

C++ Club UK

Gleb Dolgich

2019-03-14

Making C++ Exception Handling Smaller On x64

<https://devblogs.microsoft.com/cppblog/making-cpp-exception-handling-smaller-x64/>

*Visual Studio 2019 Preview 3 introduces a new feature to reduce the binary size of C++ exception handling (try/catch and automatic destructors) on x64. Dubbed **FH4** (for `_CxxFrameHandler4`, see below), I developed new formatting and processing for data used for C++ exception handling that is ~60% smaller than the existing implementation, resulting in overall binary reduction of up to 20% for programs with heavy usage of C++ exception handling.*

https://www.reddit.com/r/cpp/comments/ayeg0b/making_c_exception_handling_smaller_on_x64/

C++, it's not you. It's me.

Blog post: <https://c0de517e.blogspot.com/2019/02/c-its-not-you-its-me.html>

The crux of the issue <...> is the growing disconnect between people working on big, complex, performance-sensitive and often monolithic and legacy-ridden codebases that we find in game development, and the ideas of "modernity" of the C++ standard community.

Reddit

I'm just going to throw it out there: I'm tired of reading this kind of stuff from game devs.

Acting like they are the only ones with performance issues. The only ones with complex codebases. The only ones who actually ship code. The ones who are just solving more unique and difficult problems than anyone else, for which C++ falls short by more.

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

The image shows a presentation slide titled "C++ Function Templates: How Do They Really Work? - Walter E. Brown". The slide contains text and code snippets. On the right side of the slide, there is a photograph of a man with glasses and a beard, wearing a dark suit and tie, standing and gesturing with his hands. In the bottom right corner of the slide, there is a logo for "C++ On Sea 2019" featuring a stylized orange and yellow square above blue wavy lines.

Instructive example

- First, let's overload the name `g` (braced def'n bodies omitted):
 - ① `template< class T > void g(T const&) //function template`
 - ② `template < > void g(int const&) // explicit specialization`
 - ③ `void g(double) // ordinary function`
- Q1: How many *declarations* are in the above?
- Q2: How many names do they *introduce* (declare)?
- Answers:
 - There are **3** declarations (numbered above), but ...
 - Only **2** (mangled) names are being introduced.
☞ An explicit specialization does not introduce a new name; it re-uses the (existing) name of its primary template!

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

13

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

The image shows a video frame. On the left is a presentation slide titled "C++ Function Templates: How Do They Really Work? - Walter E. Brown". The slide features a "SPOILER ALERT!" section with a list of bullet points and code snippets. On the right is a photograph of Walter E. Brown, an older man with a beard and glasses, wearing a dark suit and tie, standing and gesturing while speaking.

SPOILER ALERT!

- From those same 3 declarations ...
~~template< class T > void g(T const&) //function template~~
~~template <> void g(int const&) // explicit specialization~~
✓ void g(double) // ordinary function
- ... at most 2 candidate declarations will be considered!
- What are those 2 candidate declarations?
 - The ordinary function declaration void g(double); , and ...
 - A synthesized function declaration void g<>(D const&);
 - Type *D* is often inferred (e.g., from the arg's type in the call).
 - Then the primary template's declaration is copied, with each instance of template param *T* replaced by the inferred type *D*.
 - Can ensure this substitution by calling as g<*D*>(...); .

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

15

cpponsea.uk @cpponsea

C++ 2019

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*



Wait, what?

- Why is the (primary) function template not a candidate?
`X template< class T > void g(T const&) //function template`
`X template < > void g(int const&) // explicit specialization`
`✓ void g(double) // ordinary function`
- Recall that the goal is to select a function (to be called), but:
 - A primary template can't be called.
 - Why not? Because a template isn't a function!
 - Nor is a template any other kind of callable entity.
- ∴ Templates are never candidates for overload resolution.

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

16

cpponsea.uk

@cpponsea



C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Another way to think about it

A cookie cutter :: a function template.



Instantiation

Different dough?
Different cookie!

Cookie dough :: template arg's. A cookie :: a function.

