Back to Basics:

Exceptions

Klaus Iglberger, CppCon 2020

klaus.iglberger@gmx.de

C++ Trainer since 2016

Author of the blaze C++ math library

(Co-)Organizer of the Munich C++ user group

Regular presenter at C++ conferences

Email: klaus.iglberger@gmx.de

Klaus Iglberger

Content

Content

Why Another Talk on Exception Safety?

Exception Handling and Exception Safety

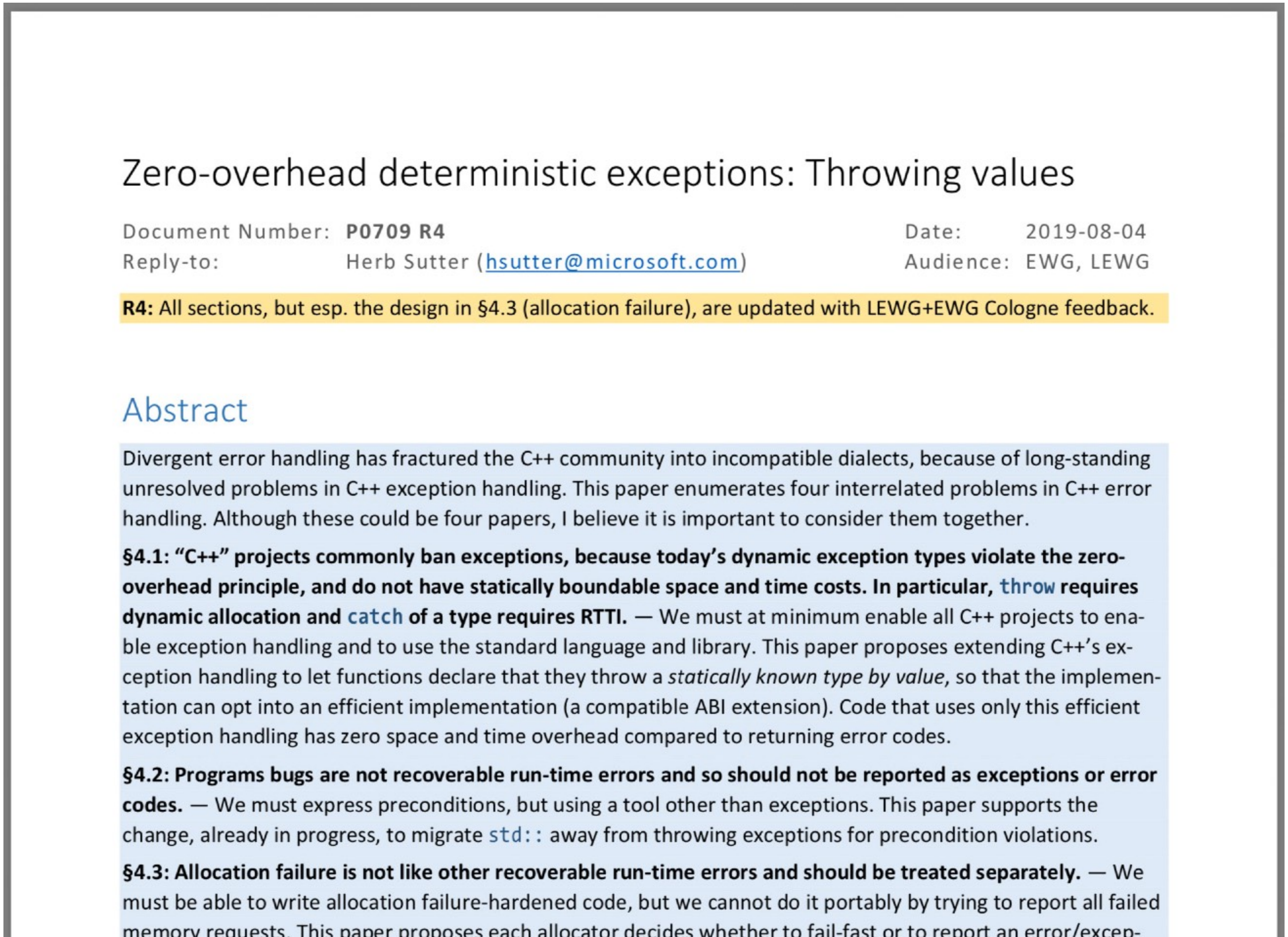
Ben Saks
Saks & Associates
www.benjaminsaks.com

Copyright © 2019 by Dan and Ben Saks

1

Cppcon
The C++ Conference
2019
cppcon.org
Ben Saks
Exception Handling and Exception Safety
Video Sponsorship Provided By:
ansatz
0:33 / 1:01:29

https://wg21.link/p0709



Zero-overhead deterministic exceptions: Throwing values

Document Number: **P0709 R4** Date: 2019-08-04
Reply-to: Herb Sutter (hsutter@microsoft.com) Audience: EWG, LEWG

R4: All sections, but esp. the design in §4.3 (allocation failure), are updated with LEWG+EWG Cologne feedback.

Abstract

Divergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems in C++ exception handling. This paper enumerates four interrelated problems in C++ error handling. Although these could be four papers, I believe it is important to consider them together.

§4.1: "C++" projects commonly ban exceptions, because today's dynamic exception types violate the zero-overhead principle, and do not have statically boundable space and time costs. In particular, `throw` requires dynamic allocation and `catch` of a type requires RTTI. — We must at minimum enable all C++ projects to enable exception handling and to use the standard language and library. This paper proposes extending C++'s exception handling to let functions declare that they throw a *statically known type by value*, so that the implementation can opt into an efficient implementation (a compatible ABI extension). Code that uses only this efficient exception handling has zero space and time overhead compared to returning error codes.

§4.2: Programs bugs are not recoverable run-time errors and so should not be reported as exceptions or error codes. — We must express preconditions, but using a tool other than exceptions. This paper supports the change, already in progress, to migrate `std::` away from throwing exceptions for precondition violations.

§4.3: Allocation failure is not like other recoverable run-time errors and should be treated separately. — We must be able to write allocation failure-hardened code, but we cannot do it portably by trying to report all failed memory requests. This paper proposes each allocator decides whether to fail-fast or to report an error/excep-

2 Why do something: Problem description, and root causes

2.1 Exceptions have not replaced error codes, and vice versa

"There are still people who argue against all use of exceptions and people who claim that exceptions should be used consistently instead of error codes." — [P0939R0]

Exceptions are the error handling model that is required by key parts of the language (for constructors and operators) and by the standard library, but are widely banned. This means that a large fraction of the C++ community is not actually using 'real' C++, but are using a language dialect, and either a nonstandard library or none at all.

Even though exceptions are required, and have been available for some 25 years, they have not replaced error codes for error handling in C++. Therefore, they never will unless they are changed in some way to address the reasons they cannot be used universally (see §2.5, "Root causes"). The community are voting with their feet:

- Major coding guidelines ban exceptions, including common modern guidelines endorsed by the world's top advocates of C++ exceptions. For example, the Google C++ Style Guide [GSG] bans exceptions. The Joint Strike Fighter Air Vehicle C++ Coding Standards (JSF++) [JSF++ 2005] was produced by a group that included Bjarne Stroustrup and is published on Stroustrup's personal website, and bans exceptions.
- Many projects ban exceptions. In [SC++F 2018], **52%** of C++ developers reported that exceptions were banned in part or all of their project code — i.e., **most** are not allowed to freely use C++'s primary recommended error handling model that is required to use the C++ standard language and library.
- Committee papers such as [P0829R2] and [P0941R0] embrace standard support for disabling exceptions.
- The C++ Core Guidelines' Guidelines Support Library [GSL] requires exceptions, and cannot be used in such projects. We are already getting requests for a nonthrowing version of GSL, which changes some of its interfaces (e.g., `narrow` reports errors by throwing `narrowing_error` and would have to change).
- Non-throwing dialects of the STL and the rest of the standard library proliferate, and C++ implementation vendors continue to receive requests to support those nonstandard dialects.
- Every C++ compiler supports a mode that disables exception handling (e.g., `-fno-exceptions`).

This is an intolerable rift: Large numbers of "C++" projects are not actually using standard C++.

But switching to error codes isn't the answer either — error codes cannot be used in constructors and operators, are ignored by default, and make it difficult to separate error handling from normal control flow.

2.2 Instead, we're actively proliferating dual interfaces that do *both*

"Filesystem library functions often provide two overloads, one that throws an exception to report file system errors, and another that sets an `error_code`*." — [N3239]*

2 **Why** do something: Problem description, and root causes

2.1 Exceptions have not replaced error codes, and vice versa

"There are still people who argue against all use of exceptions and people who claim that exceptions should be used consistently instead of error codes." — [P0939R0]

Exceptions are the error handling model that is required by key parts of the language (for constructors and operators) and by the standard library, but are widely banned. This means that a large fraction of the C++ community is not actually using 'real' C++, but are using a language dialect, and either a nonstandard library or none at all.

Even though exceptions are required, and have been available for some 25 years, they have not replaced error codes for error handling in C++. Therefore, they never will unless they are changed in some way to address the reasons they cannot be used universally (see §2.5, "Root causes"). The community are voting with their feet:

- Major coding guidelines ban exceptions, including common modern guidelines endorsed by the world's top advocates of C++ exceptions. For example, the Google C++ Style Guide [GSG] bans exceptions. The Joint Strike Fighter Air Vehicle C++ Coding Standards (JSF++) [JSF++ 2005] was produced by a group that included Bjarne Stroustrup and is published on Stroustrup's personal website, and bans exceptions.
- Many projects ban exceptions. In [SC++F 2018], **52%** of C++ developers reported that exceptions were banned in part or all of their project code — i.e., **most** are not allowed to freely use C++'s primary recommended error handling model that is required to use the C++ standard language and library.
- Committee papers such as [P0829R2] and [P0941R0] embrace standard support for disabling exceptions.
- The C++ Core Guidelines' Guidelines Support Library [GSL] requires exceptions, and cannot be used in such projects. We are already getting requests for a nonthrowing version of GSL, which changes some of its interfaces (e.g., `narrow` reports errors by throwing `narrowing_error` and would have to change).
- Non-throwing dialects of the STL and the rest of the standard library proliferate, and C++ implementation vendors continue to receive requests to support those nonstandard dialects.
- Every C++ compiler supports a mode that disables exception handling (e.g., `-fno-exceptions`).

This is an intolerable rift: Large numbers of "C++" projects are not actually using standard C++.

But switching to error codes isn't the answer either — error codes cannot be used in constructors and operators, are ignored by default, and make it difficult to separate error handling from normal control flow.

2.2 Instead, we're actively proliferating dual interfaces that do *both*

"Filesystem library functions often provide two overloads, one that throws an exception to report file system errors, and another that sets an `error_code`." — [N3239]

2 **Why** do something: Problem description, and root causes

2.1 Exceptions have not replaced error codes, and vice versa

"There are still people who argue against all use of exceptions and people who claim that exceptions should be used consistently instead of error codes." — [P0939R0]

Exceptions are the error handling model that is required by key parts of the language (for constructors and operators) and by the standard library, but are widely banned. This means that a large fraction of the C++ community is not actually using 'real' C++, but are using a language dialect, and either a nonstandard library or none at all.

