Exceptions Demystified

Andreas Weis

BMW AG

CppNow 2019

About me

- ComicSansMS
- @DerGhulbus
- Co-organizer of the Munich C++ User Group
- Currently working as a Software Architect for BMW

Overview

- Implementing Exception Handling
- Working around problems with today's implementations
- Alternative mechanisms (static exceptions)

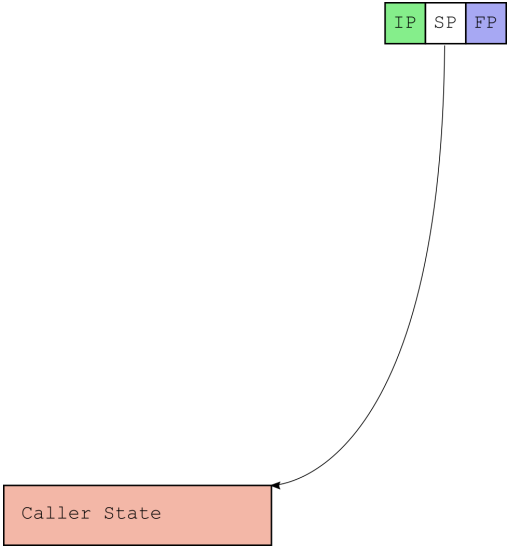
Why exceptions?

- They are at the heart of the error handling of many libraries, including the standard library
- Separating error handling code from the normal execution path is often desirable
- Handling errors originating in constructors or operators is difficult without them

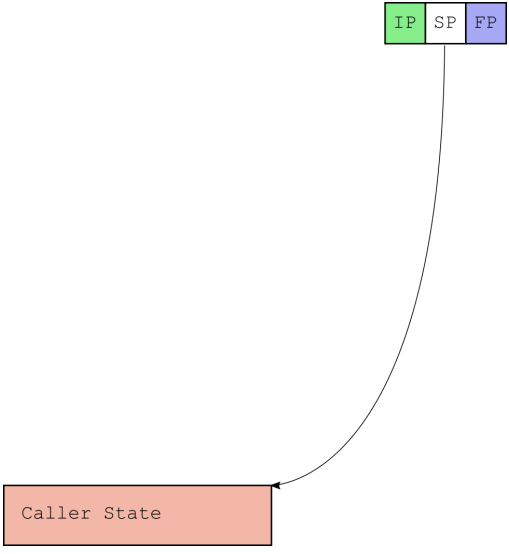
The goal

Make exceptions usable for everyone.