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

17



cpponsea.uk

@cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

The image shows a video frame. On the left is a presentation slide with a dark blue header containing the title "C++ Function Templates: How Do They Really Work? - Walter E. Brown". The main content of the slide is a bulleted list under the heading "So why can't we call a function template?". The list discusses the non-callability of function templates due to their nature as templates and the compiler's role in generating callable declarations. At the bottom of the slide, there is a copyright notice: "Copyright © 2018, 2019 by Walter E. Brown. All rights reserved." and the number "18". On the right side of the frame, a man with a white beard and glasses, wearing a dark suit and tie, stands in a room with light-colored walls, holding a small device. In the bottom right corner of the slide area, there is a logo for "C++ On Sea 2019" featuring stylized waves and the year "2019".

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

So why can't we call a function template?

- Well, would you try to eat a cookie cutter?
 - A cookie cutter isn't edible; it's not a cookie.
 - A function template isn't callable; it's not a function.
- So, a template can never be a candidate to be called:
 - But just as we use cookie cutters + dough to make cookies, ...
 - A compiler uses templates + arguments to make functions!
- *I.e., the compiler will make/create/**synthesize** a function declaration from a (primary) function template declaration:*
 - But first inspects the call, **deduces** argument type(s), then ...
 - **Substitutes** those argument types for template parameters ...
 - Resulting in a **synthesized** candidate for overload resolution.

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

18

 2019

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Well, okay, but then ...

- Why is the explicit specialization not a candidate?
~~template< class T > void g(T const&) //function template~~
~~template < > void g(int const&) // explicit specialization~~
✓ void g(double) // ordinary function
- Yes, it's a declaration, but:
 - Some declarations introduce no new names!
 - E.g., `static_assert` (see [dcl.dcl]/6).
 - E.g., a redeclaration (repeats a name previously introduced).
 - Of more interest, “[an] explicit specialization of a template does not introduce a name” (see [namespace.memdef]/1).
 - Hence “**specializations don’t participate in overloading**”

— Herb Sutter, 2001

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

19

cpponsea.uk @cpponsea



C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

We still need a function def'n to compile/call — where is it?

- Suppose *overload resolution* selects a candidate declaration synthesized from `template< class T > void g(T const&);`:
 - E.g., for a call `g(42)`, makes/selects `void g<>(int const&);`.
 - E.g., for a call `g('4')`, makes/selects `void g<>(char const&);`.
- ① If there's a pre-existing specialization of that template ...
 - Whose template arg's match those of the chosen candidate, ...
 - That specialization is our function to compile (once)/(later) call.
- ② Otherwise, the compiler must/will (only now) **instantiate** a specialization matching the chosen candidate declaration:
 - By duplicating the primary template's definition and ...
 - Substituting template arguments for template parameters ...

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

21

cpponsea.uk

@cpponsea

C++ On Sea 2019

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

The image shows a presentation slide titled "C++ Function Templates: How Do They Really Work? - Walter E. Brown". The slide content is as follows:

Controlling instantiation ①

- It's possible to demand a template's instantiation:
 - Even if there's no corresponding call or other use.
 - Will guarantee which compilation unit contains the function.
 - Useful when creating a library, for example.
- Syntax:
 - After the primary template:
`template< class T > go(T const &) { ... }`
 - Declare an explicit instantiation:
`template< int > go(int const &);`.
 - Can omit the template argument if it's otherwise obtainable.
- Explicit instantiation is rarely useful in a header.

At the bottom left of the slide, there is a copyright notice: "Copyright © 2018, 2019 by Walter E. Brown. All rights reserved." At the bottom right, there is a page number "22".

To the right of the slide, a man with a beard and glasses, wearing a dark suit and tie, is standing and speaking. He is holding a small object in his hands. In the bottom right corner of the slide area, there is a logo for "C++ On Sea 2019" featuring a stylized "C" and "S" with wavy lines underneath.

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

Controlling instantiation ②

- It's possible to prohibit a template's instantiation:
 - Even if there is a corresponding call or other use.
 - You must ensure instantiation elsewhere, if needed, ...
 - So the compiler needn't implicitly instantiate here.
- Syntax:
 - After the primary template:
`template< class T > go(T const &) { ... }`
 - Declare an extern explicit instantiation:
`extern template go<int>(int const &);`
 - Can omit the template argument if it's otherwise obtainable.
- Avoiding instantiation is sometimes useful in a header.

