```
template <typename T>
struct invalid
{
    void foo()
    {
        static_assert(sizeof(char) == 0);
    }
};
```

```
invalid<int> inv_var;
```

**No, it does not**

# Dependent types

# Dependent types : typename

**A dependent name is a name that depends on an unknown template type**

```
template <typename T>
void foo(int x)
{
    auto v = T::U(x);
}
```

Is it a function call?

Is it a variable declaration?

# Dependent types : typename

**A dependent name is a name that depends on an unknown template type**

```
template <typename T>
void foo(int x)
{
    auto v = typename T::U(x);
}
```

**Ok, it is a declaration**



# Dependent types : template

Can also have dependent names that are templates

```
template <typename T, int N>
void foo(int x)
{
    auto v = T::U<N>(x);
}
```

Value comparison?

Function call?

Constructor call?

# Dependent types : template

Can also have dependent names that are templates

```
template <typename T, int N>
void foo(int x)
{
    auto v = T::template U<N>(x);
}
```

↑  
Ok, function call

# Dependent types : template

Can also have dependent names that are templates

```
template <typename T, int N>
void foo(int x)
{
    auto v = typename T::template U<N>(x);
}
```

Ok, type constructor



# Template deduction



# Template deduction : by value

Most straight forward is to explicitly specify the template types

```
template <typename T, typename U>  
auto max(T a, U b);
```

```
max<int,int>(1, 5.);
```



The diagram illustrates the deduction of template parameters T and U from the function call max<int,int>(1, 5.). Two arrows point from the explicit type 'int' in the angle brackets to the deduced types 'T' and 'U' in the text below. The first arrow points from the first 'int' to 'T', and the second arrow points from the second 'int' to 'U'. The text 'T = int' and 'U = int' is displayed below the arrows.

T = int

U = int

# Template deduction : by value

Most straight forward is to explicitly specify the template types

```
template <typename T, typename U>  
auto max(T a, U b);
```

```
max<int,int>(1, 5.);
```

**Return type  
doesn't contribute  
to type deduction**

**Irrelevant for type deduction**

# Template deduction : by value

**Most straight forward is to explicitly specify the template types**

**If a template is explicitly declared it does not enter into argument type deduction**

# Template deduction : by value

**Every type that is not explicitly defined must be deduced from the function arguments**

## **Deduction flow**

- 1** Explicit types are set
- 2** Each argument factors into type deduction separately
  - This happens "in parallel"
  - Resulting deduction is either full or partial
- 3** The deduced types are checked for consistency
- 4** Template checks that all types are fully deduced

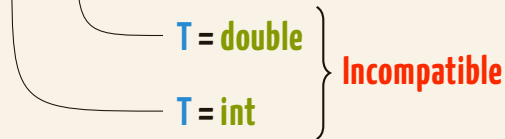
# Template deduction : by value

```
template <typename T>  
T max(T a, T b);  
  
max(4, 5.);
```

# Template deduction : by value

```
template <typename T>  
T max(T a, T b);
```

```
max(4, 5.);
```



# Template deduction : by value

```
template <typename T>  
T max(T a, T b);
```

```
max<double>(4, 5.);
```

**T = double**

Irrelevant



# Template deduction : by value

## Puzzle #1

```
template <typename T, bool B>
struct My_Type{};

template <typename T, typename U>
void foo(
    My_Type<T, !std::is_same_v<T, U>>,
    My_Type<U, sizeof(U) == sizeof(T)>);

int main()
{
    foo(My_Type<int,true>{},
        My_Type<double,true>{});
}
```



# Template deduction : by value

## Puzzle #1

```
template <typename T, bool B>
struct My_Type{};

template <typename T, typename U>
void foo(
    My_Type<T, !std::is_same_v<T, U>>,
    My_Type<U, sizeof(U) == sizeof(T)>);

int main()
{
    foo(My_Type<int,true>{},
        My_Type<unsigned,true>{});
}
```

# Template deduction : by value

## Puzzle #2

```
template <typename T, typename U>
void foo(
    std::array<T, sizeof(U)>,
    std::array<U, sizeof(T)>);

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

# Template deduction : by value

## Puzzle #2

```
template <typename T, typename U>
void foo(
    std::array<T, sizeof(U)>,
    std::array<U, sizeof(T)>);

