Exception-Safe Coding

Website
http://exceptionsafecode.com

- Bibliography
- Video
- Comments
Contact

- Email
  jon@exceptionsafecode.com
- Follow
  @JonathanKalb
- Résumé
  jonkalb@a9.com

Dedication

To the great teacher of Exception-Safe coding…
The Promise

• Easier to Read
  Easier to Understand and Maintain
• Easier to Write
• No time penalty
• 100% Robust

A Word on C++11

• I will cover both C++ 2003 and C++ 2011
• Solid on classic C++
• Some things still to learn about C++11
• No fundamental change in exception-safety
• Some new material
• Some material no longer necessary
Session Preview

• The problem
• Solutions that don’t use exceptions
• Problems with exceptions as a solution
• How not to write Exception-Safe code
• Exception-Safe coding guidelines
• Implementation techniques

What’s the Problem?

• Separation of Error Detection from Error Handling
Solutions without Exceptions

- Addressing the problem without exceptions
- Error flagging
- Return codes
Error Flagging

- errno
- "GetError" function

```c
errno = 0;
old_nice = getpriority(PRIO_PROCESS, 0);
/* check errno */
if (errno)
{
    /* handle error */
}
```
Problems with the Error Flagging Approach

- Errors can be ignored
- Errors are ignored by default
- Ambiguity about which call failed
- Code is tedious to read and write

Return Codes

- Return values are error/status codes
- (Almost) every API returns a code
- Usually int or long
- Known set of error/status values
- Error codes relayed up the call chain
Problems with the Return Code Approach

- Errors can be ignored
- Are ignored by default
- If a single call “breaks the chain” by not returning an error, errors cases are lost
- Code is tedious to read and write
- Exception based coding addresses both of these issues…
  … but has issues of its own.

The Dark Side

Broken error handling leads to bad states,
bad states lead to bugs,
bugs lead to suffering.

— Yoda
The Dark Side

Code using exceptions is no exception.

T& T::operator=(T const& x)
{
    if (this != &x)
    {
        this->~T(); // destroy in place
        new (this) T(x); // construct in place
    }
    return *this;
}
The Dark Side

Early adopters reluctant to embrace exceptions

The Dark Side

- Implementation issues are behind us
- Today’s compilers:
  - Reliable, Performant, and Portable
- What causes concerns today?
Code Path Disruption

- Having error conditions that can’t be ignored implies that the functions we are calling have unseen error returns.

“Counter-intuitively, the hard part of coding exceptions is not the explicit throws and catches. The really hard part of using exceptions is to write all the intervening code in such a way that an arbitrary exception can propagate from its throw site to its handler, arriving safely and without damaging other parts of the program along the way.”

– Tom Cargill
Counter-intuitively, this is true of any error handling system.

Cargill’s Article

- “Exception Handling: A False Sense of Security”
- Analyzed a templated Stack class
- Found problems, but no solution
Cargill’s Stumper

template <class T> T Stack<T>::pop()
{
    if ( top < 0 )
        throw "pop on empty stack";
    return v[top--];
}

Standard’s Solution

template <class T> T& stack<T>::top();

template <class T> void stack<T>::pop();
Cargill’s Article

• Spread Fear, Uncertainty, and Doubt
• Some said, “Proves exceptions aren’t safe”

Cargill’s Conclusions

• Didn’t say exceptions were unsafe
• Didn’t say exceptions were too hard to use
• Did say he didn’t have all the answers
Cargill’s Conclusions

We don’t know how to be exception-safe. (1994)

Sure we do! (1996)

Abrahams’ Conclusions

“Exception-handling isn’t hard. Error-handling is hard. Exceptions make it easier!”
dosomething();
cleanup();

“…exceptions are extremely dangerous.”
– Joel Spolsky
dosomething();
cleanup();

“That code is wrong.”
– Jon Kalb

First Steps

• Carefully check return values/error codes to detect and correct problems.
• Identify functions that can throw and think about what to do when they fail
• Use exception specifications so the compiler can help create safe code.
• Use try/catch blocks to control code flow
The Hard Way

• Carefully check return values/error codes to detect and correct problems.
• Identify functions that can throw and think about what to do when they fail
• Use exception specifications so the compiler can help create safe code.
• Use try/catch blocks to control code flow

The Wrong Way

• Carefully check return values/error codes to detect and correct problems.
• Identify functions that can throw and think about what to do when they fail
• Use exception specifications so the compiler can help create safe code.
• Use try/catch blocks to control code flow

“You must unlearn what you have learned.”

— Yoda
The Right Way

- Think structurally
- Maintain invariants

Exception-Safe!

