User-defined exception information and diagnostic information in exception objects

Document: P0640R0
Date: 2017-04-15
Project: Programming Language C++
Audience: Library Evolution Working Group
Authors: Emil Dotchevski, emildotchevski@gmail.com
Peter Dimov, pdimov@pdimov.com

I. Overview

This document proposes a new function template called `throw_with_info` to be used as an alternative to the `throw` keyword. It allows exception-neutral contexts to use a `catch(exception_info &)` to intercept any thrown exception and to store into it arbitrary additional data needed to handle it (the propagation of the original exception object can then resume by a `throw` without argument). This frees user-defined exception types from the burden of explicitly storing data needed to handle failures, which effectively decouples the data transported by exception objects from their C++ type. This accurately reflects the nature of such data, since it depends only on the context provided by the call stack at the point of the `throw` but not on the type of the failure that is being reported.

This proposal is a refinement of the Boost Exception library and incorporates valuable feedback from the Boost community.

II. Motivation

1. Reporting failures in low level libraries

Consider the following exception type:

```
class file_read_error: public std::exception {
  std::string file_name_;
  public:
  explicit file_read_error( std::string const & fn ): file_name_(fn) { }
  std::string const & file_name() const noexcept { return file_name_; }
};
```

A `catch` statement that handles `file_read_error` exceptions:

```
catch(file_read_error & e) {
  std::cerr << "Error reading \"" << e.file_name() << "\"\n";
}
```

Finally, a function that may throw `file_read_error` exceptions:

```
void read_file(FILE * f) {
  ....
  size_t nr=fread(buf,1,count,f);
  if(ferror(f))
    throw file_read_error(???); //File name not available here!
  ....
}
```

The issue is that the `catch` needs a file name, but at the point of the `throw` a file name is not available (only a `FILE` pointer is). In general the error might be detected in a library which can not assume that a meaningful name is available for any `FILE` it reads, even if a program that uses the library could reasonably make the same assumption.

Using `exception_info` a file name may be added to any exception after it has been thrown, while anything available at the point of the throw (e.g. `errno`) may be passed directly to `throw_with_info`:

```
class file_read_error: public std::exception { };
struct xi_file_name { typedef std::string type; };
struct xi_errno { typedef int type; };

void read_file(FILE * f) {
  ....
  size_t nr=fread(buf,1,count,f);
  if(ferror(f))
    std::throw_with_info(file_read_error(),
      std::exception_info().set<xi_errno>(errno));
  ....
}
```

```
void process_file(char const * name) {
  try {
    if(FILE * fp=fopen(name,"rt")) {
      std::shared_ptr<FILE> f(fp,fclose);
      ....
      read_file(fp); //throws using std::throw_with_info
      ....
    }
    else
      std::throw_with_info(file_open_error());
  }
  catch(std::exception_info & xi) {
    xi.set<xi_file_name>(name);
    throw;
  }
}
```

Now the final `catch` statement may look like this:

```
catch(file_io_error & e) {
  std::cerr << "I/O error!\n";
  std::exception_info const * xi=std::get_exception_info(e);
  assert(xi!=0); //In this program all file_io_error exceptions
                 //are thrown by std::throw_with_info.
  std::string const * fn=xi.get<xi_file_name>();
  assert(fn!=0); //In this program all files have names.
  std::cerr << "File name: " << *fn << "\n";
}
```

2. Decoupling error classification from related program-specific data

Because the C++ `catch` statement intercepts thrown exceptions based on their type, programs should `throw` different types to report different kinds of failures. Naturally, if exception types are organized in a hierarchy, programmers can `catch` a base type to intercept any exception that derives from it.

