C++ Essentials: Templates

2. C++ Templates

Klaus Iglberger
June 20th, 2025

Content

2.1. Motivation

What Is Generic Programming?

- Generic programming is the approach to implement algorithms and data structures in the most general sensible way.
- Algorithms are written in terms of types to-be-specified later.
- The term **generic programming** was originally coined by David Musser and Alexander Stepanov.
- Generic programming helps us to reduce redundancy and programming effort, while it increases reusability and flexibility.

Templates

- Templates are a kind of pattern for the compiler.
- We can **instantiate** templates with different types or values
    - Each instantiation for a new type or value results in additional code, the fill-in template is generated with the given template argument.
- Templates reduce a lot of writers' work. We do not have to implement functions multiple times just because it's a slightly different type.
- There are different types of templates
    - Function templates
    - Class templates
    - Variable templates (since C++14).
- Templates are always initiated by the keyword `template`.

Guidelines

Core Guideline T.1: Use templates to raise the level of abstraction of code

Core Guideline T.2: Use templates to express algorithms that apply to many argument types

Core Guideline T.3: Use templates to express containers and ranges

Core Guideline T.5: Combine generic and OO techniques to amplify their strengths, not their costs

2.2. Function Templates

Function Templates

Task (2_Templates/Max)

Step 1: Write a `max()` function template that takes a single template parameter for its two function parameters.

Step 2: Write a `max()` function template that takes two template parameters instead of one (one for each function parameter).

Step 3: Overload the `max()` function template for integers.

Step 4: Specialize the `max()` function template(s) for integers.

Max with One Template Parameter

Template parameter list (can be type parameters or non-type parameters)

```
template< typename T >
inline T max( T const& a, T const& b )
{
    return ( a < b ) ? b : a;
}
```

The two template parameters of type T are deduced based on the given arguments. Both arguments are deduced independently and only if the type of both arguments match, the template can be instantiated for that type (i.e. T can be replaced by that type).

```
max( 1, 5 );
```

Uses of the `max()` algorithm. Both template arguments match, which instantiates `max<int>()`

```
template< >
inline int max<int>( int const& a, int const& b )
{
    return ( a < b ) ? b : a;
}
```

Max with One Template Parameter

Quick Task: Which function is called?

```
max( 7.0, 42.0 );          // Calls max<double> (by argument deduction)
max( 'a', 'b' );           // Calls max<char>   (by argument deduction)
max( 7, 42 );              // Calls max<int>    (by argument deduction)
max<>( 7, 42 );            // Calls max<int>    (by argument deduction)
max<double>( 7, 42 );// Calls max<double> (no argument deduction)
```

Max with Two Template Parameter

It is possible to define `max()` with two template parameters:

```
template< typename T1, typename T2 >
inline ??? max( T1 const& a, T2 const& b )
{
    return ( a < b ) ? b : a;
}
```

What should the return type be?

The return type can be specified in five different ways:

- A third template parameter
- A return type deduction based on the arguments (C++11)
- The `common_type` type trait (C++11)
- The `auto` keyword (C++14)
- `decltype(auto)` (C++14)

Max with Three Template Parameters

Isn’t it possible to define `max()` with three template parameters?

```
template< typename T1, typename T2, typename RT >
inline RT max( T1 const& a, T2 const& b )
{
    return ( a < b ) ? b : a;
}
```

Unfortunately, the compiler cannot deduce the type of RT with the given template arguments. And since RT is the last parameter, usage is cumbersome:

```
max<double,int,double>( 1.2, -4 );  // Returns 1.2
```

Max with Three Template Parameters

Let's move RT to the front…

```
template< typename RT, typename T1, typename T2 >
inline RT max( T1 const& a, T2 const& b )
{
    return ( a < b ) ? b : a;
}
```

The compiler still cannot deduce RT, but now only RT has to be specified:

```
max<double>( 1.2, -4 );  // Returns 1.2
```

Return Type Deduction Based on Arguments

C++11

template parameter list

```
template< typename T1, typename T2 >
inline auto max( T1 const& a, T2 const& b ) -> decltype(a+b)
{
    return ( a < b ) ? b : a;
}
```

function/call parameter list

trailing return type

The std::common_type trait

template parameter list

function/call parameter list

```
template< typename T1, typename T2 >
inline auto max( T1 const& a, T2 const& b )
    -> std::common_type_t<T1,T2>
{
    return ( a < b ) ? b : a;
}
```

trailing return type

Return Type Deduction with auto

C++14

template parameter list

function/call parameter list

```
template< typename T1, typename T2 >
inline auto max( T1 const& a, T2 const& b )
{
    return ( a < b ) ? b : a;
}
```

return type deduction

decltype(auto)

14

template parameter list

```
template< typename T1, typename T2 >
inline decltype(auto) max( T1 const& a, T2 const& b )
{
    return ( a < b ) ? b : a;
}
```

function/call parameter list

return type deduction

Overloading Function Templates

It is possible to overload function templates

```
// Template version of max
template< typename T1, typename T2 >
inline auto max( T1 const& a, T2 const& b )
{
    return ( a < b ) ? b : a;
}


// Overload for int
inline int max( int a, int b )
{
   return ( a < b ) ? b : a;
}
```

Overloading Function Templates

Quick Task: Which function is called?

```
max( 7.0, 42.0 );       // Calls max<double> (by argument deduction)
max( 'a', 'b' );        // Calls max<char>   (by argument deduction)
max( 7, 42 );           // Calls the nontemplate for two ints
max<>( 7, 42 );         // Calls max<int>    (by argument deduction)
max<double>( 7, 42 );// Calls max<double> (no argument deduction)
```

Function Call Resolution

Remember the first two steps of the compiler to resolve a function call:

1. **Name resolution**: Select all (visible) candidate functions with a certain name within the current scope. If none is found, proceed into the next surrounding scope.

2. **Overload resolution**: Find the best match among the selected candidate functions:
    1. If a non-template is a perfect match, select it.
    2. If a template is available, try to make it a perfect match
        1. If this is possible, select it.
        2. If this is not possible, try to find the best match among the non-template functions; use argument conversions as allowed and necessary.

Specializing Function Templates

Function templates can also be specialized. Given the following base template ...

```
template< typename T1, typename T2 >
void f( T1, T2* );
```

... a specialization for the built-in type `int` can be created like this:

```
template<>
void f( int, double* );
```

`T1` is specialized as `int`, `T2` as `double`.