- Guidelines for code that is Exception-Safe
- Few enough to fit on one slide
- Hard requirements
- Sound advice
Exception-Safety Guarantees (Abrahams)

- Basic
  - invariants of the component are preserved, and no resources are leaked
- Strong
  - if an exception is thrown there are no effects
- No-Throw
  - operation will not emit an exception

---

Exception-Safety Guarantees (Abrahams)

- Basic
  - invariants of the component are preserved, and no resources are leaked
- Strong
  - Yoda:
    “Do or do not.”
- No-Throw
  - operation will not emit an exception
Exception-Safety Assumptions

- Basic guarantee
  - Cannot create robust code using functions that don’t provide at least the Basic guarantee – fixing this is priority zero
- All code throws unless we know otherwise
- We are okay with this

Mechanics

- How exceptions work in C++
  - Error detection / throw
  - Error handling / catch
  - New in C++11
Error Detection

{  
    /* A runtime error is detected. */  
    ObjectType object;  
    throw object;  
}

Is object thrown?
Can we throw a pointer?
Can we throw a reference?

Error Detection

{  
    std::string s("This is a local string.");  
    throw ObjectType(constructor parameters);  
}
Mechanics

- How exceptions work in C++
- Error detection / throw
- Error handling / catch
- New in C++11

```cpp
try
{
    code_that_might_throw();
}
catch (A a) //== works like a function argument
{
    error_handling_code_that_can_use_a(a);
}
catch (...) //== “catch all” handler
{
    more_generic_error_handling_code();
}
more_code();
```
... catch (A a) {
    ...

    • Issues with catching by value
      • Slicing
      • Copying

    catch (A& a) {
        a.mutating_member();
        throw;
    }
}
try
{
    throw A();
}
catch (B) {}  // if B is a public base class of A
catch (B&) {}
catch (B const&) {}
catch (B volatile&) {}
catch (B const volatile&) {}
catch (A) {}
catch (A&) {}
catch (A const&) {}
catch (A volatile&) {}
catch (A const volatile&) {}
catch (void*) {}  // if A is a pointer
catch (...) {}

Guideline

- Throw by value.
- Catch by reference.
Performance Cost of try/catch

- No throw — no cost.
- In the throw case…
  - Don’t know. Don’t care.

```cpp
void F(int a)
{
  try
  {
    int b;
    ...
  }
  catch (std::exception const& ex)
  {
    ...
    // Can reference a, but not b
    ...
    // Can throw, return, or end
  }
}
```
void F(int a)
try
{
    int b;
    ...
}
catch (std::exception const& ex)
{
    ... // Can reference a, but not b
    ... // Can throw, but can’t “return”
}
Function Try Block for a Constructor

Foo::Foo(int a)
try :
Base(a),
member(a)
{
}
catch (std::exception& ex)
{
... // Can reference a, but not Base or member
// Can modify ex or throw a different exception...
// but an exception will be thrown
}

Function Try Blocks

- Only use is to change the exception thrown by the constructor of a base class or data member constructor
Mechanics

- How exceptions work in C++
- Error detection / throw
- Error handling / catch
- New in C++11

C++11 Supported Scenarios

- Moving exceptions between threads
- Nesting exceptions
Moving Exceptions Between Threads

- Capture the exception
- Move the exception like any other object
- Re-throw whenever we want

Capturing is easy
<exception> declares:

```cpp
exception_ptr current_exception() noexcept;
```
std::exception_ptr ex(nullptr);
try {
    ... 
}
catch(...) {
    ex = std::current_exception();
    ...
}
if (ex) {
    ...
}
Re-throwing is easy
<exception> declares:

[[noreturn]] void rethrow_exception(exception_ptr p);

A related scenario

```cpp
int Func(); // might throw

std::future<int> f = std::async(Func());

int v(f.get()); // If Func() threw, it comes out here
```
Nesting Exceptions

- Nesting the current exception
- Throwing a new exception with the nested one
- Re-throwing just the nested one

Nesting the current exception is easy

<exception> declares:

\[
\text{class nested_exception;}
\]

Constructor implicitly calls current_exception() and holds the result.
Nesting Exceptions

Throwing a new exception with the nested is easy
<exception> declares:

```cpp
[[noreturn]] template <class T>
void throw_with_nested(T&& t);
```

Throws a type that is inherited from both T and std::nested_exception.

```cpp
try {
    try {
        ...
    } catch(...) {
        std::throw_with_nested(MyException());
    }
    catch (MyException&ex) {
        ... handle ex
        ... check if ex is a nested exception
        ... extract the contained exception
        ... throw the contained exception
    }
}
```
Nesting Exceptions

One call does all these steps

<exception> declares:

template <class E>
void rethrow_if_nested(E const& e);

try {
    try {
        ...
    } catch(...) {
        std::throw_with_nested(MyException());
    }
    } catch (MyException&ex) {
        ... handle ex
        ... check if ex is a nested exception
        ... extract the contained exception
        ... throw the contained exception
    }

Nesting Exceptions

```cpp
try {
    try {
        ...
    } catch(...) {
        std::throw_with_nested(MyException());
    }
} catch (MyException& ex) {
    ... handle ex
    std::rethrow_if_nested(ex);
}
```

• The “Terminate” Handler
  • Calls std::abort()
  • We can write our own ...
    • …but it is too late.