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

23

cpponsea.uk @cpponsea



C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

The screenshot shows a video player interface. The top bar displays the title "C++ Function Templates: How Do They Really Work? - Walter E. Brown". The main area contains a presentation slide with the following content:

Compiler-generated function templates (since C++14)

- Recall that evaluating a *lambda-expression* produces a *closure* object that “behaves like a function object”:
 - auto lambda1 = [] (int x) { return x < 0; };
 - struct { // lambda1's type; such a type's name can't be uttered auto operator() (int x) { return x < 0; } ;};
- Evaluating a *generic lambda-expression* produces a closure object whose *operator()* is a *function template*:
 - auto lambda2 = [] (auto x) { return x < 0; };
 - struct { // lambda2's type; such a type's name can't be uttered template< class _T > auto operator() (_T x) { return x < 0; } ;};

At the bottom of the slide, there is a copyright notice: "Copyright © 2018, 2019 by Walter E. Brown. All rights reserved." and the number "24".

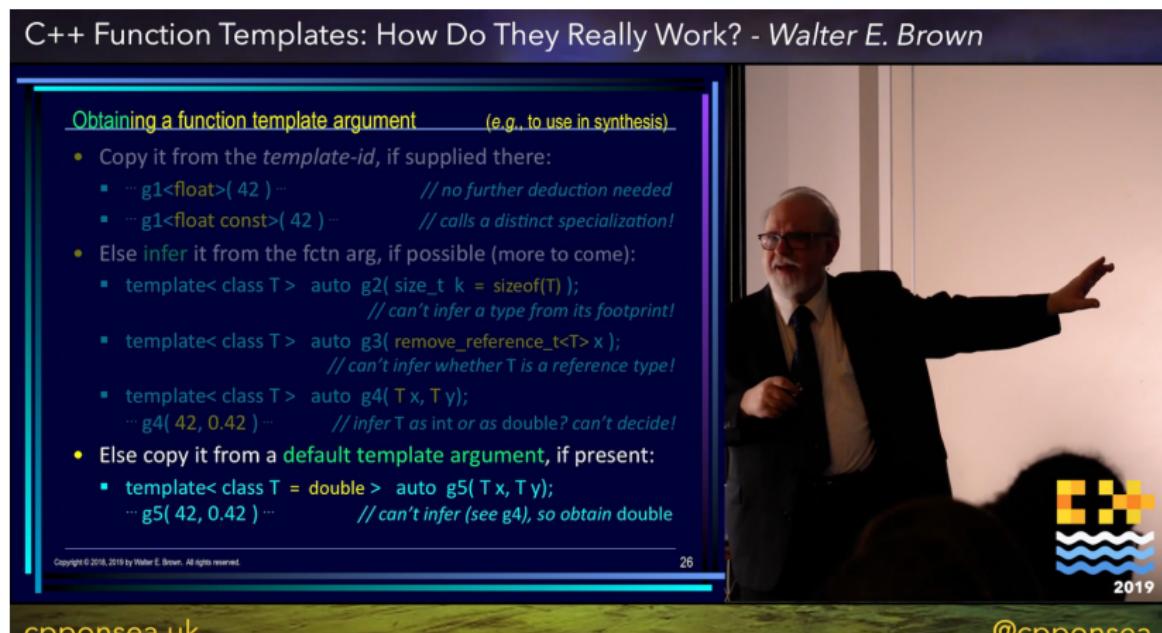
The video frame on the right shows Walter E. Brown, an elderly man with a white beard and glasses, wearing a dark suit and tie, pointing his right hand towards the audience.

At the bottom right of the video frame is a logo for "C++ On Sea 2019" featuring a stylized orange and yellow square above three blue wavy lines.

The footer of the slide includes the URL "cpponsea.uk" and the handle "@cpponsea".

