Demystifying C++ Exceptions

10 Tips to Effectively Design Exception-Safe Code

Benefits of Using Exceptions

- Exceptions are used in many common C++ libraries
- Exceptions separate error-handling code from the normal program flow thus making code more readable, robust and extensible
- Exceptions are difficult to ignore
- Exceptions are the cleanest way to report errors from constructors and operators
- Exceptions often carry more information than error codes
- Exceptions are easily propagated from deeply nested functions
- Return values become free for purposes other than reporting errors

Challenges of Using Exceptions

- Writing error-safe code with or without the use of exceptions is challenging
- Exceptions are difficult to ignore
- Exceptions create invisible exit-points from methods
- Exceptions can easily lead to resource leaks and unstable objects
- Exceptions are easily abused
- Developers must learn how to write exception safe code
  - Many articles and recommendations on writing exception-safe code is daunting to a large percentage of developers
  - There is a lack of a complete set of guidelines for writing exception-safe code along with a comprehensive library that pulled all the required idioms and helper-code together.

My Primary Goal Today

Show that using C++ Exceptions can be Practical for a Majority of Developers and Companies who develop code in C++

How I Plan to Accomplish this Goal

- Discuss levels of exception-safety and propose a level that is practical for many companies and developers
- Provide a comprehensive set of guidelines for writing exception-safe code
- Discuss idioms that are particular useful for writing exception-safe code
- Discuss a helper library along with specific recommendations for other libraries that make writing exception safe code significantly less challenging along with adding additional benefits to using exceptions

Exception Guarantees

(re-defined with slight modifications)

- No Guarantee

  Component throwing exception (possibly even the system) may be in an unstable state and no actions may be safely performed on the component (including destroying or resetting).

- Basic Guarantee

  Invariants of a component throwing exceptions are preserved, no resources are leaked and the system is in a stable state.

  Note: There is no guarantee about the components state other than it is safe to use after the exception. Therefore, the options for recovery are generally limited to destruction or resetting the component to some known state before use.

- Strong Guarantee

  An operation will either completely succeed or all actions performed will be rolled-back to the state the system was in before calling the failing operation.

- No-Throw Guarantee

  An operation will not throw an exception. Often implies that the operation is guaranteed to succeed.

Current Guidelines for Exception-Safe Code

(with respect to Exception Guarantees)

"Exception-safe code must offer one of the three guarantees (basic, strong, no-throw)" and "As a general rule, you want to offer the strongest guarantee that's practical".

- Effective C++ by Scott Myers

"A function should always support the strictest guarantee that it can without penalizing users who don't need it."

- GotW #82 by Herb Sutters

"The basic guarantee is a simple minimum standard for exception-safety to which we can hold all components."

- Exception-Safety in Generic Components by David Abrahams

"This (basic guarantee) is the basic level you should strive for in all code."

- ACCU Overload Journal #86 by Anthony Williams

Exception Guarantees Expanded

(re-defined with slight modifications)

- No Guarantee

  Component throwing exception (possibly even the system) may be in an unstable state and no actions may be safely performed on the component (including destroying or resetting).

- Minimal (Destructible) Guarantee

  Component throwing exception may store invalid data but can be safely destroyed or reset (including through assignment) without causing a crash or leaking resources and that the system is in a stable state.

- Basic Guarantee

  Adds to the minimal guarantee the requirement that the component is still safe to use (invariants are preserved).

  Note: There is no guarantee about the components state.

- Strong Guarantee

  An operation will either completely succeed or all actions performed will be rolled-back to the state the system was in before calling the failing operation.

- No-Throw Guarantee

  An operation will not throw an exception. Often implies that the operation is guaranteed to succeed.

TIP #1 - Question the Current Wisdom

- Don’t always worry about the basic guarantee, just make sure to provide the minimal guarantee.
  - Since the basic guarantee doesn't guarantee the state of an object, current guidelines suggest you destroy or reset the component which is the same actions required by the minimal guarantee.
- In many situations and code, don’t worry about striving to achieve the strong or no-throw guarantee.
  - This can be difficult to achieve and often has a performance penalty. Most code written today does not provide a strong guarantee whether using or not using exceptions.
- Sometimes it is OK to allow some classes of exceptions to be fatal. If a fatal exception is thrown then the only viable option is to exit the program as cleanly as possible.
  - This allows components to be designed with a stated guarantee except in the case of a fatal exception.
  - For example, in many programs bad_alloc is fatal and will cause the program to terminate anyway so why spend extra time worrying about it.
  - Even with a Fatal Exception, it is usually fairly easy for most developers to guarantee at least the Minimal Guarantee if they are following a few suggested idioms.