Using the proposed `throw_with_info`, exception types are no longer burdened with explicitly holding data members; reflecting the logical classification of failures in an exception class hierarchy is much simplified. For example, an I/O library may define the following hierarchy:

```
//Exception class hierarchy
struct io_error: std::exception { };
struct read_error: virtual io_error { };
struct write_error: virtual io_error { };
struct file_error: virtual io_error { };
struct file_read_error:  virtual file_error, virtual read_error  { };
struct file_write_error: virtual file_error, virtual write_error { };

//Tag types for storing data into any exception regardless of type
```

```
struct xi_user_name { typedef std::string type; };
struct xi_file_name { typedef std::string type; };
```

With this hierarchy in place, a `catch(io_error &)` would intercept any of the above exception types, a `catch(file_error &)` would intercept read or write file errors, while `catch(read_error &)` would intercept any read error, not only file read errors.

Independently, exception-neutral functions can store into any exception thrown by `throw_with_info` any data available to them that may be needed by a final `catch` to handle the error. Which data is relevant depends on the call stack at the point of the `throw`.

In the example below, a `file_name` is relevant to any exception that originates within `compute_file`, even exceptions that report failures that are not classified as “reading”, “parsing” or “computing” errors (for example, `std::bad_alloc`). The `user_name` is also relevant, but only when `compute_file` is called from `process_file` (note that the structure of the data that needs to be transported by exception objects is independent from and in general can not be reflected in the error classification defined by the exception type hierarchy):

```
void process_file(char const * user_name, char const * file_name) {
  try {
    login(user_name);
    compute_file(file_name);
    write_output();
  }
  catch(std::exception_info & xi) {
    xi.set<xi_user_name>(user_name);
    xi.set<xi_file_name>(file_name);
    throw;
  }
}

void compute_file(char const * file_name) {
  try {
    shared_ptr<FILE> f=file_open(file_name);
    compute(parse(f));
  }
  catch(std::exception_info & xi) {
    xi.set<xi_file_name>(file_name);
    throw;
  }
}
```

III. Impact

This proposal extends the standard library, adding a new standard header `<exception_info>` without affecting existing ABIs and requires no new language features.

IV. Proposed text

1. Header <exception_info> synopsis

```
namespace std {
  template <class E> [[noreturn]]
  void throw_with_info(E && e, exception_info && xi = exception_info());

  template <class E> [[noreturn]]
  void throw_with_info(E && e, exception_info const & xi);

  template <class E>
  exception_info const * get_exception_info(E const & e);

  template <class E>
  exception_info * get_exception_info(E & e);

  template <class E>
  string exception_diagnostic_info(E const & e);

  string exception_diagnostic_info(exception_ptr const & p);

  class exception_info;

  struct xi_file_name;
  struct xi_file;
  struct xi_fileno;
  struct xi_errno;
  struct xi_api_function;
}
```

2. throw_with_info

```
template <class E> [[noreturn]]
void throw_with_info(E && e, exception_info && xi = exception_info());

template <class E> [[noreturn]]
void throw_with_info(E && e, exception_info const & xi);
```

Requires: `E` shall be a valid base class and shall not derive from `exception_info`, or the program is ill-formed.

Effects: Throws an exception of unspecified type that derives publicly and non-virtually from both `E` and `exception_info`. The `E` subobject of the exception object is initialized with `std::forward<E>(e)`. The `exception_info` subobject of the exception object is initialized with `std::move(xi)` for the first overload and `xi` for the second.

[Note: Implementations are encouraged to capture `typeid(E)` and store it into the `exception_info` subobject of the exception object, for later use in `exception_info::diagnostic_info`.*—end note]*

3. get_exception_info

```
template <class E>
exception_info const * get_exception_info(E const & e);

template <class E>
exception_info * get_exception_info(E & e);
```

Requires: `E` shall be polymorphic, or the program is ill-formed.

Returns: If the dynamic type of `e` derives from `exception_info`, returns a pointer to the `exception_info` subobject of `e`. Otherwise returns 0.

4. exception_diagnostic_info

```
template <class E>
string exception_diagnostic_info(E const & e);
```

Returns: A string of unspecified format that contains human-readable technical diagnostic information about `e`.

Throws: `bad_alloc` or any exception thrown by `exception_info::diagnostic_info`.