Even though exceptions are required, and have been available for some 25 years, they have not replaced error codes for error handling in C++. Therefore, they never will unless they are changed in some way to address the reasons they cannot be used universally (see §2.5, "Root causes"). The community are voting with their feet:

- Major coding guidelines ban exceptions, including common modern guidelines endorsed by the world's top advocates of C++ exceptions. For example, the Google C++ Style Guide [GSG] bans exceptions. The Joint Strike Fighter Air Vehicle C++ Coding Standards (JSF++) [JSF++ 2005] was produced by a group that included Bjarne Stroustrup and is published on Stroustrup's personal website, and bans exceptions.
- Many projects ban exceptions. In [SC++F 2018], **52%** of C++ developers reported that exceptions were banned in part or all of their project code — i.e., **most** are not allowed to freely use C++'s primary recommended error handling model that is required to use the C++ standard language and library.
- Committee papers such as [P0829R2] and [P0941R0] embrace standard support for disabling exceptions.
- The C++ Core Guidelines' Guidelines Support Library [GSL] requires exceptions, and cannot be used in such projects. We are already getting requests for a nonthrowing version of GSL, which changes some of its interfaces (e.g., `narrow` reports errors by throwing `narrowing_error` and would have to change).
- Non-throwing dialects of the STL and the rest of the standard library proliferate, and C++ implementation vendors continue to receive requests to support those nonstandard dialects.
- Every C++ compiler supports a mode that disables exception handling (e.g., `-fno-exceptions`).

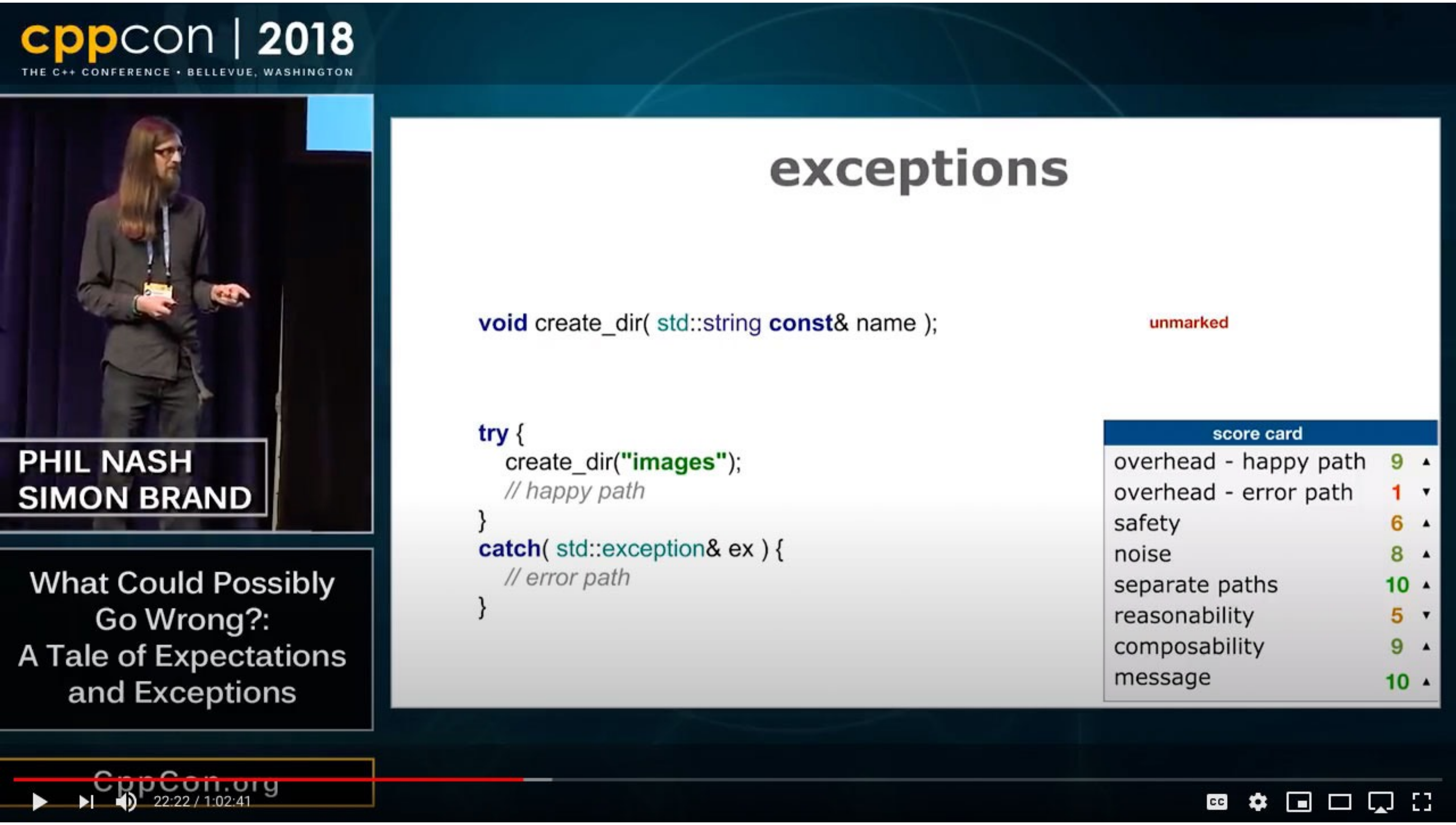
This is an intolerable rift: Large numbers of "C++" projects are not actually using standard C++.

But switching to error codes isn't the answer either — error codes cannot be used in constructors and operators, are ignored by default, and make it difficult to separate error handling from normal control flow.

2.2 Instead, we're actively proliferating dual interfaces that do *both*

"Filesystem library functions often provide two overloads, one that throws an exception to report file system errors, and another that sets an `error_code`." — [N3239]

Rating Exceptions


cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
PHIL NASH
SIMON BRAND
What Could Possibly Go Wrong?: A Tale of Expectations and Exceptions
CppCon.org
22:22 / 1:02:41

exceptions

```
void create_dir( std::string const& name );
```

unmarked

```
try {
    create_dir("images");
    // happy path
}
catch( std::exception& ex ) {
    // error path
}
```

score card		
overhead - happy path	9	▲
overhead - error path	1	▼
safety	6	▲
noise	8	▲
separate paths	10	▲
reasonability	5	▼
composability	9	▲
message	10	▲

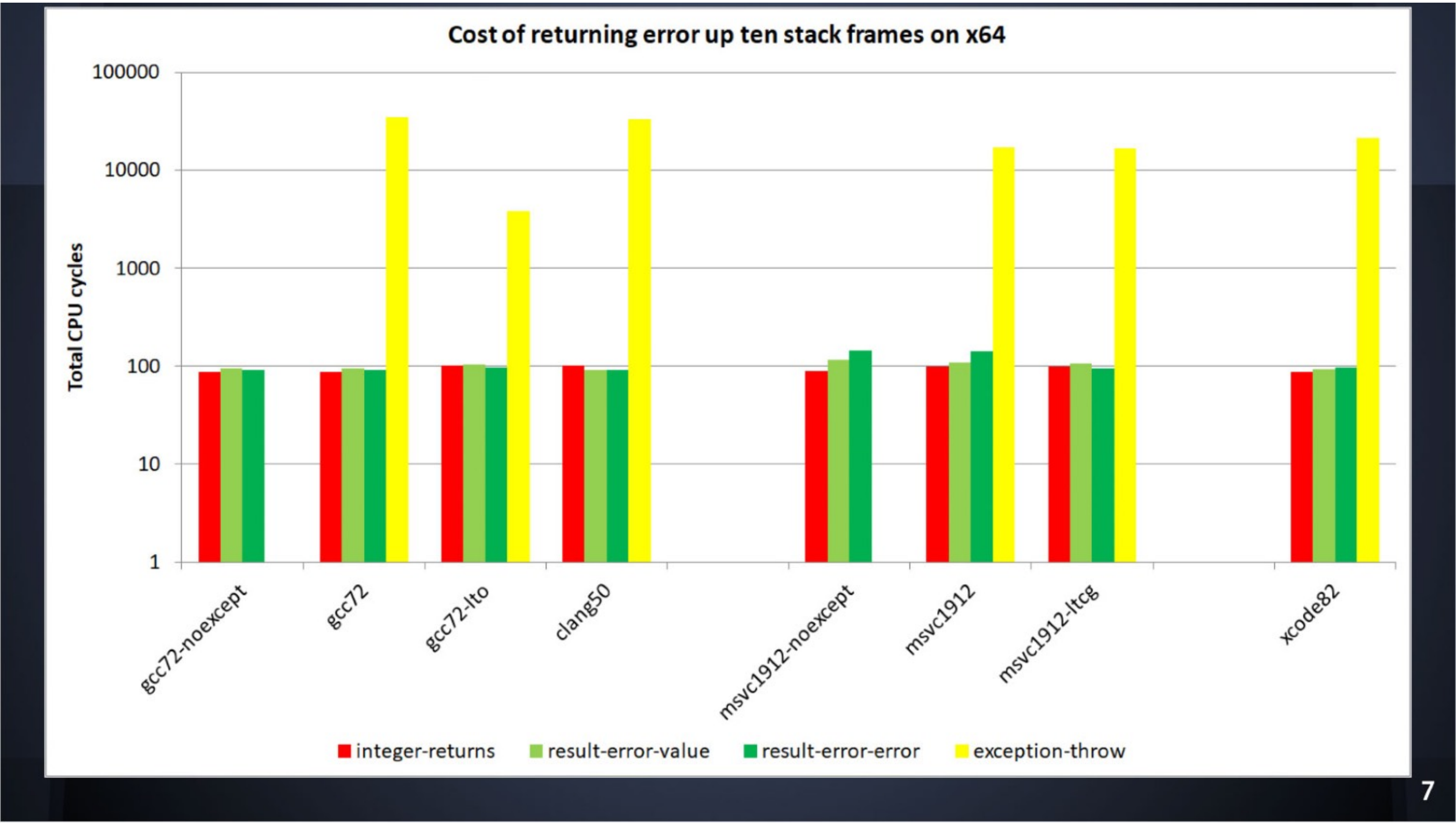
What is the Problem?

- Exceptions incur an extreme performance overhead in the failure case
- Exceptions make it harder to reason about functions
- Exceptions rely on dynamic memory
- Exceptions make the binary size grow (not zero overhead)

Performance of Exceptions

Meeting C++ 2017
Niall Douglas
Introduction to proposed std::expected<T,E>
Introduction to proposed std::expected<T, E>
Niall Douglas
0:01 / 1:36:43

Performance of Exceptions



What is the Problem?

