# Rethinking Exceptions

## Jason Turner

- Co-host of CppCast http://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Co-creator of ChaiScript http://chaiscript.com
- Curator of http://cppbestpractices.com
- Microsoft MVP for C++ 2015-present

# Jason Turner

Independent and available for training or contracting

- http://articles.emptycrate.com/idocpp
- http://articles.emptycrate.com/training.html
- Next training: December 13-15th in Denver

## About my Talks

- Move to the front!
- Please interrupt and ask questions

## Returning Values From Functions

## Return Expression

```cpp
auto get_value()
{
  return Type{}; /// is this...
}
```

- a copy?
- a move?
- something else?

## Return Expression

```cpp
auto get_value()
{
  return Type{}; /// c++17 says this is elided
}
```

- a copy?
- a move?
- something else?

## Return Value

```cpp
auto get_value()
{
  Type obj{};
  return obj; /// Is this...
}
```

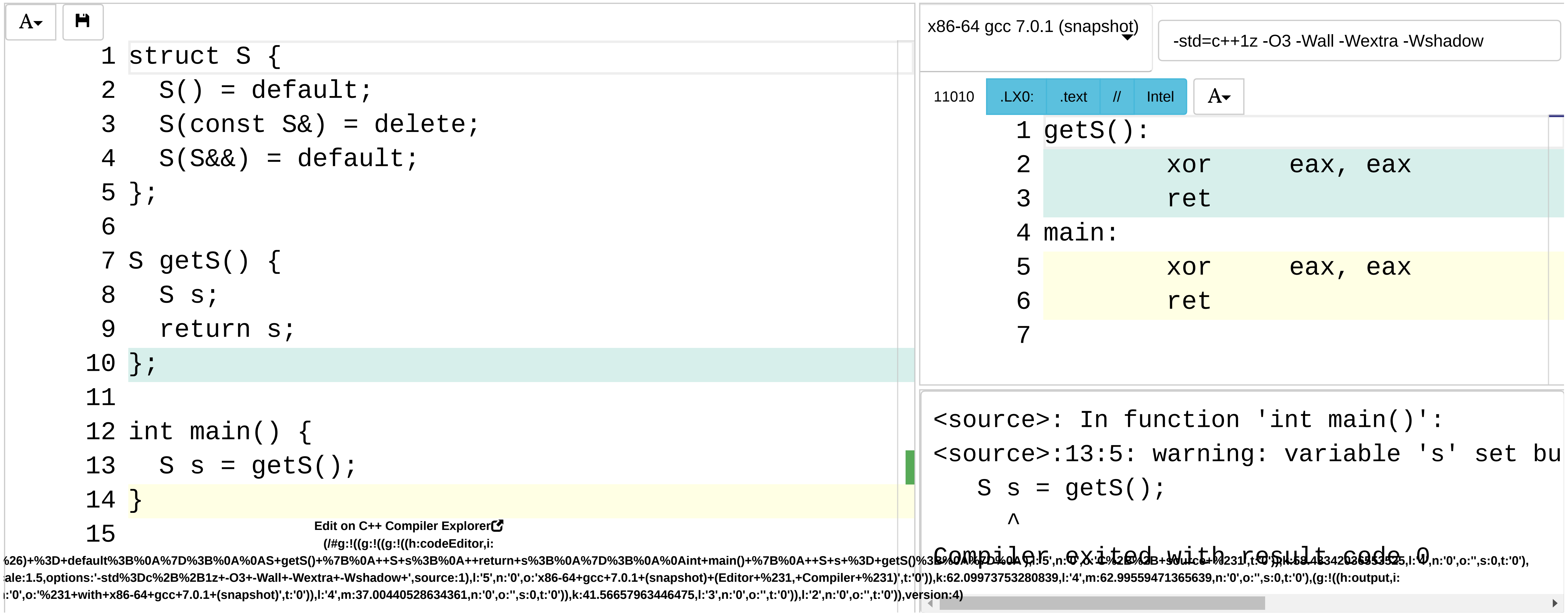
- a copy?
- a move?
- something else?

# Return Value

```cpp
auto get_value()
{
  Type obj{};
  return obj; /// Move if possible, probably elided
}
```

- a copy?
- a move?
- something else?

# Return Value



```
-std=c++1z -O3 -Wall -Wextra -Wshadow
```

```cpp
struct S {
  S() = default;
  S(const S&) = delete;
  S(S&&) = default;
};

S getS() {
  S s;
  return s;
};

int main() {
  S s = getS();
}

```

```
getS():
        xor     eax, eax
        ret
main:
        xor     eax, eax
        ret

```

```
<source>: In function 'int main()':
<source>:13:5: warning: variable 's' set bu
   S s = getS();
     ^
Compiler exited with result code 0
```

Edit on C++ Compiler Explorer
(/#g:!((g:!((g:!((h:codeEditor,i:
%26)+%3D+default%3B%0A%7D%3B%0A%0AS+getS()+%7B%0A++S+s%3B%0A++return+s%3B%0A%7D%3B%0A%0Aint+main()+%7B%0A++S+s+%3D+getS()%3B%0A%7D%0A'),l:'5',n:'0',o:'C%2B%2B+source+%231',t:'0')),k:58.43342036553525,l:'4',n:'0',o:'',s:0,t:'0'),
ale:1.5,options:'-std%3Dc%2B%2B1z+-O3+-Wall+-Wextra+-Wshadow+',source:1),l:'5',n:'0',o:'x86-64+gcc+7.0.1+(snapshot)+(Editor+%231,+Compiler+%231)',t:'0')),k:62.09973753280839,l:'4',m:62.99559471365639,n:'0',o:'',s:0,t:'0'),(g:!((h:output,i:
:'0',o:'%231+with+x86-64+gcc+7.0.1+(snapshot)',t:'0')),l:'4',m:37.00440528634361,n:'0',o:'',s:0,t:'0')),k:41.56657963446475,l:'3',n:'0',o:'',t:'0')),l:'2',n:'0',o:'',t:'0')),version:4)

## Return `optional`

```cpp
auto get_value()
{
  return std::optional<Type>{Type{}}; // is this...
}
```

- a copy?
- a move?
- something else?

## Return `optional`

```cpp
auto get_value()
{
  /// the optional is "something else"
  /// the contained `Type{}` is move constructed
  return std::optional<Type>{Type{}};
}
```

- a copy?
- a move?
- something else?

## Return `optional`

```cpp
auto get_value()
{
  /// is this...?
  return std::optional<Type>{std::in_place_t{}};
}
```

- a copy?
- a move?
- something else?

## Return `optional`

```cpp
auto get_value()
{
  /// No moves or copies at all.
  /// `std::in_place_t{}` constructs the `Type{}` in place
  /// then the return is elided
  return std::optional<Type>{std::in_place_t{}};
}
```

- a copy?
- a move?
- something else?

## Return with possible throw

```cpp
auto get_value()
{
  if (!somethinghappened) {
    return Type{}; /// is this...?
  } else {
    throw std::runtime_error("something happened");
  }
}
```

- a copy?
- a move?
- something else?

# Return with possible throw

```cpp
auto get_value()
{
  if (!somethinghappened) {
    return Type{}; /// Same rules apply, throw doesn't affect it
  } else {
    throw std::runtime_error("something happened");
  }
}
```

- a copy?
- a move?
- something else?

## Error Handling Possibilities

## Return Value Code

```cpp
int do_something();

void my_func() {
  auto error_code = do_something();
  if(error_code == 0) {
    // no error
  } else {
    // handle error
  }
}
```

- easily ignored
- interrupts normal logic flow

## Return Value Code C++17 Style

```cpp
[[nodiscard]] int do_something();

void my_func() {
  if(auto error_code = do_something(); error_code == 0) {
    // no error
  } else {
    // handle error
  }
}
```

- harder to ignore
- interrupts normal logic flow

## Global Values

```cpp
do_something();

if (errno != 0) {
  // an error occurred
}
```

- even more easily ignored
- same issues as return codes
- before C++11, `errno` was not thread safe

## `std::optional<>` Return Value

```cpp
if (auto result = do_something();
      result) {
  // no error, use result.value()
} else {
  // handle error
}
```

- cannot be ignored (throws exception on invalid access)
- harder to reason about lifetime
  - `&&` qualified members should make this efficient
  - but copy/move elision cannot work in the same way
  - (store and forward of data)
- Cannot carry error description, attempts to have some sort of error code leads to thread race condition issues again, like `errno`

# The future: `std::expected<>`?

```cpp
if (auto result = do_something();
    result) {
  // no error, use result
} else {
  // do something with contained error
}
```

- Cannot be ignored
- Same lifetime issues as `std::optional<>`
- Can carry exception / error description

## Who

- Compiles with exceptions disabled and why?