Specializing Function Templates

The specialization for the `max()` function looks like this:

```cpp
// Template version of max
template< typename T1, typename T2 >
inline auto max( T1 const& a, T2 const& b )
{
    return ( a < b ) ? b : a;
}

// Specialization of the max function template
template<>
inline int max<int,int>( int const& a, int const& b )
{
    return ( a < b ) ? b : a;
}
```

Overloading vs. Specialization

Mind the difference between overloading and specialization:

```
template< typename T >
int f( T );              // (1) Base template

template< typename T >
int f( T* );             // (2) Overloading of (1)

template<>
int f( int );            // Specialization of (1)

template<>
int f( int* );           // Specialization of (2)
```

Overloading vs. Specialization

Quick Task: Which version of `f()` will be invoked by the last line?

```
template< typename T >
void f( T );                    // (1)

template<>
void f<int*>( int* );           // (2)

template< typename T >
void f( T* );                   // (3)

// ...

int* p;
f( p );      // Calls (3)
```

Function Call Resolution

Remember the first two steps of the compiler to resolve a function call:

1. **Name resolution:** Select all (visible) candidate functions with a certain name within the current scope. If none is found, proceed into the next surrounding scope.

2. **Overload resolution:** Find the best match among the selected candidate functions:
    1. If a non-template is a perfect match, select it.
    2. If a template is available, try to make it a perfect match
        1. If this is possible, select it.
        2. If this is not possible, try to find the best match among the non-template functions; use argument conversions as allowed and necessary.
    3. If a template is selected, take all its specialization into consideration

Function Templates

Task (2_Templates/MinMax)

Step 1: Write a `minmax()` function template that takes a single template parameter for its two function parameters.

Step 2: Write a `minmax()` function template that takes two template parameters instead of one (one for each function parameter).

Step 3: Overload the `minmax()` function template for integers.

Step 4: Specialize the `minmax()` function template(s) for integers.

Function Templates

Task (2_Templates/StringViewAdd): The following code example contains a serious bug. Explain the bug and discuss a solution.

Function Templates

Task (2_Templates/Compare)

Step 1: Write a generic `compare()` function that returns a negative number if the left-hand side argument is smaller, 0 if both arguments are equal, and a positive number if the left-hand side argument is larger.

Step 2: Specialize the `compare()` function for pointers to char.

Function Templates

Task (2_Templates/ArraySize): Write a `constexpr` function template `size()` that returns the size of a given array.

Function Templates

Task (2_Templates/Accumulate_1)

Step 1: Implement the `accumulate()` algorithm. The algorithm should take a pair of iterators, an initial value for the reduction operation, and a binary operation that performs the elementwise reduction.

Step 2: Implement an overload of the `accumulate()` algorithm that uses 'std::plus' as the default binary operation.

Step 3: Implement an overload of the `accumulate()` algorithm that uses the default of the underlying data type as initial value and `std::plus` as the default binary operation.

Step 4: Test your implementation with a custom binary operation (e.g. `Times`).

Function Templates

Task (2_Templates/Find): Write a function template that acts like the standard library `find()` algorithm. The function needs two template type parameters: One represents the function's iterator parameters, one represents the type of the value to find. Use your function to find a given value in a `std::vector<int>` and a `std::list<std::string>`.

Function Template Guidelines

Core Guideline T.2: Use templates to express algorithms that apply to many argument types

Guideline: In case a non-template function and a template function are equally well matched, the non-template function is preferred.

Guideline: When accessing a nested type within a template, don't forget to disambiguate with `typename`.

Core Guideline T.144: Don't specialize function templates

2.3. Template Argument Deduction

Template Argument Deduction

```
template< typename T >
void f( ParamType param );

f( expr );
```

Case 1: ParamType is Neither Pointer Nor Reference

```
template< typename T >
void f( T param );          // ParamType is 'T'

f( expr );
```

1. If `expr`'s type is a reference, ignore the reference part.
2. Ignore `const` and `volatile` qualifiers on `expr`.
3. Then pattern-match `expr`'s type against `ParamType` to determine `T`.

```
int        x  = 42;
int const  cx = x;
int const& rx = x;

f( x );    // T is 'int', ParamType is 'int'

f( cx );   // T is 'int', ParamType is 'int'

f( rx );   // T is 'int', ParamType is 'int'
```

Case 2: ParamType is Pointer

```
template< typename T >
void f( T* param );        // ParamType is 'T*'

f( expr );
```

1. If `expr`'s type is a reference, ignore the reference part.
2. Then pattern-match `expr`'s type against `ParamType` to determine `T`.

```
int         x  = 42;
int const* px = x;

f( &x );   // T is 'int', ParamType is 'int*'

f( cx );   // T is 'int const', ParamType is 'int const*'
```

Case 3: ParamType is (Non-Forwarding) Reference

```
template< typename T >
void f( T& param );          // ParamType is 'T&'

f( expr );
```

1. If expr's type is a reference, ignore the reference part.
2. Then pattern-match expr's type against ParamType to determine T.

```
int         x  = 42;
int const   cx = x;
int const&  rx = x;

f( x );    // T is 'int', ParamType is 'int&'

f( cx );   // T is 'int const', ParamType is 'int const&'

f( rx );   // T is 'int const', ParamType is 'int const&'
```

Case 3: ParamType is (Non-Forwarding) Reference

```
template< typename T >
void f( T const& param ); // ParamType is 'T const&'

f( expr );
```

1. If `expr`'s type is a reference, ignore the reference part.
2. Then pattern-match `expr`'s type against `ParamType` to determine `T`.

```
int          x  = 42;
int const    cx = x;
int const&   rx = x;

f( x );     // T is 'int', ParamType is 'int const&'

f( cx );    // T is 'int', ParamType is 'int const&'

f( rx );    // T is 'int', ParamType is 'int const&'
```

Case 4: ParamType is Forwarding Reference

```
template< typename T >
void f( T&& param );       // ParamType is 'T&&'

f( expr );
```