Anatomy of the call stack

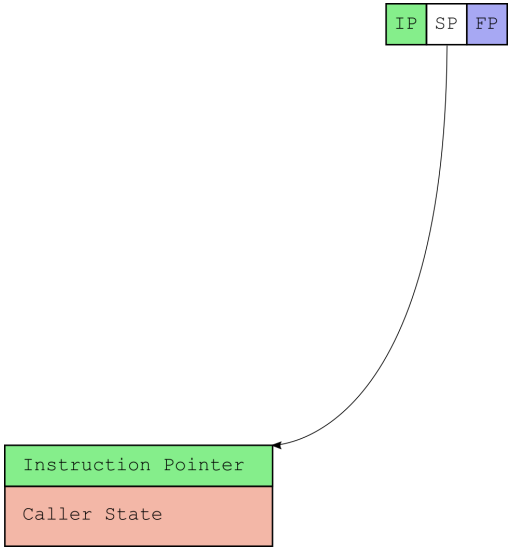


Anatomy of the call stack



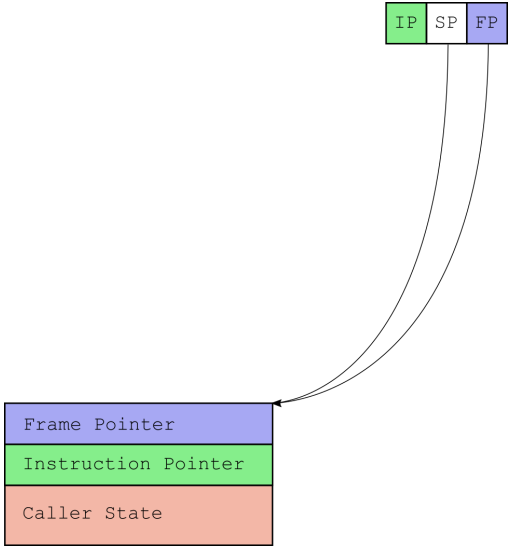

```
f(); <= IP
```

Anatomy of the call stack



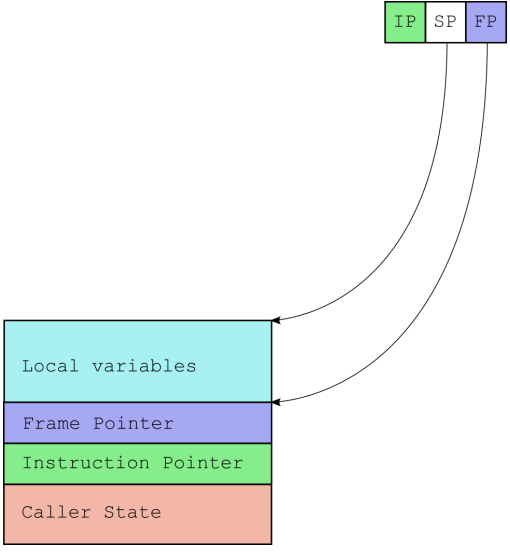

```
f(); <= IP
```

Anatomy of the call stack



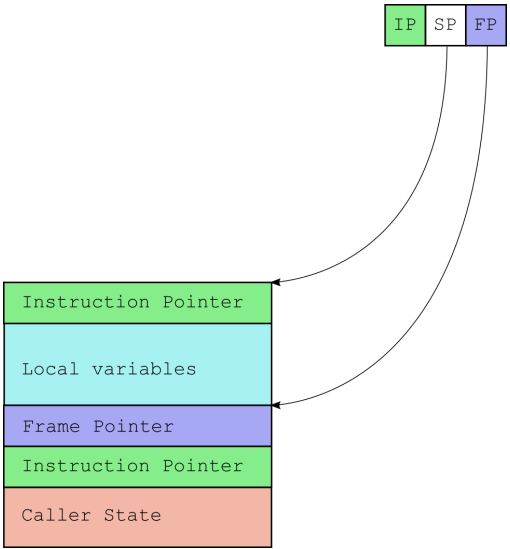

```
f(); <= IP
```

Anatomy of the call stack



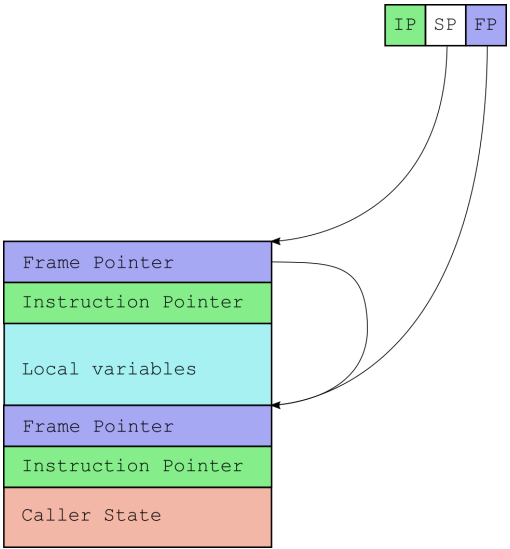

```
f();
void f() {
    int x, y, z;
    g(); <= IP
}
```

Anatomy of the call stack



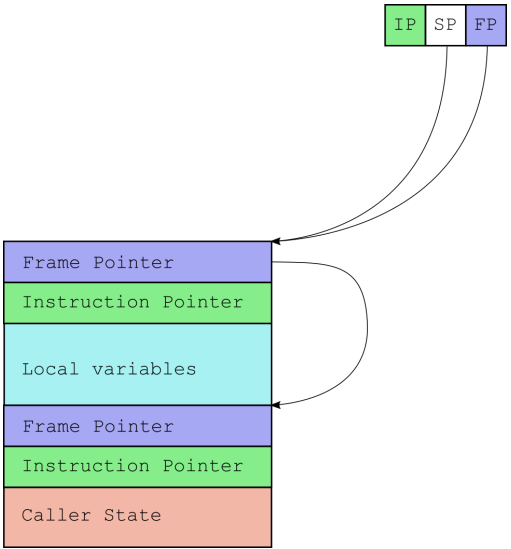

```
f();
void f() {
    int x, y, z;
    g(); <= IP
}
```