• The “Unexpected” Handler
  • Calls the terminate handler
  • We can write our own ...
    • …but it is too late.

Standard Handlers
Standard Handlers

- The “Unexpected” Handler
- Called when throwing an exception outside of (dynamic) exception specifications

Exception Specifications

- Two flavors
- C++ 2003
  - Exception Specifications
  - Now technically called Dynamic Exception Specifications
Exception Specifications

- Two flavors
- C++ 2011
  - Introduces “noexcept” keyword
  - Deprecates Dynamic Exception Specifications

Dynamic Exception Specifications

```cpp
void F(); // may throw anything

void G() throw (A, B); // may throw A or B

void H() throw (); // may not throw anything
```
Dynamic Exception Specifications

- Not checked at compile time.
- Enforced at run time.
- By calling the "unexpected" handler and aborting.

Guideline

- Do not use dynamic exception specifications.
• Two uses of “noexcept” keyword in C++11
• noexcept specification (of a function)
• noexcept operator

As a noexcept exception specification

```cpp
void F(); // may throw anything
void G() noexcept(Boolean constexpr);  
void G() noexcept; // defaults to noexcept(true)
```

Destructors are noexcept by default.
noexcept

- As an operator

```cpp
static_assert(noexcept(2 + 3) , "");  
static_assert(not noexcept(throw 23) , "");
inline int Foo() {return 0;}
static_assert(noexcept(Foo()), ""); // ???
```

```cpp
// C++ 2011
```

---

noexcept

- As an operator

```cpp
static_assert(noexcept(2 + 3) , "");  
static_assert(not noexcept(throw 23) , "");
inline int Foo() {return 0;}
static_assert(noexcept(Foo()), ""); // assert fails!
```

```cpp
// C++ 2011
```
noexcept

• As an operator
static_assert(noexcept(2 + 3), "");
static_assert(not noexcept(throw 23), "");
inline int Foo() noexcept {return 0;}
static_assert(noexcept(Foo()), ""); // true!

How will noexcept be used?
• Operator form for no-throw based optimizations
  • move if no-throw, else do more expensive copying
• Unconditional form for simple user-defined types
  struct Foo { Foo() noexcept {} };
• Conditional form for templates with operator form
  template <typename T> struct Foo<T {
    Foo() noexcept(noexcept(T())) {} };

Guideline

• Do not use dynamic exception specifications.
• Do use noexcept.

Standard Handlers

• The “Terminate” Handler
• Called when re-throw and there is no exception
• Called when a “noexcept” function throws
• Called when throwing when there is already an exception being thrown
How to not “Terminate”

- Don’t re-throw outside of a catch block
  - ✔
- Don’t throw from a “noexcept” function
  - ✔
- Don’t throw when an exception is being thrown
  - When would that happen? After throw comes catch. What else happens?
  - Destructors!

Guideline

- Destructors must not throw.
  - Must deliver the No-Throw Guarantee.
  - Cleanup must always be safe.
  - May throw internally, but may not emit.
Safe Objects

- Exception-Safe Code is Built on Safe Objects

Object Lifetimes

- Order of construction:
  - Base class objects
    - As listed in the type definition, left to right
  - Data members
    - As listed in the type definition, top to bottom
    - Not as listed in the constructor’s initializer list
  - Constructor body
- Order of destruction:
  - Exact reverse order of construction
- When does an object’s lifetime begin?
Aborted Construction

- How?
  - Throw from constructor of base class, constructor of data member, constructor body
- What do we need to clean up?
  - Base class objects?
  - Data members?
  - Constructor body?
    - We need to clean up anything we do here because the destructor will not be called.
- What about new array?
- What about the object’s memory?

Aborted Construction

- Throwing from a constructor
- Leaking object memory
- Placement new
Placement New

- Any use of new passing additional parameter
- Standard has “original placement new”
- Overload for “newing” an object in place
  ```cpp
  Object* obj = new(&buffer) Object;
  ```
- “Placement” can be misleading

Aborted Construction

- Throwing from a constructor
- Leaking object memory
- Placement new
- *Effective C++, 3rd Ed.*
  - Item 52:
    - Write placement delete if you write placement new.
Placement Delete

- We can’t pass parameters to the delete operator
- Only called if constructor throws during the “corresponding” placement new
- Not an error if not defined
- It’s just a hard to find bug

RAII

- Resource Acquisition Is Initialization
RAII Examples

- Most smart pointers
- Many wrappers for
  - memory
  - files
  - mutexes
  - network sockets
  - graphic ports

```cpp
template <typename U> struct ArrayRAII
{
    ArrayRAII(int size): array_(new U[size]) {}  
    ~ArrayRAII() {delete [] array_;}
    U* array() {return array_;}
    ...

private:
    // Cannot be default constructed or copied.
    ArrayRAII();
    ArrayRAII(ArrayRAII const&);
    ArrayRAII& operator=(ArrayRAII const&);

    U* array_;  
};
```
What happens to the object if acquisition fails?