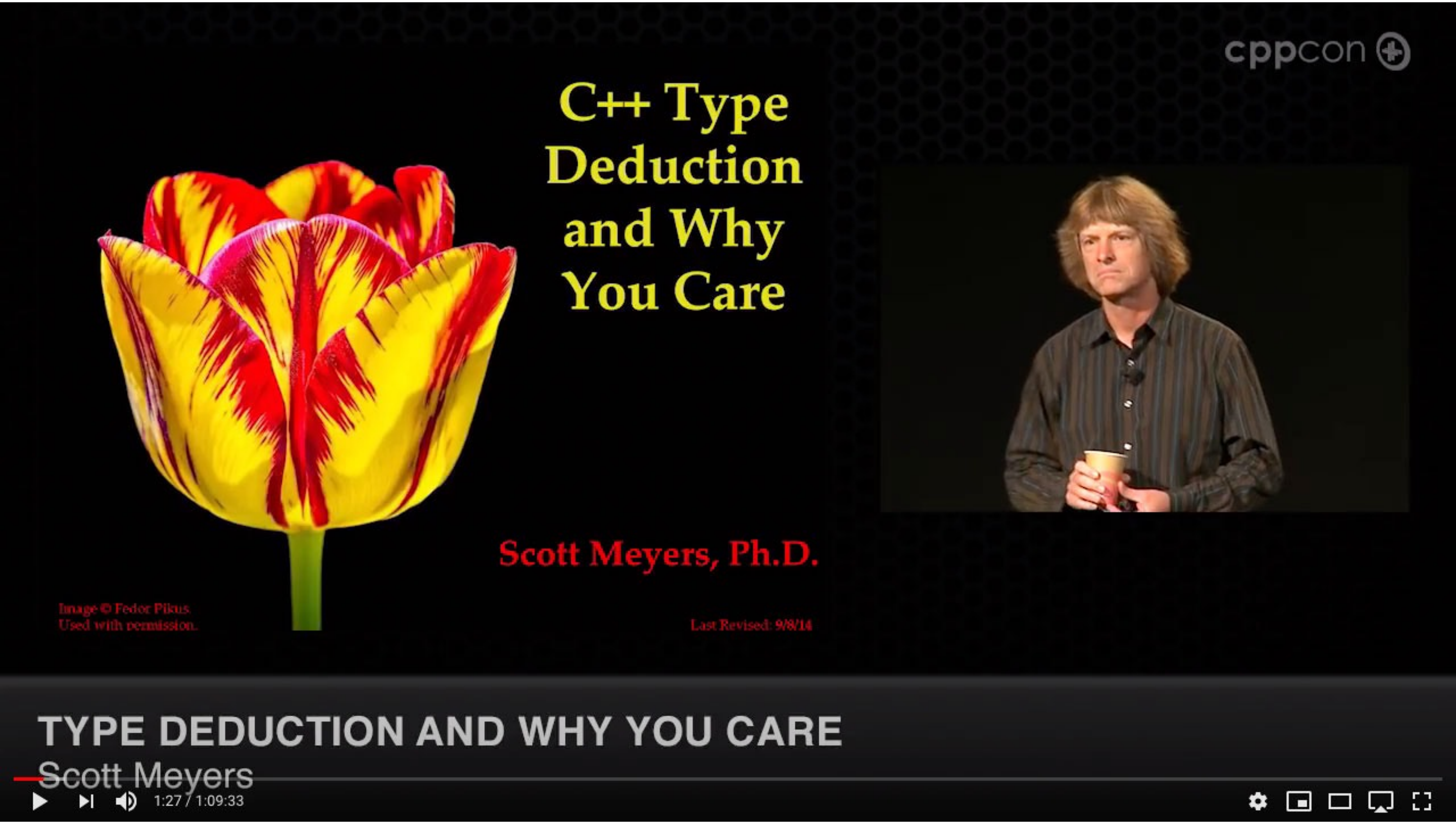
1. If `expr` is an lvalue, deduce both `T` and `ParamType` as lvalue reference.
2. If `expr` is an rvalue, deduce `T` and `ParamType` as “normal” reference.

```
int        x  = 42;
int const  cx = x;
int const& rx = x;

f( x );    // T is 'int&', ParamType is 'int&'

f( cx );   // T is 'int const&', ParamType is 'int const&'

f( rx );   // T is 'int const&', ParamType is 'int const&'

f( 42 );   // T is 'int', ParamType is 'int&&'
```

Guidelines

```
template< typename T >
void f( ParamType param );

f( expr );
```

Guideline: For non-forwarding references, `T` is never deduced to be a reference.


cppcon
C++ Type Deduction and Why You Care
Scott Meyers, Ph.D.
Image © Fedor Pikus. Used with permission.
Last Revised: 9/8/14
TYPE DEDUCTION AND WHY YOU CARE
Scott Meyers
1:27 / 1:09:33

2.4. Class Templates

Programming Task

Task (2_Templates/UniquePtr1): `Reimplement a (simplified)` std::unique_ptr class template.

Step 1: Implement the `unique_ptr` base template for single resources.

Step 2: Add a template parameter for a deleter.

Step 3: Implement a `unique_ptr` specialization for arrays.

Step 4: Implement the copy and move operations.

Step 5: Implement move operations for different pointer types.

Step 6: Implement the `reset()` and `release()` functions.

Defining a Class Template

```
template< typename T >
class unique_ptr
{
 public:
   using pointer = T*;
   using element_type = T;

   constexpr unique_ptr();
   explicit  unique_ptr( T* ptr );
             unique_ptr( unique_ptr const& u ) = delete;
             unique_ptr( unique_ptr&& u ) noexcept;

   // ...

 private:
   T* ptr_;

   // ...
};
```

Template parameter list (can be type parameters or non-type parameters)

Nested type aliases

Member function declarations

Declaration of member data

Instantiating a Class Template

```
int main()
{
   // Instantiates the unique_ptr class template for ‘int’
   // This line of code will only instantiate the according
   // constructor and destructor. No other function is instantiated
   // (on-demand instantiation)
   unique_ptr<int> iptr( new int(42) );

   // Instantiates the unique_ptr class template for ‘std::string’
   // Note that this is a separate class and has nothing to do with
   // the instantiation for ‘int’
   unique_ptr<std::string> sptr( new std::string( "42" ) );

   return EXIT_SUCCESS;
}
```

Member Functions of Class Templates

```
template< typename T >
class unique_ptr
{
 public:
   // ...
   pointer release() noexcept { return std::exchange( ptr_, {} ); }

   void reset( pointer ptr = pointer{} ) noexcept;

 private:
   T* ptr_;
};
```

Function definition within the class body

Function definition outside the class body

```
template< typename T, typename D >
void unique_ptr<T,D>::reset( pointer ptr ) noexcept
{
   unique_ptr tmp( std::exchange( ptr_, ptr ) );
}
```

Member Functions of Class Templates

```
template< typename T >
class unique_ptr
{
 public:
   // ...

   pointer release() noexcept;

 private:
   T* ptr_;
};

template< typename T >
pointer unique_ptr<T>::release() noexcept
{
   return std::exchange( ptr_, {} );
}
```

Function declaration within the class body, definition outside

Compilation error: Type ‘pointer’ is not known in this context

Member Functions of Class Templates

```
template< typename T >
class unique_ptr
{
 public:
   // ...

   pointer release() noexcept;

 private:
   T* ptr_;
};

template< typename T >
unique_ptr<T>::pointer unique_ptr<T>::release() noexcept
{
   return std::exchange( ptr_, {} );
}
```