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>



C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Obtaining a function template argument (e.g., to use in synthesis)

- Copy it from the *template-id*, if supplied there:
 - `... g1<float>(42) ...` // no further deduction needed
 - `... g1<float const>(42) ...` // calls a distinct specialization!
- Else *infer* it from the fctn arg, if possible (more to come):
 - `template< class T > auto g2(size_t k = sizeof(T));` // can't infer a type from its footprint!
 - `template< class T > auto g3(remove_reference_t<T> x);` // can't infer whether T is a reference type!
 - `template< class T > auto g4(T x, T y);`
`... g4(42, 0.42) ...` // infer T as int or as double? can't decide!
- Else copy it from a *default template argument*, if present:
 - `template< class T = double > auto g5(T x, T y);`
`... g5(42, 0.42) ...` // can't infer (see g4), so obtain double

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

26

 2019

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

The screenshot shows a video player interface. At the top, a dark bar displays the title "C++ Function Templates: How Do They Really Work? - Walter E. Brown". The main area is divided into two sections: a dark blue slide on the left and a video feed on the right.

Slide Content:

- Section Header:** What does type inference consider?
- List:** The success and result of type inference considers:
 - The function parameter's type (ref-qual? cv-qual?), vis-à-vis the function argument's type, ...
 - So that that argument can initialize that parameter.
 - Conversions? Never considered when inferring a type!

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

Video Feed: A man with glasses and a beard, wearing a dark suit and tie, is speaking. He is gesturing with his hands. In the bottom right corner of the video feed, there is a logo for "C++ On Sea 2019" featuring stylized waves and the year 2019.

Bottom Navigation:

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Inferring template arg **T** from fctn param/arg combinations

g(?)	T	T &	T const&	T const &&	T &&
42	int	<i>int</i>	int	int	int
k	int	int	int	<i>int</i>	int &
kl	int	int	int	<i>int</i>	int &
kr	int	int	int	<i>int</i>	int &
kc	int	int const	int	<i>int</i>	int const &
kcl	int	int const	int	<i>int</i>	int const &
kcr	int	int const	int	<i>int</i>	int const &

☞ Type inference happens in other C++ contexts, too, and follows analogous rules. E.g., `auto x = 42;`.

☞ Recall that parameter passage is initialization; thus, a call's argument is just an initializer for the fctn param.

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

28

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>



C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

The role of partial ordering (adapted from [https://docs.microsoft.com/...](https://docs.microsoft.com/))

- Sometimes, candidates can be synthesized from each of several function templates that match a call's arguments:
 - C++ performs **partial ordering** of such templates/candidates to determine which function is preferred.
 - (The ordering is **partial** because some templates/candidates can be considered equally specialized.)
- The compiler seeks the **most specialized** of the possible (**viable**) candidates:
 - *E.g.,* suppose one fctn template has a **fctn** param of type **T**, while a second fctn template has a **fctn** param of type **T***.
 - Given a call with a **fctn** arg of pointer type, the **T*** param is deemed **more specialized**, hence preferred over the **T** param (although both would be viable matches).

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

30

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Test your understanding ① (the "Dimov/Abrahams example")

- First, let's overload the name `h` (no tricks; this is valid code):
 - `template< class T > void h(T) { ... } // (a)`
 - `template< > void h(int *) { ... } // (b)`
 - `template< class T > void h(T *) { ... } // (c)`
- Now, let's call `h` — but which `h`? (This matters; bodies may differ.)
 - `int * p = nullptr;`
`h(p); // will call an implicit specialization of (c)`
- What templates/candidates will be considered?
 - ~~✗~~ Synthesized from (a): `void h<int *>(int *);`
 - ✓ Synthesized from (c): `void h<int >(int *); // more specialized`
 - Note that (b) is an explicit specialization of (a), not of (c).
Why? Where (b) is declared, only (a) is in scope; haven't yet seen (c).