Anatomy of the call stack



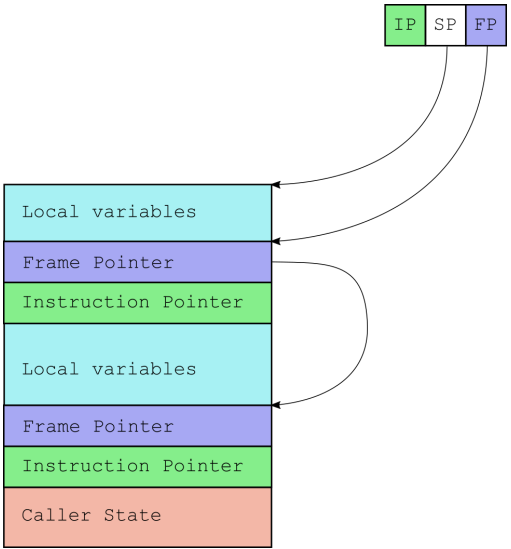

```
f();
void f() {
    int x, y, z;
    g(); <= IP
}
```

Anatomy of the call stack



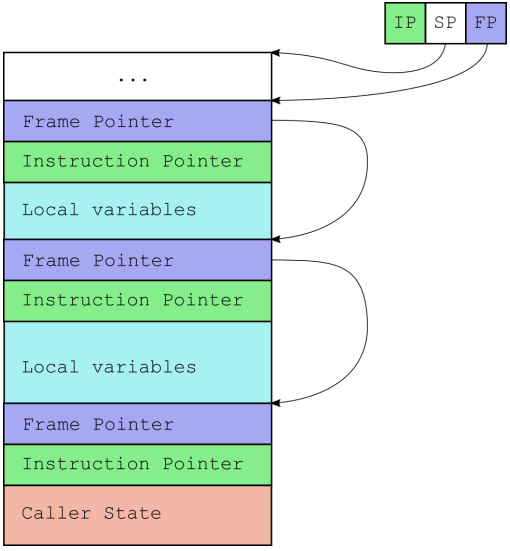

```
f();
void f() {
    int x, y, z;
    g(); <= IP
}
```

Anatomy of the call stack



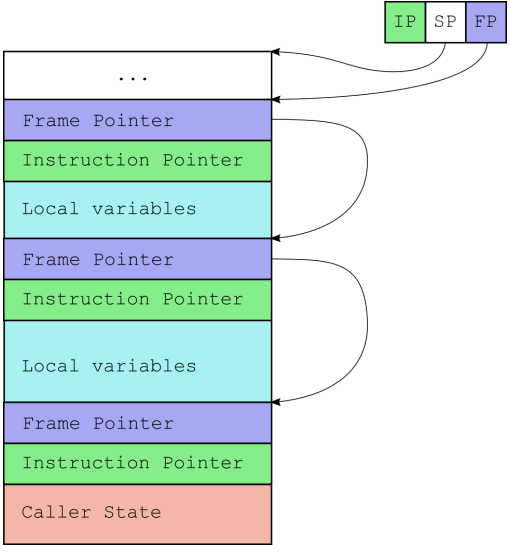

```
f();
void f() {
    int x, y, z;
    g();
}
void g() {
    int x; <= IP

}
```

Anatomy of the call stack




```
f();
void f() {
    int x, y, z;
    g();
}
void g() {
    int x;
    h(); <= IP
}
```

A few observations...



- The state of the current function is always located between the frame pointer and the stack pointer.
- The stack frames form a singly linked list that can be traversed by following the frame pointer
- When the size of the frame is static, the compiler may generate code that does not rely on the frame pointer to free up a register.

Lifetime of an exception

```
void g() {
    throw MyException{};  // starts here
}

void f() {
  try {
    g();
  } catch(MyException& e) {
  } // dies here
}
```

Responsibilities of `throw`

- Create the exception object
  - Memory for the exception is allocated in an unspecified way [except.throw]
  - As a consequence, there is no official customization point for this allocation
  - Many implementations today simply perform a heap allocation (even for a single `int`)
- Transfer control to the exception handler
  - The standard does not mention how this transfer of control is achieved
  - It does mandate though, that destructors must be invoked for objects along the path from `throw` to the handler $\Rightarrow$ *stack unwinding* [except.ctor]

