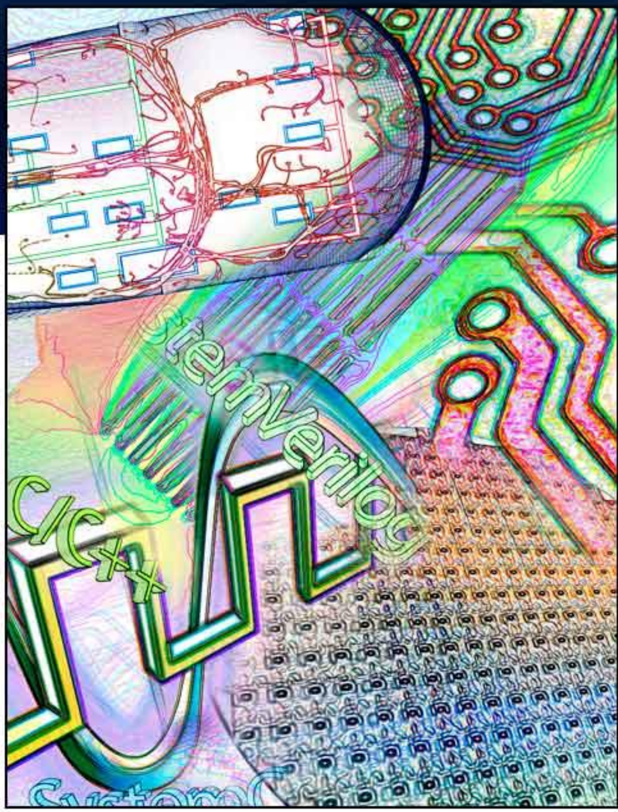

The Unexceptional Exceptions

Fedor G Pikus, Chief Scientist

Mentor Graphics, Design2Silicon Division

CPPCon, September 2015

Outline

- The motivation
- Introduction to exceptions
- Other ways to handle errors
- Handle errors not exceptions
- Simple guidelines for error handling
  - with and without exceptions
- Using exceptions in legacy code
- Additional reading

The Motivation

- The main goal of this class is not to teach how to use exceptions
  — it's not that hard
- The main goal is to teach to not be afraid to use exceptions
  — also not to be awed by them

Exceptions

- Exceptions are a language facility for handling abnormal situations at run time
  - Usually these situations are errors
- Exceptions interrupt normal control flow
  - They are used when normal control flow becomes impossible or undesired
- Exceptions create alternative control flow
  - Exception handling code is executed only when exception occurs

Basic Exceptions

```
int main() {
    try {
        Widget* w = create_widget();
        if(w=NULL) throw 1;
    }
    catch (int e) {
        cout<<”Exception “ << e << endl;
    }
}
```

Exceptional Tools

- Exception is generated using “throw”
  - throw *object*;
  - Any data type can be thrown
  - Standard provides exception base class std::exception
  - Exception must have copy constructor
  - Exception may contain data
    - throw myStatus(“Oops, bad file”);
    - You can throw a temporary object

Exceptional Circumstances

- What happens when exception is thrown?
- Execution of current scope ends
  - Code between throw and closing "}" is skipped
  - Control jumps to the closing "}"
  - Local variables are destructed
  - Stack frame is unwound (exiting the scope)
- The process is repeated until exception is caught
- If exception is not caught in main() the program terminates
  - Compiler can determine that the exception is not caught, and terminate the program immediately (without unwinding stack)

Basic Exceptions Example

```
void do_stuff() {
    Widget w;
    while (1) {
        Gadget g;
        throw 1;
        transmogrify(g,w);  // Not called
    }                       // g destroyed (may be)
}                           // w destroyed (may be)
int main() {
    do_stuff();
    more_stuff();           // Not called
}                           // Not caught
```

Basic Exceptions Example

```
void do_stuff() {
    Widget w;
    while (1) {
        Gadget g;
        throw 1;
        transmogrify(g,w);  // Not called
    }                       // g destroyed
}                           // w destroyed
int main() {
    try {
      do_stuff();
    } catch { int i } { cout << "Caught " << i << endl; }
    more_stuff();           // Called
}
```

