Runtime type introspection with `std::exception_ptr`

Aaryaman Sagar (aary@instagram.com)

February 7, 2018
Document #: p0933
Library Evolution Working Group

1. Introduction

Exceptions often manifest themselves via the type erased `std::exception_ptr` handle. Many of the relevant programming models and environments however, tend to not mix well with exceptions because of the limited capabilities of `std::exception_ptr`. And many platforms have even disabled exceptions entirely. This paper proposes adding RTTI features to `std::exception_ptr`, which will facilitate error introspection without having to go through the overhead of stack unwinding and exception propagation.

2. Prevalence

Current interfaces that aim to generalize asynchronous I/O hook into `std::exception_ptr` as a point of error propagation. This is true for example with the current `std::future` interface. Further with asychronous continuations this feature is going to be more widely used and implemented.

```
asynchronous_io().then([](std::future<Value> future) {
    try {
        cout << "Value " << future.get() << endl;
    } catch (std::runtime_error& err) {
        cerr << err.what() << endl;
    }
});
```

This interface wherein the exception is hidden behind the discriminated asynchronous monad is convenient but quickly degrades to bad performance because of the repeated stack unwinding with `std::exception_ptr`, this is especially true when exceptions propagate through several layers of chained callbacks. Even outside user code, implementations themselves might have to put a `try catch` block around the callbacks just for this purpose.

When an API has such drawbacks people look to either using some custom form of error propagation or coming up with their own interfaces and deem exceptions underperformant. As an example Facebook's folly futures, implement `onError` callbacks and their own `folly::exception_wrapper` to avoid some pathological inefficiencies with exceptions and `std::exception_ptr`

```
asychronous_io().then([](Value value) {
    cout << "Value " value << endl;
}).onError(std::runtime_error& err) {
    cerr << "Value " value << endl;
});
```

3. Current `std::exception_ptr` implementations

Current `std::exception_ptr` implementations contain mechanisms to fetch `std::type_info` object that corresponds to the type of the exception being pointed to. This can be found in the `libstdc++` implementation here

```
const std::type_info* __cxa_exception_type() const;
```

As far as I understand the Microsoft Visual C++ standard library has similar private functionality.

Given the existence of such introspection mechanisms, the first addition described in this proposal is to make the basic `type_info` method public

```
class exception_ptr {
public:
    // ...

    /**
     * Queries the exception_ptr for the type of the exception object
     * contained internally
     */
    const std::type_info& type() const noexcept;
};
```

Given the above RTTI extraction interface, it is natural to also include a way to fetch the underlying object itself. Without which the above would be simply a read-only operation.

```
class exception_ptr {
public:
    // ...

    /**
     * Queries the exception_ptr for the type of the exception object
     * contained internally
     */
    const std::type_info& type() const noexcept;

    /**
     * Returns a pointer to the contained exception object
     */
    const void* get() const noexcept;
};
```

The `get()` method above returns a `const void*` pointer instead of a `void*` because of an existing note in the standard recommending copying to avoid issues around data races (§[propagation]p7)

> (*Note*: If `rethrow_exception` rethrows the same exception object (rather than a copy), concurrent access to that rethrown exception object may introduce a data race. Changes in the number of `exception_ptr` objects that refer to a particular exception do not introduce a data race. - *end note*)

Returning a `const void*` forces callers to avoid unsafe concurrent mutations on the underlying exception object.

Existing implementations of `rethrow_exception` however do not adhere to this guideline and rethrow the same exception object. Whether or not this guideline still holds is better discussed in another paper. For the rest of this paper, `get()` returns a `const void*` (subject to change based on the conclusions around possible misinterpretations and/or misguided justifications of the above note)

4. Interoperability with `std::any`

C++17 also provided a convenient utility to generalize discriminated monadic storage - `std::any`. Both the implementations of `std::exception_ptr` and `std::any` allow fetching `std::type_info` objects for the underlying object or exception.

