# Boost Exception

Emil Dotchevski

emil@revergestudios.com

## Background

- Error handling strategies
  - “Nothing can go wrong, ever”
  - “All hell breaks loose anyway, just die now”
  - “Let someone else worry about it”
    - Error codes
    - Exceptions

## Background

- Why use exceptions?
  - Separating error-handling code
    - try / catch
  - Discriminating on error types
    - catch(foo) / catch(bar)
    - exception types hierarchy
  - Propagating errors up the call stack
    - most contexts in a program can't handle errors

## Background

- Why use exceptions?
  - Separating error-handling code
    - try / catch
  - Discriminating on error types
    - catch(foo) / catch(bar)
    - exception types hierarchy
  - Propagating errors up the call stack
    - most contexts in a program can't handle errors

## Background

- Exception safety
  - What happens with invariants when exceptions are thrown or propagated?
  - Even if you do not use exceptions you should think about invariants when reporting errors.
- Neutrality
  - If a (generic) context is not handling an error, it should not interfere.

## The “traditional” exception handling approach

- The throw site
  - creates an exception object of appropriate type
  - stuffs it with data relevant to the detected error
- The catch site
  - selects failures based on exception types
  - inspects exception objects for data required to deal with the problem

## Problem: no file name!

- The catch site:

```
catch( file_read_error & e )
{
    std::cerr << e.file_name();
}
```

- The throw site:

```
void read_file( FILE * f )
{
    ....
    size_t nr=fread(buf,1,count,f);
    if( ferror(f) )
        throw file_read_error(???);
    ....
}
```

## Pass the file name to read_file()?

- Presumably, read_file() may take a file name:

```
void read_file( FILE * f, char const * name )
{
    ....
    size_t nr=fread(buf,1,count,f);
    if( ferror(f) )
        throw file_read_error(name);
    ....
}
```

- Issues:
  - Will the immediate caller have a file name?
  - Can a library designer reasonably provide all data required by a given application to handle library exceptions?

## The “traditional” exception handling approach is flawed

- The throw site
  - creates an exception object of appropriate type
  - stuffs it with data relevant to the detected error
- The catch site
  - selects failures based on exception types
  - inspects exception objects for data required to deal with the problem

## Solution: wrapping

- To wrap an exception object, we must copy it!
  - in general requires “cloning” (or may slice)
- Interferes with exception neutrality
  - practically impossible in generic components

## The Boost way

- Simply derive your exception types from boost::exception.
- Confidently limit the throw site to provide only data that is available naturally.
- Use exception-neutral contexts between the throw and the catch to augment exceptions with more relevant data as they bubble up.

## Example: the throw

```
void read_file( FILE * f )
{
    ....
    size_t nr=fread(buf,1,count,f);
    if( ferror(f) )
        throw file_read_error() << errno_code(errno);
    ....
}
```

## Example: adding the file name

```
try
{
    if( FILE * fp=fopen(“foo.txt”,”rt”) )
    {
        shared_ptr<FILE> f(fp,fclose);
        ....
        read_file(fp); //throws types deriving from boost::exception
        do_something();
        ....
    }
    else
        throw file_open_error() << errno_code(errno);
}
catch( boost::exception & e )
{
    e << file_name(“foo.txt”);
    throw;
}
```

## Example: the catch

```
catch( io_error & e )
{
    std::cerr << “I/O Error!\n”;

    if( std::string const * fn=get_error_info<file_name>(e) )
        std::cerr << “File name: “ << *fn << “\n”;

    if( int const * c=get_error_info<errno_code>(e) )
        std::cerr << “OS says: “ << strerror(*c) << “\n”;
}
```

## Deriving from boost::exception

- Typically you can add it as a base of types that derive from std::exception directly:

```
struct exception_base:                   struct exception_base:
    virtual std::exception       ──►         virtual std::exception,
{                                ──►         virtual boost::exception
};                                       {
                                         };
```

- All other exception types can also be free of any members:

```
struct io_error: virtual exception_base { };
struct file_error: virtual exception_base { };
struct file_open_error: virtual io_error, virtual file_error { };
struct file_read_error: virtual io_error, virtual file_error { };
```

## boost::enable_error_info()

- Use when exception types definitions can't be modified to add boost::exception as a base.
- Call directly in the throw statement:

```
throw enable_error_info(std::runtime_error(“Error!”)) << more_info(....);
```

- Above, the returned object is of unspecified type that can be caught as boost::exception or std::runtime_error.

## boost::error_info

- Used to define things like errno_code and file_name in header files:

```
namespace boost { template <class,class> class error_info; }

typedef boost::error_info<struct tag_errno,int> errno_code;
typedef boost::error_info<struct tag_file_name,std::string> file_name;
```

- Type-safe
- The data types do not need a no-throw copy constructor
- Provides a nested value_type for use with generic components

## boost::get_error_info()

- Returns a (possibly null) pointer of the correct type:

```
catch( boost::exception & e )
{
    std::string const * fn=get_error_info<file_name>(e);
    ....
}
```

- The returned pointer becomes invalid when the exception object is destroyed...
- ...but the type of the pointee has an accessible copy constructor, so you can copy it.

## boost::throw_exception()

- The new (as of 1.37) behavior:
  - same as the old behavior if the passed type derives from boost::exception
  - if not, enable_error_info is used to inject boost::exception as a base anyway
- This allows users to add error_info to most exceptions emitted by Boost libraries.

## BOOST_THROW_EXCEPTION

- Function, file, line information

```
typedef error_info<struct tag_throw_function,char const *> throw_function;
typedef error_info<struct tag_throw_file,char const *> throw_file;
typedef error_info<struct tag_throw_line,int> throw_line;
```

- Definition:

```
#define BOOST_THROW_EXCEPTION(e)\
    ::boost::throw_exception( ::boost::enable_error_info(e) <<\
    ::boost::throw_function(BOOST_CURRENT_FUNCTION) <<\
    ::boost::throw_file(__FILE__) <<\
    ::boost::throw_line(__LINE__) );
```

## boost::diagnostic_information()

- Typical use:

```
catch( boost::exception & e )
{
    std::cerr << "OMG!" << diagnostic_information(e);
}
catch( ... )
{
    std::cerr << "OMG!";
}
```

- A possible output:

```
example_io.cpp(83): Throw in function void parse_file(const char *)
Dynamic exception type: class file_open_error
std::exception::what: example_io error
[struct tag_errno_code *] = 2, OS says "No such file or directory"
[struct tag_file_name *] = tmp1.xml
[struct tag_function *] = fopen
[struct tag_open_mode *] = rb
```

## Transporting of exceptions between threads (N2179)

- Support for:
  - exception_ptr
  - current_exception()
  - copy_exception()
  - rethrow_exception()
- Requires enable_current_exception()
- However, enable_current_exception() is integrated in boost::throw_exception().

## More information

- Formal documentation:

  http://www.boost.org/doc/libs/release/libs/exception/doc/boost-exception.html

- Boost users or Boost developers mailing lists:

  http://lists.boost.org/mailman/listinfo.cgi/boost-users
  http://lists.boost.org/mailman/listinfo.cgi/boost

- Ask questions now