## Who

- Strongly avoids all exceptions and why?

## Why Not Exceptions?

- Can bloat binary sizes, which affects cache utilization
- Handling exceptions can be much slower (~60x) than not
- Exceptions are even slower when executing inside a debugger that reports on them
- The compiler cannot elide or optimize out exceptions

# Why Exceptions?

- Thread safe with no additional work
- Carry context and additional information
- Work perfectly with RVO/NRVO and copy and move elision
- Normal, non-exception cases have fewer conditionals and more straightforward logic
- Fewer conditionals are more optimizable and more readable
- Impossible to ignore

## Reconsidering Exceptions

- My goal today is to convince you to reconsider exceptions.
- They are clearly not the best solution for every case, but can be used wisely and effectively.

## Enter `noexcept`: A Specifier, An Operator, and A Promise

## Oh, and a note about user's groups

## The `noexcept` Specifier

## `noexcept` Specifications

```cpp
void func() {} // may throw
```

```cpp
void func() noexcept {} // may not throw
```

```cpp
void func() noexcept(constant-expr) {} // may throw if expr false
```

`noexcept` specifications are used to tell the compiler if a function is allowed to throw an exception or not. If a `noexcept(true)` function does throw, the program is required to call `terminate`.

## `noexcept` Lambda Specifications

```
auto func = []() {} // may throw
```

```
auto func = []() noexcept {} // may not throw
```

```
auto func = []() noexcept(constant-expr) {} // may throw if expr false
```

`noexcept` specifications also apply to lambdas.

## The `noexcept` Operator

## `noexcept` operator

```cpp
noexcept(expression); // returns true if expression cannot throw
```

`noexcept` is a unary operator [expr.unary.noexcept] that returns a constant expression bool. `true` if the expression cannot throw, `false` if it might.

```cpp
void func1();
void func2() noexcept;
```

```cpp
constexpr bool b1 = noexcept(3+4); // true
constexpr bool b2 = noexcept(func1()); // false
constexpr bool b3 = noexcept(func2()); // true
```

## The `noexcept` Promise

## `noexcept`

[except.spec]

*Whenever an exception is thrown and the search for a handler (18.3) encounters the outermost block of a function with a non-throwing exception specification, the function `std::terminate()` is called (18.5.1). [Note: An implementation shall not reject an expression [...] because, [...], it throws or might throw an exception [...] ] [Example:*

```cpp
extern void f(); // potentially-throwing
void g() noexcept {
  f(); // valid, even if f throws
  throw 42; // valid, effectively a call to std::terminate
}
```

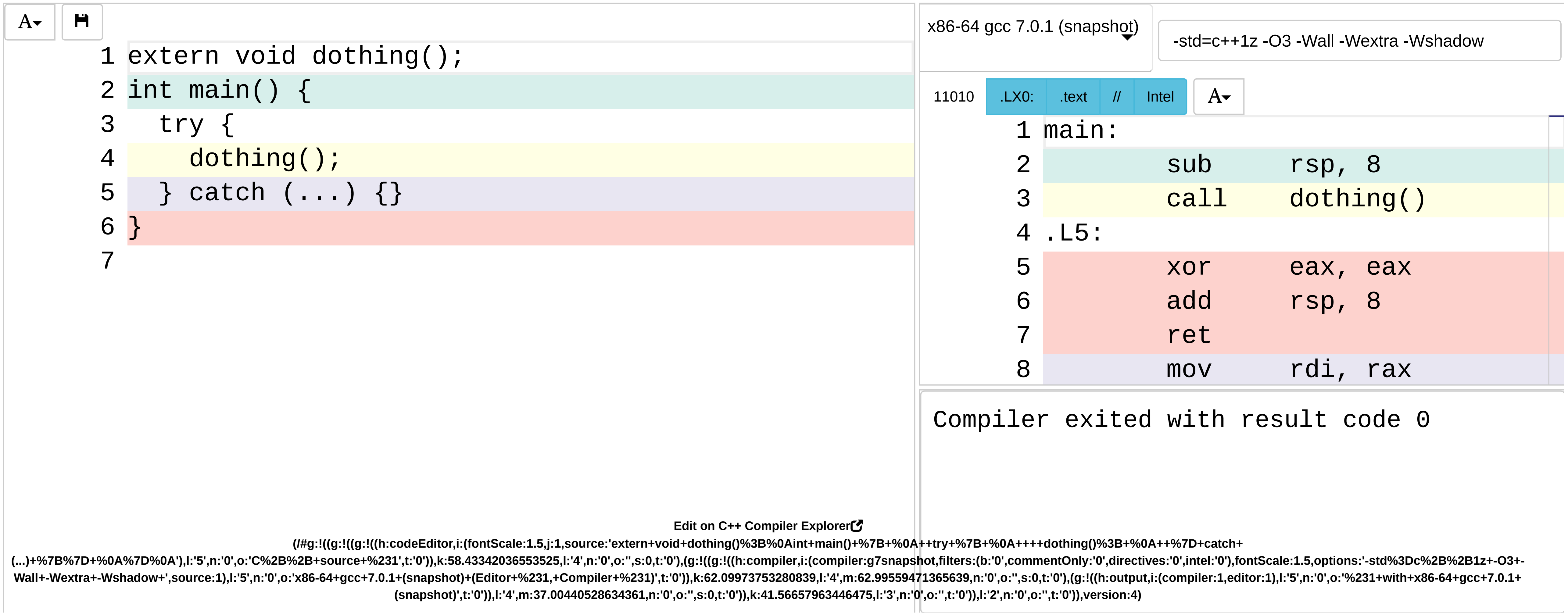
## `std::terminate`

[except.terminate] (When an exception cannot be handled...)

> *In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before `std::terminate()` is called. In the situation where the search for a handler (18.3) encounters the outermost block of a function with a non-throwing exception specification (18.4), it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before `std::terminate()` is called. In all other situations, the stack shall not be unwound before `std::terminate()` is called. An implementation is not permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to `std::terminate()`.*