- Nothing

- The object never exists.
- If you have the object, you have the resource.
- If the attempt to get the resource failed, then the constructor threw and we don’t have the object.
RAII Cleanup

- Destructors have resource release responsibility.
- Some objects may have a “release” member function.
- Cleanup cannot throw
  - Destructors cannot throw

Design Guideline

- Each item (function or type) does just one thing.
- No object should manage more than one resource.
Every Resource in a Object

- If it isn’t in an object, it isn’t going to be cleaned up in a destructor and it may leak.
- Smart Pointers are your friend.

shared_pointer

- The smart pointer
  - From Boost
  - Was in the TR1
  - Is in C++ 2011
- Ref-counted
- Supports custom deleters
Smart Pointer “Gotcha”

- Is this safe?

```cpp
FooBar(smart_ptr<Foo>(new Foo(f)),
       smart_ptr<Bar>(new Bar(b)));
```

“There’s many a slip twixt the cup and the lip”

Smart Pointer “Gotcha”

- What is the rule?

“No more than one `new` in any statement.”

```cpp
a = FooBar(smart_ptr<Foo>(new Foo(f))) + Bar();
```

where we assume Bar() can throw

(Why do we assume Bar() can throw?)
Smart Pointer “Gotcha”

• What is the rule?

“Never incur a responsibility as part of an expression that can throw.”

```cpp
smart_ptr<T> t(new T);
```

Does both, but never at the same time.

Smart Pointer “Gotcha”

• But what about this?

```cpp
smart_ptr<Foo> t(new Foo( F() ));
```

Does it violate the rule?

It is safe.
What is the rule?

Assign ownership of every resource, immediately upon allocation, to a named manager object that manages no other resources.

Dimov’s rule

A better way

auto r(std::make_shared<Foo>(f));
auto s(sutter::make_unique<Foo>(f));

• More efficient.
• Safer
Smart Pointer “Gotcha”

- Is this safe?

```cpp
FooBar(std::make_shared<Foo>(f),
       std::make_shared<Bar>(b));
```

Yes!

Smart Pointer “Gotcha”

- A better rule

  “Don’t call new.”
Smart Pointer “Gotcha”

- A better rule
  
  "Don’t call new."
  "Avoid calling new."

Lesson Learned

- Keep your resources on a short leash to not go leaking wherever they want.
Manage State Like a Resource

- Use objects to manage state in the same way that we use objects to manage any other resource.

RAII

- Resource Acquisition Is Initialization
RAII

- Resource Acquisition Is Initialization
  - “Resource” includes too much
  - “Resource” includes too little
- Responsibility Acquisition Is Initialization
  - Responsibility leaks
  - Responsibility management

Guideline

- Use RAII.
  - Responsibility Acquisition Is Initialization.
- Every responsibility is an object
- One responsibility per object
Don’t write cleanup code that isn’t being called by a destructor.

Destructors must cleanup all of an object’s outstanding responsibilities.

Be suspicious of cleanup code not called by a destructor.

Joel on Software

dosomething();
cleanup();

“...exceptions are extremely dangerous.”
– Joel Spolsky
Jon on Software

```c
{
    CleanupType cleanup;
    dosomething();
}
```

“...Exception-Safe code is exceptionally safe.”
– Jon Kalb

Guideline

- All cleanup code is called from a destructor.
- An object with such a destructor must be put on the stack as soon as calling the cleanup code become a responsibility.
class Widget
{
    Widget& operator=(Widget const& rhs);
        // Strong Guarantee ???
        // ...
private:
    T1 t1_;  
    T2 t2_; 
};

Widget& Widget::operator=(Widget const& rhs) {
    T1 original(t1_);
    t1_ = rhs.t1_;  
    try {
        t2_ = rhs.t2_; 
    } catch (...) {
        t1_ = original;
        throw;  
    }  
}
The Cargill Widget Example

Widget& Widget::operator=(Widget const& rhs) {
    T1 original(t1_);
    t1_ = rhs.t1_;  
    try {
        t2_ = rhs.t2_;  
    } catch (...) {
        t1_ = original; //== can throw
        throw;
    }
}

Cargill’s Points
• Exception-safety is harder than it looks.
• It can’t be “bolted on” after the fact.
  • It need to be designed in from the beginning.

Cargill’s answer to the challenge:
• No, it can’t be done.