Lifetime of an exception

```
void g() {
    throw MyException{};
}

void f() {
  try {
    g();
  } catch(MyException& e1) {
    try {
      g();
    } catch(MyException& e2) {
      // e1 and e2 are both alive in here
    }
  }
```

Lifetime of an exception

```
void g() {
    throw MyException{};
}

std::exception_ptr g_exc;

void f() {
  try {
    g();
  } catch(...) {
    g_exc = std::current_exception();
  }
}
```

Lifetime of an exception

```
void g() {
    throw MyException{};
}

std::vector<std::exception_ptr> g_excs;

void f() {
  try {
    g();
  } catch(...) {
    g_excs.push_back(std::current_exception());
  }
}
```

Storage of the exception object

- An unbounded number of exception objects can be alive at the same time
- Exception objects can be of arbitrary size. Allocate on the heap?
- `std::bad_alloc` is a common exception
- Stack?
- We still need to perform calls to destructors during unwinding (each of which can have their own, nested exceptions)
- Since C++11, an exception can outlive the handler (`exception_ptr`), so these for sure cannot be left on the stack

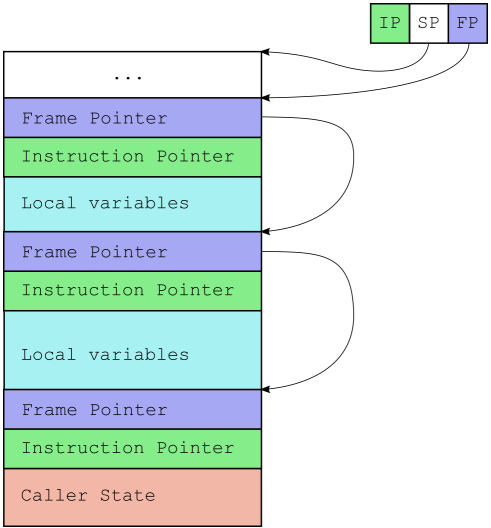
Customizing storage of the exception object

- Most implementations today simply invoke malloc for allocating storage for the exception object
- Replacing malloc can be an option for certain scenarios, requires patching the runtime
- Is it a problem to allocate exceptions from a fixed-size arena? It might be, as that turns each `throw` and `exception_ptr` use into a potential `terminate`
- In practice, we already have a similar situation today with `bad_alloc`

Responsibilities of `throw`

- Create the exception object
  - Memory for the exception is allocated in an unspecified way [except.throw]
  - As a consequence, there is no official customization point for this allocation
  - Many implementations today simply perform a heap allocation (even for a single `int`)
- Transfer control to the exception handler
  - The standard does not mention how this transfer of control is achieved
  - It does mandate though, that destructors must be invoked for objects along the path from `throw` to the handler $\Rightarrow$ *stack unwinding* [except.ctor]

Finding the right `catch`



- The stack frames form a linked list that can be iterated over
- Traverse the call stack up and check at each level if a matching catch handler is present
- Once a catch handler has been found, transfer control to that stack frame

What's the catch?

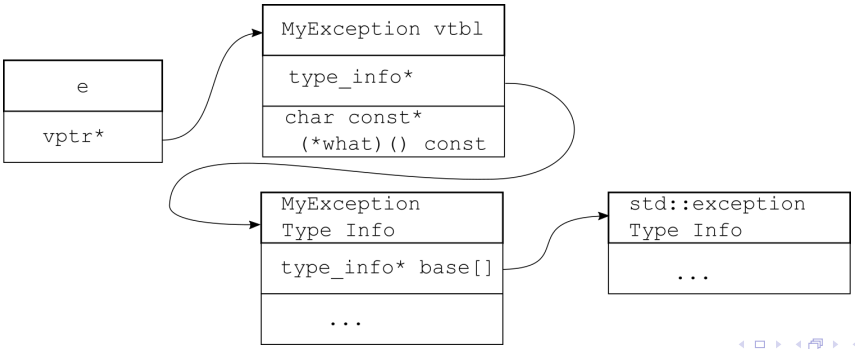
```
class MyException : public std::exception
 { /* ... */ };
void g() {
  throw MyException{};
}

void f() {
  try {
    g();
  } catch(std::exception& e) {
    printError(e.what());
  }
}
```