- Exceptions incur an extreme performance overhead in the failure case
- Exceptions make it harder to reason about functions
- Exceptions rely on dynamic memory
- Exceptions make the binary size grow (not zero overhead)

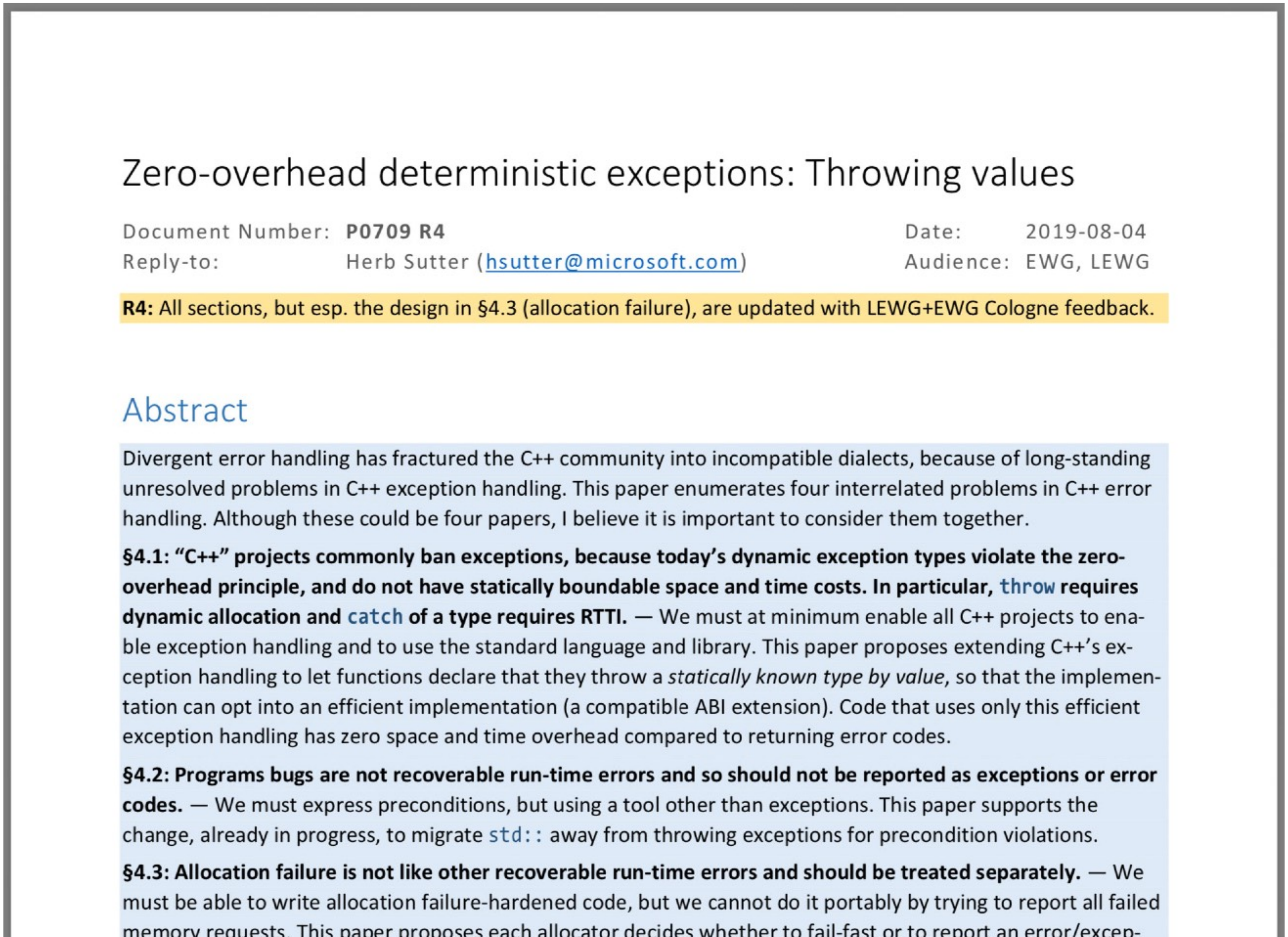
What is the Problem?

- Exceptions incur an extreme performance overhead in the failure case
- Exceptions make it harder to reason about functions
- Exceptions rely on dynamic memory
- Exceptions make the binary size grow (not zero overhead)

What is the Problem?

- Exceptions incur an extreme performance overhead in the failure case
- Exceptions make it harder to reason about functions
- Exceptions rely on dynamic memory
- Exceptions make the binary size grow (not zero overhead)

https://wg21.link/p0709



Zero-overhead deterministic exceptions: Throwing values

Document Number: **P0709 R4** Date: 2019-08-04
Reply-to: Herb Sutter (hsutter@microsoft.com) Audience: EWG, LEWG

R4: All sections, but esp. the design in §4.3 (allocation failure), are updated with LEWG+EWG Cologne feedback.

Abstract

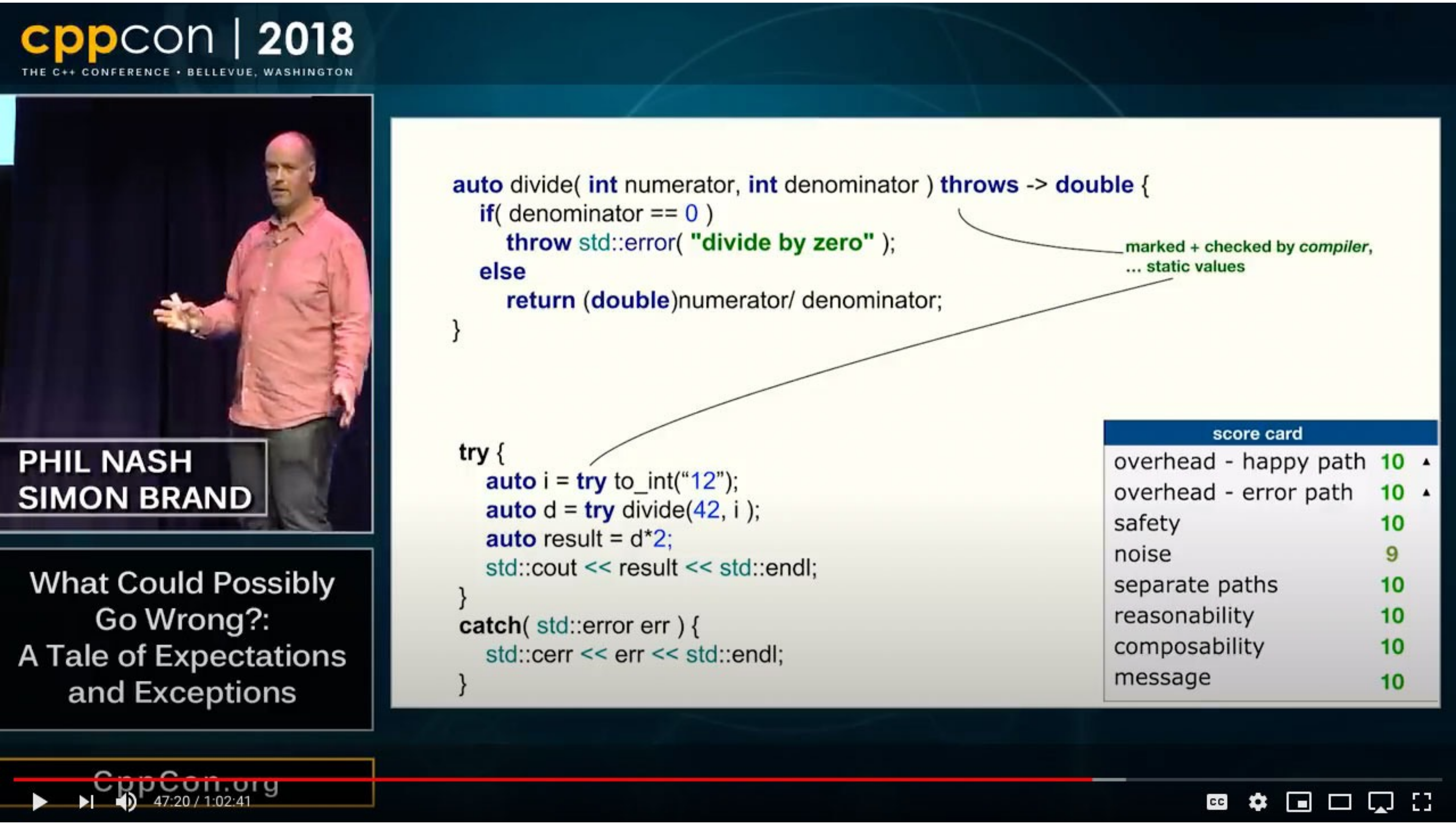
Divergent error handling has fractured the C++ community into incompatible dialects, because of long-standing unresolved problems in C++ exception handling. This paper enumerates four interrelated problems in C++ error handling. Although these could be four papers, I believe it is important to consider them together.

§4.1: "C++" projects commonly ban exceptions, because today's dynamic exception types violate the zero-overhead principle, and do not have statically boundable space and time costs. In particular, `throw` requires dynamic allocation and `catch` of a type requires RTTI. — We must at minimum enable all C++ projects to enable exception handling and to use the standard language and library. This paper proposes extending C++'s exception handling to let functions declare that they throw a *statically known type by value*, so that the implementation can opt into an efficient implementation (a compatible ABI extension). Code that uses only this efficient exception handling has zero space and time overhead compared to returning error codes.

§4.2: Programs bugs are not recoverable run-time errors and so should not be reported as exceptions or error codes. — We must express preconditions, but using a tool other than exceptions. This paper supports the change, already in progress, to migrate `std::` away from throwing exceptions for precondition violations.

§4.3: Allocation failure is not like other recoverable run-time errors and should be treated separately. — We must be able to write allocation failure-hardened code, but we cannot do it portably by trying to report all failed memory requests. This paper proposes each allocator decides whether to fail-fast or to report an error/excep-

Rating Exceptions


cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
PHIL NASH
SIMON BRAND
What Could Possibly Go Wrong?: A Tale of Expectations and Exceptions
CppCon.org
47:20 / 1:02:41

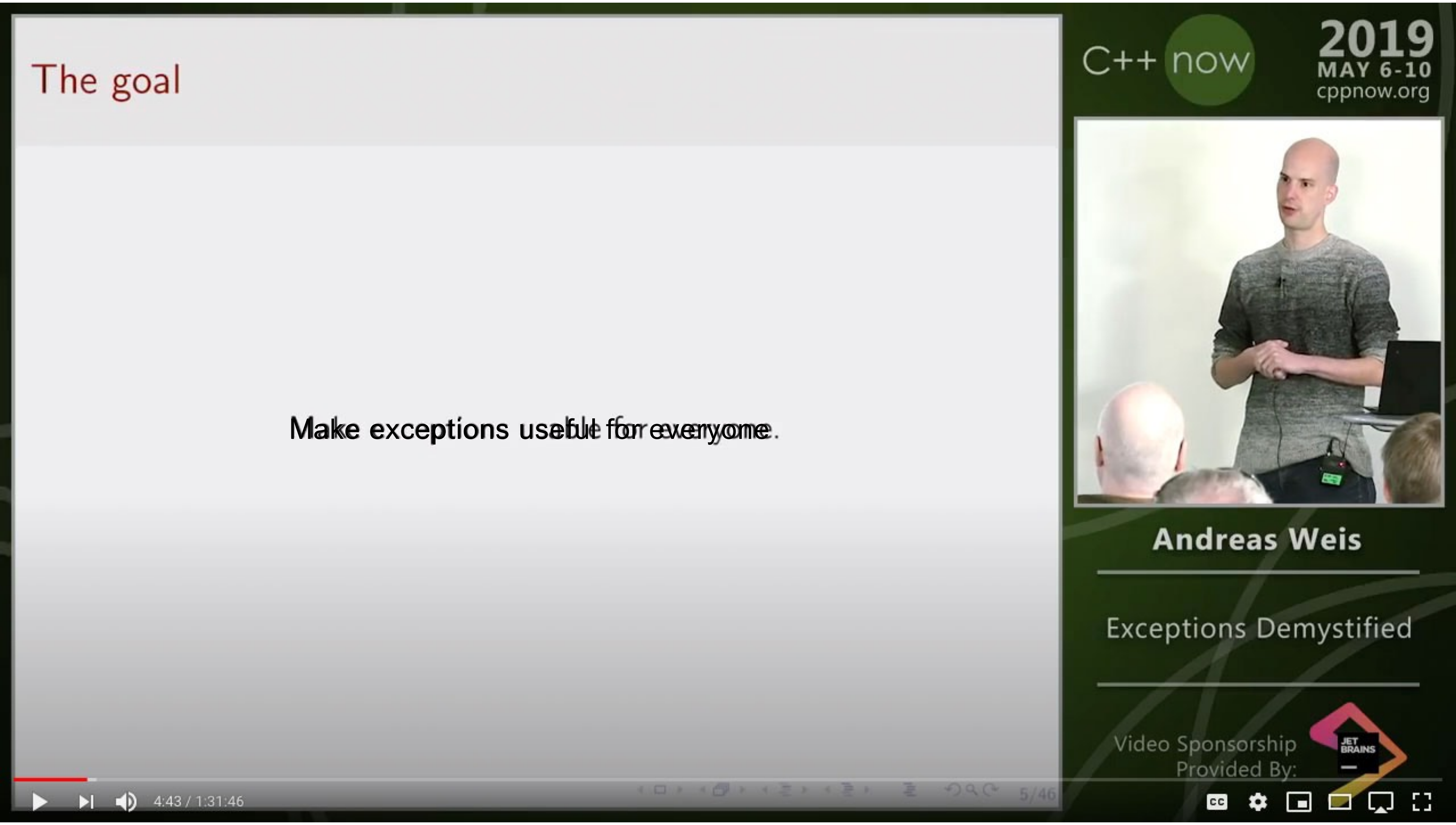
```
auto divide( int numerator, int denominator ) throws -> double {
   if( denominator == 0 )
      throw std::error( "divide by zero" );
   else
      return (double)numerator/ denominator;
}
```

marked + checked by *compiler*,
... static values

```
try {
   auto i = try to_int("12");
   auto d = try divide(42, i );
   auto result = d*2;
   std::cout << result << std::endl;
}
catch( std::error err ) {
   std::cerr << err << std::endl;
}
```

score card		
overhead - happy path	10	▲
overhead - error path	10	▲
safety	10	
noise	9	
separate paths	10	
reasonability	10	
composability	10	
message	10	