Example #1

Stack::push()

```
template< typename T >
Stack< T >::push( T element )
{
  if ( m_top == m_size )
    {
    T* new_array = new T[ m_size *= 2 ];      // T1
    for ( int i = 0; i < m_top; ++i )
      new_array[ i ] = m_array[ i ];          // T2
    delete[] m_array;                         // T3
    m_array = new_array;
    }
  m_array[ m_top++ ] = element;               // T4
}
```

T1 – A throw from new or T's default constructor only satisfies the minimal guarantee (invariant m_size changed).
T2 – A throw from T's assignment operator will satisfy the minimal guarantee if we take care of the memory leak.
T3 – A throw from T's destructor provides no guarantee (m_array is unstable) so T's cannot be allowed to throw.
T4 – A throw from T's assignment operator only satisfies the minimal guarantee (invariant m_top changed).

So, What's the Point

- If you follow an idiom that takes care of the memory leak in case 2 and you do not allow destructors to throw then the minimal guarantee is satisfied with no conscious thought.
- It is fairly easy to satisfy the basic requirement, but does require more conscious thought
  - You have to use a temporary for m_size and only modify it after copying all elements from m_array to new_array.
  - You have to realize that it is OK that m_size gets modified even though Case 4 may still throw because the invariants are consistent (m_array is the size of m_size).
  - You have to use a temporary for m_top and only modify m_top after copying the element to the stack has succeeded.
- Even if you do support the basic requirement, often someone will come along and "re-optimize the code" to make it look "better" breaking the basic guarantee.
- Generally, satisfying the strong guarantee requires even more thought, more code and often incurs a performance penalty.

Example #1

Stack::push() with (almost) Basic and Strong Guarantees

```
// Basic Guarantee
template< typename T >
Stack< T >::push( T element )
{
  if ( m_top == m_size )
    {
    size_t new_size = m_size * 2;
    T* new_array = new T[ new_size ];
    for ( int i = 0; i < m_top; ++i )
      new_array[ i ] = m_array[ i ];
    delete[] m_array;
    m_array = new_array;
    m_size = new_size;
    }
  m_array[ m_top ] = element;
  ++m_top;
}
// strong-guarantee is not satisfied
// because function may throw after
// capacity has changed (assuming
// capacity is an invariant).
```

```
// Strong Guarantee
template< typename T >
Stack< T >::push( T element )
{
  if ( m_top == m_size )
    {
    size_t new_size = m_size * 2;
    T* new_array = new T[ new_size ];
    for ( int i = 0; i < m_top; ++i )
      new_array[ i ] = m_array[ i ];
    new_array[ m_top ] = element;

    delete[] m_array;
    m_array = array;
    m_size = new_size;
    }
  else
    m_array[ m_top ] = element;
  ++m_top;
}
```

Note: These examples ignore the memory leak problem and assume that deallocation and destructors do not throw.

Example #2

Stack::pop()

```
// Minimal Guarantee

template< typename T >
T Stack< T >::pop()
{
  T result;
  if ( m_top > 0 )
    result = m_array[ --m_top];

  return result;
}
```

```
// Strong Guarantee?

template< typename T >
T Stack< T >::pop()
{
  T result;
  if ( m_top > 0 )
    {
    result = m_array[ m_top - 1 ];
    --m_top;
    }

  return result;
}
```

In this example, why is the strong guarantee really no better than the minimal guarantee?

Retrieving the return value forces client to use T's copy-constructor. If this throws then the stack has been changed (m_top is decremented) yet return value has been lost.