Finding the right `catch`

- Exception types are not required to match exactly
- Polymorphic catching of exceptions requires determining the types of all base classes of an object at runtime
- Exception class hierarchies are typically complex (one of the few places where multiple inheritance is still common)
- There exists a solution for this problem in C++ already: RTTI

Runtime Type Identification (RTTI)

```
class MyException : public std::exception
 { /* ... */ };
MyException e;
```



Problems with RTTI

- Traversing the Type Info structure is expensive
- The type info structures are open for extension - pulling in a shared library can increase cost of traversals in existing code
- Runtime analysis only possible if all types are known beforehand
- RTTI always brings a non-zero overhead for the binary size:
  - Type information is stored for all types, not just the ones participating in exception handling
  - The type info structure contains more information than is needed to find the matching catch handler

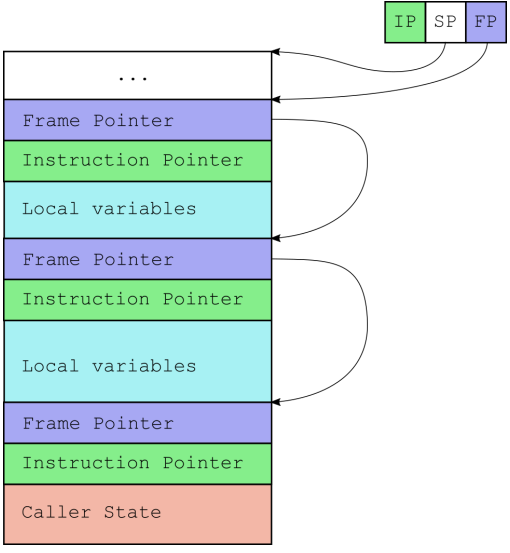
An implementation is not required to use RTTI for exception handling. But they typically do.

Unwinding the stack

The state of the program at any point is determined by

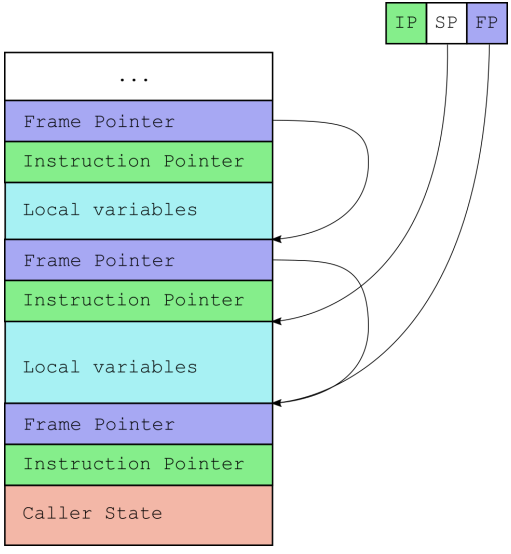
- The contents of its memory
- The values stored in its registers

Unwinding the stack



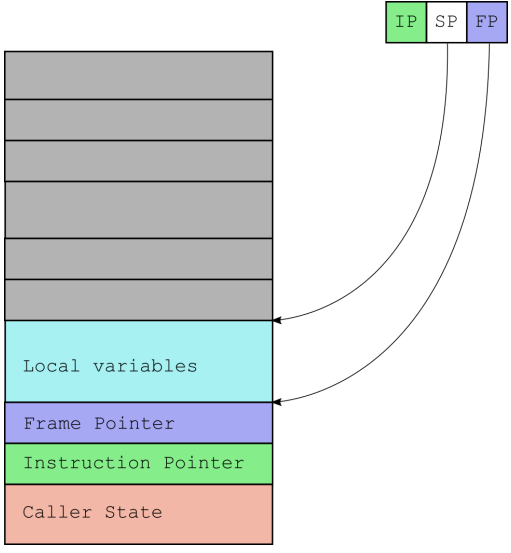

```
IP = old_IP;
SP = old_SP;
FP = old_FP;
```

