Runtime type introspection with `std::exception_ptr`

Aaryaman Sagar (aary@instagram.com)

February 7, 2018

1. Introduction

Exceptions manifest themselves in non-linear asynchronous programming via the `std::exception_ptr` handle. Asynchronous and non-linear programming models, however tend to not mix well with exceptions because of the limited capabilities of `std::exception_ptr`. This paper proposes adding RTTI features to `std::exception_ptr`, which will facilitate error introspection giving users more control of exceptions without having to go through the overhead of stack unwinding.

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, this is especially true when exceptions propagate through several layers of chained callbacks. Even outside user code, implemnetations themselves might have to put a `try catch` block around the callbacks just for this purpose.

When a core implementation has such drawbacks people will look to either using some custom form of error propagation or will come up with their own interfaces and deem exceptions underperformant. We should solve this problem for them and provide a well performing primitive that is already built into the language. 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;
```

[TODO - section about relevant portions of the itanium ABI]

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. Further there is another discriminated type std::variant that stores one of many types. While these interfaces solve similar problems, they have drastically different interfaces. This paper will go into depth about how to unify these interfaces and allow std::exception_ptr to become a well performing first choice primitive

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

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

    /**
     * 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. 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::remove_cvref_t<Exc> exception_ptr::extract() {
    try {
        std::rethrow_exception(*this);
    } catch(Exc& err) {
```

```
        return err;
    } catch(...) {
        std::rethrow_exception(*this);
    }
}
```

Where the underlying exception is copied and returned (possibly more than once). The rules listed in [except.handle] apply. If an exception was already propagating and an incompatible exception type template was passed to the extract method, as if to trigger the default catch clause std::terminate() is called

std::exception_ptr would now look like this

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

    /**
     * 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, std::rethrow_exception(*this) is called
     */
    template <typename Exc>
    Exc extract();
};
```

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

```
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.

std::exception_ptr would now look like this

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

    /**
     * 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, std::rethrow_exception(*this) is called
     */
    template <typename Exception>
    Exception extract();

    /**
     * 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, the exception is
     * rethrown as if via std::rethrow_exception(*this)
     *
     * A terminal closure or function that accepts elipses may be passed to
     * override the default behavior of rethrowing the exception
     */
    template <typename... HandleClauses>
    void handle(HandleClauses&&... handle_clauses);
};
```

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) {
    // Failure case if applicable
    if constexpr (is_elipses_arg_type<Head>) {
        static_assert(sizeof...(Tail) == 0);
        head();
    }

    // handle exceptions
    try {
        head(this->extract<extract_arg_type_t<Head>>(*this));
    } catch (...) {
        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

8. Dealing with data races

The current standard goes into depth about how to prevent data races when accessing the same exception object through an `std::exception_ptr` instance. This is why `std::exception_ptr::any()` `std::exception_ptr::extract` and

`std::exception_ptr::handle` should return copies of the underlying exception to user code. Making it safe for threads to concurrently access the same `exception_ptr` instance. (Provided that copy constructor implementations for the given objects don't introduce a race themsleves)