Jon’s answer:
• Yes, it can.
Fundamental Object Functions

- Construction
- Default
- Copy
- Destruction
- (Copy) Assignment operator
  - Value class
- The Rule of Three
- The Rule of Four
  - One more fundamental operator…

The Swapperator

- swap()
  - No-Throw swapping is a key exception-safety tool
  - swap() is defined in std, but...
    - std::swap<>() not No-Throw (in classic C++)
  - swap() for types we define can (almost) always be written as No-Throw
The Swapperator

- Spelled “swap()”
- Write a one-parameter member function and two-parameter free function in the “std” namespace
  - If your type is a template, do not put in “std”
- Both take parameters by (non-const) reference
- Does not throw!
- Is not written like this: swap() throw ()
  - Do not use dynamic exception specifications

Swapperator Examples

```cpp
struct BigInt {
    ...
    void swap(BigInt&); // No Throw
    // swap bases, then members
    ...
};
namespace std {
    template <> void swap<BigInt>(BigInt&a, BigInt&b)
    {a.swap(b);}
}
```
Swapperator Examples

template <typename T>
struct CircularBuffer {
    ...
    void swap(CircularBuffer<T>&); // No Throw
    // Implementation will swap bases then members.
    ...
};

// not in namespace std
template <typename T>
void swap(CircularBuffer<T>&a, CircularBuffer<T>&b)
{a.swap(b);}

Why No-Throw?

- That is the whole point
- std::swap<>() is always an option
  - But it doesn’t promise No-Throw
    - It does three copies–Copies can fail!
- Our custom swaps can be No Throw
  - Don’t use non-swapping base/member classes
  - Don’t use const or reference data members
  - These are not swappable
Guideline

• Create swapperator for value classes.
  • Must deliver the No-Throw guarantee.

The Swapperator

• Swappertor new and improved for C++11
• std::swap() now with moves!
• can be noexcept...
  • for objects with noexcept move operations
The Swapperator

- To define swap() or not to define swap()
  - Not needed for exception-safety
    - noexcept move operators are enough
  - May be wanted for performance
  - If defined, declared as noexcept

The Swapperator

- New rules for move operations
  - Kind of based on Rule of Three
    - If we create copy operations we must create our own move operations
  - How to know we’ve done it right?
    - Call Jon!
      - (925) 890...
The Swapperator

esc::check_swap() will verify at compile time that its argument’s swapperator is declared noexcept

#include "esc.hpp"

template <typename T>
void check_swap(T* = 0);

(Safe, but useless, in C++ 2003)
#include "esc.hpp"

struct MyType
{
  ...

  void AnyMember() {esc::check_swap(this); ...}

  ...

};

template <typename T> void check_swap(T* const t = 0)
{
  static_assert(noexcept(delete t), "msg...");
  static_assert(noexcept(T(std::move(*t))), "msg...");
  static_assert(noexcept(*t = std::move(*t)), "msg...");
  using std::swap;
  static_assert(noexcept(swap(*t, *t)), "msg...");
}
The Swapperator

template <typename T> void check_swap(T* const t = 0)
{
    ...
    static_assert(
        std::is_nothrow_move_constructible<T>::value, "msg...");
    static_assert(
        std::is_nothrow_move_assignable<T>::value, "msg...");
    ...
}

Calling swap in a template

template...
{
    ...
    using std::swap;
    swap(a, b);
    ...
}
Calling swap in a template (alternative)

```cpp
#include "boost/swap.hpp"

boost::swap(a, b);
```

Guideline

- Create swapperator for value classes.
- Must deliver the No-Throw guarantee.
Guideline

- Create swapperator for value classes.
  - Must deliver the No-Throw guarantee.

Guideline

- Support swapperator for value classes.
  - Must deliver the No-Throw guarantee.
Guideline

• Support swapperator for value classes.
• Must deliver the No-Throw guarantee.

Guideline

• Do not use dynamic exception specifications.
  • Do use noexcept.
    • Cleanup
      • Destructors are noexcept by default
  • Move/swap
  • Where else?
    • Wherever we can?
Guideline

- Do not use dynamic exception specifications.
  - Do use noexcept.
    - Cleanup
      - Destructors are noexcept by default
  - Move/swap
  - Where else?
    - Wherever it is “natural” and free?

Guideline

- Do not use dynamic exception specifications.
  - Do use noexcept.
    - Cleanup
      - Destructors are noexcept by default
  - Move/swap
  - Where else?
    - No where!
The Critical Line

- Implementing the Strong Guarantee
- Deferring the commit until success is guaranteed

```cpp
struct ResourceOwner
{
    // ...

    ResourceOwner& operator=(ResourceOwner const& rhs)
    {
        delete mResource;
        mResource = new Resource(*rhs.mResource);
        return *this;
    }

    // ...

private:
    // ...

    Resource* mResource;
};
```
struct ResourceOwner
{
    // ...
    ResourceOwner& operator=(ResourceOwner const&rhs)
    {
        if (this != &rhs)
        {
            delete mResource;
            mResource = new Resource(*rhs.mResource);
            return *this;
        }
    }
    // ...
private:
    // ...
    Resource* mResource;
};
struct ResourceOwner
{
    // …
    ResourceOwner& operator=(ResourceOwner const& rhs)
    {
        Resource temp(*rhs.mResource);
        temp.swap(*mResource);
        return *this;
    }
    // …
private:
    // …
    Resource* mResource;
};

void FunctionWithStrongGuarantee()
{
    // Code That Can Fail
    ObjectsThatNeedToBeModified.MakeCopies(OriginalObjects);
    ObjectsThatNeedToBeModified.Modify();