Remarks:

- If `E` is not polymorphic, implementations are encouraged to include in the returned string the type name of `E`.
- If `E` is polymorphic, implementations are encouraged to include in the returned string the type name of the dynamic type of `e`.
- If `E` is polymorphic and the dynamic type of `e` derives from `class exception`, implementations are encouraged to include in the returned string the string returned by `exception::what`, called at the time of the call to `exception_diagnostic_info`.
- If `E` is polymorphic and `xi` is the pointer returned from `get_exception_info(e)`, if `xi!=0`, implementations are encouraged to include in the returned string the value returned from `xi->diagnostic_info()`, called at the time of the call to `exception_diagnostic_info`.

[Example:

```
Dynamic exception type: struct file_open_error
what: example_io error
example_io.cpp(70): throw_with_info in function class std::shared_ptr<FILE>
open_file(const char *,const char *)
std::xi_api_function = fopen
std::xi_errno = 2, "No such file or directory"
std::xi_file_name = "tmp1.txt"
```

—end example]

```
string exception_diagnostic_info(exception_ptr const & p);
```

Returns: A string of unspecified format that contains human-readable technical diagnostic information as if by calling the `exception_diagnostic_info` function template with the exception object contained in `p`, or an empty string if `p` is empty.

Throws: `bad_alloc` or any exception thrown by `exception_info::diagnostic_info`.

5. Class exception_info

```
namespace std {
  class exception_info {
  public:

    exception_info() noexcept;
    exception_info(char const * file, int line, char const * function=0)
noexcept;

    exception_info(exception_info const & r);
    exception_info(exception_info && r) noexcept;

    virtual ~exception_info() noexcept;

    exception_info & operator=(exception_info const & r);
    exception_info & operator=(exception_info && r) noexcept;

    char const * file() const noexcept;
    int line() const noexcept;
    char const * function() const noexcept;

    template <class Tag> exception_info & set(typename Tag::type const & x);
    template <class Tag> exception_info & set(typename Tag::type && x);

    template <class Tag> exception_info & unset() noexcept;

    template <class Tag> typename Tag::type const * get() const noexcept;
    template <class Tag> typename Tag::type * get() noexcept;

    string diagnostic_info() const;
  };
}
```

The class `exception_info` provides storage for objects of arbitrary `CopyConstructible` types. The same `exception_info` object can store objects of different types.

To store an object into an `exception_info` object, instantiate the `set` member function template with a user-defined `Tag` type and pass the object to be stored. `Tag` types are required to define a member type called `type` which specifies the type of the stored object.

To access a stored object, instantiate the `get` member function template with the `Tag` type used with the `set` member function template when the object was stored.

```
exception_info() noexcept;
```

Postconditions: `file()==0 && line()==0 && function()==0`. `get<Tag>()==0` for any `Tag`.

```
exception_info(char const * file_, int line_, char const * function_=0)
noexcept;
```

Postconditions: `file()==file_ && line()==line_ && function()==function_`. `get<Tag>()==0` for any `Tag`.

[Example:

```
std::throw_with_info(std::out_of_range("Invalid argument"),
  std::exception_info(__FILE__,__LINE__,__func__));
```

—end example]

```
exception_info(exception_info const & r);
```

Effects: Copies `r` into `*this`. Each value stored into `r` by `set` is copied into `*this`.

Postconditions: `file()==r.file() && line()==r.line() && function()==r.function()`.

Throws: `bad_alloc` or any exception thrown while copying the values stored in `r` by `set`.

Remarks: `*this` and `r` do not share state.

```
exception_info(exception_info && r) noexcept;
```

Effects: Moves the state of `r` into `*this`. `r` is left empty, as if default constructed.

```
virtual ~exception_info() noexcept;
```

Note: `exception_info` is polymorphic.

```
exception_info & operator=(exception_info const & r);
```

Effects: Replaces `*this`'s state with `r`. The values stored into `*this` by `set` are destroyed and each value stored into `r` by `set` is copied into `*this`.