Function declaration within the class body, definition outside

Compilation error: ‘pointer’ is not recognized as type

Member Functions of Class Templates

```
template< typename T >
class unique_ptr
{
 public:
   // ...

   pointer release() noexcept;

 private:
   T* ptr_;
};

template< typename T >
typename unique_ptr<T>::pointer unique_ptr<T>::release() noexcept
{
   return std::exchange( ptr_, {} );
}
```

Function declaration within the class body, definition outside

Disambiguation with the ‘typename’ keyword

Member Functions of Class Templates

```
template< typename T >
class unique_ptr
{
 public:
   // ...
   pointer release() noexcept;

 private:
   T* ptr_;
};

template< typename T >
auto unique_ptr<T>::release() noexcept -> pointer
{
   return std::exchange( ptr_, {} );
}
```

Function declaration within the class body, definition outside

Alternative formulation with a trailing return type. No disambiguation is required!

Use of a Class Template Name Inside the Class

```
template< typename T >
class unique_ptr
{
 public:
   // ...
   constexpr unique_ptr();
   explicit  unique_ptr<T>( T* ptr );
   // ...

 private:
   T* ptr_;

   // ...
};
```

Within the class definition it is not necessary to repeat the class template parameters

It is possible, however, to also mention the class template parameter(s); It is not recommended, though, for readability

Use of a Class Template Name Outside the Class

```
template< typename T >
class unique_ptr
{
 public:
   // ...

   void reset( pointer ptr = pointer{} ) noexcept;

 private:
   T* ptr_;
};

template< typename T, typename D >
void unique_ptr<T,D>::reset( pointer ptr ) noexcept
{
   unique_ptr tmp( std::exchange( ptr_, ptr ) );
}
```

Outside the class template the class template parameters need to be explicitly used

Member Templates

```
template< typename T >
class unique_ptr
{
 public:
   // ...
   template< typename U >
   unique_ptr( unique_ptr<U>&& u ) noexcept;
   // ...

 private:
   T* ptr_;
};
```

Declaration of a member function template

Definition of a member function template outside the class definition

Note the two consecutive list of template parameters. The first refers to the class template, the second to the member function template.

```
template< typename T >
template< typename U >
unique_ptr<T>::unique_ptr( unique_ptr<U>&& u ) noexcept
   : ptr_( u.ptr_ )
{
   u.ptr_ = nullptr;
}
```

Class Templates and Friends

```
template< typename T >
class unique_ptr
{
 public:
   // ...
   template< typename U >
   unique_ptr( unique_ptr<U>&& u ) noexcept;
   // ...

 private:
   T* ptr_;

   friend class unique_ptr<int>;
   template< typename U > friend class unique_ptr;
};
```

Different instantiations of class templates cannot access the data members of other instantiations. If access is necessary, a friend is required.

It is possible to declare a specific instantiation a friend, or all instantiations.

```
template< typename T >
template< typename U >
unique_ptr<T>::unique_ptr( unique_ptr<U>&& u ) noexcept
   : ptr_( u.ptr_ )
{
   u.ptr_ = nullptr;
}
```

Friend access to another instantiation of the unique_ptr class template

Default Template Arguments

```
// Declaration of the unique_ptr class template
template< typename T
        , typename D = default_delete<T> >
class unique_ptr;
```

Default template parameter.

The default parameter must be specified exactly once (ODR) and must be visible even if the class template is forward declared. Therefore the default is usually part of the declaration.

```
// Definition of the unique_ptr class template
template< typename T
        , typename D >
class unique_ptr
{
   // ...
};
```

Default Template Arguments

```
template< typename T >
struct chatty_delete {
   void operator()( T* ptr ) const {
      std::cout << "chatty_delete: deleting ptr...\n";
      delete ptr;
   }
};

int main()
{
   // Instantiating the class template with the default template
   // parameter (i.e. default_delete<int>)
   unique_ptr<int> iptr1( new int(42) );

   // Instantiating the class template with another deleter. This
   // will instantiate another, independent type.
   unique_ptr<int,chatty_delete<int>> iptr2( new int(43) );

   return EXIT_SUCCESS;
}
```

Class Template Partial Specialization

```
template< typename T, typename D >
class unique_ptr<T[],D>
{
 public:
   using pointer = T*;
   using element_type = T;

   // ...
   T& operator[]( size_t index ) const;
   template< typename U > void reset( U ptr ) noexcept;
   void reset( std::nullptr_t ptr = nullptr ) noexcept;
   // ...

 private:
   T* ptr_;
};
```

Template parameter list. No separate default parameters possible!

Specialization pattern. The compiler selects the specialization based on this pattern.

Nested type aliases

Member function declarations. The specialization is a separate type and therefore can provide a different set of operations.

Declaration of member data

```
template< typename T, typename D >
T& unique_ptr<T[],D>::operator[]( size_t index ) const
{
   return ptr_[index];
}
```

Member function definition outside the class definition

Class Template Partial Specialization

```
int main()
{
   // Instantiates a the unique_ptr class template for a single ‘int’
   // This line of code will only instantiate the according
   // constructor and destructor. No other function is instantiated
   // (on-demand instantiation)
   unique_ptr<int> iptr( new int(42) );

   // Instantiates a the unique_ptr class template specialization
   // for an array of 'int's. This line of code will only instantiate
   // the according constructor and destructor.
   unique_ptr<int[]> sptr( new int[10] );

   return EXIT_SUCCESS;
}
```

Class Template Full Specialization

Empty template parameter list

```
template< >
class unique_ptr<int,default_delete<int>>
{
 public:
   using pointer = int*;

   pointer release() noexcept;
   // ...

 private:
   int* ptr_;
};

auto unique_ptr<int,default_delete<int>>::release() noexcept
   -> pointer
{
   return std::exchange( ptr_, {} );
}
```

Specialization pattern, consisting of concrete types or values that don't depend on a template parameter

release() function must be defined as ordinary function, since it is not a template anymore!

Class Template Full Specialization

```
template<  >
class unique_ptr<int,default_delete<int>>;

int main()
{
   // Instantiates a the unique_ptr class template full specialization
   // for a single 'int' and 'default_delete<int>'. This line of code
   // will only instantiate the according constructor and destructor.
   unique_ptr<int> iptr( new int(42) );


   return EXIT_SUCCESS;
}
```

Programming Task

Task (2_Templates/Vector1): Rework the `StringVector` class to a class template named `Vector`.