## How is the Promise Enforced?

# Basic Exception Handling



```cpp
extern void dothing();
int main() {
  try {
    dothing();
  } catch (...) {}
}

```

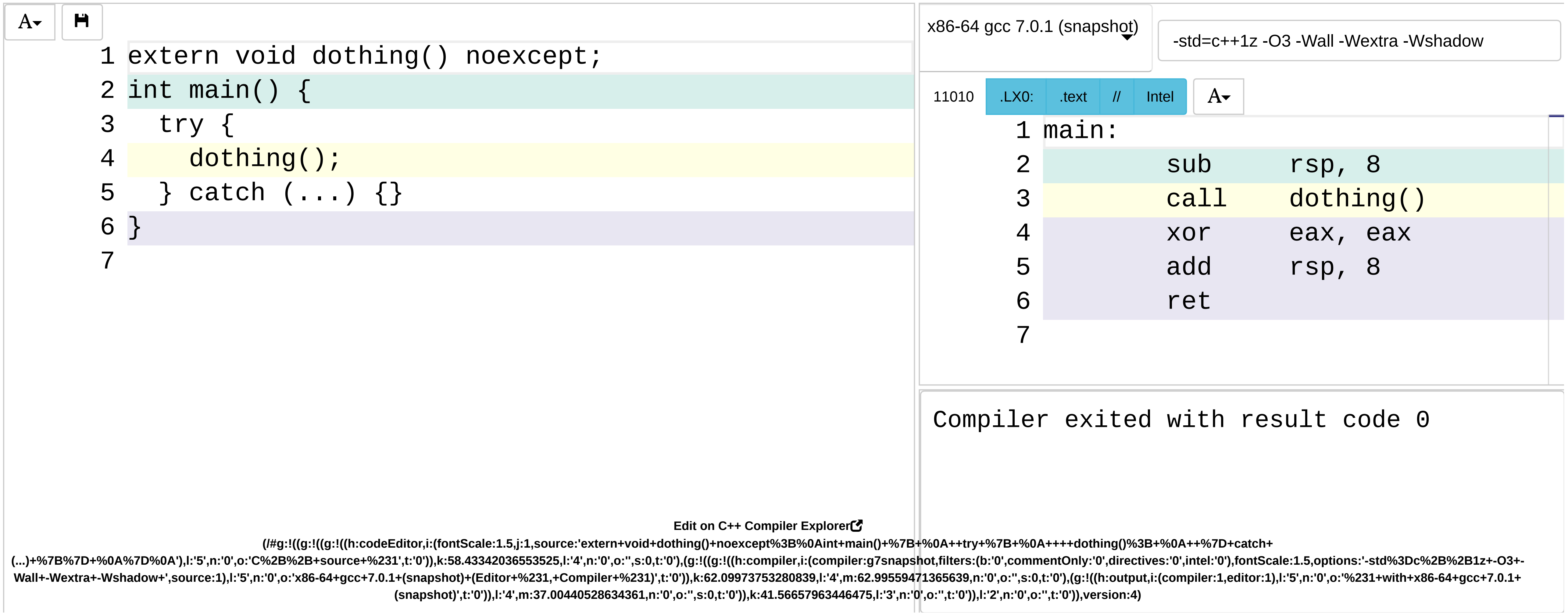
```
-std=c++1z -O3 -Wall -Wextra -Wshadow
```

```asm
main:
        sub     rsp, 8
        call    dothing()
.L5:
        xor     eax, eax
        add     rsp, 8
        ret
        mov     rdi, rax
```

```
Compiler exited with result code 0
```

Edit on C++ Compiler Explorer
(/#g:!((g:!((g:!((h:codeEditor,i:(fontScale:1.5,j:1,source:'extern+void+dothing()%3B%0Aint+main()+%7B+%0A++try+%7B+%0A++++dothing()%3B+%0A++%7D+catch+(...)+%7B%7D+%0A%7D%0A'),l:'5',n:'0',o:'C%2B%2B+source+%231',t:'0')),k:58.43342036553525,l:'4',n:'0',o:'',s:0,t:'0'),(g:!((g:!((h:compiler,i:(compiler:g7snapshot,filters:(b:'0',commentOnly:'0',directives:'0',intel:'0'),fontScale:1.5,options:'-std%3Dc%2B%2B1z+-O3+-Wall+-Wextra+-Wshadow+',source:1),l:'5',n:'0',o:'x86-64+gcc+7.0.1+(snapshot)+(Editor+%231,+Compiler+%231)',t:'0')),k:62.09973753280839,l:'4',m:62.99559471365639,n:'0',o:'',s:0,t:'0'),(g:!((h:output,i:(compiler:1,editor:1),l:'5',n:'0',o:'%231+with+x86-64+gcc+7.0.1+(snapshot)',t:'0')),l:'4',m:37.00440528634361,n:'0',o:'',s:0,t:'0')),k:41.56657963446475,l:'3',n:'0',o:'',t:'0')),l:'2',n:'0',o:'',t:'0')),version:4)

# Not Handling The Exception



```
extern void dothing() noexcept;
int main() {
  try {
    dothing();
  } catch (...) {}
}
```

-std=c++1z -O3 -Wall -Wextra -Wshadow

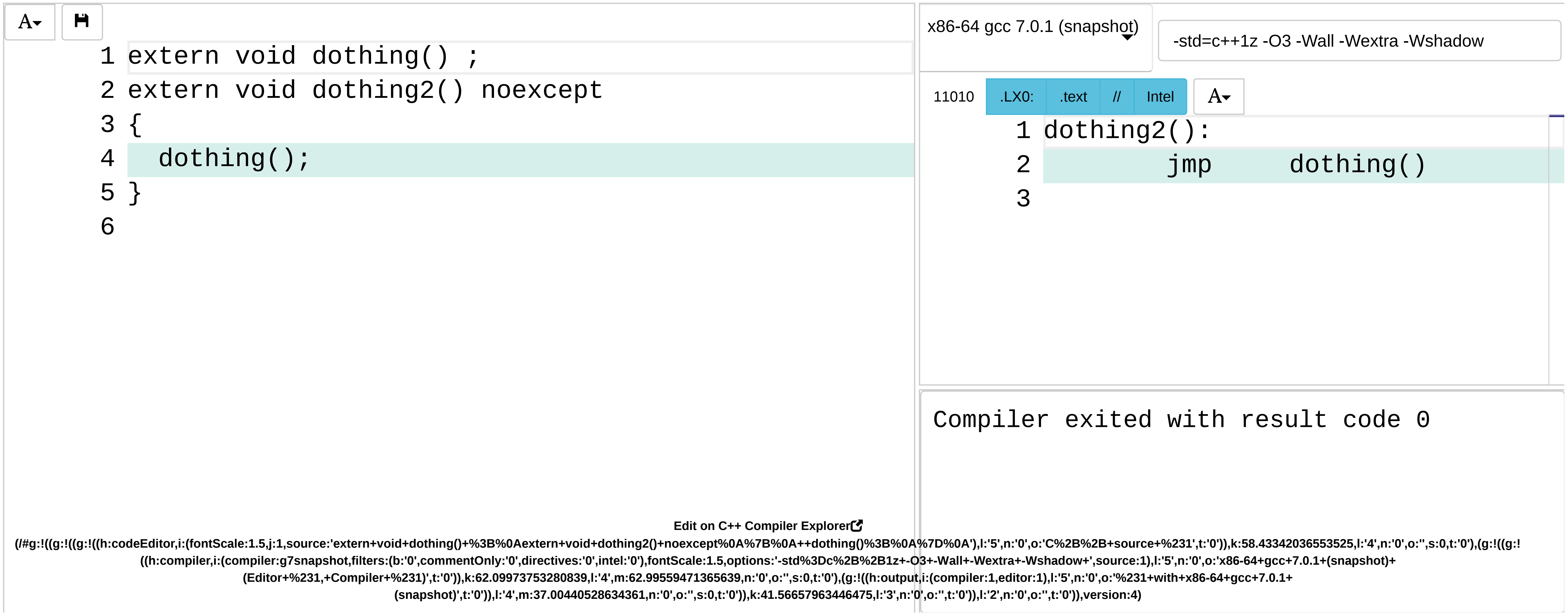
```
main:
        sub     rsp, 8
        call    dothing()
        xor     eax, eax
        add     rsp, 8
        ret
```

```
Compiler exited with result code 0
```

Edit on C++ Compiler Explorer

(/#g:!((g:!((g:!((h:codeEditor,i:(fontScale:1.5,j:1,source:'extern+void+dothing()+noexcept%3B%0Aint+main()+%7B+%0A++try+%7B+%0A++++dothing()%3B+%0A++%7D+catch+(...)+%7B%7D+%0A%7D%0A'),l:'5',n:'0',o:'C%2B%2B+source+%231',t:'0')),k:58.43342036553525,l:'4',n:'0',o:'',s:0,t:'0'),(g:!((g:!((h:compiler,i:(compiler:g7snapshot,filters:(b:'0',commentOnly:'0',directives:'0',intel:'0'),fontScale:1.5,options:'-std%3Dc%2B%2B1z+-O3+-Wall+-Wextra+-Wshadow+',source:1),l:'5',n:'0',o:'x86-64+gcc+7.0.1+(snapshot)+(Editor+%231,+Compiler+%231)',t:'0')),k:62.09973753280839,l:'4',m:62.99559471365639,n:'0',o:'',s:0,t:'0'),(g:!((h:output,i:(compiler:1,editor:1),l:'5',n:'0',o:'%231+with+x86-64+gcc+7.0.1+(snapshot)',t:'0')),l:'4',m:37.00440528634361,n:'0',o:'',s:0,t:'0')),k:41.56657963446475,l:'3',n:'0',o:'',t:'0')),l:'2',n:'0',o:'',t:'0')),version:4)

# Explicitly Calling Terminate



```cpp
extern void dothing() ;
extern void dothing2() noexcept
{
  dothing();
}

```

x86-64 gcc 7.0.1 (snapshot)