Postconditions: `file()==r.file() && line()==r.line() && function()==r.function()`.

Throws: `bad_alloc` or any exception thrown while copying the values stored in `r` by `set`.

Remarks: `*this` and `r` do not share state after the assignment.

```
exception_info & operator=(exception_info && r) noexcept;
```

Effects: Initializes a temporary `tmp` of type `exception_info` with `move(r)`, destroys the old state of `*this` and moves the state of `tmp` into it. `r` is left empty, as if default constructed.

```
char const * file() const noexcept;
int line() const noexcept;
char const * function() const noexcept;
```

Returns: The `file`, `line` and `function` arguments passed to `exception_info`'s constructor. In case `*this` was initialized by the default constructor, `file()` returns 0, `line()` returns 0 and `function()` returns 0.

```
template <class Tag>
exception_info & set(typename Tag::type const & x);

template <class Tag>
exception_info & set(typename Tag::type && x);
```

Requires: `x` shall be `CopyConstructible`, or the program is ill-formed.

Effects: `x` is stored into `*this` and can be accessed by `get<Tag>`. If `*this` already has a value stored under the specified `Tag`, the original value is overwritten.

Returns: `*this`.

Postconditions: `get<Tag>()` returns a pointer to the stored copy of `x`.

Throws: May throw `bad_alloc` or any exception thrown by `Tag::type`'s copy or move constructor.

[Example 1:

```
std::shared_ptr<FILE> file_open(char const * file_name) {
  if(FILE * f=fopen(file_name,"r"))
    return std::shared_ptr<FILE>(f,&fclose);
  else
    std::throw_with_info(file_open_error(),std::exception_info()
      .set<std::xi_api_function>("fopen")
      .set<std::xi_file_name>(file_name)
      .set<std::xi_errno>(errno));
}
```

—end example]

[Example 2:

```
void process_file(char const * file_name) {
  std::shared_ptr<FILE> f=file_open(file_name);
  try {
    read_file(f.get());
    do_work();
  }
  catch(exception_info & xi) {
    xi.set<std::xi_file_name>(file_name);
    throw;
  }
}
```

—end example]

```
template <class Tag>
exception_info & unset() noexcept;
```

Effects: If `*this` contains an object at `Tag` (of type `Tag::type`, see `set<Tag>`), the stored object is removed. Otherwise `unset` has no effect.

Returns: `*this`.

Postconditions: `get<Tag>()` returns 0.

```
template <class Tag>
typename Tag::type const * get() const noexcept;

template <class Tag>
typename Tag::type * get() noexcept;
```

Returns: If `*this` contains an object at `Tag` (of type `Tag::type`, see `set<Tag>`), `get<Tag>` returns a pointer to the stored object; otherwise it returns 0.

Remarks: Destroying the `exception_info` object or calling any instance of the `set` or `unset` member function templates invalidates the returned pointer.

[Example:

```
try {
  do_work();
}
catch(file_read_error & e) {
  std::exception_info const * xi=std::get_exception_info(e);
  assert(xi!=0); //In this program all file_read_error exceptions
                 //are thrown using std::throw_with_info.
  std::string const * fn=xi->get<std::xi_file_name>();
  assert(fn!=0); //In this program all file_read_error exceptions
                 //contain std::xi_file_name at this point.
  std::cerr << "Error reading \" << *fn << "\"";
  if(int const * err=xi->get<std::xi_errno>())
    std::cerr << "(OS says \" << strerror(*err) << "\")";
  std::cerr << '\n';
}
```

—end example]

```
string diagnostic_info() const;
```

Returns: A string of unspecified format that contains human-readable technical diagnostic information about `*this`.

Throws: May throw `bad_alloc` or any exception thrown in the attempt to convert to string any of the objects stored into `*this` by the `set` member function template.

