Nested Exceptions
and Exception Pointers

Dr. Colin Hirsch

"A corner of C++11 that I stumbled over recently"

Dr. Colin Hirsch

Nested Exceptions and Exception_ptr

Dr. Colin Hirsch

With Templates!

Exceptional Agenda

- ~~Exception Basics~~
- Exception Pointers
- Nested Exceptions

Nested Exceptions

- Motivating example (PEGTL)
- How to throw them (very easy)
- How to catch them (very easy)
- How to inspect them (the interesting bit)

Motivating Example

- Consider the PEGTL
- An unrecoverable parse error is thrown as exception:
  - `throw parse_error( "typo in file bar.in" );`
- What if **`bar.in`** was parsed via the following line from **`foo.in`**?
  - `include "bar.in"`
- Then the error message must include (no pun intended) something like:
  - `"included from file foo.in"`
- However this information is not available where the exception is thrown!

Enriching Exceptions

```
// Current solution in the PEGTL (simplified/pseudo code):

struct parse_error
{
   std::vector< position > positions;
};
```

Enriching Exceptions

```
// Current solution in the PEGTL (simplified/pseudo code):

struct parse_error
{
   std::vector< position > positions;
};

// Parse function to parse files included from other files:

bool parse_nested( const position& include_position, const std::filesystem::path& included_file )
{
   try {
     return parse( included_file );
   }
   catch( parse_error& e ) {
      e.positions.emplace_back( include_position );
      throw;
   }
}
```

Exceptionally Limited

- This approach is very limited
  - Only one exception type is used to accumulate data
    - Only one type of position information is accumulated

- PEGTL currently undergoing refactoring of input layer
  - More flexibility for the input classes, including:
    - Different input classes can use different position information

- **What we want is nesting of arbitrary exception types**

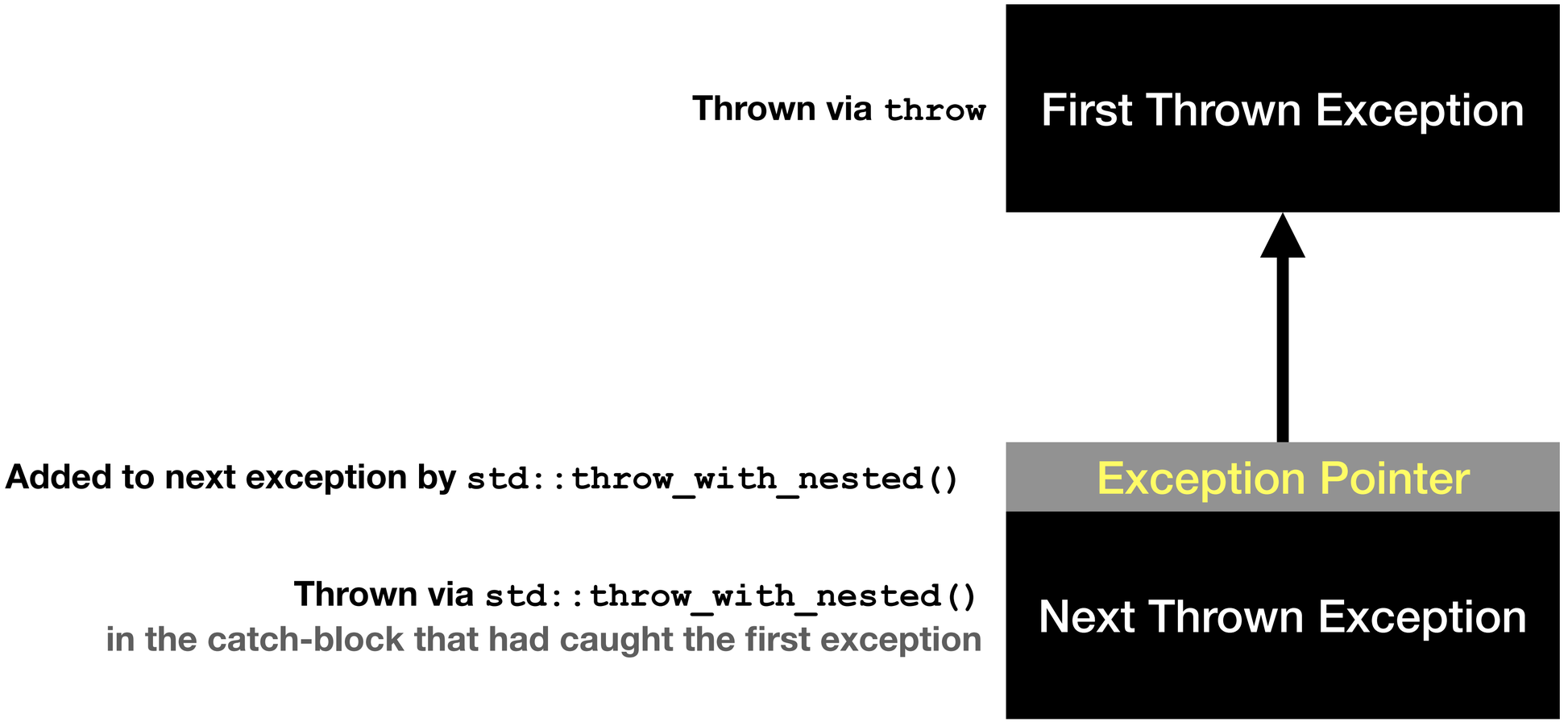
Throwing Nested Exceptions

```
// Future solution in the PEGTL (simplified/pseudo code):

template< typename Position >
struct parse_error
{
   Position position;
};

template< typename Position >
bool parse_nested( const Position& include_position, const std::filesystem::path& included_file )
{
   try {
     return parse( included_file );
   }
   catch( ... ) {
      std::throw_with_nested( parse_error< Position >( include_position ) );
  }
}
```