-std=c++1z -O3 -Wall -Wextra -Wshadow

```
dothing2():
        jmp     dothing()

```

```
Compiler exited with result code 0
```

Edit on C++ Compiler Explorer
(/#g:!((g:!((g:!((h:codeEditor,i:(fontScale:1.5,j:1,source:'extern+void+dothing()+%3B%0Aextern+void+dothing2()+noexcept%0A%7B%0A++dothing()%3B%0A%7D%0A'),l:'5',n:'0',o:'C%2B%2B+source+%231',t:'0')),k:58.43342036553525,l:'4',n:'0',o:'',s:0,t:'0'),(g:!((g:!((h:compiler,i:(compiler:g7snapshot,filters:(b:'0',commentOnly:'0',directives:'0',intel:'0'),fontScale:1.5,options:'-std%3Dc%2B%2B1z+-O3+-Wall+-Wextra+-Wshadow+',source:1),l:'5',n:'0',o:'x86-64+gcc+7.0.1+(snapshot)+(Editor+%231,+Compiler+%231)',t:'0')),k:62.09973753280839,l:'4',m:62.99559471365639,n:'0',o:'',s:0,t:'0'),(g:!((h:output,i:(compiler:1,editor:1),l:'5',n:'0',o:'%231+with+x86-64+gcc+7.0.1+(snapshot)',t:'0')),l:'4',m:37.00440528634361,n:'0',o:'',s:0,t:'0')),k:41.56657963446475,l:'3',n:'0',o:'',t:'0')),l:'2',n:'0',o:'',t:'0')),version:4)

## Enforcing `noexcept`

Can result in larger code:

- explicitly calling terminate when exceptions thrown (or may be) from `noexcept`

Can result in smaller code:

- Removing exception handlers to force terminate to be called when the `noexcept` calls can be traced by the compiler

# But Why `try` If You Can't Fail?

Simply put: if we know something cannot throw, why would we ever put a try block around it that the compiler could then remove?

## But Why `try` If You Can't Fail?

```cpp
template<typename Callable>
bool check_condition(Callable &&callable) {


  return callable();
}
```

## But Why `try` If You Can't Fail?

```cpp
template<typename Callable>
bool check_condition(Callable &&callable) {
  /// What if Callable is a std::function?

  return callable();
}
```

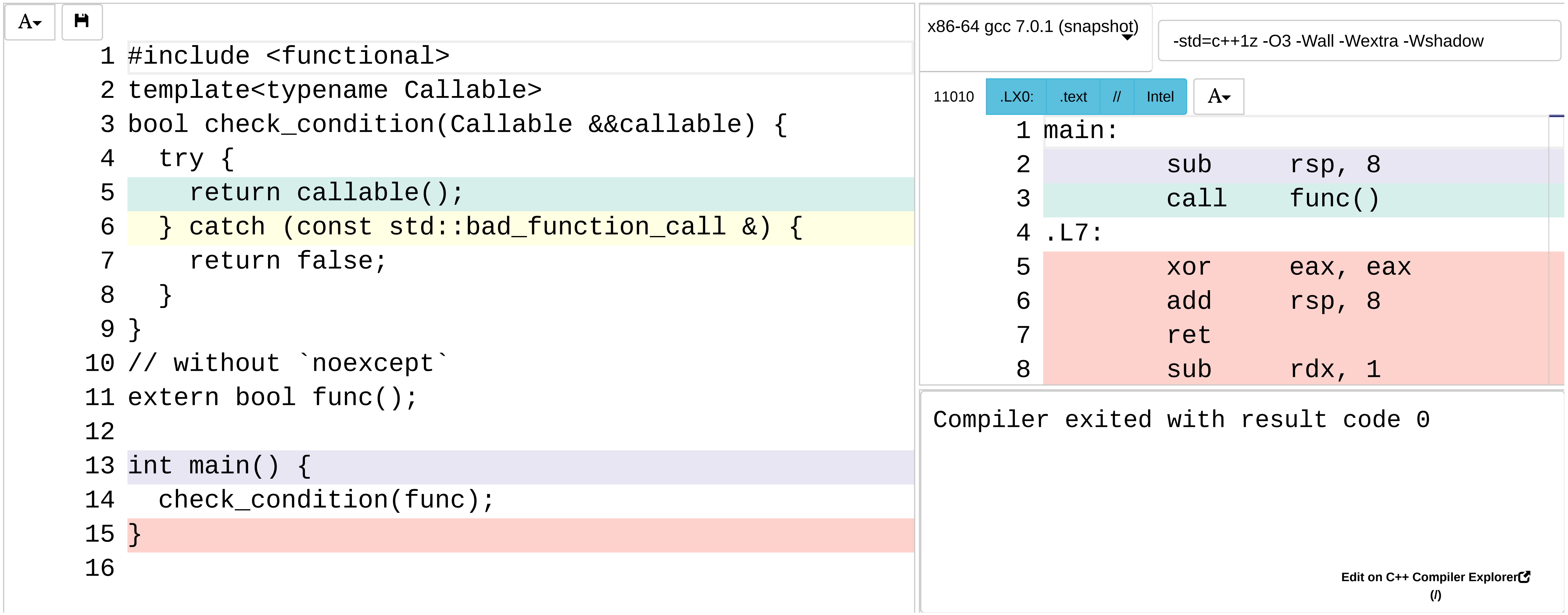
## But Why `try` If You Can't Fail?

```
template<typename Callable>
bool check_condition(Callable &&callable) {
  /// What if Callable is a std::function?
  /// that is empty?
  return callable();
}
```

# But Why `try` If You Can't Fail?

```cpp
template<typename Callable>
bool check_condition(Callable &&callable) {
  try {
    return callable();
  } catch (const std::bad_function_call &) {
    return false;
  }
}
```

# But Why `try` If You Can't Fail?



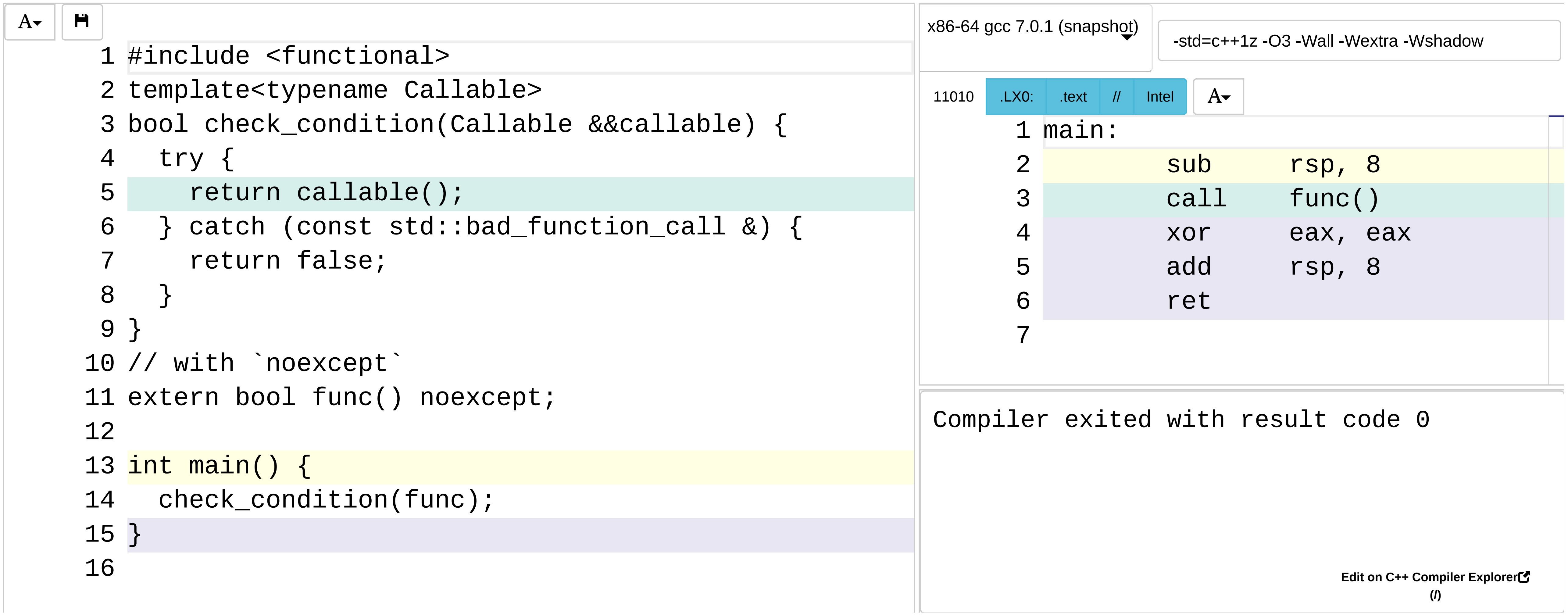
```cpp
#include <functional>
template<typename Callable>
bool check_condition(Callable &&callable) {
  try {
    return callable();
  } catch (const std::bad_function_call &) {
    return false;
  }
}
// without `noexcept`
extern bool func();

int main() {
  check_condition(func);
}