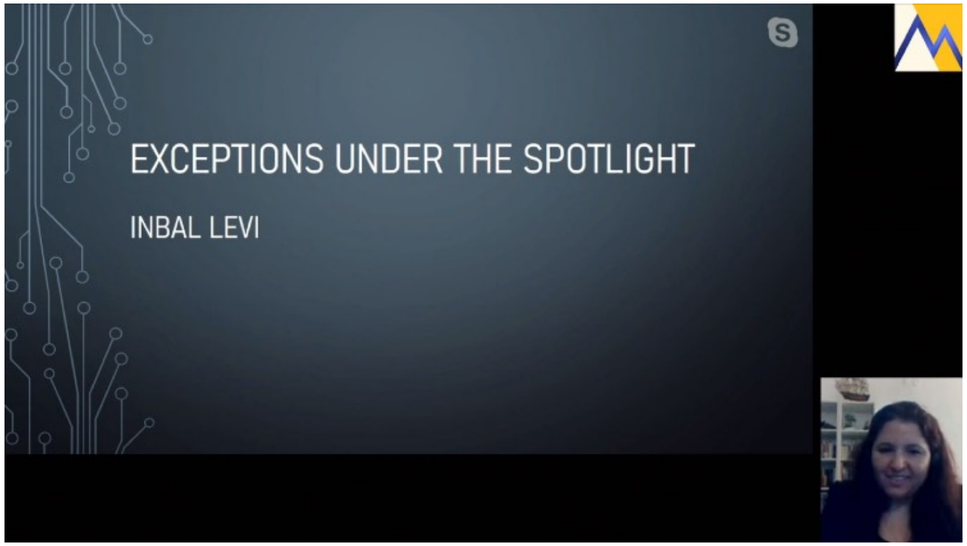
The Goal


The goal

Make exceptions useful for everyone.

The Goal

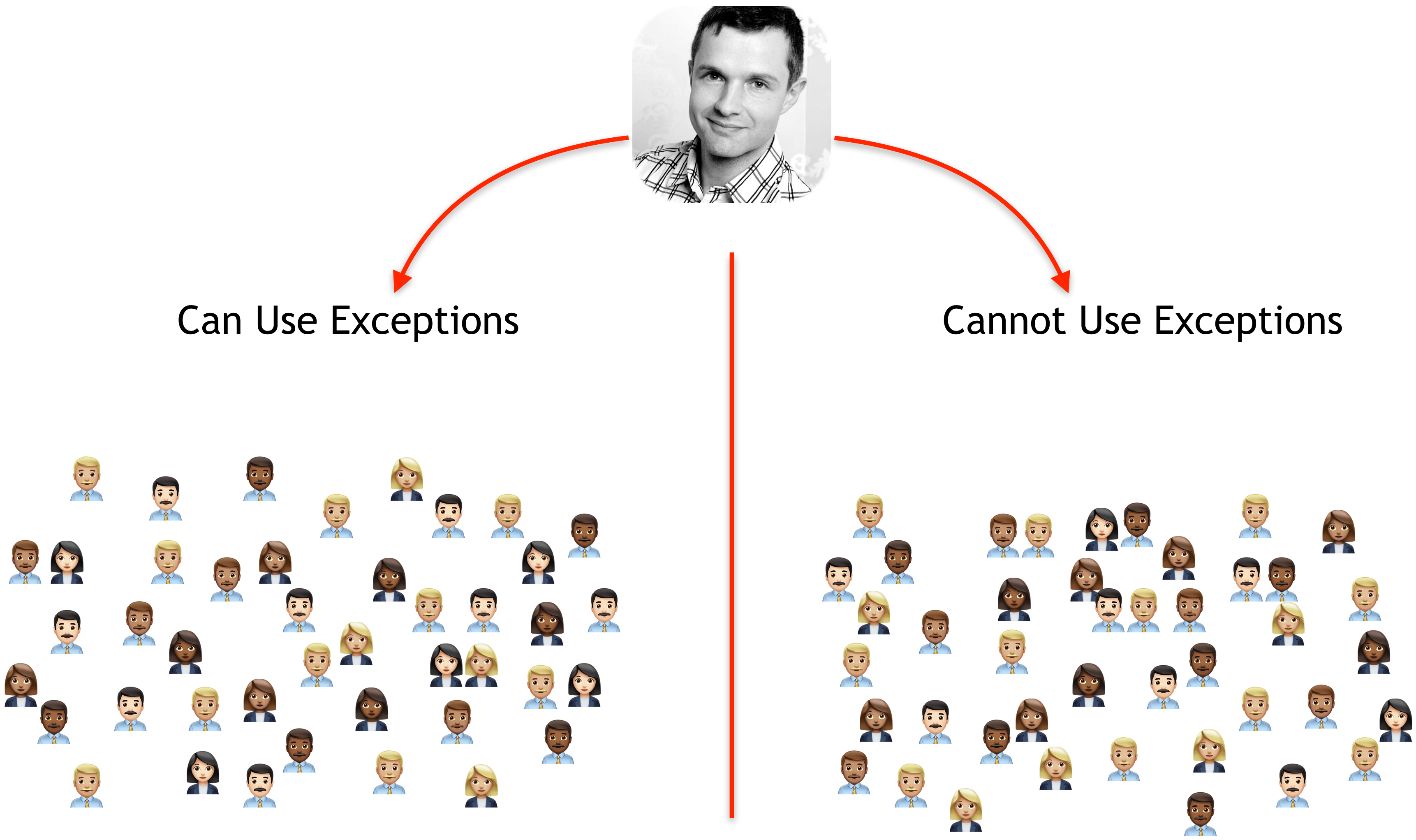
- Make exceptions useful for everyone (technically)


EXCEPTIONS UNDER THE SPOTLIGHT

INBAL LEVI

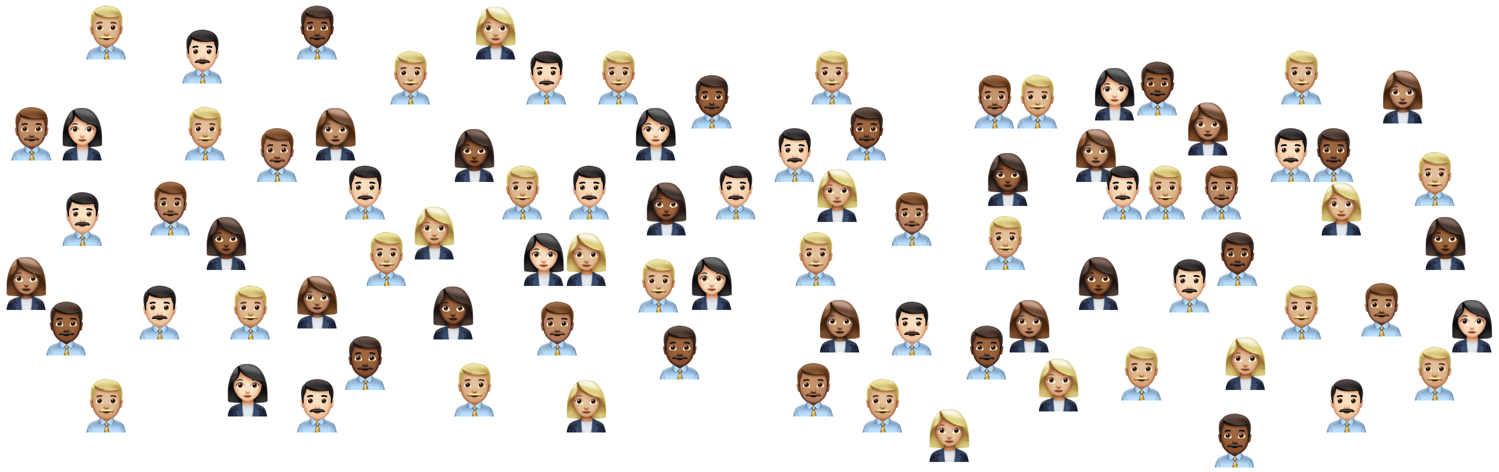
- Teach how to work with exceptions properly
    - … how to write good code
    - This talk

This Talk



This Talk

How to Write Good Code?



Content

How Do Exceptions Work

```
void f()
{
   std::string s{ “Some default initializer” };

   // …

   if( /* some condition */ ) {
      throw std::runtime_error( “…” );
   }

   // …
}

void g()
{
   std::vector<int> v{ 1, 2, 3, 4, 5, 6 };

   // …

   f();

   // …
}

void h()
{
   /* … */
}

int main()
{
   try {
      g();
      h();
   }
   catch( std::exception const& ex ) {
      /* Handle exception */
   }
}
```

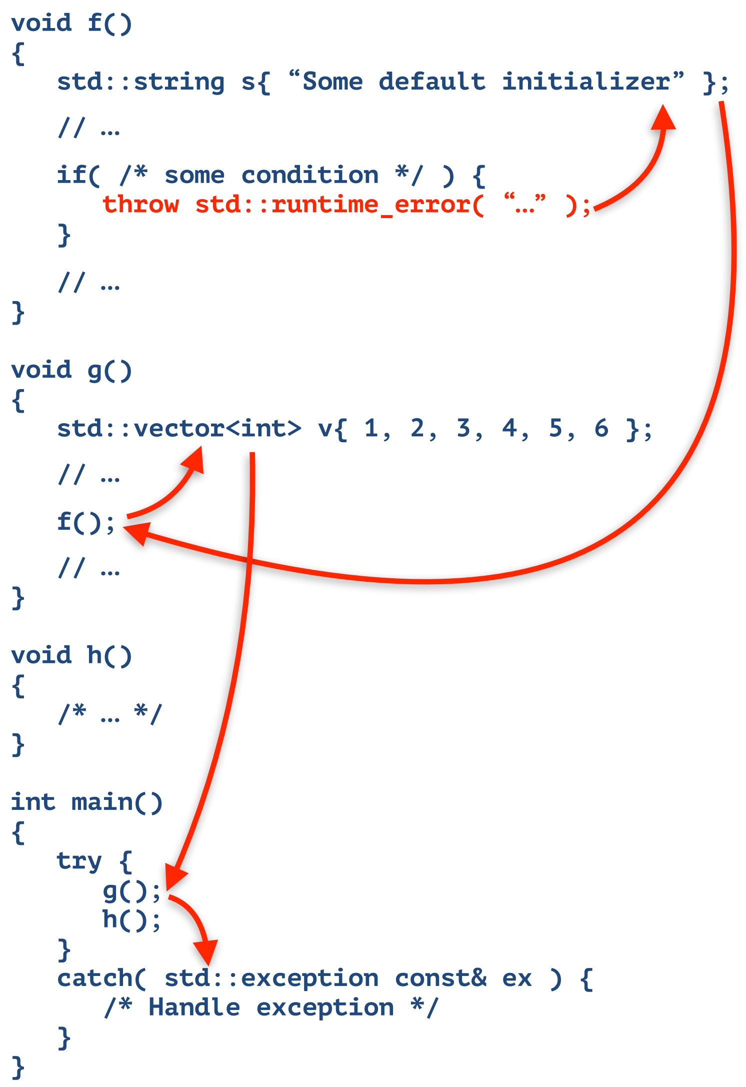
- Three keywords
  - throw
  - try
  - catch
- Stack unwinding
  - Objects on the stack are destroyed
  - Destruction happens in reverse order
- Unhandled exceptions result in a call to `std::terminate()`
  - No stack unwinding
  - No destructors are called
  - Resources are potentially leaked