    // The Critical Line

    // Code That Cannot Fail (Has a No-Throw Guarantee)
    ObjectsThatNeedToBeModified.swap(OriginalObjects);
}
struct ResourceOwner
{
    // ...
    ResourceOwner& operator=(ResourceOwner const&rhs)
    {
        Resource temp(*rhs.mResource);
        temp.swap(*mResource);
        return *this;
    }
    // ...
    private:
    // ...
    Resource* mResource;
};

struct ResourceOwner
{
    // ...
    void swap(ResourceOwner&); // No Throw
    ResourceOwner& operator=(ResourceOwner rhs)
    {
        swap(rhs);
        return *this;
    }
    // ...
    private:
    // ...
    Resource* mResource;
};
struct ResourceOwner
{
    // ...
    void swap(ResourceOwner&); // No Throw
    ResourceOwner& operator=(ResourceOwner rhs)
    {
        swap(rhs);
        return *this;
    }
    // ...

    private:
    // ...
    Resource* mResource;
};

struct ResourceOwner
{
    // ...
    void swap(ResourceOwner&) noexcept;
    ResourceOwner& operator=(ResourceOwner rhs);
    ResourceOwner& operator=(ResourceOwner&& rhs) noexcept;

    // ...

    private:
    // ...
    Resource* mResource;
};
struct ResourceOwner
{
    // ...
    void swap(ResourceOwner&) noexcept;
    ResourceOwner& operator=(ResourceOwner const&rhs);
    ResourceOwner& operator=(ResourceOwner&& rhs) noexcept;

    // ...
    private:
    // ...
    Resource* mResource;
};
Guideline

- Use “Critical Lines” for Strong Guarantees.

The Cargill Widget Example

```cpp
Widget& Widget::operator=(Widget const& rhs) {
    T1 tempT1(rhs.t1_);
    T2 tempT2(rhs.t2_);
    t1_.swap(tempT1);
    t2_.swap(tempT2);
}```
The Cargill Widget Example

Widget& Widget::operator=(Widget const& rhs) {
  T1 tempT1(rhs.t1_);
  T2 tempT2(rhs.t1_);
  The Critical Line
  t1_.swap(tempT1);
  t2_.swap(tempT2);
}

// Strong Guarantee achieved!

swap()

• The Force is strong in this one. — Yoda
Where to try/catch

- Switch
- Strategy
- Some success

Switch

- Anywhere that we need to switch our method of error reporting.
Switch Cases

- Anywhere that we support the No-Throw Guarantee
- Destructors & Cleanup
- Swapperator & Moves
- C-API
- OS Callbacks
- UI Reporting
- Converting to other exception types
- Threads

Strategy

- Anywhere that we have a way of dealing with an error such as an alternative or fallback method.
Some Success

• Anywhere that partial failure is acceptable.

Using ReadRecord() which throws when the database block is corrupt.

```cpp
void DisplayContact(Database const& db, RecordID rID)
{
    ContactPtr contact(db.ReadRecord(rID));
    ContactWindowPtr contactWindow(CreateContactWindow());
    contactWindow->LoadContactData(contact);
    ...
}
```
Using `ReadRecord()` which throws when the database block is corrupt.