Catching Exceptions

- Exception is handled with try-catch blocks
  - try {do_stuff();} catch(*type* e) {handle();}
  - You can catch
    - by reference: catch(int& e) {...}
    - by value: catch(int e) {...}
      - copy will be made, usually undesirable (except for basic types)
    - by pointer: catch(int* e) {...}
      - avoid if possible (who should delete the pointer?)
  - catch(...) catches any type

Catching Exceptions

- You can catch different types separately

```
try { do_stuff(); }
catch(int e) {
    cout << ”Error number “ << e; }
catch(std::exception& e) {
    cout << “Exception “ << e.what(); }
catch(...) {
    cout << “Something bad happened”; }
```

Throwing Polymorphically

- Can derived class be thrown?
  — Yes, anything can be thrown

Throwing Polymorphically

```
class D : public B {};
void f(B& throw_if_trouble) {
    if (trouble()) throw throw_if_trouble;
}
int main() {
    D status;
    try { f(status); }
    catch(D& e) { cout << “It's a D!”; }
    catch(...) {
        cout << “It's weird...”;
    }
}
```

Throwing Polymorphically

- Can derived class be thrown?
  - Yes, anything can be thrown
- Is throw itself polymorphic?
  - No, throw is not a virtual function
  - Type of the object to throw is determined at compile time
- Can exceptions be thrown polymorphically?
  - Yes, using virtual functions

Throwing Polymorphically

```
class B {
   virtual void raise() { throw *this; }
};
class D : public B {
   virtual void raise() { throw *this; }
};
void f(B& throw_if_trouble) {
    if (trouble()) throw_if_trouble.raise();
}
```

Re-throwing Exceptions

- Can exception handler itself throw?
  - Yes, but what?
  - The "current exception"

```
try { ... }
catch(...) {
    cout << "Oops!";
    throw;
}
```

- The exception continues to propagate

Re-throwing Exceptions

- Re-throw is often used to handle exceptions “in stages”
  - Exception can be modified along the way

```
void transmogrify() {
    try { ... }
    catch(status& s) {
        s.trace("Transmogrification failed");
        throw;
    }
}
```

Re-throwing Exceptions

- Exception dispatcher uses re-throw

```
void dispatcher() {
    try { throw; }
    catch(exception1& e) {
        ... handle exception1 ... }
    catch(exception2& e) {
        ... handle exception2 ... }
}
try { something_dangerous(); }
catch(...) { dispatcher(); }
```

- Allows code reuse in exception handling

Alternatives to Exceptions

- First, what’s wrong with exceptions?

Exceptional Problems

- Exceptions introduce separate execution flow
- Exception-handling code runs while the normal execution flow is suspended
  - The program is in who-knows-what state, interrupted at any random time, but exception-handling channel has to run anyway
  - Exceptions (usually) don't happen when things are going well, so the program is likely in a bad state already
- The goal of exception handling is to restore the program to a well-defined state
  - How well defined?
  - What can we expect from the program that throws?
- We need some guarantees, otherwise anything is possible

Alternatives to Exceptions

- First, what's wrong with exceptions?
- Exception-safe code is hard to write

Alternatives to Exceptions

- Main alternative to exceptions is return codes (or error codes)

```
int f() {
    Widget* w = create_widget();
    if (w == NULL) return -1;
}
```

- Return codes use the same control flow for normal and exceptional situations

The Main Difference

- Return codes can be ignored
  - Return codes often are ignored

    ```
    printf("This never fails\n");
    ```

    - printf() does return an error code!
  - If an error code is ignored, program continues
- Exceptions continue to propagate until they are caught
  - Propagate all the way to main() if not caught
  - If an exception is ignored, program terminates

Exceptional Flexibility

- Return codes are constrained by the type of return value
  - Return values may be used in normal flow
  - "Impossible" return value may not exist
- Exceptions can be of any type
  - Several unrelated types can be thrown
  - New types can be added any time
  - Exceptions are more flexible
- But there is a price for the flexibility

Exceptions and Interface

- Return values are part of the interface
  - At least the type is, the values must still be documented separately
- Exceptions are mostly invisible in the interface
  - what can this function throw?