How Do Exceptions Work

```
void f()
{
   std::string s{ “Some default initializer” };

   // …

   if( /* some condition */ ) {
      throw std::runtime_error( “…” );
   }

   // …
}

void g()
{
   std::vector<int> v{ 1, 2, 3, 4, 5, 6 };

   // …

   f();

   // …
}

void h()
{
   /* … */
}

int main()
{
   try {
      g();
      h();
   }
   catch( std::exception const& ex ) {
      /* Handle exception */
   }
}
```

- Three keywords
  - throw
  - try
  - catch
- Stack unwinding
  - Objects on the stack are destroyed
  - Destruction happens in reverse order
- Unhandled exceptions result in a call to `std::terminate()`
  - No stack unwinding
  - No destructors are called
  - Resources are potentially leaked

How Do Exceptions Work



```
void f()
{
   std::string s{ “Some default initializer” };
   // …
   if( /* some condition */ ) {
      throw std::runtime_error( “…” );
   }
   // …
}

void g()
{
   std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
   // …
   f();
   // …
}

void h()
{
   /* … */
}

int main()
{
   try {
      g();
      h();
   }
   catch( std::exception const& ex ) {
      /* Handle exception */
   }
}
```

- Three keywords
  - throw
  - try
  - catch
- Stack unwinding
  - Objects on the stack are destroyed
  - Destruction happens in reverse order
- Unhandled exceptions result in a call to `std::terminate()`
  - No stack unwinding
  - No destructors are called
  - Resources are potentially leaked

How Do Exceptions Work

```
void f()
{
   std::string s{ “Some default initializer” };

   // …

   if( /* some condition */ ) {
      throw std::runtime_error( “…” );
   }

   // …
}

void g()
{
   std::vector<int> v{ 1, 2, 3, 4, 5, 6 };

   // …

   f();

   // …
}

void h()
{
   /* … */
}

int main()
{

      g();
      h();



}
```

- Three keywords
  - throw
  - try
  - catch
- Stack unwinding
  - Objects on the stack are destroyed
  - Destruction happens in reverse order
- Unhandled exceptions result in a call to `std::terminate()`
  - No stack unwinding
  - No destructors are called
  - Resources are potentially leaked

How Do Exceptions Work

```
void f()
{
   std::string s{ “Some default initializer” };

   // …

   if( /* some condition */ ) {
      throw std::runtime_error( “…” );
   }

   // …
}

void g()
{
   std::vector<int> v{ 1, 2, 3, 4, 5, 6 };

   // …

   f();

   // …
}

void h()
{
   /* … */
}

int main()
{
   try {
      g();
      h();
   }
   catch( … ) {  // Catch-all handler
      /* … */
   }
}
```

- Three keywords
  - throw
  - try
  - catch
- Stack unwinding
  - Objects on the stack are destroyed
  - Destruction happens in reverse order
- Unhandled exceptions result in a call to `std::terminate()`
  - No stack unwinding
  - No destructors are called
  - Resources are potentially leaked

Questions?

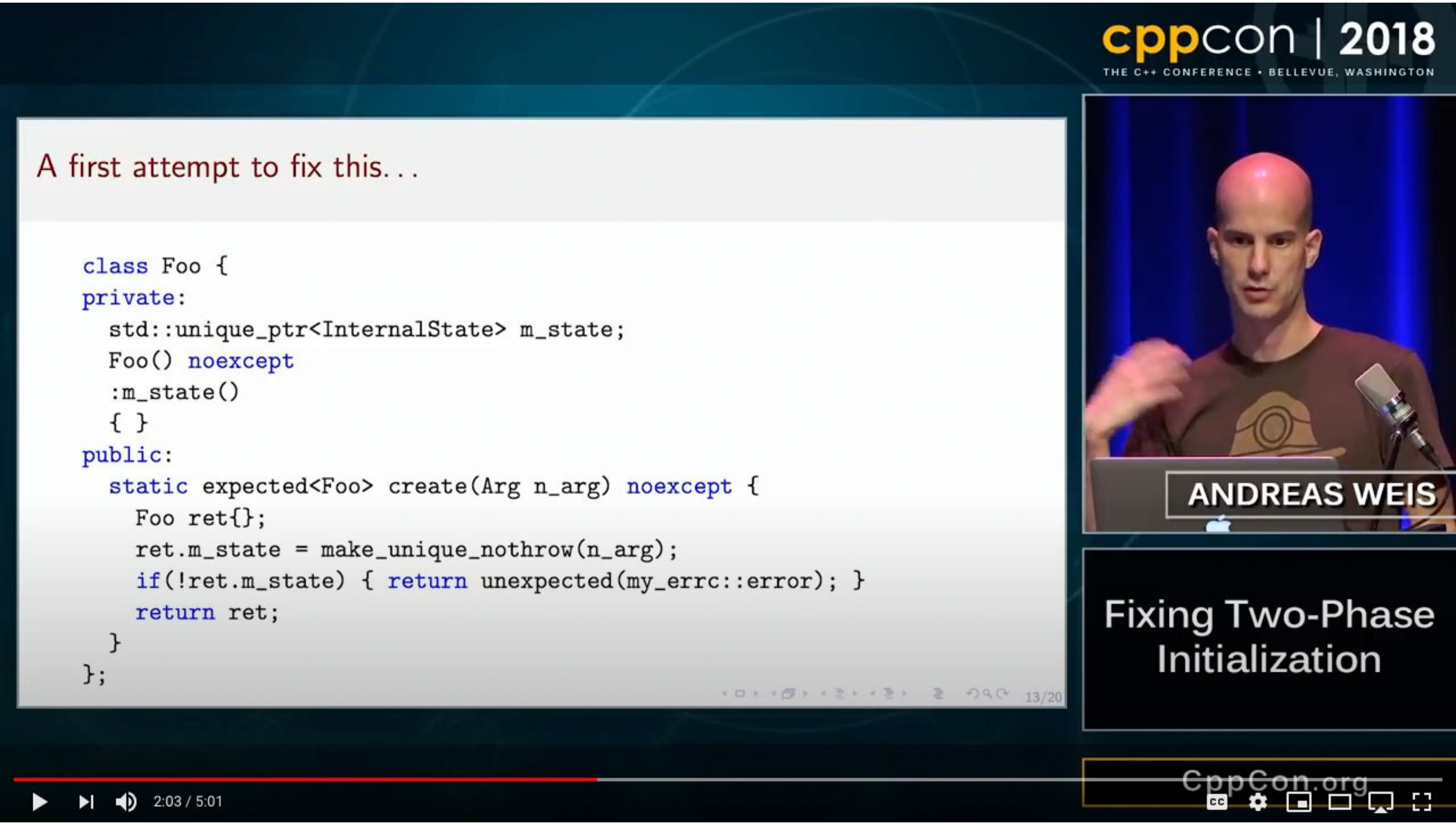
Content

When to Use Exceptions (And When Not)

Use exceptions …

- … for errors that are expected to occur rarely
- … for “exceptional cases” that cannot be dealt with locally (I/O errors)
  - File not found
  - Can’t find key in map
- … for operators and constructors (i.e. where few other mechanism works)

Dealing with Failing Constructors


cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
ANDREAS WEIS
Fixing Two-Phase Initialization
CppCon.org
2:03 / 5:01

A first attempt to fix this. . .

```
class Foo {
private:
  std::unique_ptr<InternalState> m_state;
  Foo() noexcept
  :m_state()
  { }
public:
  static expected<Foo> create(Arg n_arg) noexcept {
    Foo ret{};
    ret.m_state = make_unique_nothrow(n_arg);
    if(!ret.m_state) { return unexpected(my_errc::error); }
    return ret;
  }
};
```

13/20

When to Use Exceptions (And When Not)

Use exceptions …

- … for errors that are expected to occur rarely
- … for “exceptional cases” that cannot be dealt with locally (I/O errors)
  - File not found
  - Can’t find key in map
- … for operators and constructors (i.e. where no other mechanism works)

Don’t use exceptions …

- … for errors that are expected to occur frequently
- … for functions that are expected to fail

Dealing with Frequently Failing Functions

```
auto to_int( std::string const& s ) -> std::optional<int>

auto to_int( std::string const& s ) -> std::expected<int>
```

Dealing with Frequently Failing Functions

```
auto to_int( std::string const& s ) -> std::optional<int>

auto to_int( std::string const& s ) -> std::expected<int>

auto to_int( std::string const& s ) -> boost::outcome<int>
```

- Easier to comprehend
- Doesn’t pretend that all strings can be converted into an `int`

When to Use Exceptions (And When Not)

Use exceptions …

- … for errors that are expected to occur rarely
- … for “exceptional cases” that cannot be dealt with locally (I/O errors)
  - File not found
  - Can’t find key in map
- … for operators and constructors (i.e. where no other mechanism works)

Don’t use exceptions …

- … for errors that are expected to occur frequently
- … for functions that are expected to fail
- … if you have to guarantee certain response times, even in the error case
- … for things that should never happen
  - Dereferencing nullptrs
  - Out-of-range access
  - Use after free