```cpp
void ScavengeDatabaseRecords(Database const& src, Database& dest)
{
    Recs recs(src.GetAllRecordIDs());
    for (Recs::const_iterator b(recs.begin()), e(recs.end()); b != e; ++b)
    {
        try
        {
            RecordPtr record(src.ReadRecord(*b));
            dest.WriteRecord(record);
        }
        catch (…) { /* possibly log that we failed to read this record. */ }
    }
}
```

Guideline

- Know where to catch.
  - Switch
  - Strategy
  - Some Success
“Most Important Design Guideline”

- Scott Meyers Known for C++ Advice
- Universal Design Principle
  - Not controversial

Make interfaces easy to use correctly and hard to use incorrectly.
“Most Important Design Guideline”

ErrorCode SomeCall(...);
void SomeCall(...); // throws

Guideline

• Prefer Exceptions to Error Codes
Prefer Exceptions to Error Codes

- Throwing exceptions should be mostly about resource availability
- When possible, provide defined behavior and/or use strong pre-conditions instead of failure cases
- Don't use exceptions for general flow control
  - Exceptions getting thrown during normal execution is usually an indication of a design flaw

Exception-Safety Guidelines

- Throw by value. Catch by reference.
- No dynamic exception specifications. Use noexcept.
- Destructors that throw are evil.
- Use RAI. (Every responsibility is an object. One per.)
- All cleanup code called from a destructor
- Support swapperator (With No-Throw Guarantee)
- Draw “Critical Lines” for the Strong Guarantee
- Know where to catch (Switch/Strategy/Some Success)
- Prefer exceptions to error codes.
Implementation Techniques

- on_scope_exit
- Lippincott Functions
- boost::exception
- Transitioning from legacy code
- Before and After

on_scope_exit

- Creating a struct just to do one-off cleanup can be tedious.

- That is why we have on_scope_exit.
void CTableLabelBase::TrackMove( ...) // This function
   // needs to set the cursor to the grab hand while it
{
   // executes and set it back to the open hand afterwards.
   ...

   esc::on_scope_exit handRestore(&UCursor::SetOpenHandCursor);
   UCursor::SetGrabHandCursor();
   ...
}

Handle FillNewHandleFromDataFork( ...) // This function needs to create a
   // Handle and fill it with the data from a file. If we fail in the read, we need to
   // dispose of the Handle
{
   Handle newHandle(::NewHandle( ...));
   esc::on_scope_exit handleDisposer(bind(&::DisposeHandle, newHandle));
   ...
   if ( ... successful ... )
   {
      handleDisposer.release(); // Any code path that doesn’t go through
         // here, will result in the Handle being
      } // handle being disposed of.
   ...
}
void JoelsFunction()
{
    dosomething();
    cleanup();
}

void JoelsFunction()
{
    esc::on_scope_exit clean(cleanup);
    dosomething();
}
struct on_scope_exit
{
    typedef function<void(void)> exit_action_t;

    on_scope_exit(exit_action_t action): action_(action) {}
    ~on_scope_exit() {if (action_) action_();}
    void set_action(exit_action_t action = 0) {action_ = action;}
    void release() {set_action();}

    private:
    on_scope_exit();
    on_scope_exit(on_scope_exit const&);
    on_scope_exit& operator=(on_scope_exit const&rhs);
    exit_action_t action_; 
};

... :: ... ( ... )  // This member function needs to do things that would
   // normally trigger notifications, but for the duration of
   // this call we don't want to generate notifications.
   // We can temporarily suppress these notifications by
   // setting a data member to false but we need to remember
   // to reset the value no matter how we leave the function.

... 

esc::on_scope_exit
    resumeNotify(esc::revert_value(mSendNotifications));

mSendNotifications = false;
...
template <typename T>
void set_value(T& t, T value) {t = value;}

template <typename T>
on_scope_exit::exit_action_t revert_value(T& t)
{
    return bind(set_value<T>, ref(t), t);
}

Source for esc namespace code (check_swap and on_scope_exit) is available at

http://exceptionsafecode.com
• A technique for factoring exception handling code.

• Example in *The C++ Standard Library* 2nd Ed. by Nicolai M. Josuttis page 50

```cpp
C_APIStatus C_APIFunctionCall()
{
    C_APIStatus result(kC_APINoError);
    try
    {
        CodeThatMightThrow();
    }
    catch (FrameworkException const& ex)
    {
        result = ex.GetErrorCode();
    }
    catch (Util::OSStatusException const&ex)
    {
        result = ex.GetStatus();
    }
    catch (std::exception const&)
    {
        result = kC_APIUnknownError;
    }
    catch (...)
    {
        result = kC_APIUnknownError;
    }
    return result;
}
```
C_APIStatus C_APIFunctionCall()
{
    C_APIStatus result(kC_APINoError);
    try
    {
        CodeThatMightThrow();
    }
    catch (...)
    {
        result = ErrorFromException();
    }
    return result;
}

C_APIStatus ErrorFromException()
{
    C_APIStatus result(kC_APIUnknownError);
    try
    {
        throw; // rethrows the exception caught in the caller’s catch block.
    }
    catch (FrameworkException const& ex)
    {
        result = ex.GetErrorCode();
    }
    catch (Util::OSStatusException const& ex)
    {
        result = ex.GetStatus();
    }
    catch (std::exception const&) { /* already kC_APIUnknownError */ }
    catch (…) { /* already kC_APIUnknownError */ }
    if (result == noErr) { result = kC_APIUnknownError; }
    return result;
}
An interesting implementation to support enhanced trouble-shooting.

Error detecting code may not have enough information for good error reporting.

boost::exception supports layers adding information to an exception and re-throwing

An exception to Switch/Strategy/Some Success?

---

Transitioning from pre-exception/exception-unsafe legacy code

Does not handle code path disruption gracefully

Sean Parent’s Iron Law of Legacy Refactoring

*Existing contracts cannot be broken!*
Sean’s Rules

1. All new code is written to be exception safe
2. Any new interfaces are free to throw an exception
3. When working on existing code, the interface to that code must be followed - if it wasn’t throwing exceptions before, it can’t start now
   a. Consider implementing a parallel call and re-implementing the old in terms of the new

Refactoring Steps

a. Consider implementing a parallel call and re-implementing the old in terms of the new
Refactoring Steps

1. Implement a parallel call following exception safety guidelines

2. Legacy call now calls new function wrapped in try/catch (…)
   a. Legacy API unchanged / doesn’t throw

3. New code can always safely call throwing code

4. Retire wrapper functions as appropriate

Refactoring Steps

• Moving an large legacy code base still a big chore
• Can be done in small bites
• Part of regular maintenance
• No need to swallow an elephant
• Can move forward with confidence
• Code base is never at risk!
Example Code

- First example I found
- Apple's FSCreateFileAndOpenForkUnicode sample code
- CreateReadOnlyForCurrentUserACL()
- "mbr_" and "acl_" APIs return non-zero error codes on error