```

```
-std=c++1z -O3 -Wall -Wextra -Wshadow
```

```
main:
        sub     rsp, 8
        call    func()
.L7:
        xor     eax, eax
        add     rsp, 8
        ret
        sub     rdx, 1
```

```
Compiler exited with result code 0
```

# `noexcept` pay for what you use



```
#include <functional>
template<typename Callable>
bool check_condition(Callable &&callable) {
  try {
    return callable();
  } catch (const std::bad_function_call &) {
    return false;
  }
}
// with `noexcept`
extern bool func() noexcept;

int main() {
  check_condition(func);
}
```

-std=c++1z -O3 -Wall -Wextra -Wshadow

```
main:
        sub     rsp, 8
        call    func()
        xor     eax, eax
        add     rsp, 8
        ret
```

```
Compiler exited with result code 0
```

## Application To ChaiScript

## ChaiScript

- Embedded scripting language co-designed by me specifically for C++
- Supports Visual Studio, clang, GCC
- Runs on Windows, Linux, MacOS, FreeBSD, Haiku
- Currently requires C++14
- Header only - no external deps
- Designed for integration with C++
- All types are the same and directly shared between script and C++ (`double`, `std::string`, `std::function<>`, etc)

## ChaiScript

- My proving ground for testing best practices.
- About 25k lines of C++
- Has evolved from C++03 + Boost -> C++11 -> C++14 and onward to C++17
- Complex template usage for automatic function type deduction

## ChaiScript

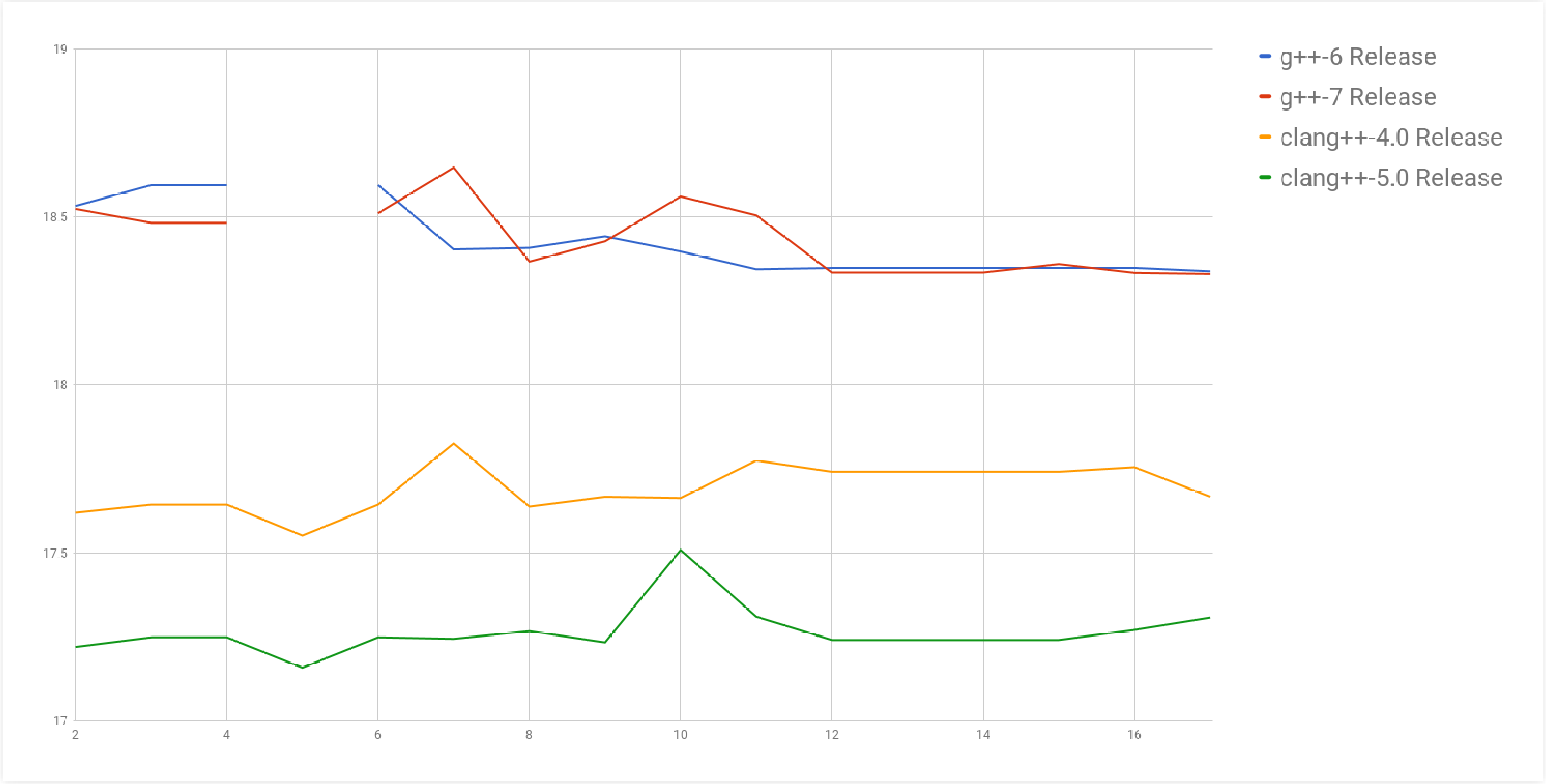
Full example:

```cpp
#include <chaiscript/chaiscript.hpp>

std::string greet(const std::string &t_name) {
  return "Hello " + t_name;
}

int main() {
  ChaiScript::chaiscript chai;
  chai.add(chaiscript::fun(&greet), "greet");
  chai.eval(R"(print(greet("Jason")))");
}
```