`std::any` provides access to an instance of any type, this is hidden behind a type erased interface. This closely resembles what exceptions do, the type of the exception is hidden behind the function until the information is made available as a part of the stack unwinding process. `std::exception_ptr` should provide a method to allow fetching of the discriminated instance as a `std::any`

```
class exception_ptr {
public:
    // ...

    /**
     * Queries the exception_ptr for the type of the exception object
     * contained internally
```

```
     */
    const std::type_info& type() const noexcept;

    /**
     * Returns a pointer to the contained exception object
     */
    const void* get() const noexcept;

    /**
     * Return an std::any object that contains a copy of the underlying stored
     * exception
     */
    std::any any() const;
};
```

This is simple and becomes a point of reusability for two separate interfaces that solve similar problems.

4.1. What about the note discussed in §[propagation]?

Like the discussion around `get()` for now, std::exception_ptr::any() returns an instance of `std::any` initialized with a copy of the underlying exception object. If the conclusion is that `exception_ptr` should provide mutable handles, then `std::exception_ptr::any()` should be modified to return an instance initialized with a pointer to the underlying exception object

4.2. Dealing with recursive exception propagation

The interaface must not return properly when an exception propagates while copying the underlying exception instance to prevent infinite exception recursion. So if a call to `std::exception_ptr::any()` causes an exception to be thrown from the underlying exception object, the implementation might throw a `std::bad_exception` object possibly causing abnormal program termination via `std::terminate`

5. Efficient representation

As the current proposal has been outlined a typical `std::exception_ptr` class is logically equivalent to a reference counted shared pointer to type erased discriminated storage - `std::shared_ptr<std::any>`. However this is just a logical representation. Implementations are free to strip away any unnnecesary indirections to make serialization to and from `std::exception_ptr` via `std::make_exception_ptr` and other `std::exception_ptr` instances performant.

Allowing `std::exception_ptr` instances to be aware of each other's internals also provides the bonus that we can now translate uniformly between different `std::exception_ptr` instances without having to go through the overhead of stack unwinding for RTTI extraction. This also means that we can now limit `std::exception_ptr` creation to a single dynamic storage allocation

6. Extracting the underlying exception

It natually follows that we need an efficient method of extracting the underlying exception from an `exception_ptr`

```
auto exception = exception_ptr.extract<std::runtime_error>();
```

This is implemented **as if** by

```
template <typename Exc>
std::optional<std::remove_cvref_t<Exc>>
exception_ptr::extract() const {
    try {
```

```
        std::rethrow_exception(*this);
    } catch(const Exc& err) {
        return err;
    } catch(...) {
        return std::nullopt;
    }
}
```

Where the underlying exception is copied (possibly more than once) and returned. The rules listed in [except.handle] apply.

6.1. Copies? What about the note discussed in §[propagation]?

If reference handles can and should be allowed from execption_ptr, the std::exception_ptr::extract method and hypothetical implementation should be appropriately modified

```
template <typename Exc>
std::optional<add_reference_wrapper_t<Exc>> exception_ptr::extract() const {
    try {
        std::rethrow_exception(*this);
    } catch(Exc err) {
        return err;
    } catch(...) {
        return std::nullopt;
    }
}
```

Note that this does not add an additional std::remove_cvref<T> to the return type to force a reference return. Users can be allowed to extract references to the underlying object by explicitly specifying an extract operation with a ref-qualified type. If the type is a refernece type then the returned optional is instantiated with a reference_wrapper indirection around the given type. For example

```
auto one = ptr.extract<std::runtime_error>();
auto two = ptr.extract<std::runtime_error&>();
```

std::exception_ptr would now look like this

```
class exception_ptr {
public:
    // ...

    /**
     * Queries the exception_ptr for the type of the exception object
     * contained internally
     */
    const std::type_info& type() const noexcept;

    /**
     * Returns a pointer to the contained exception object
     */
    const void* get() const noexcept;

    /**
     * Return an std::any object that contains a copy of the underlying stored
     * exception
     */
    std::any any() const;

    /**
     * Extracts a copy of the underlying stored exception, if an incompatible
     * type is passed, nullopt is returned
```

```
     */
    template <typename Exc>
    std::optional<std::remove_cvref_t<Exc>> extract() const;
};
```