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

# Template deduction : references

## lvalues (&)

anything with a tag  
anything that will be missed

## rvalues (&&)

intermediate values  
things no one will miss

Also called  
universal references

### Reference collapse

$$\& + \& = \&$$

$$\&\& + \& = \&$$

$$\& + \&\& = \&$$

$$\&\& + \&\& = \&\&$$

# Template deduction : references

Template deduction uses reference collapse so that the function signature matches the call

```
template <typename T>  
auto foo(T &);
```

```
int i;
```

foo(i);	←	Ok! T = int
foo(std::move(i));	←	Compile error
foo(5);	←	Compile error

# Template deduction : references

Template deduction uses reference collapse so that the function signature matches the call

```
template <typename T>  
auto foo(T const &);
```

```
int i;
```

foo(i);	←	Ok! T = int
foo(std::move(i));	←	Ok! T = int
foo(5);	←	Ok! T = int

# Template deduction : references

Template deduction uses reference collapse so that the function signature matches the call

```
template <typename T>  
auto foo(T &&);
```

```
int i;
```

foo(i);	←	Ok! T = int&
foo(std::move(i));	←	Ok! T = int
foo(5);	←	Ok! T = int

# Template deduction : variadic

Variadic template packs can hold any number of types

```
template <typename... Tp>  
auto foo(Tp...);
```

<code>foo();</code>	←	<code>Tp = &lt;&gt;</code>
<code>foo(3, 1l, 'a');</code>	←	<code>Tp = &lt;int, long, char&gt;</code>
<code>foo&lt;double&gt;(3, 1l, 'a');</code>	←	<code>Tp = &lt;double, long, char&gt;</code>
<code>foo&lt;int,int,int&gt;(2,4);</code>	←	<b>Compile error</b> Expected 3 arguments...



# Template deduction : variadic

Template packs gobble up everything to the right

```
template <typename T, typename... Tp>  
auto foo(Tp..., T);
```

```
foo(3, 1l, 'a'); ←———— Compile error  
                        Cannot determine T  
                        Tp = <int, long, char>
```

```
foo<char, int, long>(  
    3, 1l, 'a'); ←———— T = double  
                        Tp = <int, long>
```

# Template deduction : classes

Introduced in C++17, can now write

```
template <typename T>
class My_Class
{
public:
    My_Class(T val);
};
```

```
My_Class inst {5.0}; ← My_Class<double>
```

# Template deduction : classes

Introduced in C++17, can now write

```
template <typename T>  
class My_Class  
{  
public:  
    My_Class(T val);  
};
```

**Deduction guide**



```
My_Class(double) -> My_Class<int>;
```

```
My_Class inst {5.0}; ← My_Class<int>
```

# Template deduction : classes

**More or less syntactic sugar for this**

```
template <typename T>  
class My_Class  
{ ... };
```

```
template <typename T>  
auto make_cls(T t)  
{  
    return My_Class<T>{t};  
}
```

```
auto inst = make_cls(5.0);
```

# Template specialisation

# Template specialisation : full

## Can specialise templates

```
template <typename T>
T null()
{
    return T{0};
}
```

```
template <>
std::string null<std::string>()
{
    return "";
}
```

# Template specialisation : full

## Can specialise templates

```
template <typename T>
T null()
{
    return T{0};
}
```

```
template <>
std::string null<>()
{
    return "";
}
```

# Template specialisation : full

## Can specialise templates

```
template <typename T>
T null()
{
    return T{0};
}
```

```
template <>
std::string null()
{
    return "";
}
```



# Template specialisation : full

## Can specialise templates

```
template <typename T>  
T null()  
{  
    return T{0};  
}
```

```
template <>  
std::string null() = delete;
```

# Template specialisation : full

## Can specialise templates

```
template <typename T>  
T null()  
{  
    return T{0};  
}
```

```
template <>  
std::string null() = delete;
```

**These are all examples of full specialisations**

(not all that interesting)

# Template specialisation : partial

## Partial template specialisations: pattern matching \o/

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

```
template <typename T>
struct is_pointer_type<T*>
{
    static constexpr bool value = true;
};
```

```
is_pointer_type<int>::value;
```

# Template specialisation : partial

## Partial template specialisations: pattern matching \o/

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

```
template <typename T>  
struct is_pointer_type<T*>  
{  
    static constexpr bool value = true;  
};
```

```
is_pointer_type<int>::value; ← false
```

# Template specialisation : partial

## Partial template specialisations: pattern matching \o/

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

```
template <typename T>
struct is_pointer_type<T*> ← T = int
{
    static constexpr bool value = true;
};
```

```
is_pointer_type<int*>::value; ← true
```

# Template specialisation : partial

## Partial template specialisations: pattern matching \o/

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

```
template <typename T>
struct is_pointer_type<T*> ← T = int*
{
    static constexpr bool value = true;
};
```

```
is_pointer_type<int**>::value; ← true
```

# Template specialisation : partial

## Partial template specialisations: pattern matching \o/

```
template <typename T> T = std::shared_ptr<int>  
struct is_pointer_type  
{  
    static constexpr bool value = false;  
};
```

```
template <typename T>  
struct is_pointer_type<T*>  
{  
    static constexpr bool value = true;  
};
```

```
is_pointer_type<  
    std::shared_ptr<int>>::value; false
```