Remarks:

- If available, implementations are encouraged to include in the returned string the `file`, `line` and `function` arguments passed to `exception_info`'s constructor. The formatting may match the format of compile-time diagnostic messages.
- At the time `diagnostic_info` is called, implementations may convert to string any of the objects stored in `*this` by `set`, by calling a suitable `operator<<` overload that takes `ostream` object on the left and `Tag::type` object on the right, bound at the point of instantiation of `set<Tag>`. Implementations are not allowed to issue a diagnostic if a suitable overload could not be bound.
- At the time `diagnostic_info` is called, objects for which a suitable `operator<<` overload could not be bound may be converted to string if a different reasonable string conversion is possible. For example, if an `operator<<` overload suitable for converting objects of type `T` can be bound, a `vector<T>` may be converted to string by calling that overload for each element of the vector and concatenating the results. In such cases implementations should limit the maximum number of the included elements (for performance and space reasons).
- At the time `diagnostic_info` is called, objects for which no suitable string conversion could be bound, implementations may use a generic string conversion (e.g. a partial hex dump of the stored object's memory) or substitute a stub string.
- At the time `diagnostic_info` is called, objects of type `exception_ptr` stored in this by `set` may be converted to string by nesting the result of calling `exception_diagnostic_info` with the object they point to.
- Implementations are encouraged to pair each converted to string object with a string representation of its tag type, for example by means of `typeid(Tag).name()`, and to include both in the returned string.
- Implementations may include in the returned string any other relevant information, such as the (partial or even single-level) stack trace collected at the point of the call to `throw_with_info`.
- Even if the dynamic type of `*this` derives from `class exception`, implementations must not include in the returned string the string returned by `exception::what`.
- Implementations may limit the total size of the returned string for performance and/or space reasons.

[Example:

```
example_io.cpp(70): throw_with_info in function class std::shared_ptr<FILE>
open_file(const char *,const char *)
std::xi_api_function = fopen
std::xi_errno = 2, "No such file or directory"
std::xi_file_name = "tmp1.txt"
```

—end example]

6. Standard tag types

The following tag types are suitable for instantiating the `set` and `get` member function templates of class `exception_info`.

```
struct xi_file_name { typedef string type; };
```

Specifies a relevant file name for exceptions used to report file errors, using UTF-8 encoding.

```
struct xi_file { typedef FILE * type; };
struct xi_fileno { typedef int type; };
```

These types can be used to specify a relevant file descriptor or `FILE` pointer in exceptions used to report file errors. One possible use case is in contexts that operate on more than one file, to determine the correct file name to pass to `set<xi_file_name>` depending on the file information reported by a lower level function.

[Example:

```
std::shared_ptr<FILE> f1=file_open("f1.txt");
std::shared_ptr<FILE> f2=file_open("f2.txt");
try {
  compare_files(f1.get(),f2.get());
}
catch(std::exception_info & xi) {
  if(FILE * const * xf=xi.get<std::xi_file>()) {
    if(*xf==f1.get()) {
      xi.unset<std::xi_file>();
      xi.set<std::xi_file_name>("f1.txt");
    }
    else if(*xf==f2.get()) {
      xi.unset<std::xi_file>();
      xi.set<std::xi_file_name>("f2.txt");
    }
  }
  throw;
}
```

—end example]

```
struct xi_errno { typedef int type; };
```

Specifies a relevant `errno` code. Implementations are encouraged to recognize `xi_errno` in `exception_info::diagnostic_info` and convert the numerical value to string using an appropriate conversion routine, for example `strerror`.

```
struct xi_api_function { typedef char const * type; };
```

Used when throwing exceptions in case a call to a no-throw API function fails, to indicate the name of that function.

[Example:

```
fread(ptr,size,count,f);
if(ferror(f))
  std::throw_with_info(file_error(),std::exception_info()
    .set<std::xi_api_function>("fread")
    .set<std::xi_errno>(errno));
```

—end example]

V. Implementability

A conforming implementation based on Boost Exception is available in Boost. Pull the `exception_info` branch of https://github.com/boostorg/exception.git and https://github.com/boostorg/throw_exception.git.