Tip #2 - Never Let a Resource go Unmanaged

- This is one of the most critical idioms to follow. Done correctly, this insures that resources are always released even in the presence of exceptions.
  - If resources are not managed, then developers would need to resort to using try-catch blocks which quickly become repetitive, tedious and error-prone.
- This also prevents dangling pointers from breaking the minimal guarantee (e.g. deleting a pointer then having a throw happen before re-assigning the pointer can cause the object to become un-destructible).
- Tip #2a - Perform every resource allocation in its own statement which immediately gives the new resource to a manager object.
  - This avoids problems related to how a compiler performs evaluation of multiple expressions on the same line (see GOTW #56).

Use RAII for Resource Management

- Some examples for RAII classes are:
  - SmartPointer classes to manage dynamically allocated memory (i.e. auto_ptr, boost::shared_ptr, etc.)
    - Memory is automatically deleted when object is destructed
  - Lock classes to manage locks (i.e. boost::lock)
    - Locks are automatically released when object is destructed
  - File classes to manage files
    - Files are automatically flushed and closed when object is destructed

```
// Example using a Boost Smart Ptr

boost::shared_array< T > new_array( new T[ m_size *= 2 ] );

// Example using an auto_ptr

std::auto_ptr< char > name_arg( new char[ name.size() + 2 );
strcpy( name_arg.get(), "-" );
strcat( name_arg.get(), name.c_str() );
argv->push_back( name_arg.release() );
```

Example #3

Minimal and Basic Guarantees

This guideline is how the problem of the memory leak in Example #1 gets resolved with very little thought.

```
// Minimal Guarantee

typedef boost::shared_array< T > Array;

template< typename T >
Stack< T >::push( T element )
{
  if ( m_top == m_size )
    {
    Array new_array( new T[m_size *= 2]);
    for ( int i = 0; i < m_top; ++i )
      new_array[ i ] = m_array[ i ];
    m_array = new_array;
    }
  m_array[ m_top++ ] = element;
}
```

```
// Basic Guarantee

typedef boost::shared_array< T > Array;

template< typename T >
Stack< T >::push( T element )
{
  if ( m_top == m_size )
    {
    size_t new_size = m_size * 2;
    Array new_array( new T[ new_size ] );
    for ( int i = 0; i < m_top; ++i )
      new_array[ i ] = m_array[ i ];
    m_array = new_array;
    m_size = new_size;
    }
  m_array[ m_top ] = element;
  ++m_top;
}
```

Tip #3 - Use Scope Guards

- One drawback with RAII is the need to create many small classes whose sole purpose is to manage a specific resource type. This in itself will lead to repetitive, tedious and error-prone code.
- There is an idiom called “ScopeGuard” that should be used in situations where a specific RAII class does not already exist.
- I will use examples based upon a helper library that implements the ScopeGuard idiom. There are numerous other implementations, including the original work by Andrei Alexandrescu and Peter Marginean and also the Boost Scope Exit library.

Example #4

Scope Guards

```
// Example 4a

HANDLE file_handle = CreateFile( … );
if ( file_handle == INVALID_HANDLE_VALUE )
  // handle error, perhaps throw?

// close handle on exit
XceptON_BLOCK_EXIT(( ::CloseHandle, file_handle ));

// Example 4b

sigaction( SIGCHLD, &sig_act, &old_sig_act );

// Reset signal upon exit
XceptON_BLOCK_EXIT(( sigaction, SIGCHLD, &old_sig_act, &sigact ));

// Example 4c

// set m_stream to NULL upon block exit
XceptON_BLOCK_EXIT(( XceptSetPtrToNULL< void >, boost::ref( m_stream ));
```

Scope Guard Details

(page 1)

```
class XceptSGBase
{
 public:
  void Dismiss() const { m_dismissed = true; }

 protected:

  XceptSGBase() : m_dismissed( false ) {}
 ~XceptSGBase() {}

  XceptSGBase( const XceptSGBase& other )
  : m_dismissed( other.m_dismissed )
  {
    // take over responsibility
    other.Dismiss();
  }

  template< typename J >
  static void Execute( J& j )
  {
    if ( !j.m_dismissed )
      {
      // always mark dismissed
      j.m_dismissed = true;

      j.CallFunctor();
      }
  }
```

```
 template< typename J >
  static void SafeExecute( J& j )
  {
    try
      {
      Execute( j );
      }
    catch( ... )
      {
      }
  }

 private:

  mutable bool m_dismissed;

 private:

  // assignment operator is not allowed
  XceptSGBase& operator=( const XceptSGBase& );

  // these objects should never be created on the
heap
  void* operator new( std::size_t );

  // cannot creater pointer aliases to these
objects
  XceptSGBase* operator&();
  const XceptSGBase* operator&() const;
};
```