7. Visitation with std::exception_ptr

Given that we have a mechanism to extract runtime type information from an exception_ptr we should have an efficient mechanism to handle errors without going through the overhead of stack unwinding with the same conditions as with regular exception handling. This should look and feel familiar to users

```
exception_ptr.handle(
    [&](std::runtime_error& exc) {
        cerr << exc.what() << endl;
    },
    [&](std::logic_error& exc) {
        cerr << exc.what() << endl;
    },
    [&](std::exception& exc) {
        cerr << exc.what() << endl;
    },
    [&](...) {
        std::terminate();
    });
```

This would need to follow the same rules as exception catching via catch clauses. The rules listed in [except.handle] apply. Here E is the type of the exception stored in the exception_ptr either via std::make_exception_ptr or via a call to std::current_exception() in the presence of exception propagation where E is std::remove_cvref_t<CE>, CE being the cv-ref qualified type of the exception in the current catch clause, or the type of the object initially thrown.

The handle clauses must be unary functions that accept a type E and return void. Polymorphic lambdas, functors with templated operator() methods or invocables accepting more than one argument (either templated or not) do not qualify as valid arguments and the resulting program is ill formed if those are passed.

If the catch-all clause is not included and none of the handlers are a good match for the exception as determined by the rules in [except.handle], std::terminate() is called.

std::exception_ptr would now look like this

```
class exception_ptr {
public:
    // ...

    /**
     * Queries the exception_ptr for the type of the exception object
     * contained internally
     */
    const std::type_info& type() const noexcept;

    /**
     * Returns a pointer to the contained exception object
     */
    const void* get() const noexcept;

    /**
     * Return an std::any object that contains a copy of the underlying stored
     * exception
     */
    std::any any() const;

    /**
```

```
     * Extracts a copy of the underlying stored exception, if an incompatible
     * type is passed, nullopt is returned
     */
    template <typename Exc>
    std::optional<std::remove_cvref_t<Exc>> extract() const;

    /**
     * Handles the exception as if by the same rules as normal exception
     * handling
     *
     * The HandleClauses clauses must be unary functions that accept one
     * cv-ref qualified argument and return void
     *
     * In the case where none of the handle clauses match, std::terminate() is
     * called
     *
     * A terminal closure or function that accepts elipses may be passed to
     * override the default behavior of std::terminate() being called in the
     * default case
     */
    template <typename... HandleClauses>
    void handle(HandleClauses&&... handle_clauses) const;
};
```

Given the `std::exception_ptr::extract` method, `std::exception_ptr::handle` performs **as if** implemented as so

```
template <typename Head, typename... Tail>
void handle(HeadClause& head, Tail&&... handle_clauses) const {
    // Failure case if applicable
    if constexpr (is_elipses_arg_type<Head>) {
        static_assert(
            sizeof...(Tail) == 0,
            "Ellipses handler must be last argument");
        head();
    }

    auto exception = this->extract<extract_arg_type_t<Head>&>();
    if (exception) {
        head(*exception);
        return;
    }
    this->handle(std::forward<Tail>(tail)...);
}
```

Of course implementations are highly encouraged not to cause repeated stack unwinding, as the premise of providing such introspection was to avoid repeated stack unwinding while still allowing multiplexing many different error types with the same exception

7.1. §[propagation]p7?

Whether or not the note interpretation holds as discussed in the above sections the hypothetical implementation stays the same. An additional copy will be created in the situation where the note interpretation holds because of the additional `std::remove_cvref` on the return type of `extract`. If it doesn't, a reference to the underlying exception will be passed