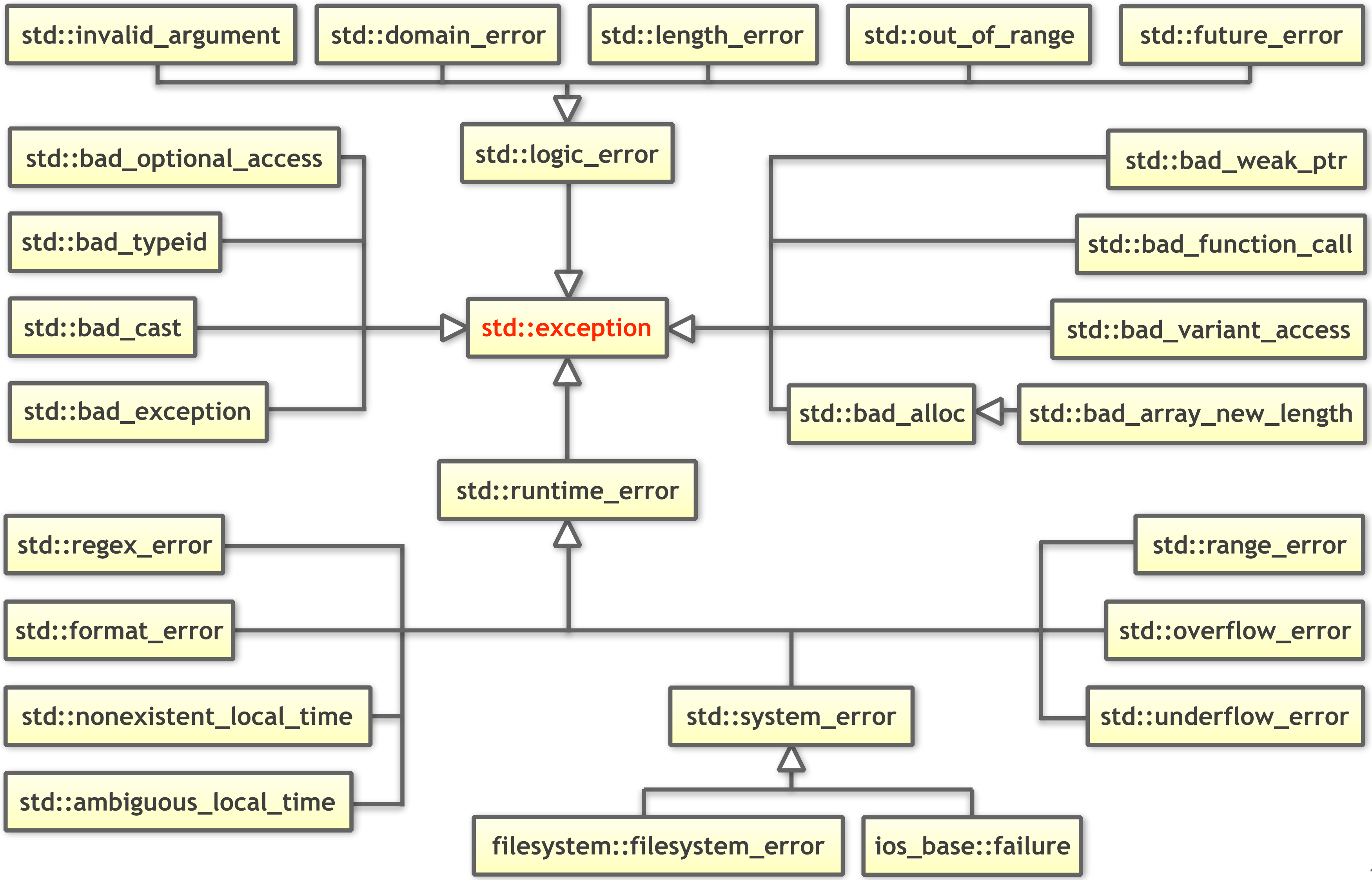
Questions?

Content

How to Use Exceptions

- Build on the `std::exception` hierarchy
- Throw by rvalue
- Catch by reference

Build on the `std::exception` Hierarchy



How to Use Exceptions

- Build on the `std::exception` hierarchy
- Throw by rvalue
- Catch by reference

Throw By Rvalue

```
void f( /* ... */ )
{
   // ...

   if( /* some condition */ ) {
      std::runtime_error error( “Error message” );
      throw error;
   }

   // ...
}
```

- Don’t throw by pointer or reference (bad semantics)
- `throw` makes a copy of the value to throw

Throw By Rvalue

```
void f( /* ... */ )
{
   // ...
   if( /* some condition */ ) {
      throw std::runtime_error( “Error message” );
   }
   // ...
}
```

- Don’t throw by pointer or reference (bad semantics)
- `throw` makes a copy of the value to throw
- throw by rvalue

How to Use Exceptions

- Build on the `std::exception` hierarchy
- Throw by rvalue
- Catch by reference

Catch By Reference

```
void f( /* ... */ )
{
   // ...

   try {
      /* ... */
   }
   catch( std::exception ex ) {
      /* ... */
   }

   // ...
}
```

- Don’t catch by value, it …
  - … creates an unnecessary copy
  - … potentially slices the exception 😱

Catch By Reference

```
void f( /* ... */ )
{
   // ...

   try {
      /* ... */
   }
   catch( std::exception const& ex ) {
      /* ... */
   }

   // ...
}
```

- Don’t catch by value, it …
  - … creates an unnecessary copy
  - … potentially slices the exception 😱
- Catch by reference (to `const`)

Questions?

Content

The Exception Safety Guarantees

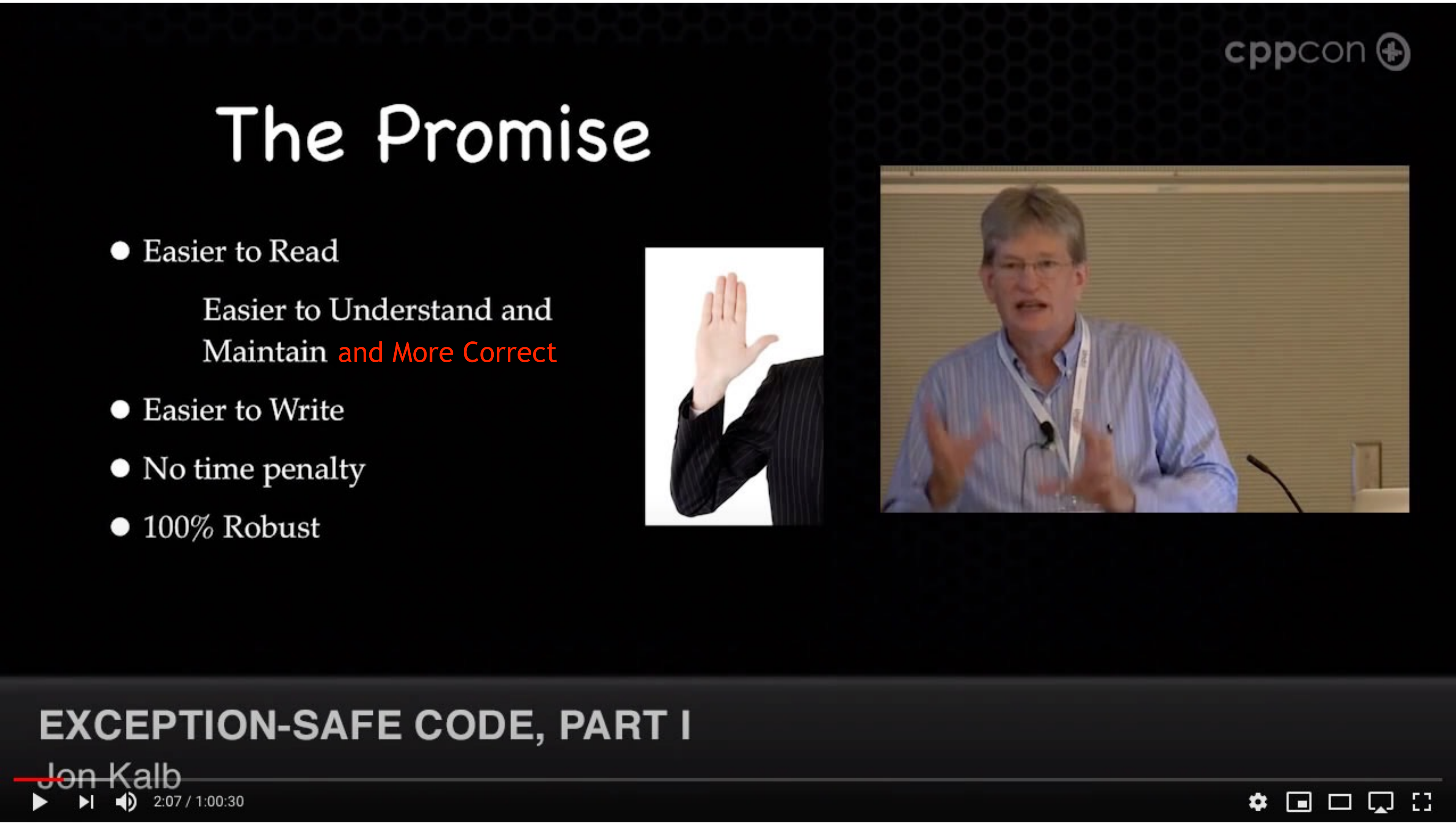
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible (e.g. sockets, streams, etc.)
- No-Throw Guarantee
  - The operation cannot fail
  - Expressed in code with `noexcept`

Content

- The Exception Situation
- How Do Exceptions Work
- Best Practices of Exception Handling
  - When to Use Exceptions (And When Not)
  - How to Use Exceptions
  - The Exception Safety Guarantees
  - How to Write Exception-Safe Code
  - How to Refactor Non-Exception-Safe Code

Content

How to Write Exception-Safe Code



The Promise

- Easier to Read

  Easier to Understand and Maintain and More Correct

- Easier to Write
- No time penalty
- 100% Robust

EXCEPTION-SAFE CODE, PART I

Jon Kalb

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …

























   // …
};

```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }
















   // …
};

```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {










   }

   // …
};

```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {









      return *this;
   }

   // …
};
```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {


      i = w.i;






      return *this;
   }

   // …
};

```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {


      i = w.i;
      s = w.s;






      return *this;
   }

   // …
};

```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {


      i = w.i;
      s = w.s;


      pr = new Resource( *w.pr );


      return *this;
   }

   // …
};

```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {


      i = w.i;
      s = w.s;


      if( w.pr ) pr = new Resource( *w.pr );


      return *this;
   }

   // …
};

```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {


      i = w.i;
      s = w.s;
      delete pr;

      if( w.pr ) pr = new Resource( *w.pr );


      return *this;
   }

   // …
};

```

How to Write Exception-Safe Code

```cpp
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {


      i = w.i;
      s = w.s;
      delete pr;

      if( w.pr ) pr = new Resource( *w.pr );
      else pr = nullptr;

      return *this;
   }

   // …
};
```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;
      delete pr;

      if( w.pr ) pr = new Resource( *w.pr );
      else pr = nullptr;

      return *this;
   }

   // …
};

```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;
      delete pr;

      if( w.pr ) pr = new Resource( *w.pr );
      else pr = nullptr;

      return *this;
   }

   // …
};
```

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;
      delete pr;     ← Dangling pointer, invariant violated!

      if( w.pr ) pr = new Resource( *w.pr );   // Both new and the Resource constructor could throw!
      else pr = nullptr;

      return *this;
   }

   // …
};
```

How to Write Exception-Safe Code

How to Write Exception-Safe Code

```
class BankAccount {
   // …

   void withdrawMoney( int amount )
   {
      // …

      reduceBalance( amount );


      prepareCash();






      releaseCash();

      // …
   }

   // …
};
```