Scope Guard Details

(part 2)

```
template< typename FUNCTOR >
class XceptSG0 : public XceptSGBase
{
 public:

  explicit XceptSG0( FUNCTOR functor )
  : m_functor( functor )
  {}

 ~XceptSG0() { SafeExecute( *this ); }

  void CallFunctor() { m_functor(); }

  const FUNCTOR& Functor() const
  { return m_functor; }

 private:

  FUNCTOR m_functor;
};

template< typename FUNCTOR >
inline XceptSG0< FUNCTOR >
XceptMakeGuard( FUNCTOR functor )
{
  return XceptSG0< FUNCTOR >( functor );
}
```

```
typedef const XceptSGBase& XceptScopeGuard;

#define XceptMAKE_GUARD( fun_n_args_tuple ) \
  XceptMakeGuard( boost::bind fun_n_args_tuple )

#define XceptJOIN_DIRECT( str1, str2 ) str1##str2
#define XceptJOIN( str1, str2 ) \
  XceptJOIN_DIRECT( str1, str2 )
#define XceptANON_VAR( str ) \
  XceptJOIN( str, __LINE__ )

#if defined( __GNUC__ )
#define XceptON_BLOCK_EXIT \
  XceptScopeGuard XceptANON_VAR( sg ) \
    __attribute__((unused)) = XceptMAKE_GUARD
#else
#define XceptON_BLOCK_EXIT \
  XceptScopeGuard XceptANON_VAR( sg ) = \
    XceptMAKE_GUARD
#endif

#if defined( __GNUC__ )
#define XceptANON_SCOPE_GUARD \
  XceptScopeGuard XceptANON_VAR( sg ) \
    __attribute__((unused))
#else
#define XceptANON_SCOPE_GUARD \
  XceptScopeGuard XceptANON_VAR( sg )
#endif
```

Scope Guard

Implementation Highlights

- The ScopeGuard class is simple and efficient
  - stack based, all inline, no virtual functions
- Can easily create anonymous scope guards

```
XceptON_BLOCK_EXIT(( CloseHandle, file_handle ));
XceptANON_SCOPE_GUARD( boost::bind< BOOL >( CloseHandle, handle ) );
```

- Can create named scope guards

```
XceptScopeGuard guard1 = XceptMAKE_SCOPE_GUARD(( CloseHandle, handle ));
XceptScopeGuard guard2 = XceptSG0( boost::bind< BOOL >( CloseHandle, handle ) );
```

- Can call Dismiss() before exiting the current scope to disable it

```
guard1.Dismiss();  // will no longer do anything upon exiting scope
```

- Implicitly swallows exceptions but can bypass this behavior by directly calling Execute()

```
guard1.Execute();  // may throw if underlying functor throws
```

- C++ Trivia Question
  - How does the XceptSC0 destructor get called when all we are storing is the XceptSGBase (remember, we are not using virtual destructors)?

Tip #4 - Know When to Throw

- Only throw when an exceptional case happens
  - Exceptions are meant to specify an error. They should not be used to return an expected result, even if that result is a failure.
  - If an exception is expected each time you run the program, then rethink the exception.
  - Why not use exceptions for everything?
    - Throwing exceptions can be expensive. In general, this shouldn't be a problem if they are only being thrown in exceptional cases.
      - Later tips (e.g. logging) make it even more expensive.
    - Catching and handling can be tedious and will complicate your code if done everywhere.
    - Many debuggers provide an ability to catch an exception before stack unwinding takes place. This capability is severely impacted if the program is throwing exceptions during normal program execution (e.g. boost::lexical_cast).
- Never throw in a destructor or de-allocation routine
  - It is a generally accepted that if destructors can throw then many operations are difficult or impossible to implement safely

Tip #5 - Know How to Throw

- Always throw by value (not pointer)
  - This avoids having to deal with memory management issues at the catch site
- If you re-throw, use “throw;” instead of “throw e;”
  - The first form will always preserve polymorphism
- Do not directly throw on a logic-error
  - See tip #10