Unwinding the stack



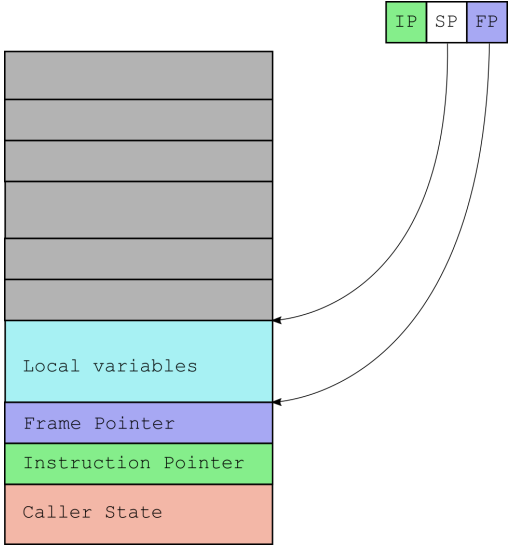

```
IP = old_IP;
SP = old_SP;
FP = old_FP;
```

Unwinding the stack




```
IP = old_IP;
SP = old_SP;
FP = old_FP;
```

Unwinding the stack




```
IP = old_IP;
SP = old_SP;
FP = old_FP;
```

Any problems with this solution?

What about destructors?

Frame-based Exception Handling

- The program code actively maintains a dynamic data structure for unwinding
- Has to keep track of active objects with non-trivial destructors and any enclosing exception handlers
- Compiler inserts code to update these data structures accordingly while stepping through the code

Drawbacks of this approach:

- Maintaining the data structure is costly and has to be done even if no exception ever occurs
- Binary code size increases significantly

What about destructors?

Table-based Exception Handling

- The set of active local objects at a certain point in code is known at compile time
- Compiler generates a table from code locations to active objects
- During unwinding, for each stack frame use the IP as a key for a lookup into the table
- If no exception occurs, no additional code for error handling is executed (zero-overhead exceptions)

Drawbacks of this approach:

- Binary size increases even more to store tables
- Table lookup for executing exceptions is very costly

Frame-based vs. table-based

- Frame-based has better performance when exceptions occur frequently
- Table-based has better performance if exceptions are rare
- Both have a significant impact on binary size, even if exceptions are never used (not zero-overhead)

Who is responsible for unwinding?

- In current implementations, this is the callee (ie. the `throw`-ing side)
- Can we shift the responsibility to the caller?
- Currently unwinding traverses the stack frames from an external piece of code and transfers control flow directly to the handler
- Instead transfer control with each step of the unwinding to the respective function that is being unwound?

Caller-driven unwinding

- Control-flow is close to normal function return, no jumping over multiple stack frames
- Requires inserting code at every call site to handle the exception case
- Generated code overhead is at worst equivalent to hand-written return code propagation
- But: Does only work if all functions along the call stack support this

Could be used to implement unwinding, but only in cases where backward compatibility is not an issue

The goal

Make exceptions usable for everyone.

Exceptions in restricted environments

- Memory may be limited (both RAM and ROM)
- Runtime must be predictable

Restrictions - Memory Consumption

Increase in binary size can be a killer for small controllers

- RTTI is too wasteful. Prefer a different mechanism that only stores the relevant information needed for exception handling
- Restrict the types that can be used as exception types
- Use a stack unwind mechanism that favors size over speed
- Get a bigger controller

Compilers so far don't seem to be very enthusiastic about supporting points 1 & 2. Realistically, if you want to use exceptions today, you need enough ROM to be able to deal with the size increase.

Restrictions - Dynamic Memory

Dynamic memory management may not be available or not acceptable due to real-time constraints

- Replace the memory management with a custom routine
- Restrict size of exception types and number of exception objects that can be active at the same time

The last point is difficult to enforce, even with tool support. It might require banning `exception_ptr`.

Restrictions - Runtime

Finding the matching `catch` clause has unpredictable runtime behavior

- Scanning of the type hierarchy using RTTI is difficult to reason about
- All exception types need to be known beforehand
- Polymorphism should be limited, ideally we only catch exceptions by their true dynamic type

Restricting Exception Types

- Polymorphism is used to allow catching exceptions at different levels of generalization
- If we disallow catching by base class, equivalent logic can be implemented manually
- In practice, how much is this feature actually used?
- Restricting the error type does not imply restrictions on the amount of information that can be transported (see Boost.Exception)
- Again in practice, how much information do we actually want to transport?

Worst-case execution time analysis

Real-time systems have to be able to analyze worst-case execution time

- For a given `throw` statement, how long does it take in the worst-case to reach an exception handler?
- Build a global call graph of the application, annotate each node with `throw`/`catch` information
- Find the paths from each `throw` to all possible `catches`
- Determine worst-case execution time for each path

Problematic cases for analysis

- Cycles in the call graph
- Indirect function calls
- Functions with no source code

Worst-case execution time - Fallback solution

What if we cannot guarantee a worst-case execution time through analysis?