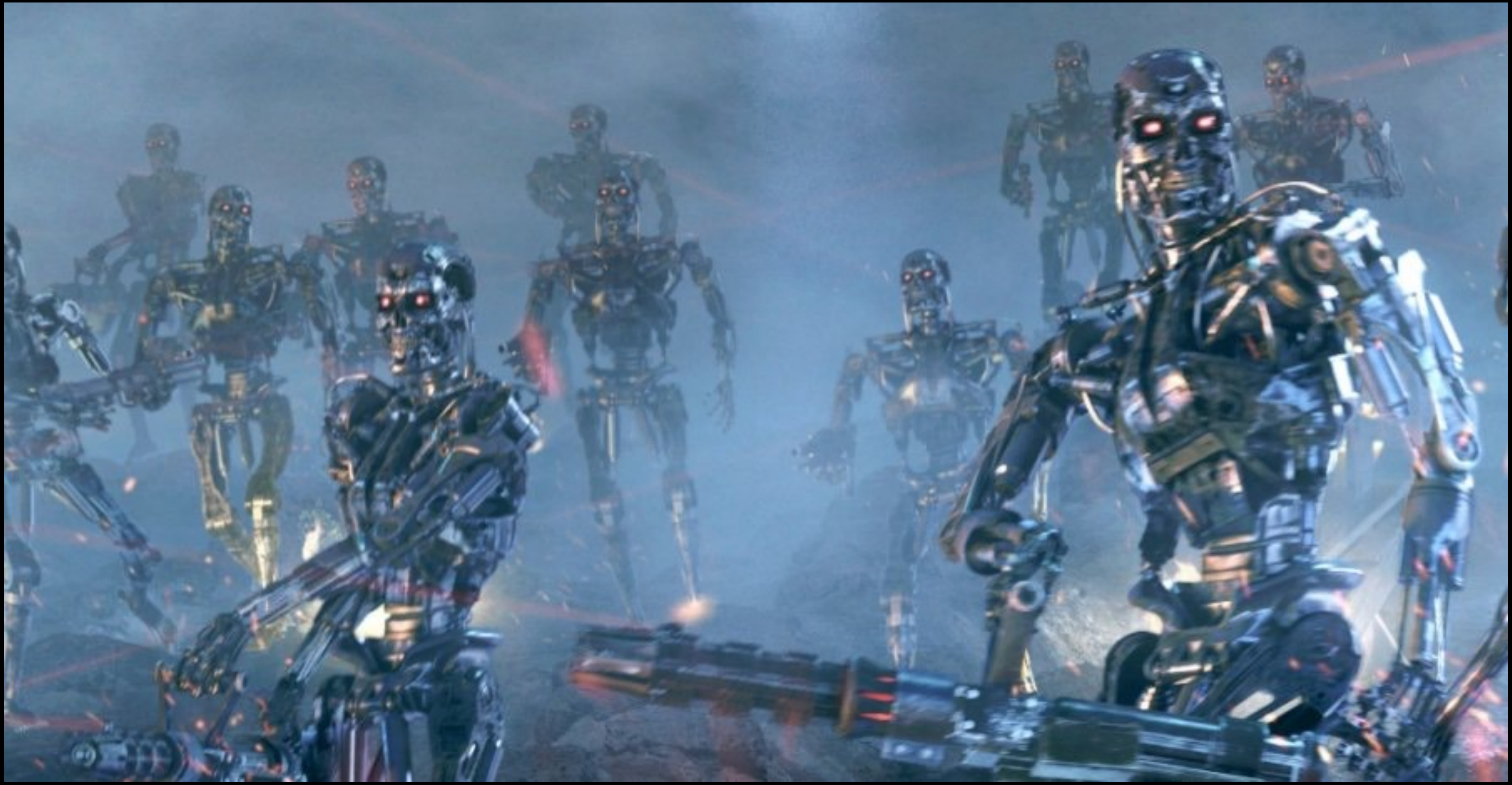
How to Write Exception-Safe Code

```
class BankAccount {
   // …

   void withdrawMoney( int amount )
   {
      // …

      reduceBalance( amount );     Balance already reduced


      prepareCash();     Throws an exception





      releaseCash();

      // …
   }

   // …
};
```

How to Write Exception-Safe Code

```
class BankAccount {
   // …

   void withdrawMoney( int amount )
   {
      // …

      reduceBalance( amount );

      try {
         prepareCash();
      }
      catch( std::exception const& ) {
         increaseBalance( amount );
      }

      releaseCash();

      // …
   }

   // …
};
```

LEHMAN BROTHERS
NEW YORK



"There is no try."
(Yoda, Star Wars, & Jon Kalb, CppCon 2014)

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;          Dangling pointer, invariant violated!
      delete pr;

      if( w.pr ) pr = new Resource( *w.pr );   // Both new and the Resource constructor could throw!
      else pr = nullptr;

      return *this;
   }

   // …
};
```

How to Write Exception-Safe Code

```cpp
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;
      delete pr;

      if( w.pr ) pr = new Resource( *w.pr );
      else pr = nullptr;

      return *this;
   }

   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```cpp
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;
      delete pr;
      pr = nullptr;
      if( w.pr ) pr = new Resource( *w.pr );


      return *this;
   }

   // …
};

```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   Resource* pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = new Resource( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;
      delete pr;
      pr = nullptr;
      if( w.pr ) pr = new Resource( *w.pr );


      return *this;
   }

   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};  // May be nullptr

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;


      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );


      return *this;
   }

   // …
};
```

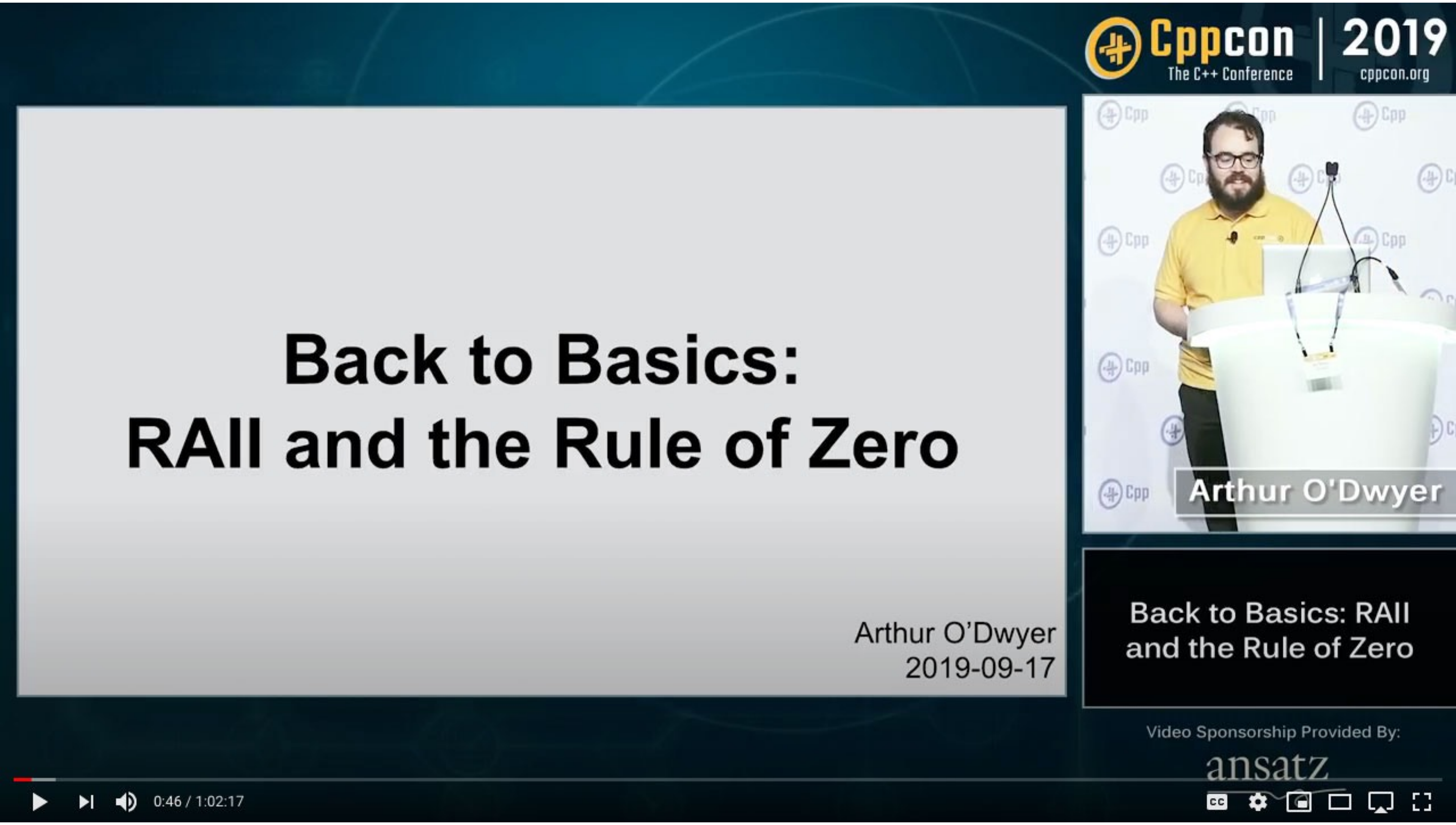
- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

Resource Acquisition Is Initialization (RAII)

RAII

(Resource Acquisition Is Initialization)

Resource Acquisition Is Initialization (RAII)


CppCon The C++ Conference | 2019 cppcon.org
Arthur O'Dwyer
Video Sponsorship Provided By:
ansatz
0:46 / 1:02:17

Back to Basics:
RAII and the Rule of Zero

Arthur O'Dwyer
2019-09-17

Back to Basics: RAII and the Rule of Zero

Resource Acquisition Is Initialization (RAII)

”Keep your resources on a short leash to not go leaking wherever they want.”

(Jon Kalb, “Exception-Safe Code”, CppCon 2014)

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;                          Can still fail

      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );

      return *this;
   }




   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;

      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );

      return *this;
   }




   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      i = w.i;
      s = w.s;

      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );

      return *this;
   }




   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```cpp
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      // RAII-based approach
      Widget tmp( w );



      return *this;
   }




   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      // RAII-based approach
      Widget tmp( w );  // Temporary-move idiom
      *this = move(tmp);


      return *this;
   }




   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      // RAII-based approach
      Widget tmp( w );  // Temporary-move idiom
      *this = move(tmp);


      return *this;
   }

   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept;

   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      // RAII-based approach
      Widget tmp( w );  // Temporary-move idiom
      *this = move(tmp);


      return *this;
   }

   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept;

   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```cpp
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      // RAII-based approach
      Widget tmp( w );  // Temporary-move idiom
      *this = move(tmp);


      return *this;
   }

   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept;

   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

Functions That Should Never Fail

- Destructors
  - Are called during stack unwinding
  - If an exception is thrown while another one is flying, `terminate()` is called
  - Are implicitly marked as `noexcept` since C++11
  - Cleanup must be safe!
- Move Operations
  - Should be implemented by means of non-failing operations

Core Guideline C.66: Make move operations `noexcept`.

- `swap` Operations
  - Can usually be implemented in terms of non-failing (basic) operations

The Benefits of `noexcept`

- `noexcept` makes the promise to never throw visible in the code
- `noexcept` can lead to (slightly) faster code
- If an exception leaves a function marked with `noexcept`, `terminate()` is called
  - The compiler does not check this promise
- A `noexcept` promise cannot not be taken back
  - Only few functions should be marked with `noexcept`
- Destructors are implicitly marked with `noexcept`

How to Write Exception-Safe Code

```cpp
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      // RAII-based approach
      Widget tmp( w );  // Temporary-move idiom
      *this = move(tmp);


      return *this;
   }

   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept;

   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      if( this == &w ) return *this;

      // RAII-based approach
      Widget tmp( w );  // Temporary-move idiom
      *this = move(tmp);


      return *this;
   }

   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept;

   // …
};
```