Nested Exception Structure



Nested Exception Implementation

```
// Current solution in the libstdc++ (simplified/pseudo code):

struct std::nested_exception
{
   std::exception_ptr nested = std::current_exception();
};

template< typename Exception >
struct std::detail::unspecified_exception
   : Exception, nested_exception
{
   using Exception::Exception;
};

template< typename Exception >
[[noreturn]] void std::throw_with_nested( const Exception& e )
{
   throw std::detail::unspecified_exception< Exception >( e );
}
```

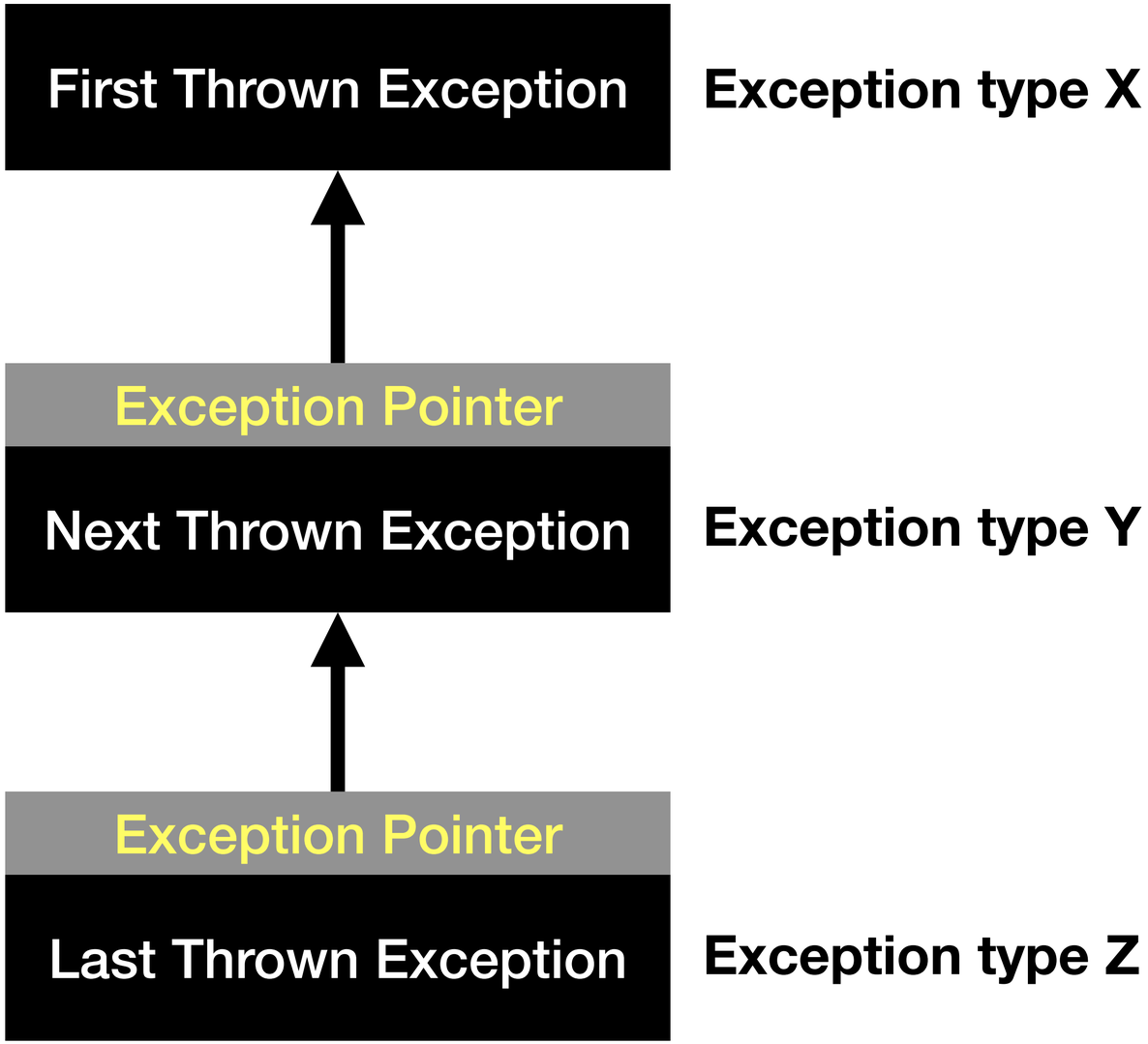
Catching Nested Exceptions

```
// Ignoring the nested exceptions:

void function()
{
   try {
      can_throw_nested_exceptions();
   }
   catch( const std::exception& e ) {
      // Catches throw and std::throw_with_nested().
   }
};
```