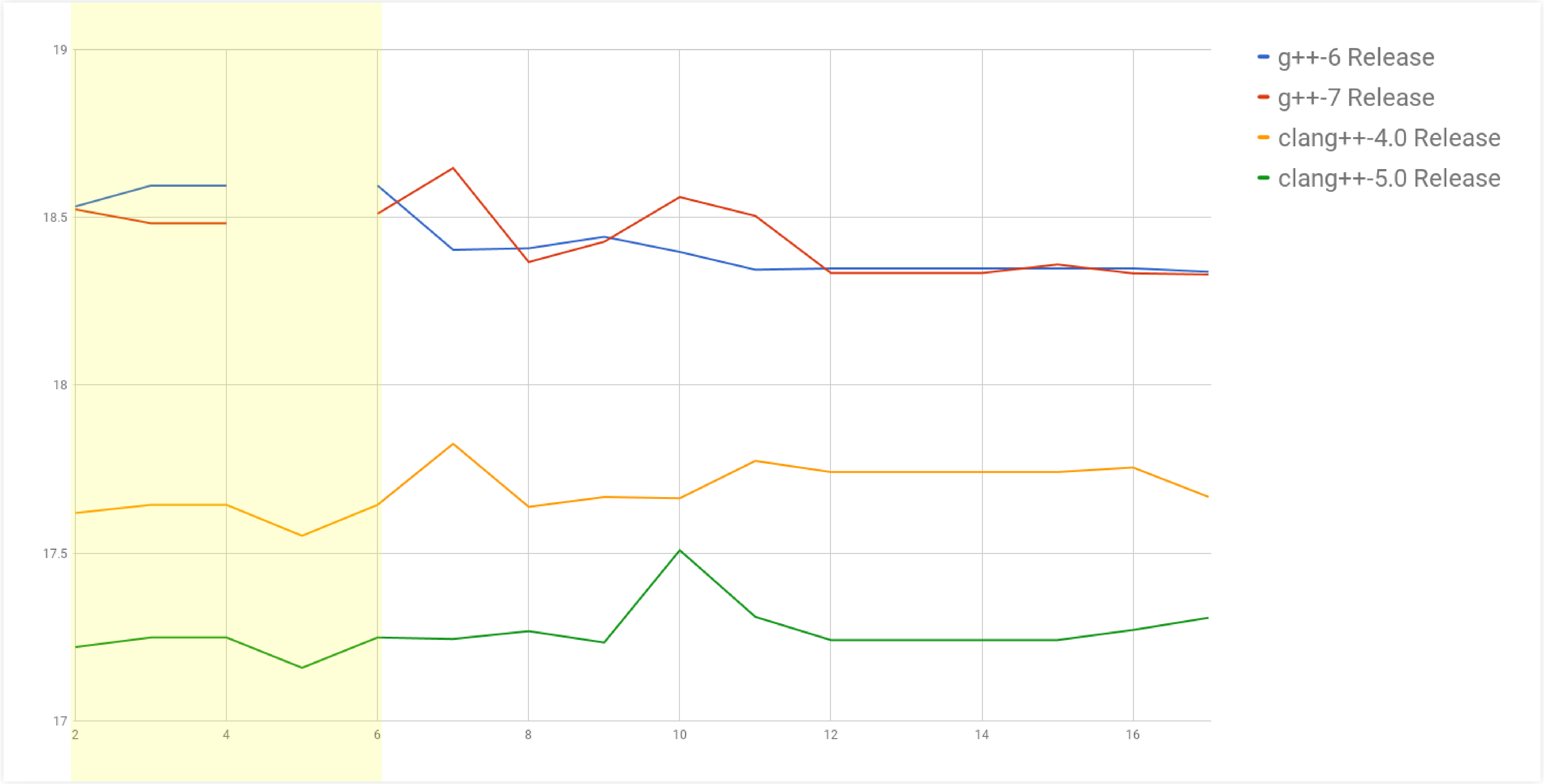
## Applying `noexcept` in ChaiScript

Using exceptions while paying the least cost, and only when we need to means deploying `noexcept` through the code base.

# Applying `noexcept` in ChaiScript



# Basic Application of `noexcept`



## Code That Should Be `noexcept`

Stubs for locks when threading is disabled:

```cpp
template<typename T>
class unique_lock
{
  public:
    explicit unique_lock(T &) {}
    void lock() {}
    void unlock() {}
};
```

Does everyone agree that these member functions can never throw an exception?

# Code That Should Be `noexcept`

Becomes:

```cpp
template<typename T>
class unique_lock
{
  public:
    explicit unique_lock(T &) noexcept {}
    void lock() noexcept {}
    void unlock() noexcept {}
};
```

## Code That Should Be `noexcept`

Simple operations that are obviously `noexcept`, such as:

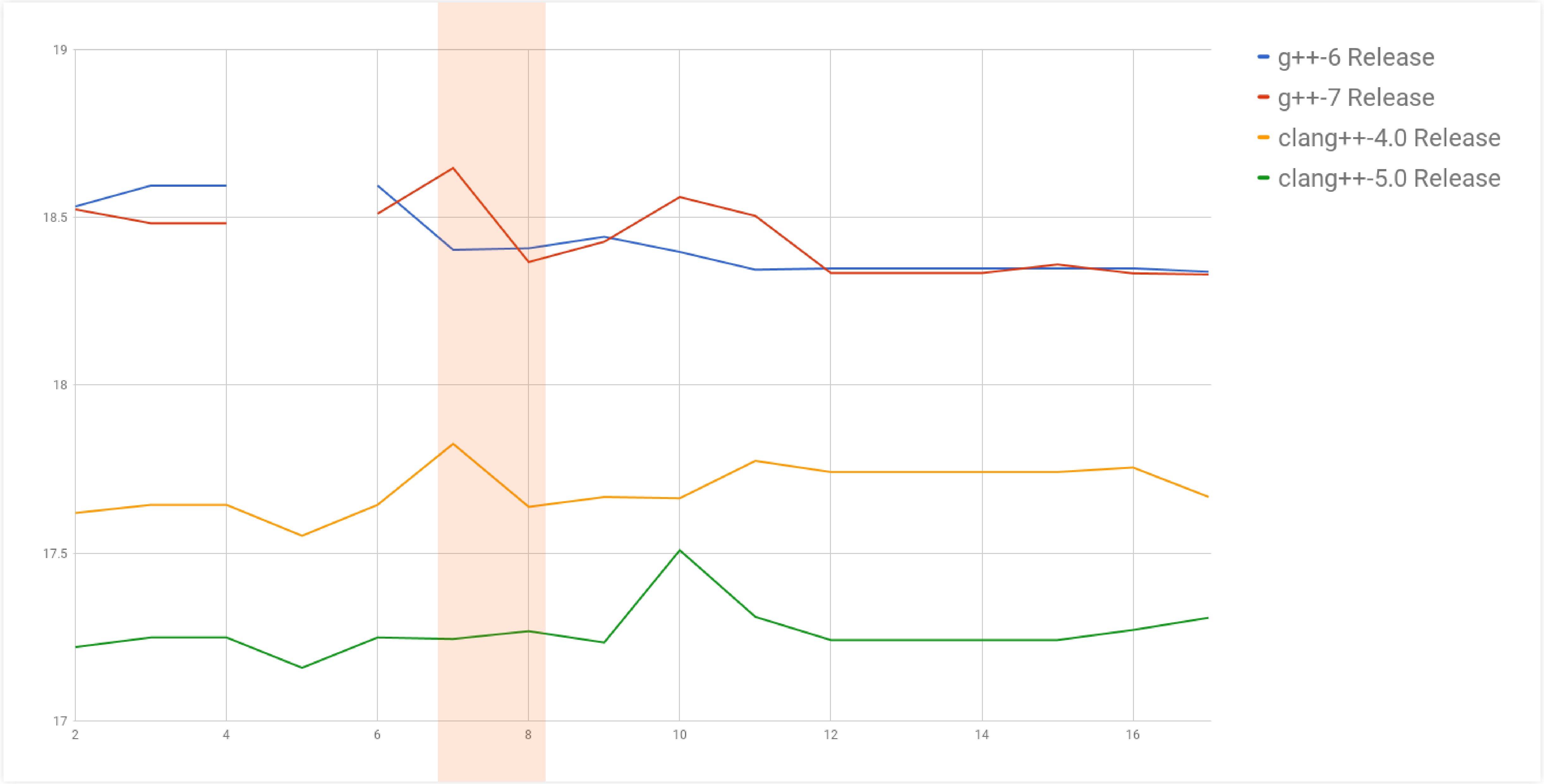
```cpp
static bool type_match(const Boxed_Value &l,
                       const Boxed_Value &r)
{
  return l.get_type_info() == r.get_type_info();
}
```

## Code That Should Be `noexcept`

Becomes:

```
static bool type_match(const Boxed_Value &l,
                       const Boxed_Value &r) noexcept
{
  return l.get_type_info() == r.get_type_info();
}
```

# Attempts to enable `noexcept`



# Realizing I'm Creating Too Many Strings

```cpp
std::string name() const
{
  if (!is_undef())
  {
    /// m_type_info is of type `std::type_info *`
    return m_type_info->name();
  } else {
    return "";
  }
}
```

# Realizing I'm Creating Too Many Strings

```cpp
const char * name() const noexcept
{
  if (!is_undef())
  {
    return m_type_info->name();
  } else {
    return "";
  }
}
```

## Overcomplicated Code

Thoughts on this?