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

31

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Test your understanding ②

- Here are the same declarations, but in a different order:
 - `template< class T > void h(T) { ... } // (a)`
 - `template< class T > void h(T*) { ... } // (c)`
 - `template< > void h(int*) { ... } // (b)`
- Let's call `h` the same way — same `h` as before?
 - `int * p = nullptr;`
`h(p); // will call (b), now an explicit specialization of (c)`
- What changed by moving (b) to follow both (a) and (c)?
 - Both (a) and (c) are now in scope when (b) is seen, so ...
 - Partial ordering of function templates picks (c) as the primary template corresponding to the explicit specialization (b).
Why pick (c) over (a)? Just as before, (c)'s parameter is already a pointer type, hence (c) is **more specialized**.

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

32



C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Test your understanding ③

- Finally, let's restore the original order, then append (d):
 - template< class T > void h(T) { ... } // (a)
 - template< > void h(int *) { ... } // (b)
 - template< class T > void h(T *) { ... } // (c)
 - template< > void h(int *) { ... } // (d) ↗
- Note the similarity of (b) and (d):
 - Is the above still legal code? If so, which `h` is called now?
 - `int * p = nullptr;`
`h(p); // this time will call (d), an explicit specialization of (c)`
- Note the substitutions in the explicit specializations:
 - template< > void h<int *>(int *) { ... } // (b)
 - template< > void h<int >(int *) { ... } // (d)

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

33

cpponsea.uk @cpponsea



C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Why write your own specialization?

- Typically, to improve an algorithm's performance in a particular case, via instructions particular to that case.
- Example:
 - The generic `swap` algorithm takes 3 `copy/move` operations.
 - Sometimes can swap uints faster via 3 `xor` operations.
- Often termed **tuning**, **refining**, **special-casing**, **adapting**, **customizing**, **tailoring**, or simply **specializing** an algorithm:
 - Do take care to obtain an outcome equivalent to that yielded by the algorithm for the general/generic case.
 - (Not enforced by C++, but Bad Things will otherwise ensue.)

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

35

cpponsea.uk @cpponsea

C++ On Sea 2019

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

Placement of specializations

- As of C++17, “an explicit specialization may be declared in any scope in which the corresponding primary template may be defined” [temp.expl.spec]/2.
- Examples:
 - template<>
void A::g<int>() { … }
// specializes g, a member function template of class A
 - template< class T >
template< >
void B<T>::h<int>() { … }
// specializes h, a member function template
// of class template B

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

36

cpponsea.uk @cpponsea



C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

But beware! ▲

- Customizing function templates via explicit specialization is widely considered a poor programming practice:
 - “it’s wrong to use function template specialization [because] it interacts in bad ways with overloads”
— David Abrahams, 2011
- Main reason:
 - “Overload resolution does not take into account explicit specializations of function templates.
 - “Only after overload resolution [including partial ordering, if needed] has chosen a function template will any explicit specializations be considered.”
— High Integrity C++ Coding Standard, 2013

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

37

cpponsea.uk @cpponsea



C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - Walter E. Brown

Advice

- ① Avoid explicit specialization of function templates as a customization technique.
- ② If you need to customize function template `f` for type `M`:
 - Prefer to overload it ...
 - Via an ordinary function `f` taking parameter(s) of type `M`.
- ③ If you must specialize to customize:
 - Recall: class template specializations can't affect overload resolution — names of class templates can't be overloaded!
 - So, invent a helper class template `H` with an `operator()` ...
 - Customize `operator()` in an explicit specialization `H<M>`, ...
 - Then have `f<T>` forward (instantiate & call) to a `function object` of type `H<T>`: e.g., `H<T>{ }(...)`.

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

38

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Consider `constexpr-if` in lieu of specialization (C++17)

- An ordinary `if`, but evaluated at compile-time:
 - Branch not taken becomes a `discarded statement`.
 - Even if discarded, each branch must be parseable.
- E.g., x^N using fewest mult's w/out explicit specializations:
 - template< unsigned N, class T > // N is a non-type parameter
T power(T x) {
 if constexpr(N == 0) return T(1);
 else if constexpr(N % 2 == 0)
 return power<N/2>(x * x); // O(log N) multiplications
 else // N is odd
 return x * power<N-1>(x);
}