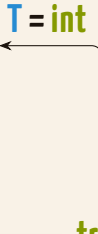
# Template specialisation : partial

## Partial template specialisations: pattern matching \o/

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

template <typename T>
struct is_pointer_type<std::shared_ptr<T>> < T = int
{
    static constexpr bool value = true;
};

is_pointer_type<
    std::shared_ptr<int>>::value; < true
```





# Template specialisation : partial

**Template deduction picks out the most specialised template**

```
template <typename T>
struct num_ptr
{ static constexpr std::size_t value = 0; };
```

```
template <typename T>
struct num_ptr<T*>
{ ... value = 1; };
```

```
template <typename T>
struct num_ptr<T**>
{ ... value = 2; };
```

```
num_ptr<int*>::value;
num_ptr<int**>::value;
```

# Template specialisation : partial

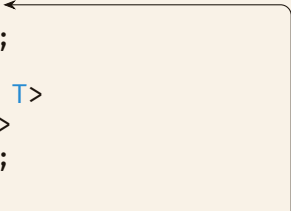
Template deduction picks out the most specialised template

```
template <typename T>
struct num_ptr
{ static constexpr std::size_t value = 0; };
```

```
template <typename T>      T = int
struct num_ptr<T*> ←
{ ... value = 1; };
```

```
template <typename T>
struct num_ptr<T**>
{ ... value = 2; };
```

```
num_ptr<int*>::value; ← value = 1
num_ptr<int**>::value;
```



# Template specialisation : partial

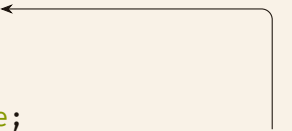
Template deduction picks out the most specialised template

```
template <typename T>
struct num_ptr
{ static constexpr std::size_t value = 0; };
```

```
template <typename T>
struct num_ptr<T*>
{ ... value = 1; };
```

```
template <typename T>
struct num_ptr<T**> ← T = int
{ ... value = 2; };
```

```
num_ptr<int*>::value;
num_ptr<int**>::value; ← value = 2
```



# Template specialisation : partial

**Template deduction picks out the most specialised template**

```
template <typename T>  
struct num_ptr  
{ static constexpr std::size_t value = 0; };
```

```
template <typename T>  
struct num_ptr<T*>  
{ ... value = 1 + num_ptr<T>::value; };
```

# Template specialisation : partial

**Template deduction picks out the most specialised template**

```
template <typename T>
struct num_ptr
{ static constexpr std::size_t value = 0; };
```

```
template <typename T>
struct num_ptr<T*>
{ ... value = 1 + num_ptr<T>::value; };
```

```
template <typename T>
struct num_ptr<std::shared_ptr<T>>
{ ... value = 1 + num_ptr<T>::value; };
```

# Template specialisation : partial

**Template deduction picks out the most specialised template**

```
template <typename T>
struct num_ptr
{ static constexpr std::size_t value = 0; };
```

```
template <typename T>
struct num_ptr<T*>
{ ... value = 1 + num_ptr<T>::value; };
```

```
template <typename T>
struct num_ptr<std::shared_ptr<T>>
{ ... value = 1 + num_ptr<T>::value; };
```

```
num_ptr<std::shared_ptr<int**>*>::value;
```

# Template specialisation : partial

Template deduction picks out the most specialised template

```
template <typename T>
struct num_ptr
{ static constexpr std::size_t value = 0; };
```

```
template <typename T>
struct num_ptr<T*>
{ ... value = 1 + num_ptr<T>::value; };
```

```
template <typename T>
struct num_ptr<std::shared_ptr<T>>
{ ... value = 1 + num_ptr<T>::value; };
```

```
num_ptr<std::shared_ptr<int**>*>::value; ← value = 6
```

I am getting ahead of myself

# Template specialisation : partial

**Only class templates have partial specialisation**

**Cannot have it for function templates due to overloading**



# Fundamental theorem of templates

**Any finite problem can be solved with a finite number of structs.**

Jonas R. Glesaaen

# Template specialisation : partial

```
template <typename T>
struct function_impl
{
    static auto _(T t) { ... };
};
```

```
template <typename T>
struct function_impl<T[]>
{
    static auto _(T[] t) { ... };
};
```

```
template <typename T>
auto foo(T t)
{
    return function_impl<T>::_(t);
}
```

**Summary and wrap up**

# Template feature table

	Type deduction	Full spec.	Partial spec.
Function	Yes	Yes	No
Class	C++17	Yes	Yes
Alias	No	No	No
Variable	No	Yes	Yes

# Resources

[1] **C++ reference.**

<http://cppreference.com>.

[2] **The c++ standard draft.**

<https://github.com/cplusplus/draft/tree/f68ca58>.

[3] **S. Meyers.**

Effective Modern C++.  
O'Reilly Media, **2014**.

[4] **Arthur O'Dwyer.**

Template normal programming.  
CppCon 2016.

# Thanks!



**lrubataru/talks**