- Treat each `throw` as a call to `terminate()` conceptually
- Establish a fallback path that ensures that the critical system invariants hold if a throw is encountered
- Instead of terminating the process directly, have the runtime trigger a grace period upon entering the unwind process
- If the unwind finishes within the grace period, the process will recover. Otherwise it will be terminated, either by the runtime itself, or by an external mechanism, upon expiry of the grace timer

Worst-case execution time - Fallback solution

How reasonable is it to treat `throws` as potential `terminate()`s?

- Developer needs to design two full error handling paths
- Exceptions can no longer be used for non-critical errors
- Allows easy transitioning to an `-fno-exceptions` mode
- Is good enough for most parts of the standard library

The story so far...

Making exceptions usable for everyone requires

- Restricting the types of exceptions that can be thrown and the total number of active exceptions
- Controlling the behavior of the dynamic memory allocation
- Knowledge of the complete type hierarchies for all types participating in exception handling
- Access to the complete source code of the program to perform execution time analysis (or sufficient tool support to allow this on closed source)

No good solution for implementing stack unwinding

A thought experiment

Imagine a code base with only `void` functions

- Allows use of the return channel of functions for error reporting
- A function can only return a single error type. Different errors are distinguished based on the return value, not the type
- Errors can be propagated up the stack as long as return values are convertible to the return type of the caller function
- Error propagation is done uniformly for every function call

Deterministic Exceptions (P0709)

Fundamental idea: Throw values instead of types
Signature of a function is changed to reflect that it can throw a deterministic exception

```
int f() throws {
  if(something_wrong()) {
    throw errc::oh_noes;
  }
  return 42;
}
```

New calling convention: `f()` returns a discriminated union of `int` and the error type.

Deterministic Exceptions (P0709)

```
int i;
try { i = f(); } catch(error e) {
  if(e == errc::oh_noes) {
    // handle error
  }
}
```

- Error is returned through the same channel as an ordinary return value ⇒ no storage problem
- Catching of the exceptions requires comparison of values, not of types ⇒ no RTTI or similar mechanisms required
- Unwinding is just the same as returning from the function ⇒ no additional data structures needed, no overhead compared to hand-written check of return codes

Deterministic Exceptions (P0709)

But we still get

- Automatic propagation of errors up the stack
- Separation of error handling from other program logic
- Ability to distinguish between different categories of exceptions (albeit with a slightly more verbose syntax)
- Interop with traditional exception handling mechanism

On many architectures this approach can be implemented more efficiently than traditional error return codes, by choosing a suitable ABI for the calling convention

Deterministic Exceptions (P0709)

Potential downsides compared to exceptions

- Transporting arbitrary amounts of information to the catching side is more difficult
- Limited backward-compatibility:
  - Signature change implies ABI break
  - Deterministic exceptions cannot be propagated through plain functions (interop required)
- Runtime overhead in absence of exceptions may be bigger than for table-based implementations

Deterministic Exceptions - Possible extensions

- Global error propagation handler (`set_error_propagation()`)
- Explicit `try` expressions for callers
- Arbitrary exception types (`throws{E}`)

Wrapping up...

- Exceptions pose a number of unsolved challenges still today
- Making exceptions usable in limited environments requires considerable effort and for some problems, it is questionable whether a solution can be found at all
- P0709 proposes an alternative approach based on values that gets rid of the problems while still preserving most of the important properties of exceptions
- Restrictions required by P0709 are not significantly worse than what users of traditional exceptions on restricted environments would have to do today

References

- Dave Watson – C++ Exceptions and Stack Unwinding (*CppCon 2017)*
- James McNellis – Unwinding the Stack: Exploring How C++ Exceptions Work on Windows (*CppCon 2018)*
- Herb Sutter – De-fragmenting C++: Making exceptions more affordable and usable (*ACCU 2019)*
- Phil Nash – The Dawn Of A New Error (*ACCU 2019)*
- **P0709** — Zero-overhead deterministic exceptions: Throwing values
- **P1640** — Error size benchmarking
- **P1095/N2289** — Zero overhead deterministic failure - A unified mechanism for C and C++

Thanks for your attention.




ComicSansMS /

@DerGhulbus