Tip #6 - Know When to Catch

- Only catch if one of the following is true:
  - You have sufficient knowledge to handle the error
  - You are going to translate an exception to
    - a higher level semantic exception
    - convert from an external library exception to an abstracted exception
    - change the error handling mechanisms (e.g. in C API, catch C++ exception and returning an error code instead)
  - you are adding additional data and then re-throwing
  - You are absorbing errors in the bodies of destructors or deallocation routines (see tip on logging)
  - You are enforcing boundaries where exceptions should not propagate past (e.g. program exit, thread exit, COM function, etc.)

Tip #7 - Know How to Catch

- Always catch by (const) reference
  - Catching by value can result in slicing
- In general, do not catch (…)
  - This not only catches C++ exceptions but may also catch exceptions emitted by the OS.
  - Often, developers do this to swallow exceptions even in places where they should not be swallowed.
- Do not swallow exceptions unless it is necessary due to enforcing a boundary condition or in bodies of destructors and deallocation routines.
  - Always log a swallowed exception (see tip on logging)

Tip #8 - Writing Exception Classes

- Exception classes should generally be derived from std::exception (or std::runtime_exception, std::logic_exception).
  - This allows clients to catch “all” exceptions without resorting to catch (…) unless that is what is intended
- Use virtual inheritance.
  - This was originally recommended by Andrew Koenig (to prevent ambiguity at the catch site).
- Be careful when embedding a std::string object or any other data member whose copy-constructor may throw.
  - Doing so could lead to std::terminate.
  - If a bad_alloc is considered fatal, one can argue that it is OK to do this and just let the second throw terminate the program.
  - An alternative is to use a shared_ptr; however, the constructing of the exception may then throw.
- Expose relevant information about the cause of the error in the exception classes public interface.
- Be careful not to overly complicate the depth of your exception hierarchy or the number of exception types (see example #5)

Example #5

Exception Classes

```
// One top-level exception class (so all your exceptions can be caught with
// a single catch statement "catch ( const XceptError& )"
class XceptError: public std::exception, public boost::exception

  // provide information related to "recoverable" logic errors
  class XceptLogicError: public XceptError

  // provide information related to system (e.g. boost::system) errors
  class XceptSystemError: public XceptError
    // this class contains a boost::system:errc member

    // provide abstracted system information
    class SysSystemError: public XceptSystemError
      // this class contains an enum for higher level system codes

// One top-level exception class that will not be caught with your
// generic catch statement "catch (const XceptError& )"
class XceptFatalError: public std::exception, public boost::exception
```

Tip #9 - Allow Exceptions for Logic Errors

- Questions to Consider
  - Should assertions be disabled in release code?
    - If there are no reasons to remove the assertion (e.g. performance concerns) then isn't it better to detect a bad situation than let the logic error move to an unpredictable runtime error?
    - Could it be possible to recover from a logic-error; perhaps even disabling certain functionality?
  - Do assertions always provide a core-dump?
    - How many Windows developers have ever seen a core dump?
    - How many Unix users disable core dumps because of file size?
    - When a core dump is generated by someone other than the developer, how often does the developer actually get the core dump?

Macros for Logic-Errors

- Define macros to use for logic-checking
  - One set for recoverable (at the time they are detected) logic-errors that will log the error and continue on.
  - One set for client recoverable logic-errors that will either assert or throw. If throw, client can try and recover.
  - One set for fatal logic-errors that will either assert or throw.
- Each set should contain both a release and a debug version
- Using macros make it easy to decide later on whether a logic-error should assert or throw an exception
  - You may even want to create compile options that determine whether the macros assert or throw.
  - Another approach could use error handlers that clients could either set at compile time (e.g. BOOST_ASSERT) or runtime (e.g. set_new_handler).