We might be catching a rat's nest of nested exceptions of different types (pun intended)

At the point of catch we know neither the types nor the number of nested exceptions



Inspecting Nested Exceptions

```
// Ignoring the nested exceptions (simplified/pseudo code):

void function()
{
   try {
      can_throw_nested_exceptions();
   }
   catch( const std::exception& e ) {
      // Works for throw e and std::throw_with_nested( e ).
   }
};
```

- **How to get at the nested exceptions?**
- **Something that works with arbitrary nestings!**

Inspecting Nested Exceptions

```
// Ignoring the nested exceptions (simplified/pseudo code):

void function()
{
   try {
      can_throw_nested_exceptions();
   }
   catch( const std::exception& e ) {
      // Works for throw e and std::throw_with_nested( e ).
   }
};
```

Inspecting Nested Exceptions

```
// What we want to do:

void function()
{
   try {
      can_throw_nested_exceptions();
   }
   catch( const std::exception& e ) {
      // Do stuff with e
      // Do stuff with exception nested in e
   }
};
```

Inspecting Nested Exceptions

```
// What we can do:

void function()
{
    try {
       can_throw_nested_exceptions();
    }
    catch( const std::exception& e ) {
       // Do stuff with e
       std::rethrow_if_nested( e );
    }
};
```

Inspecting Nested Exceptions

```
// What we need to do but it doesn't scale:

void function()
{
   try {
      can_throw_nested_exceptions();
   }
   catch( const std::exception& e ) {
      // Do stuff with e
      try {
         std::rethrow_if_nested( e );
      }
      catch( const std::exception& e ) {
         // Do stuff with e
         // Do stuff with exception nested in e
      }
   }
};
```

Inspecting Nested Exceptions

```
catch( ... ) {
   std::exception_ptr p = std::current_exception();
   do {
      try {
         std::rethrow_exception( p );
      }
      catch( const std::exception& e ) {
         // Do stuff with e
         try {
            std::rethrow_if_nested( e );
            p = std::exception_ptr();
         }
         catch( ... ) {
            p = std::current_exception();
         }
      }
   } while( p );
}
```

Inspecting Nested Exceptions

```
catch( ... ) {
   std::exception_ptr p = std::current_exception();
   do {
      try {
         std::rethrow_exception( p );
      }
      catch( const std::exception& e ) {
         // Do stuff with e
         try {
            std::rethrow_if_nested( e );
            p = std::exception_ptr();
         }
         catch( ... ) {
            p = std::current_exception();
         }
      }
   } while( p );
}
```

```
template< typename E >
[[nodiscard]] std::exception_ptr get_nested( const E& e )
{
   try {
      std::rethrow_if_nested( e );
      return std::exception_ptr();
   }
   catch( ... ) {
      return std::current_exception();
   }
}
```