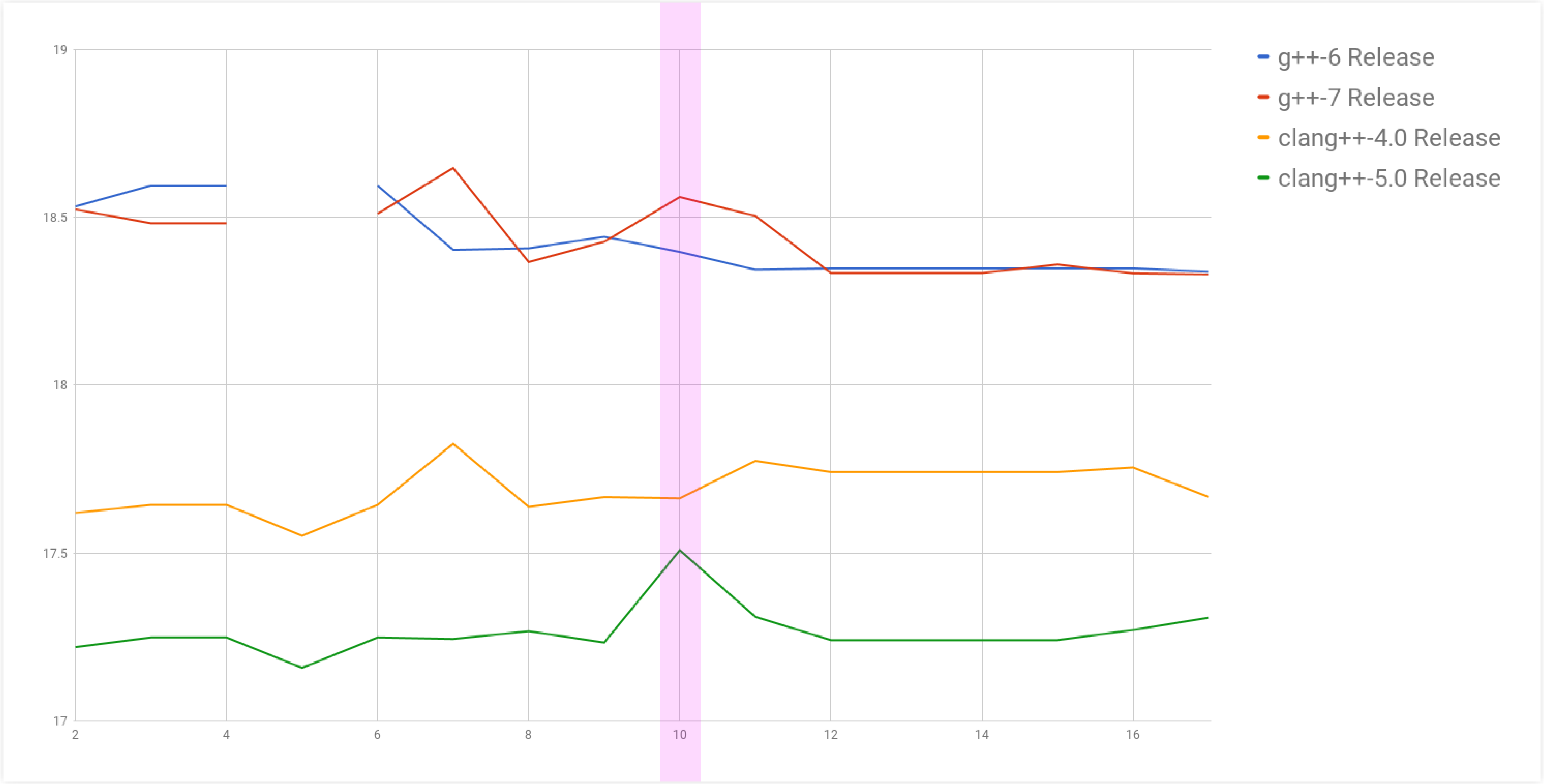
```cpp
class bad_any_cast : public std::bad_cast
{
  public:
    bad_any_cast() = default;
    bad_any_cast(const bad_any_cast &) = default;
    ~bad_any_cast() noexcept override = default;

    // \brief Description of what error occurred
    const char * what() const noexcept override {
      return m_what.c_str();
    }

  private:
    std::string m_what = "bad any cast";
};
```

## Overcomplicated Code

Easily simplified to:

```
class bad_any_cast : public std::bad_cast
{
  public:
    /// \brief Description of what error occurred
    const char * what() const noexcept override {
      return "bad any cast";
    }
};
```

# Trying Too Hard



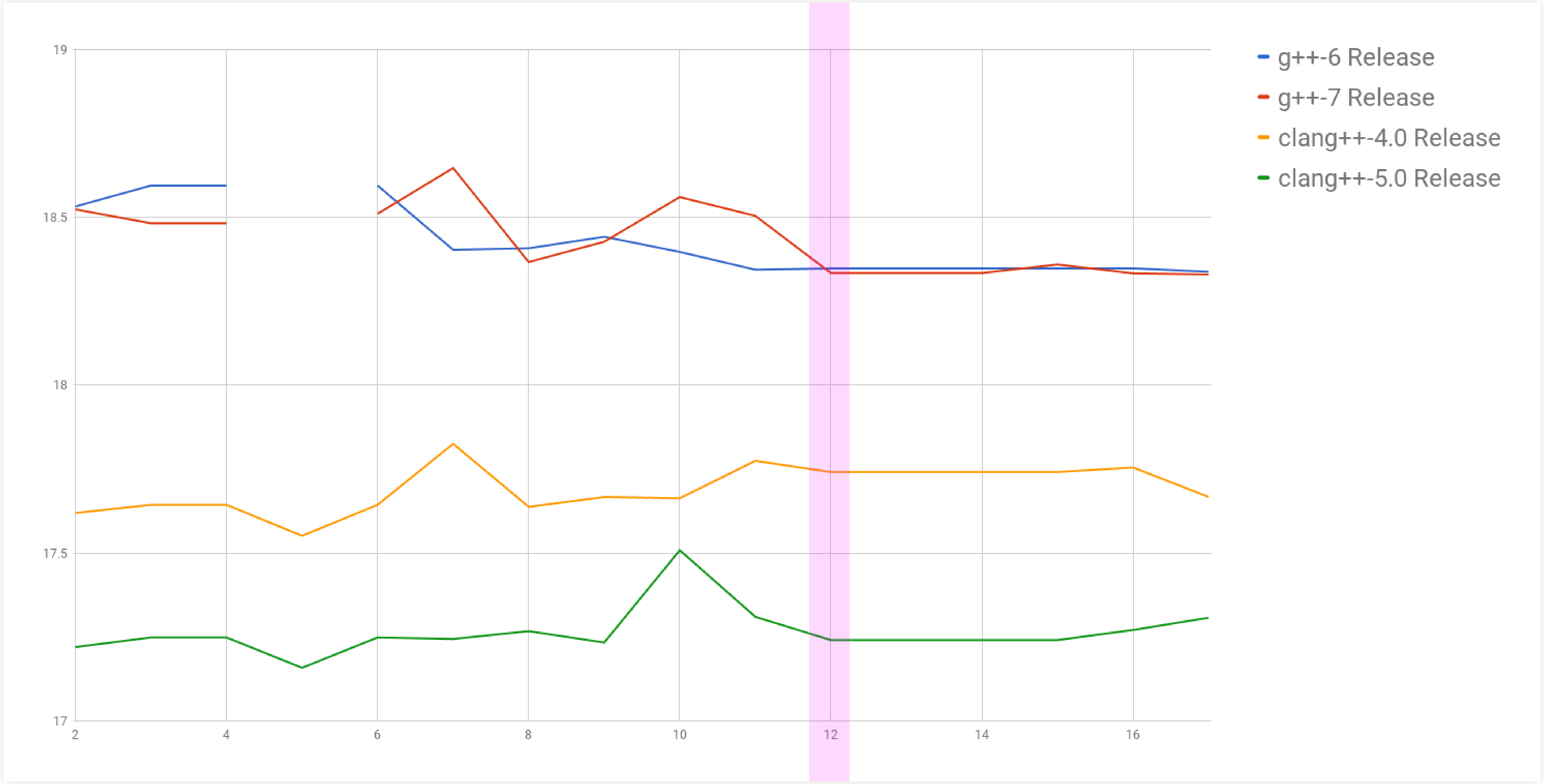
# Trying Too Hard

```cpp
static bool is_reserved_word(const std::string &name)
{
  static const std::set<std::string> m_reserved_words
    = {"def", "fun", "while", "for", "if", "else", "&&", "||", ",",
       "auto", "return", "break", "true", "false", "class", "attr",
       "var", "global", "GLOBAL", "_", "__LINE__", "__FILE__",
       "__FUNC__", "__CLASS__"};
  return m_reserved_words.count(name) > 0;
}
```