Example #6

Xcept Macros

- XceptASSERT and XceptDBG_ASSERT
  - will throw an XceptError
  - clients can choose to recover from these logic-errors
- XceptASSERT_FATAL and XceptDBG_ASSERT_FATAL
  - will throw an XceptFatalError
    - catching XceptError will not catch these
  - clients should “gracefully” terminate the application
- XceptASSERT_THROW and XceptDBG_ASSERT_THROW
  - will throw an exception client specifies when calling
- XceptASSERT_WARNING and XceptDBG_ASSERT_WARNING
  - will log the warning and continue on

Example #6

Xcept Macro Definitions (page 1)

```
template< typename T >
T XceptAssertion( const T& e, const char* expr, const char* file, const char* function, unsigned int line )
{
  e << boost::throw_file( file );
  if ( function )
    e << boost::throw_function( function );
  e << boost::throw_line( line );
  return e;
}

#if !defined( _DEBUG )

#define XceptASSERT( expr ) \
  ((expr) ? ((void) 0) : throw XceptAssertion( XceptError(), NULL, __FILE__, NULL, __LINE__ ))

// XceptASSERT_FATAL( expr ) passes XceptFatalError() as the first argument
// XceptASSERT_THROW( expr, e ) takes an exception and passes it as the first argument

#define XceptASSERT_WARNING( expr ) ((expr) ? ((void) 0) : XceptWarning( NULL, __FILE__, NULL, __LINE__ ))
#define XceptTHROW( e ) throw XceptException( e, __FILE__, NULL, __LINE__ )

#define XceptDBG_ASSERT( expr ) ( (void) 0 )
// remaining XceptDBG_ are defined exactly the same
```

Example #6

Xcept Macros Definitions (page 2)

```
#else  // defined( _DEBUG )

#define XceptASSERT( expr ) \
( (expr) ? ( (void) 0 ) : \
           throw XceptAssertion( XceptError(), #expr, __FILE__, BOOST_CURRENT_FUNCTION, __LINE__ ) )

// remaining non-DBG macros defined very similar

#define XceptDBG_ASSERT( expr )  XceptASSERT(  (expr)  )

// remaining XceptDBG_ macros are defined  very similar

#endif

#define XceptInfo( name, type ) \
typedef boost::error_info< struct tag_##name, type > name##Info; \
\
inline const name##Info::value_type* name##InfoGet( const boost::exception& e ) \
{ return boost::get_error_info< name##Info >( e ); }

// e.g. To Define - XceptInfo( SysXceptFilename2, std::string );
// e.g. To Set    - e << SysXceptFilename2Info( name );
// e.g. To Get    - std::string* name = SysXceptFilename2InfoGet( e );
```

Tip #10 - Exception Logging

- Exception Logging Should Never Throw
  - It is better to disable the logging and lose the information than to have logging be the cause of your program crashing (especially true in released code).
- Always Log when Swallowing Exceptions
  - If you don't, exception becomes invisible (except when running under a debugger with breaking enabled) and may hide problematic code.
- Logging Exceptions
  - Considerations
    - Debug vs. Release
    - Invasibility
    - Performance
    - Portability
    - Symbol Availability

How to Log Exception

- Log Location of Thrown Exception
  - PROS: Non-invasive, portable and does not rely on symbols being available.
  - CONS: Minor performance impact when exception is thrown.
- Log Location of Catches
  - PROS: Portable and does not rely on symbols
  - CONS: Somewhat invasive, minor performance impact when exception is caught.
- Log the Stack Trace of Thrown Exceptions
  - Using Execution Context Statements
    - PROS: Very portable and does not rely on symbols being available.
    - CONS: Is very invasive and can be expensive.
  - Using Stack Trace APIs
    - PROS: Lots of Information
    - CONS: Non-Portable, Expensive and Relies on Symbols being Available

Example #7

Execution Context Statements

```
class XceptScope
{
  XceptScope( const char* file, unsigned int line )
  : m_file( file ), m_line( line ), m_uncaught_exception_on_entry( std::uncaught_exception() )
  {}
 ~XceptScope()
  {
  if ( !m_uncaught_exception_on_entry && std::uncaught_exception() )
    {
    // assume log is non-throwing
    log << "Unwinding stack due to exception file: " << m_file << " line: " << m_line << "\n";
    }
  }
 private:
  const char*  m_file;
  unsigned int m_line;
  bool         m_uncaught_exception_on_entry;
};

#define XceptSCOPE XceptScope( __FILE__, __LINE__ )

void func()
{
  XceptSCOPE;

  // do stuff

  // if exception is thrown will log when scope exits during stack unwinding
}
```