Programming Task

Task (2_Templates/Vector2): Add an `emplace_back()` function to the `Vector` class template.

Programming Task

Task (2_Templates/FixedVector1): Implement the class template `FixedVector`. A fixed vector represents a hybrid between `std::vector` and `std::array`, i.e. it holds a maximum number of elements in static memory but can be resized within this bound.

```
template< typename Type         // Type of the elements
        , size_t Capacity >    // Maximum number of elements
class FixedVector;
```

Type Traits – std::is_const

```
namespace std {

template< typename T >          // The primary template
struct is_const                 // represents the default
   : public false_type          // behavior
{};

template< typename T >          // Each specialization
struct is_const<T const>        // handles a specific true
   : public true_type           // case
{};

template< typename T >
constexpr bool is_const_v = is_const<T>::value;

} // namespace std
```

Example: IsConst Type Trait

Task (2_Templates/TypeTraits/IsConst): Implement the `is_const` type trait in the following example. The type trait should determine if the given type is a `const`-qualified type. In case the given type is a `const`-qualified type, the type trait should inherit from `std::true_type`, else it should derive from `std::false_type`.

Type Traits — Usage of std::remove_const

```
#include <type_traits>

int main()
{
   using Type1 = int;
   using Type2 = int const;
   using Type3 = int volatile;
   using Type4 = int const volatile;

   using Result1 = std::remove_const<Type1>::type;
   using Result2 = std::remove_const<Type2>::type;
   using Result3 = std::remove_const<Type3>::type;
   using Result4 = std::remove_const<Type4>::type;

   static_assert( std::is_same_v< Result1, int          > );
   static_assert( std::is_same_v< Result2, int          > );
   static_assert( std::is_same_v< Result3, int volatile > );
   static_assert( std::is_same_v< Result4, int volatile > );

   return EXIT_SUCCESS;
}
```

Type Traits – Usage of std::remove_const

```
#include <type_traits>

int main()
{
   using Type5 = int*;
   using Type6 = int* const;
   using Type7 = int* volatile;
   using Type8 = int* const volatile;

   using Result5 = std::remove_const<Type5>::type;
   using Result6 = std::remove_const<Type6>::type;
   using Result7 = std::remove_const<Type7>::type;
   using Result8 = std::remove_const<Type8>::type;

   static_assert( std::is_same_v< Result5, int*          > );
   static_assert( std::is_same_v< Result6, int*          > );
   static_assert( std::is_same_v< Result7, int* volatile > );
   static_assert( std::is_same_v< Result8, int* volatile > );

   return EXIT_SUCCESS;
}
```

Type Traits – Usage of std::remove_const

```
#include <type_traits>

int main()
{
   using Type9  = int&;
   using Type10 = int const&;
   using Type11 = int volatile&;
   using Type12 = int const volatile&;

   using Result9  = std::remove_const<Type9 >::type;
   using Result10 = std::remove_const<Type10>::type;
   using Result11 = std::remove_const<Type11>::type;
   using Result12 = std::remove_const<Type12>::type;

   static_assert( std::is_same_v< Result9 , int&                > );
   static_assert( std::is_same_v< Result10, int const&          > );
   static_assert( std::is_same_v< Result11, int volatile&       > );
   static_assert( std::is_same_v< Result12, int const volatile& > );

   return EXIT_SUCCESS;
}
```

Type Traits – Usage of std::remove_const

```
#include <type_traits>

int main()
{
   using Type9  = int&;
   using Type10 = int const&;
   using Type11 = int volatile&;
   using Type12 = int const volatile&;

   using Result9  = std::remove_const<Type9 >::type; // Using '::type'
   using Result10 = std::remove_const<Type10>::type; // often quickly
   using Result11 = std::remove_const<Type11>::type; // becomes a
   using Result12 = std::remove_const<Type12>::type; // nuisance ...

   static_assert( std::is_same_v< Result9 , int&                > );
   static_assert( std::is_same_v< Result10, int const&          > );
   static_assert( std::is_same_v< Result11, int volatile&       > );
   static_assert( std::is_same_v< Result12, int const volatile& > );

   return EXIT_SUCCESS;
}
```

Type Traits – Usage of std::remove_const

```
#include <type_traits>

int main()
{
   using Type9  = int&;
   using Type10 = int const&;
   using Type11 = int volatile&;
   using Type12 = int const volatile&;

   using Result9  = std::remove_const_t<Type9 >;  // … therefore we
   using Result10 = std::remove_const_t<Type10>;  // abbreviate it
   using Result11 = std::remove_const_t<Type11>;  // with '_t' after
   using Result12 = std::remove_const_t<Type12>;  // the type trait.

   static_assert( std::is_same_v< Result9 , int&                > );
   static_assert( std::is_same_v< Result10, int const&          > );
   static_assert( std::is_same_v< Result11, int volatile&       > );
   static_assert( std::is_same_v< Result12, int const volatile& > );

   return EXIT_SUCCESS;
}
```

Type Traits – std::remove_const

```
namespace std {

template< typename T >              // The primary template
struct remove_const                 // represents the default
{                                   // behavior
   using type = T;
};

template< typename T >              // Each specialization
struct remove_const<T const>        // handles a specific true
{                                   // case
   using type = T;
};

template< typename T >
using remove_const_t = typename remove_const<T>::type;

} // namespace std
```

Example: RemoveConst Type Trait

Task (2_Templates/TypeTraits/RemoveConst): Implement the `remove_const` type trait in the following example. The type trait should remove any top level `const` qualifier from the given type.

Example: IsPointer Type Trait

Task (2_Templates/TypeTraits/IsPointer): Implement the `is_pointer` type trait in the following example. The type trait should determine if the given type is a pointer type. In case the given type is a pointer type, the type trait should inherit from `std::true_type`, else it should derive from `std::false_type`.

Class Template Guidelines

Core Guideline T.42: Use template aliases to simplify notation and hide implementation details

Core Guideline T.64: Use specialization to provide alternative implementations of class templates

2.5. CTAD and Deduction Guides

Deducing a Class Template Parameter

In order to instantiate a class template, every template argument must be known, but not every template argument has to be specified.

```
// deduces to std::pair<int, double> p(2, 4.5);
std::pair p(2, 4.5);

// same as auto t = std::make_tuple(4, 3, 2.5);
std::tuple t(4, 3, 2.5);

// same as std::less<void> l;
std::less l;
```

Deducing a Class Template Parameter

```
template< typename T >
class Widget
{
 public:
   using value_type = T;

   Widget( T const& ) {}
};



int main()
{
   Widget<int> w1( 42 );   // Explicit template parameter
   Widget      w2( 42 );   // Class Template Argument Deduction (CTAD)

   return EXIT_SUCCESS;
}
```

Deducing a Class Template Parameter

```
template< typename T >
class Widget
{
 public:
   using value_type = T;

   Widget( T const& ) {}
};

template< typename T >
Widget( T const& ) -> Widget<T>;

int main()
{
   Widget<int> w1( 42 );   // Explicit template parameter
   Widget      w2( 42 );   // Class Template Argument Deduction (CTAD)

   return EXIT_SUCCESS;
}
```

The class template parameter of w2 is deduced from the given constructor. This constructor implicitly provides a rule for how to deduce the class template parameter T. This rule is called a deduction guide.