Inspecting Nested Exceptions

```
catch( ... ) {
   std::exception_ptr p = std::current_exception();
   do {
      try {
         std::rethrow_exception( p );
      }
      catch( const std::exception& e ) {
         // Do stuff with e
         p = get_nested( e );
     }
   } while( p );
}
```

Inspecting Nested Exceptions

```
catch( ... ) {
   std::exception_ptr p = std::current_exception();
   do {
      try {
         std::rethrow_exception( p );
      }
      catch( const std::logic_error& e ) {
         // Do stuff with e
         p = get_nested( e );
      }
      catch( const std::runtime_error& e ) {
         // Do stuff with e
         p = get_nested( e );
      }
      catch( const std::exception& e ) {
         // Do stuff with e
         p = get_nested( e );
      }
   } while( p );
}
```

```
// Copyright (c) 2022 Dr. Colin Hirsch

#include <cstdint>
#include <exception>
#include <iostream>
#include <stdexcept>
#include <typeinfo>

namespace cpp
{
   template< typename... >
   struct rethrower;

   template<>
   struct rethrower<>
   {
      template< typename Processor, typename Caught, typename Visitor >
      static void rethrow( const Caught& caught, Visitor&& /*unused*/, const std::size_t /*unused*/ )
      {
         std::rethrow_if_nested( caught );
      }

      template< typename Processor, typename Visitor >
      static void rethrow( const std::exception_ptr& caught, Visitor&& /*unused*/, const std::size_t /*unused*/ )
      {
         std::rethrow_exception( caught );
      }
   };

   template< typename Exception, typename... Exceptions >
   struct rethrower< Exception, Exceptions... >
   {
      template< typename Processor, typename Caught, typename Visitor >
      static void rethrow( const Caught& caught, Visitor&& visitor, const std::size_t level )
      {
         try {
            rethrower< Exceptions... >::template rethrow< Processor >( caught, visitor, level );
         }
         catch( const Exception& exception ) {
            Processor::process( exception, visitor, level );
         }
      }
   };

   template< typename Rethrower >
   struct processor
   {
      template< typename Exception, typename Visitor >
      static void process( const Exception& exception, Visitor&& visitor, const std::size_t level )
      {
         Rethrower::template rethrow< processor >( exception, visitor, level + 1 );
         visitor( exception, level );
      }
   };

   template< typename... Exceptions >
   struct inspector
   {
      using Rethrower = rethrower< Exceptions... >;
      using Processor = processor< Rethrower >;

      template< typename Visitor >
      static void inspect( Visitor&& visitor )
      {
         Rethrower::template rethrow< Processor >( std::current_exception(), visitor, 0 );
      }
   };

}  // namespace cpp

int main()
{
   try {
      try {
         try {
            throw std::runtime_error( "runtime error" );
         }
         catch( ... ) {
            std::throw_with_nested( std::invalid_argument( "invalid argument" ) );
         }
      }
      catch( ... ) {
         std::throw_with_nested( std::logic_error( "logic error" ) );
      }
   }
   catch( ... ) {
      cpp::inspector< std::exception, std::runtime_error, std::logic_error >::inspect(
         []< typename T >( const T& e, const std::size_t level ) {
            std::cout << level << ": " << typeid( T ).name() << " " << typeid( e ).name() << " " << e.what() << std::endl;
      } );
   }
   return 0;
}
```

During the live presentation this source code was shown in an Emacs session; please copy to your editor of choice to read at a more decent size and with syntax highlighting.

Employed Standard Facilities

```
using std::exception_ptr = ...; // shared_ptr-like handle to an exception (object)


// Returns current "in-flight" exception (or copy), typically used in catch-block:
std::exception_ptr std::current_exception() noexcept;


// Throws t (or copy) as exception with std::current_exception() as nested exception:
template< class T >
[[noreturn]] void std::throw_with_nested( T&& t );


// If e "contains" a nested exception f then f is thrown again (or a copy):
template< class E >
void std::rethrow_if_nested( const E& e );


// Throws a previously captured exception again (or a copy):
[[noreturn]] void rethrow_exception( std::exception_ptr p );
```

Thank You!