Bonus Tips

Guidelines for Implementing a Strong Guarantee

- Do all work on the side and then commit with non-throwing operations.
  - Example of this technique is the “create a temporary and swap idiom”.
  - See GotW #59.
- If using a class that isn’t exception-safe, you can use the “hide the details being a pointer” technique to still make your class exception-safe.
  - See GotW #59
- Prefer “one class or function to one responsibility”.
  - Example #2 demonstrates how a function with more than one responsibility is harder to make exception safe.
  - Many exception safety problems can be made simpler or eliminated without conscious thought simply by following this guideline.
  - See GotW #8 and GotW #21
- Prefer delegation to inheritance when given a choice.
  - Lower coupling makes it easier to insure exception safety
  - See GotW #60
- Use Transactional Guards

Example #8

Using Transactional Guards

```
// portion of a function that copies a file

// we use this to commit actions when function completes or to automatically
// rollback actions when we have an exception.
XceptTxGuard tx_guard;

if ( dst_path.Exists() )
  {
  // if a file already exists of the same name, we need to move to a temporary
  // file so if copy command is unsuccessful, we can restore original file

  SysPath tmp_dst_path = SysPath::CreateTmpPath( dst_path.ParentDir() );

  SysNativeMoveFile( dst_path, tmp_dst_path );

  // restore original file is function fails
  tx_guard.AddRollbackAction( XceptMAKE_ACTION((SysNativeMoveFile, tmp_dst_path, dst_path )) );
  // need to remove temporary file if function is successful
  tx_guard.AddCommitAction( XceptMAKE_ACTION(( SysDeleteFile, tmp_dst_path )) );
  }

SysFile dest( dst_path, "wb" );

// if we fail, need to remove the destination file we just created
tx_guard.AddRollbackAction( XceptMAKE_ACTION(( SysDeleteFile, dst_path )) );

// … more work

// function done so do commit
tx_guard.Commit();

// One furthur note, failure to allocate memory (for tx_guard and when copying strings/paths to actions)
// would cause a failure (strong-guarantee would not be guaranteed). This could be fixed by pre-
// allocating space for the tx_guard and using shared_ptr for string/paths that are part of actions.
```

Example #8

Transactional Guard Class

```
class XceptTxGuard
{
 public:

  typedef boost::function0<void> XceptAction;
  typedef std::vector< XceptAction > Actions;

  XceptTxGuard() : m_committed( false ) {}
 ~XceptTxGuard();

  void AddCommitAction( const Action& action )
    { m_commits.push_back( action ); }
  void AddRollbackAction( const Action& action )
    { m_rollbacks.push_back( action ); }

  template< typename SCOPE_GUARD >
  void AddRollbackGuard( const SCOPE_GUARD& guard )
    {
      m_rollbacks.push_back( guard.Functor() );
      guard.Dismiss();
    }

  void Commit();

 private:

  bool m_committed;

  Actions m_commits;
  Actions m_rollbacks;
};

#define XceptMAKE_ACTION( fun_n_args_tuple ) \
boost::bind fun_n_args_tuple
```

```
#define foreach    BOOST_FOREACH
#define foreach_r  BOOST_REVERSE_FOREACH

XceptTxGuard::~XceptTxGuard()
{
  if ( !m_committed )
    {
    try
      {
      foreach_r( Action action, m_rollbacks )
        action();
      }
    catch( ... )
      { // TODO: Log failure }
    }
}


void XceptTxGuard::Commit()
{
  m_committed = true;

  try
    {
    foreach( Action action, m_commits )
      action();
    }
  catch( ... )
    { // TODO: Log failure }

  m_commits.clear();
}
```

Conclusion

- Many developers and companies do not use exceptions because of fear, belief that exceptions are more difficult to use than error codes and that error-codes provide the same benefit (error-safe code) as exceptions.
- Current literature and wisdom portray that using exceptions are difficult unless you understand all the intricacies. This scares developers and companies and they decide to never allow the use of exceptions in their code.
- With a few simple guidelines and idioms, exceptions can be a powerful alternative to error-codes.
- Used correctly, exceptions will simply your code, simply development and maintenance, and provide higher quality (less defects) products to your customers.