Deducing a Class Template Parameter

```
namespace std {

template< typename T, /*...*/ >
class vector
{
 public:
   using value_type = T;

   template< typename Iter >
   vector( Iter begin, Iter end );

   // ...
};

template< typename Iter >
vector(Iter begin, Iter end)
   -> vector< typename std::iterator_traits<Iter>::value_type >;

} // namespace std

int main()
{
   vector<int> v1{ 1, 2, 3, 4, 5 };           // Explicit template parameter
   vector      v2( v1.begin(), v1.end() );   // CTAD via explicit deduction guide

   return EXIT_SUCCESS;
}
```

This explicit deduction guide describes the rule on how to deduce T from a given type of iterator.

2.6. Variadic Templates

Programming Task

Task (2_Templates/Print):

Task 1: Extend the given `println()` function by variadic templates to enable an arbitrary number of function arguments.

Task 2: Extend the `println()` function with a parameter to format the given values.

```
template< typename... Ts >
void println( std::ostream& os, Ts const&... values )
{
   ( os << ... << values ) << '\n';
}
```

Solution 1: Tail Recursion with Overloading

C++11

This overload serves as termination criterion for the tail recursion

```
template< typename T >
void println( std::ostream& os, T const& value )
{
   os << value << '\n';
}

template< typename T, typename... Ts >
void println( std::ostream& os, T const& value, Ts const&... values )
{
   os << value;
   println( os, values... );
}
```

Solution 2: Tail Recursion with constexpr if

```
template< typename T, typename... Ts >
void println( std::ostream& os, T const& value, Ts const&... values )
{
   os << value;

   if constexpr( sizeof...(Ts) > 0 ) {
      println( os, values... );
   }
   else {
      os << '\n';
   }
}
```

Solution 3: Fold Expressions

C++17

```
template< typename... Ts >
void println( std::ostream& os, Ts const&... values )
{
   ( os << ... << values ) << '\n';
}
```

Fold Expressions

The Situation up to C++14

```cpp
// Sum up all the given numbers
template< typename... Ns >
auto sum( Ns... ns );
```

The Situation up to C++14

```
auto sum()
{
   return 0;
}

template< typename N >
auto sum( N n )
{
   return n;
}

template< typename N0, typename... Ns >
auto sum( N0 n0, Ns... ns )
{
   return n0 + sum( ns... );
}
```

C++17 Fold Expressions

```
// Sum up all the given numbers
template< typename... Ns >
auto sum( Ns... ns )
{
   return ( ns + ... + 0 );
}
```

Note the parenthesis around the fold expression. They are required!

Example:

```
auto a = sum( 3.14, 1E7, -42, 17 );
   // 3.14 + ( 1E7 + ( -42 + ( 17 + 0 ) ) )
```

C++17 Fold Expressions

Unary Right Fold	*(E op ...)*	E_1 *op (... op (*E_{N-1} *op* E_N*))*
Unary Left Fold	*(... op E)*	*((*E_1 *op* E_2*) op ...) op* E_N
Binary Right Fold	*(E op ... op I)*	E_1 *op (... op (*E_{N-1} *op (*E_N *op I)))*
Binary Left Fold	*(I op ... op E)*	*(((I op* E_1*) op* E_2*) op ...) op* E_N

- Fold expressions apply unary or binary operators to parameter packs
- Parentheses around the fold expression are required
- The following 32 operators are foldable:

==	!=	<	>	<=	>=	&&	||	,	.*	->*	=
+	-	*	/	%	^	&	|	<<	>>		
+=	-=	*=	/=	%=	^=	&=	|=	<<=	>>=		

C++17 Fold Expressions

For unary folds, if the parameter pack is empty then the value of the fold is

&&	*true*
||	*false*
,	*void()*

For any operator not listed above, a unary fold expression with an empty parameter pack is ill-formed.

C++17 Fold Expressions — Examples

```
template< typename... Ts >
void println( std::ostream& os, Ts&&... ts )
{
   ( os << ... << std::forward<Ts>( ts ) ) << '\n';
}
```

```
template< typename F, typename... Args >
void for_each_arg( F f, Args&&... args )
{
   ( f( std::forward<Args>( args ) ), ... );
}
```

Programming Task

Task (2_Templates/Sum):

Step 1: Write a generic `sum()` function that can compute the sum of an arbitrary number of values of any type.

Step 2: Write a generic `sum_if()` function that computes the sum of all values that fulfill a given predicate (e.g. `is_positive()`).

Programming Task

Task (2_Templates/VariadicMax):

Step 1: Extend the `max()` function template for an arbitrary number of elements.

Step 2: Use the `max()` function template to determine the size of the largest type given to the `Variant` class template.

Programming Task

Task (4_Class_Design/VariadicMax_Assembly): Copy-and-paste the following code into Compiler Explorer (i.e. godbolt.org). Analyse the resulting assembly code of the given `max()` implementations with different compilers and a high optimization level (e.g. `-O3` or `/O2`).

Programming Task

Task (2_Templates/VariadicMinMax): Extend the `minmax()` function template for an arbitrary number of elements.

Programming Task

Task (2_Templates/PrintTuple): Write an output operator for a tuple of variadic size and content.

Programming Task

Task (2_Templates/PackChallenges): Provide a solution for the following challenges:

Task 1: Sum up all the elements of a pack.

Task 2: Sum up a pack of modified elements.

Task 3: Check whether any element matches a given predicate `pred()`.

Task 4: Count the elements matching a given predicate `pred()`.

Task 5: Get the index of the first element matching a given predicate.

Task 6: Get the nth element (where `n` is a runtime value).

Task 7: Get the first element of a parameter pack.

Task 8: Get the last element of a parameter pack.

Task 9: Get the minimum element of a parameter pack.

Task 10: Get the maximum element of a parameter pack.