```c
static acl_t CreateReadOnlyForCurrentUserACL(void)
{
    acl_t theACL = NULL;
    uuid_t theUUID;
    int    result;

    result = mbr_uid_to_uuid(geteuid(), theUUID); // need the uid for the ACE
    if (result == 0)
    {
        theACL = acl_init(1); // create an empty ACL.
        if (theACL)
        {
            Boolean freeACL = true;
            acl_entry_t newEntry;
            acl_permset_t newPermSet;

            result = acl_create_entry_np(&theACL, &newEntry, ACL_FIRST_ENTRY);
            if (result == 0)
            {
                // allow
                result = acl_set_tag_type(newEntry, ACL_EXTENDED_ALLOW);
                if (result == 0)
                {
                    // the current user
                    result = acl_set_qualifier(newEntry, (const void *)theUUID);
                    if (result == 0)
                    {
                        result = acl_get_permset(newEntry, &newPermSet);
                        if (result == 0)
                        {
                            // to read data
                            result = acl_add_perm(newPermSet, ACL_READ_DATA);
                            if (result == 0)
                            {
                                result = acl_set_permset(newEntry, newPermSet);
                                if (result == 0)
                                {
                                    freeACL = false; // all set up and ready to go
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    if (freeACL)
    {
        acl_free(theACL);
        theACL = NULL;
    }

    return theACL;
}
```
Example Code

- Rewrite Assumptions
  - All "mbr_" and "acl_" APIs throw
  - acl_t RAII Wrapper Class

Example Rewrite

- Two versions of re-writes
  - intermediate.cpp
    - Does not throw
  - after.cpp
    - throws instead of returning a code
static acl_t CreateReadOnlyForCurrentUserACL()
{
    acl_t result(0);
    try
    {
        ACL    theACL(1);
        acl_entry_t newEntry;
        acl_create_entry_np(&theACL.get(), &newEntry, ACL_FIRST_ENTRY);

        // allow
        acl_set_tag_type(newEntry, ACL_EXTENDED_ALLOW);

        // the current user
        uuid_t    theUUID;
        mbr_uid_to_uuid(geteuid(), theUUID); // need the uuid for the ACE
        acl_set_qualifier(newEntry, (const void *)theUUID);
        acl_permset_t newPermSet;
        acl_get_permset(newEntry, &newPermSet);

        // to read data
        acl_add_perm(newPermSet, ACL_READ_DATA);
        acl_set_permset(newEntry, newPermSet);

        // all set up and ready to go
        result = theACL.release();
    }
    catch (...) {}
    return result;
}
Before & After Example

- Advantages
  - More white space
  - 50% fewer lines
  - 100% fewer braces
  - 100% fewer control structures
- Easier to write and read, faster, and 100% robust

What does Exception-Safe Code look like?

- There is no “try.” — Yoda
The Coder’s Fantasy

• Writing code without dealing with failure.

The Success Path

• The power of the Exception-Safe coding guidelines is the focus on the success path.
static acl_t CreateReadOnlyForCurrentUserACL()
{
    ACL theACL(1);
    acl_entry_t newEntry;
    acl_create_entry_np(&theACL.get(), &newEntry, ACL_FIRST_ENTRY);

    // allow
    acl_set_tag_type(newEntry, ACL_EXTENDED_ALLOW);

    // the current user
    uuid_t theUUID;
    mbr_uid_to_uuid(geteuid(), theUUID); // need the uuid for the ACE
    acl_set_qualifier(newEntry, (const void *)theUUID);
    acl_permset_t newPermSet;
    acl_get_permset(newEntry, &newPermSet);

    // to read data
    acl_add_perm(newPermSet, ACL_READ_DATA);
    acl_set_permset(newEntry, newPermSet);

    // all set up and ready to go
    return theACL.release();
}

The Promise

• Easier to Read
  Easier to Understand and Maintain

• Easier to Write

• No time penalty

• 100% Robust
The Promise

• Why easier to read and write?
  • Many fewer lines of code
    • No error propagation code
  • Focus on the success path only

The Promise

• Why no time penalty?
  • As fast as if errors handling is ignored!
    • No return code checking
  • Compiler knows error handling code
    • catch blocks can be appropriately (de)optimized
The Promise

- Why 100% robust?
- Errors are never ignored
- Errors do not leave us in bad states
- No leaks

Thank you

- Visit:
  http://exceptionsafecode.com
- Send me hate mail or good reviews:
  jon@exceptionsafecode.com
- Please Follow me on Twitter:
  @JonathanKalb
- Send me your résumé:
  jonkalb@a9.com
Exception-Safe Coding

Questions?

Jon Kalb (jon@kalbweb.com)