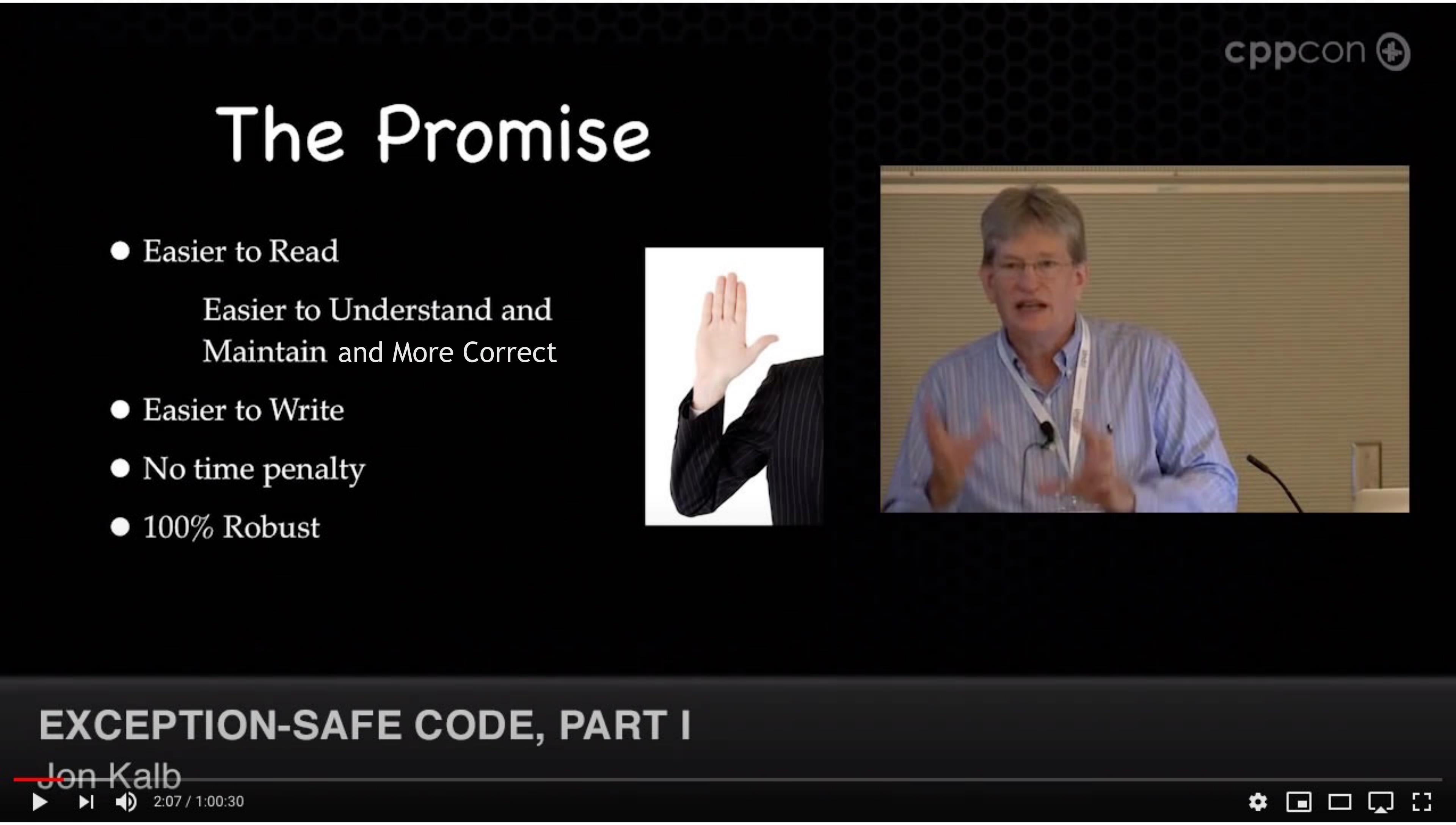
- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code

```cpp
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   std::unique_ptr<Resource> pr{};

 public:
   // …
   // Copy constructor
   Widget( Widget const& w )
      : i { w.i }
      , s { w.s }
   {
      if( w.pr ) pr = std::make_unique<Resource>( *w.pr );
   }

   // Copy assignment operator
   Widget& operator=( Widget const& w )
   {
      /* if( this == &w ) return *this; */  // Optional

      // RAII-based approach
      Widget tmp( w );  // Temporary-move idiom
      *this = move(tmp);


      return *this;
   }

   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept;

   // …
};
```

- Exception unsafe
  - No guarantees with respect to invariants and resources
- Basic Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
- Strong Exception Safety Guarantee
  - Invariants are preserved
  - No resources are leaked
  - No state change (commit-or-rollback)
  - Not always possible
- No-Throw Guarantee
  - The operation cannot fail

How to Write Exception-Safe Code



The Promise

- Easier to Read

  Easier to Understand and Maintain and More Correct
- Easier to Write
- No time penalty
- 100% Robust

EXCEPTION-SAFE CODE, PART I

Jon Kalb

Guidelines

Guideline: RAII is the single most important idiom of the C++ programming language. Use it!

Guideline: All functions should at least provide the basic exception safety guarantee, if possible and reasonable the strong guarantee.

Guideline: Consider the no-throw guarantee, but only provide it if you can guarantee it even for possible future changes.

How to Deal With Failing Cleanup Functions

How to Deal With Failing Cleanup Functions

- Destructors must not throw
- But what if a function called in a destructor can fail?

```
class File {
 public:
   // …
   ~File()
   {
      fclose( pf );   // fclose returns an error code; may fail




   }
   // …
 private:
   std::FILE* pf;
};
```

How to Deal With Failing Cleanup Functions

- Destructors must not throw
- But what if a function called in a destructor can fail?

```
class File {
 public:
   // …
   ~File()
   {
      int const result = fclose( pf );

      if( !result ) {
         /* Deal with the error, but don’t throw! */
      }
   }
   // …
 private:
  std::FILE* pf;
};
```

- std::ofstream ignores failure during closing the file
- For handling the error case differently, write your own RAII class

Questions?

Content

- The Exception Situation
- How Do Exceptions Work
- Best Practices of Exception Handling
    - When to Use Exceptions (And When Not)
    - How to Use Exceptions
    - The Exception Safety Guarantees
    - How to Write Good Code
    - How to Refactor Non-Exception-Safe Code

How to Refactor Non-Exception Safe Code

- Transition from pre-exception/exception-unsafe legacy code (which doesn't handle code path disruption gracefully) to exception-safe code
- Sean Parent's **Iron Law of Legacy Refactoring:**

Existing contracts cannot be broken!

Sean’s Rules

- All new code is written to be exception safe
- Any new interfaces are free to throw an exception
  - When working on existing code the interface to that code must be followed – **if it wasn’t throwing exceptions before, it can’t start now**.
  - Consider implementing a new function and re-implementing the old in terms of the new

Refactoring Steps

1. Implement a new function following exception safety guidelines
2. Legacy function now calls new function wrapped in try/catch(…)
   - Legacy API unchanged / doesn’t throw
3. New code can always safely call throwing code
4. Retire wrapper functions as appropriate

Refactoring Example

```
bool FileReader::LoadFile(ByteStream& input)
{
   fileSizeType aSize;
   IOError iErr;
   File f(m_FileName, GENERIC_READ, FILE_SHARE_READ, OPEN_EXISTING);
   iErr = f.Open();
   if ( iErr == IOE_OK )
   {
      iErr = f.GetSize(&aSize);
      if ( iErr == IOE_OK )
      {
         uint16_t inBufferSize = (uint16_t) aSize + 1U;
         uint32_t bytesRead = 0U;
         input.SetSize(inBufferSize);
         iErr = f.Read(&input[0], inBufferSize, &bytesRead);
      }
      f.Close();
   }
   return ( iErr == IOE_OK );
}
```

Refactoring Assumptions

- The `File` class follows RAII
  - Constructor opens, throws on error
  - Destructor calls `Close()` when necessary
- `File::GetSize()` returns value, throws on error
- `File::Read()` returns bytes read, throws on error

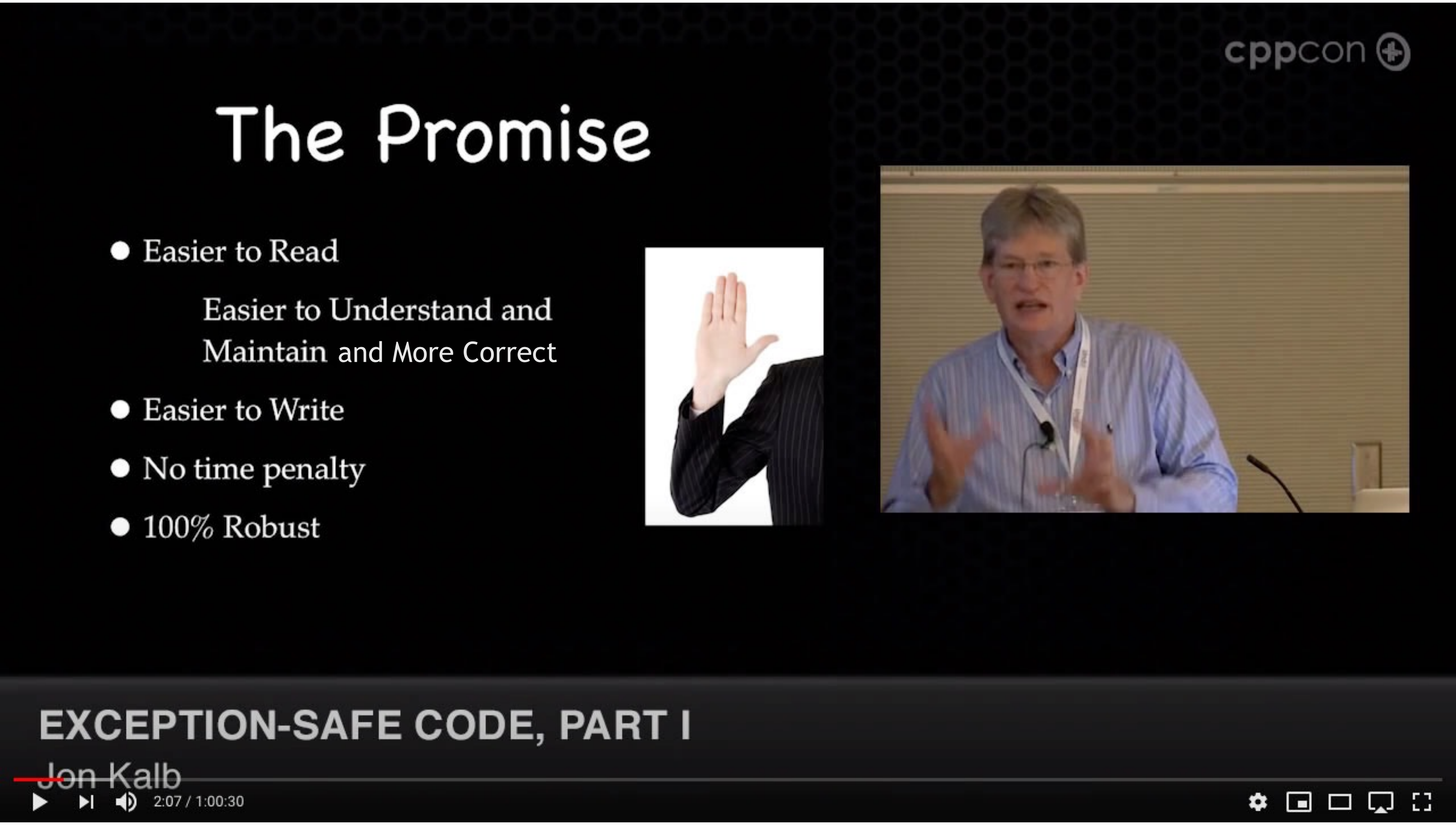
Refactoring Example

```
ByteStream const load_file(char const* filename)
{
   File f(filename, GENERIC_READ, FILE_SHARE_READ, OPEN_EXISTING);
   ByteStream input(f.GetSize() + 1);
   f.Read(input.data(), input.size());
   return input;
}


bool FileReader::LoadFile(ByteStream& input)
{
   try {
      input = load_file(m_FileName);
      return true;
   }
   catch(...) {}
   return false;
}
```

Refactoring Steps

- Moving a large code base is still a big chore
- Can be done in small bites
- No need to swallow an elephant
- Part of regular maintenance
- Can move forward with confidence
- Code base is never at risk!

How to Write Exception-Safe Code



The Promise

- Easier to Read

  Easier to Understand and Maintain and More Correct
- Easier to Write
- No time penalty
- 100% Robust

Questions?

Back to Basics:

Exceptions

Klaus Iglberger, CppCon 2020

klaus.iglberger@gmx.de