Programming Task

Task (2_Templates/HigherOrder): Implement the given `count_if()` call in terms of reusable higher order functions called `youngerThan()`, `hasName()` and `when_all()`. `youngerThan()` should provide a reusable predicate to check for a given age, `hasName()` should provide a reusable predicate to check for a given name, and `when_all()` should provide a generic way to combine an arbitrary number of predicates.

Programming Task

Task (2_Templates/AddSub): Implement the `addsub()` function, which performs an alternating addition/subtraction of the given arguments.

Programming Task

Task (2_Templates/VariantIndex): Write the `variant_index` class template to evaluate the index of an alternative within a given variant V. In case the given type `T` is found in the list of alternatives, `variant_index` should return the index. Otherwise `variant_index` should return `std::variant_npos`.

Programming Task

Task (2_Templates/CartesianProduct): Implement the following `cartesian_product()` function to determine the cartesian product of several given input ranges. For example, the given `main()` function should result in the following output:

```
1-a
1-b
2-a
2-b
3-a
3-b
```

Examples from the Standard Library

cppreference.com Create account Search

Page Discussion View Edit History

C++ / Utilities library / Dynamic memory management / std::unique_ptr

std::make_unique, std::make_unique_for_overwrite

Defined in header <memory>

`template< class T, class... Args >` `unique_ptr<T> make_unique( Args&&... args );`	(1)	(since C++14) (only for non-array types)
`template< class T >` `unique_ptr<T> make_unique( std::size_t size );`	(2)	(since C++14) (only for array types with unknown bound)
`template< class T, class... Args >` `/* unspecified */ make_unique( Args&&... args ) = delete;`	(3)	(since C++14) (only for array types with known bound)
`template< class T >` `unique_ptr<T> make_unique_for_overwrite( );`	(4)	(since C++20) (only for non-array types)
`template< class T >` `unique_ptr<T> make_unique_for_overwrite( std::size_t size );`	(5)	(since C++20) (only for array types with unknown bound)
`template< class T, class... Args >` `/* unspecified */ make_unique_for_overwrite( Args&&... args ) = delete;`	(6)	(since C++20) (only for array types with known bound)

Constructs an object of type T and wraps it in a `std::unique_ptr`.

Programming Task

Task (2_Templates/MakeUnique): Implement the `make_unique()` function for the `std::unique_ptr` class template.

Examples from the Standard Library

cppreference.com Create account Search

Page Discussion View Edit History

C++ / Containers library / std::vector

std::vector<T,Allocator>::emplace

```
template< class... Args >
iterator emplace( const_iterator pos, Args&&... args );
```
(since C++11) (until C++20)

```
template< class... Args >
constexpr iterator emplace( const_iterator pos, Args&&... args );
```
(since C++20)

Inserts a new element into the container directly before pos.

The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at a location provided by the container. However, if the required location has been occupied by an existing element, the inserted element is constructed at another location at first, and then move assigned into the required location.

The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`. `args...` may directly or indirectly refer to a value in the container.

If the new `size()` is greater than `capacity()`, all iterators and references are invalidated. Otherwise, only the iterators and references before the insertion point remain valid. The past-the-end iterator is also invalidated.

Parameters

pos - iterator before which the new element will be constructed

Programming Task

Task (2_Templates/Vector2): Add an `emplace_back()` function to the `Vector` class template.

Examples from the Standard Library

C++ / Utilities library / std::variant

std::variant

Defined in header <variant>

```
template <class... Types>
class variant;
```
(since C++17)

The class template `std::variant` represents a type-safe union. An instance of `std::variant` at any given time either holds a value of one of its alternative types, or in the case of error - no value (this state is hard to achieve, see `valueless_by_exception`).

As with unions, if a variant holds a value of some object type T, the object representation of T is allocated directly within the object representation of the variant itself. Variant is not allowed to allocate additional (dynamic) memory.

A variant is not permitted to hold references, arrays, or the type void. Empty variants are also ill-formed (`std::variant<std::monostate>` can be used instead).

A variant is permitted to hold the same type more than once, and to hold differently cv-qualified versions of the same type.

Consistent with the behavior of unions during aggregate initialization, a default-constructed variant holds a value of its first alternative, unless that alternative is not default-constructible (in which case the variant is not default-constructible either). The helper class `std::monostate` can be used to make such variants default-constructible.

Template parameters

Programming Task

Task (2_Templates/VariantIndex): Write the `variant_index` class template to evaluate the index of an alternative within a given variant V. In case the given type `T` is found in the list of alternatives, `variant_index` should return the index. Otherwise `variant_index` should return `std::variant_npos`.

Examples from the Standard Library

cppreference.com   Create account   Search

Page Discussion   View Edit History

C++ / Utilities library / std::tuple

std::tuple

Defined in header <tuple>

```
template< class... Types >
class tuple;
```
(since C++11)

Class template `std::tuple` is a fixed-size collection of heterogeneous values. It is a generalization of `std::pair`.

If `std::is_trivially_destructible<Ti>::value` is `true` for every `Ti` in `Types`, the destructor of `tuple` is trivial.

Template parameters

Types... - the types of the elements that the tuple stores. Empty list is supported.

Member functions

(constructor) (C++11)	constructs a new `tuple` (public member function)
operator= (C++11)	assigns the contents of one `tuple` to another (public member function)
swap (C++11)	swaps the contents of two `tuple`s (public member function)

Variadic Template Guidelines

Core Guideline T.100: Use variadic templates when you need a function that takes a variable number of arguments of a variety of types

Core Guideline T.103: Don’t use variadic templates for homogeneous arguments lists

2.7. Non-Type Template Parameters

An Example

template parameter list

```
template< typename T
        , size_t MaxSize = 100UL >
class Stack {
private:
    T m_elems[MaxSize];
    size_t m_numElems;
public:
    void push ( T const& );
    void pop  ();
    T    top  () const;
    bool empty() const {
        return m_numElems == 0UL;
    }
    bool full() const {
        return m_numElems == MaxSize;
    }
};
```

template default argument

function definitions within the class body

Restrictions

Non-Type template parameters may be ...

- ... constant integral values (constants, `constexpr`, literals, ...)
- ... constant floating point values (since C++20)
- ... pointers to objects with external linkage

Non-Type template parameters may not be ...

- ... class-type objects
- ... pointers to objects with internal linkage (e.g. string literals)

2.8. Tricky Basics

The `typename` Keyword

The `typename` keyword has two distinct uses. First it is used to declare a template type parameter ...

```
template< typename T >
class MyClass {
   // ...
};
```

... and second it is used to disambiguate nested values/objects from types:

```
template< typename T >
class MyClass {
   typename T::SubType* ptr;
   // ...
};
```

`typename` is required whenever a name that depends on a template parameter is a type.