# Trying Too Hard

```cpp
static bool is_reserved_word(const std::string &name) noexcept
{
  static const char *m_reserved_words[]
    = {"def", "fun", "while", "for", "if", "else", "&&", "||", ",",
       "auto", "return", "break", "true", "false", "class", "attr",
       "var", "global", "GLOBAL", "_", "__LINE__", "__FILE__",
       "__FUNC__", "__CLASS__"};

  return std::any_of(std::begin(m_reserved_words),
                     std::end(m_reserved_words),
      [&name](const char *str){ return str == name; });
}
```

# Real Fixes Discovered



## Real Fixes Discovered

How to make this `noexcept`?

```cpp
bool is_operator(const std::string &t_s) const {
  return std::any_of(m_operator_matches.begin(),
                     m_operator_matches.end(),
    [t_s](const auto &opers) {
      return std::any_of(opers.begin(), opers.end(),
        [t_s](const utility::Static_String &s) {
          return t_s == s.c_str();
        });
      });
}
```

# Real Fixes Discovered

Many unnecessary string copies.

```
bool is_operator(const std::string &t_s) const {
  return std::any_of(m_operator_matches.begin(),
                        m_operator_matches.end(),
    [&t_s](const auto &opers) {
      return std::any_of(opers.begin(), opers.end(),
        [&t_s](const utility::Static_String &s) {
          return t_s == s.c_str();
        });
      });
}
```

## Lessons Learned

## Debugging Can Be Harder

Taking this signature:

```cpp
std::pair<bool, bool> call_match_internal(
   const std::vector<Boxed_Value> &vals,
   const Type_Conversions_State &t_conversions) const noexcept;
```

Turns out `call_match_internal` can throw.

## Debugging Can Be Harder

In a few cases this caused an "unhandled exception" error which was very difficult to locate which change I had made that caused it.

## Sometimes Specifying `noexcept` can be complex

```
noexcept(noexcept(check_divide_by_zero(u)))
```

But this really only happens in generic code.

## `noexcept` and Forwarding references

This may be a move or a copy, depending on what is passed

```
template<typename T>
struct S {
  template<typename Param>
  S(Param &&param) noexcept(noexcept(T{std::forward<Param>(param)}))
    : value(std::forward<Param>(param))
  {
  }
  T value;
};
```

## noexcept and Forwarding references

A helper to indicate the intent might be helpful to add to the standard:

```cpp
template<typename From, typename To>
struct is_nothrow_forward_constructible
 : std::bool_constant<noexcept(To{std::declval<From>()})>
{
};

template< class From, class To >
inline constexpr bool is_nothrow_forward_constructible_v
  = is_nothrow_forward_constructible<From, To>::value;
```

## noexcept and Forwarding references

Used as:

```cpp
template<typename T>
struct S {
  template<typename Param>
  S(Param &&param)
    noexcept(is_nothrow_forward_constructible_v<decltype(param), T>)
    : value{std::forward<Param>(param)}
  {
   // static_assert(std::is_const_v<T>);
  }

  T value;
};
```

## `noexcept` makes you think about your code

When you ask "can this function be `noexcept`?" you are forced to consider why or why not, and make fixes, like in my capture of `std::string` objects by copy.

## `noexcept` can help you find design flaws

If you think a function should be `noexcept`, but it cannot be, you've probably found a design flaw in your code.

## Guidelines

## Consider the contract you make

Is this a good example of using `const`?

```cpp
struct S {
  mutable int count = 0;
  void update_count() const {
    ++count;
  }
};
```

## Consider the contract you make

By marking a method `const` you are promising to the user of your library that you are not modifying your object.

You are also making an implicit promise that the method call is thread safe for multiple readers.

# Consider the contract you make

In the same way:

```cpp
struct S {
  mutable int count = 0;
  void update_count() noexcept {
    ++count;
    if (count > 100) {
      throw count;
    }
  }
};
```

This breaks the contract you have made with the user of your library by crashing in an unhandlable way.

# Consider the contract you make

In the opposite direction:

```cpp
struct S {
  int count = 0;
  void update_count() noexcept {
    ++count;
  }
};
```

You can create a library / program that compiles with no exception handling code if you properly guarantee that all your functions are `noexcept`

Consider this alternative to `-fno-exceptions`

# Don't Force `noexcept`

It's entirely possible that an exception-prone solution, with allocations, is the more efficient and best solution, don't dismiss it.

```cpp
static bool is_reserved_word(const std::string &name)
{
  static const std::set<std::string> m_reserved_words
    = {"def", "fun", "while", "for", "if", "else", "&&", "||", ",",
       "auto", "return", "break", "true", "false", "class", "attr",
       "var", "global", "GLOBAL", "_", "__LINE__", "__FILE__",
       "__FUNC__", "__CLASS__"};
  return m_reserved_words.count(name) > 0;
}
```

## If An Argument Might Throw on Copy, it's not `noexcept`

```cpp
void copy_string(std::string s)
{
  std::string my_s(std::move(s));
}
```

`std::string` is no throw move constructible, but it's impossible to guarantee that a call to this string will not throw, because of the copy required (Herb Sutter Back to The Basics CppCon 2014).

## If An Argument Might Throw on Copy, it's not `noexcept`

A generic version of this might look like:

```cpp
template<typename T>
void copy_thing(T t)
  noexcept(std::is_nothrow_move_constructible_v<T>
        && std::is_nothrow_copy_constructible_v<T>)
{
  T my_t(std::move(t));
}
```

# Code with no reasonable way to handle exception

Looking at our `set` example again:

```cpp
static bool is_reserved_word(const std::string &name) noexcept
{
  static const std::set<std::string> m_reserved_words
    = {"def", "fun", "while", "for", "if", "else", "&&", "||", ",",
       "auto", "return", "break", "true", "false", "class", "attr",
       "var", "global", "GLOBAL", "_", "__LINE__", "__FILE__",
       "__FUNC__", "__CLASS__"};
  return m_reserved_words.count(name) > 0;
}
```

Why or why not make this `noexcept`?

# Sometimes `noexcept` is really hard

```
template<typename Itr, typename Predicate>
constexpr Itr find_if(Itr begin, Itr end, Predicate predicate)
   noexcept( noexcept(begin != end)
             && noexcept(predicate(*begin))
             && noexcept(++begin)
             && std::is_nothrow_copy_constructible_v<Itr>)
{
  while (begin != end) {
    if (predicate(*begin)) { break; }
    ++begin;
  }
  return begin;
}
```

But does it matter for code like this?

## Sometimes `noexcept` is hard

If the compiler inlines the code, then it will make its own `noexcept` optimizations determinations.

# Declaring Lambdas

Remember to declare your lambdas which are used more than once (and less likely to get inlined) `noexcept` (when appropriate).

```cpp
auto lambda = []() noexcept {
  // do no-exception things
};
```

## Conclusion

## This is not `noexcept` all the things

- `noexcept` is a goal in the same way `const` is for your methods
- Critically think about your code that cannot be `noexcept` and look for simplification opportunities
- Don't use `noexcept` when it is too much of a burden to
- Consider what the standards does:
    - Use a no-throw move constructor when possible for performance (Example: `std::vector::resize`)
    - Be aware of `std::move_if_noexcept`

## `noexcept` Best Practices

- Prefer `noexcept` over `-fno-exceptions`
- Be aware that `noexcept` becomes part of the type system in C++17
- Olivier Giroux (in reference to using `par`, `seq` and `par_unseq` with parallel algorithms): "Give ... as much semantic information as possible." This is really what `noexcept` is about, providing semantic information to your users and the compiler
- `noexcept` correctness helps standard containers provide exception guarantees

## Jason Turner

- Co-host of CppCast http://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Co-creator of ChaiScript http://chaiscript.com
- Curator of http://cppbestpractices.com
- Microsoft MVP for C++ 2015-present

# Jason Turner

Independent and available for training or contracting

- http://articles.emptycrate.com/idocpp
- http://articles.emptycrate.com/training.html
- Next training: December 13-15th in Denver