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

39

 2019

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Review

- Recall that a `friend` declaration `injects` (introduces) the friend's name into the nearest enclosing namespace:
 - But `un/qualified name lookup` won't find that name ...
 - Unless/until there's a declaration directly in that namespace.
- Examples:
 - `class A { friend void go() {…}; };`
`go(); // error: go is a hidden friend and thus not visible here`
`void go();`
`go(); // okay: go is now visible in this scope`
 - `void go();`
`class B { friend void go() {…}; };`
`go(); // okay`

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

42

cpponsea.uk

@cpponsea

 2019

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

The image shows a video frame. On the left is a presentation slide with a dark blue background. The title 'C++ Function Templates: How Do They Really Work? - Walter E. Brown' is at the top in white. Below it is a section titled 'Function templates & friendship' in yellow. A bulleted list follows, with some items in yellow. On the right is a photograph of a man with glasses and a beard, wearing a dark suit and tie, standing and gesturing with his hands. In the bottom right corner of the slide area, there is a logo for 'C++ On Sea 2019' featuring stylized waves and the year '2019'. At the bottom of the slide, there is small text: 'Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.' and the number '43'. The bottom of the video frame has a green decorative border.

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Function templates & friendship

- A class (or class template) can grant `friendship`:
 - To any particular specialization of a function template, or ...
 - To all specializations of a function template.
- Same name visibility rules hold.
- Examples; assume `template< class T > void go() { ... } ;`
 - `class A { friend void go<int>(); }; //just go<int>`
 - `class B { template< class T > friend void go(); }; // all gos`
 - `template< class T >`
`class C { friend void go<T>(); }; // just the corresponding go`

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

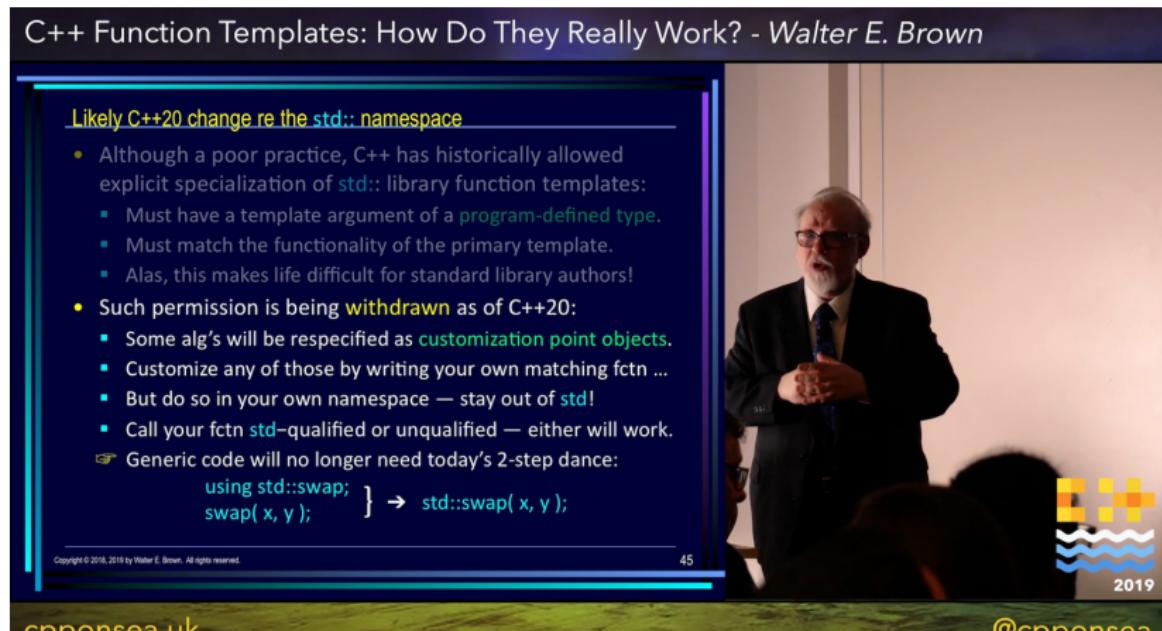
43

 2019

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>



Likely C++20 change re the `std::` namespace

- Although a poor practice, C++ has historically allowed explicit specialization of `std::` library function templates:
 - Must have a template argument of a `program-defined type`.
 - Must match the functionality of the primary template.
 - Alas, this makes life difficult for standard library authors!
- Such permission is being **withdrawn** as of C++20:
 - Some alg's will be respecified as `customization point objects`.
 - Customize any of those by writing your own matching fctn ...
 - But do so in your own namespace — stay out of `std`!
 - Call your fctn `std`-qualified or unqualified — either will work.