Using this->

Consider the following example:

```
template< typename T >
class Base {
 public:
   void bar();
};

template< typename T >
class Derived : Base<T> {
 public:
   void foo() {
      bar();
   }
};
```

Using `this->`

Consider the following example:

```cpp
template< typename T >
class Base {
 public:
   void bar();
};

template< typename T >
class Derived : Base<T> {
 public:
   void foo() {
      bar();  // <- Results in a compilation error
   }
};
```

Using `this->`

Output from the GNU C++ compiler:

```
$ g++ -o DependentNames DependentNames.cpp
DependentNames.cpp:19:11: error: there are no arguments to ‘bar’
that depend on a template parameter, so a declaration of ‘bar’ must
be available [-fpermissive]
```

Using this->

Consider the following example:

```cpp
template< typename T >
class Base {
 public:
   void bar();
};

template< typename T >
class Derived : Base<T> {
 public:
   void foo() {
      this->bar();
   }
};
```

Using this->

The reason behind this behavior:

```
template< typename T >
struct Base {
   int basefield;
};

template< typename T >
struct Derived : public Base<T>
{
   void f() { basefield = 0; }
};

template<>
struct Base<bool>
   enum { basefield = 0; }
};

void g( Derived<bool>& d ) {
   d.f();
}
```

Zero Initialization

1. Quick task: What are the initial values of `x` and `ptr`?

```
void foo()
{
    int  x;     // x has undefined value
    int* ptr;   // ptr points to somewhere
}
```

Zero Initialization

2. Quick task: What is the initial value of `x` ?

```
template< typename T >
void foo()
{
   T x;  // x has undefined value only if T is of
         // built-in type; otherwise the according
         // constructor is called.
}
```

Zero Initialization

3. Quick task: Is the following a solution to ”zero initialize” `x` ?

```
template< typename T >
void foo()
{
   T x();  // Explicitly calling the ”constructor”


}
```

Zero Initialization

3. Quick task: Is the following a solution to ”zero initialize” `x` ?

```
template< typename T >
void foo()
{
   T x();  // Function declaration


}
```

No, unfortunately it’s not. This is a function declaration, not an explicit constructor call.

Zero Initialization

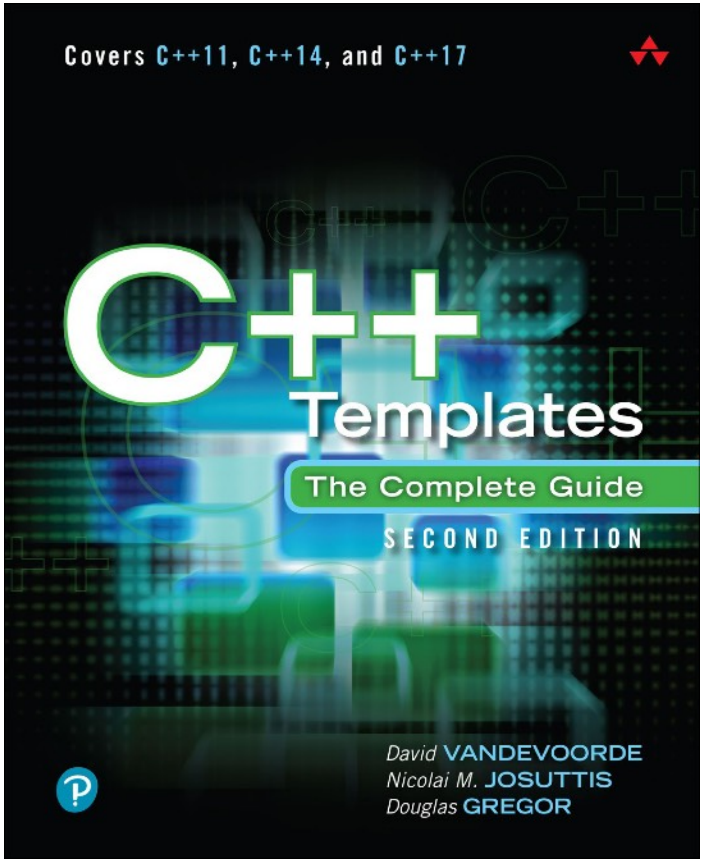
In order to ”zero initialize” type dependent values of built-in data type, you should be doing the following

```
template< typename T >
void foo()
{
   T x{};  // Explicit ”zero initialization”


}
```

Things to Remember

- Use the power of templates to create isolation points
- Mind the differences between function template overloading and specialization
- Prefer function overloading to function template specialization
- Remember that specialized class templates are new types
- Don’t forget non-type template parameters
- Remember the tricky details 😄

Literature


Covers C++11, C++14, and C++17
C++
Templates
The Complete Guide
SECOND EDITION
David VANDEVOORDE
Nicolai M. JOSUTTIS
Douglas GREGOR

References (Templates)

- Dan Saks, "Back to Basics: Function and Class Templates". CppCon 2019 (https://www.youtube.com/watch?v=LMP_sxOaz6g)
- Andreas Fertig, "Back to Basics: Templates (part 1 of 2)". CppCon 2020 (https://www.youtube.com/watch?v=VNJ4wiuxJM4)
- Andreas Fertig, "Back to Basics: Templates (part 2 of 2)". CppCon 2020 (https://www.youtube.com/watch?v=0dtjDTEE0hQ)
- Bob Steagall, "Back to Basics: Templates (part 1 of 2)". CppCon 2021 (https://www.youtube.com/watch?v=XN319NYEOcE)
- Bob Steagall, "Back to Basics: Templates (part 2 of 2)". CppCon 2021 (https://www.youtube.com/watch?v=2Y9XbltAfXs)
- Scott Meyers, "Type Deduction and Why You Care". CppCon 2014 (https://www.youtube.com/watch?v=wQxj20X-tIU)
- Walter Brown, "C++ Function Templates: How Do They Really Work?". C++ on Sea 2019 (https://www.youtube.com/watch?v=nfIX8yWlByY)
- Walter Brown, "Modern Template Programming: A Compendium, Part 1". CppCon 2014 (https://www.youtube.com/watch?v=Am2is2QCvxY)
- Walter Brown, "Modern Template Programming: A Compendium, Part 2". CppCon 2014 (https://www.youtube.com/watch?v=a0FliKwcwXE)

email: klaus.iglberger@cpp-training.com

LinkedIn: linkedin.com/in/klaus-iglberger

Xing: xing.com/profile/Klaus_Iglberger/cv