☞ Generic code will no longer need today's 2-step dance:

```
using std::swap; } → std::swap( x, y );
```

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

45

cpponsea.uk @cpponsea



C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Likely C++20 simpler template declaration syntax

- An `auto`-based syntax is en route for C++20:
 - `template< class T >`
`auto f(T& x);` // C++17 return type deduction
 - `auto f(auto& x);` // C++20 abbrev. function template
- Intended to work with constrained declarations, too ("*Constraint auto goes wherever auto goes*"):
 - `template< class T >`
 `requires Sortable<T>;` // Concepts TS → C++20
`auto f(T& x);`
 - `template< Sortable T >` // Stepanov → TS → C++20
`auto f(T& x);`
 - `auto f(Sortable auto& x);` // recent adjective syntax → C++20

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

46

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

There's always more to learn in C++

- `class`, `alias` (C++11), and `variable` (C++14) templates.
- Templates with several template parameters.
- Templates involving `parameter packs`, `non-type template parameters`, and/or `template template parameters`.
- Class template argument deduction and its guides (C++17).
- Template metaprogramming (see my CppCon 2014 videos).
- Constrained templates (concepts TS/C++20).
- Recommended reading:
 - Vandevoorde, Josuttis, & Gregor:
C++ Templates: The Complete Guide, 2nd Ed.
© 2017. ISBN 978-0321714121.

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

48

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

Online resources

- David Abrahams: Comment on "How to overload std::swap()."
 - stackoverflow.com/questions/11562/how-to-overload-stdswap#comment-5729583
- Herb Sutter: "Why Not Specialize Function Templates?"
 - www.gotw.ca/publications/mill17.htm
- Programming Research Ltd.: "Do not explicitly specialize a function template that is overloaded with other templates."
 - www.codingstandard.com/rule/14-2-2-do-not-explicitly-specialize-a-function-template-that-is-overloaded-with-other-templates

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

49

cpponsea.uk @cpponsea

C++ On Sea 2019 - Walter E. Brown - C++ Function Templates: How Do They Really Work?

<https://youtu.be/nfIX8yWIByY>

C++ Function Templates: How Do They Really Work? - *Walter E. Brown*

WG21 papers

- Walter E. Brown: "Thou Shalt Not Specialize std Function Templates!"
 - wg21.link/p0551
- Eric Niebler: "Suggested Design for Customization Points."
 - wg21.link/N4381
- Davis Herring & Roger Orr: "Differences Between Functions and Function Templates"
 - wg21.link/p1392

Copyright © 2018, 2019 by Walter E. Brown. All rights reserved.

50

cpponsea.uk @cpponsea

C++ On Sea 2019

RxCpp and Executors with Kirk Shoop

<http://cppcast.com/2019/03/kirk-shoop/>

- ▶ Ranges deal with objects distributed in space, Rx deals with objects distributed in time
- ▶ Executors need to be compatible with tasks and observables
- ▶ Works at Facebook with Eric Niebler and Lewis Baker on making ranges work with coroutines and executors
- ▶ Rx was developed for a garbage-collected object lifetime, but with C++ it's more complicated

Stackless vs. Stackful Coroutines

Article by Varun Ramesh, 18 August 2017

Announcing the Open Sourcing of Windows Calculator

- ▶ Blog post
- ▶ <https://github.com/Microsoft/calculator>
- ▶ https://www.reddit.com/r/programming/comments/ay2aq6/announcing_the_open_sourcing_of_windows/
- ▶ Counting Bugs in Windows Calculator

Everything You Never Wanted to Know About CMake

<https://izzys.casa/2019/02/everything-you-never-wanted-to-know-about-cmake/>

Quote

Unknown:

Weeks of coding can save you hours of planning.