    ```
    Widget* create_widget();
    ```
  - anything at all...
- The separate execution channel – here it is

Exceptions and Interface

- But there are throw() declarations! And noexcept()!
  - Yes, but they quite don't do what the name implies

Widget* create_widget() throw();

  - There is no compile-time check that an exception is not thrown
  - If such exception is thrown, the program will terminate (run-time check)

- No-throw guarantee can be useful, noexcept more so

void test_widget() throw(); // C++03

void test_widget() noexcept(true); // C++11

  - Still not enforced at compile time – the program may throw and terminate at run time
  - no-throw functions may throw and catch internally!

Point of No Return

- Constructors have no return values!
  - Exceptions can be thrown in constructors
  - If exception is thrown, the object was not constructed successfully
- Destructors have no return values...
  - Destructors should not throw either
  - Destructors can be called while exception is propagating and stack is unwound
    - Two exceptions cannot propagate at the same time
  - In C++11, destructors are noexcept() by default

Exceptions vs Error Codes

Exceptions	Return codes
Cannot be ignored	Can be ignored
Own execution flow	Same execution flow
Propagates until handled	Must be propagated manually
Implicit interface	Explicit interface
Flexible	Constrained
Can be used in constructors	Cannot be used in constructors

Exception Safety is Hard

```
Widget* make_widget(Log* l) {
    l->open();
    Widget* w = new Widget();
    l->record(w);
    l->close();
    return w;
}
```

- What could possibly go wrong?
  - Everything…

Exception Safety is Hard

```
Widget* make_widget(Log* l) {
    l->open();
```

- open() can throw
  - function exits, no harm

```
    Widget* w = new Widget();
    l->record(w);
    l->close();
    return w;
}
```

Exception Safety is Hard

```
Widget* make_widget(Log* l) {
    l->open();
    Widget* w = new Widget();
```

- operator new can throw
  - function exits, log file remains open

```
    l->record(w);
    l->close();
    return w;
}
```

Exception Safety is Hard

```
Widget* make_widget(Log* l) {
    l->open();
    Widget* w = new Widget();
    l->record(w);
```

- record() can throw
  - log file remains open, Widget is leaked
    - What if Widget contains a lock?

```
    l->close();
    return w;
}
```

No Exceptions - No Problems

- Rewrite everything to use error codes...

```
Widget* make_widget(Log* l) {
    l->open();
    Widget* w = new Widget();
    l->record(w);
    l->close();
    return w;
}
```

- ... and things become much simpler

See No Evil

```
Widget* make_widget(Log* l) {
    l->open();
```

- open() can return error code
  - la-la-la, I can't hear you...

```
    Widget* w = new Widget();
    l->record(w);
    l->close();
    return w;
}
```

Hear No Evil

Widget* make_widget(Log* l) {

l->open();

Widget* w = new Widget();

- memory allocation can fail, return NULL
  - la-la-la, I can't hear you...

l->record(w);

l->close();

return w;

}

Say No Evil

```
Widget* make_widget(Log* l) {
    l->open();
    Widget* w = new Widget();
    l->record(w);
```

- record(w) can fail, return error code
  - la-la-la, I can't hear you...

```
    l->close();
    return w;
}
```

No Exceptions - Same Problems

- Rewrite everything to use error codes...

```
Widget* make_widget(Log* l) {
    if (l->open()<0) return NULL;
    Widget* w = new Widget();
    if (!w) return NULL;
    if (l->record(w)<0) return NULL;
    if (l->close()<0) return NULL;
    return w;
}
```

- ... and things become just as bad, only uglier

Handle Errors, not Exceptions

- The fundamental problem is handling errors
  - not handling exceptions
- After an error is detected, the program must remain in a well-defined state
- As a minimum, no resources must be leaked
  - basic error safety guarantee
- Ideally, operation which caused error is undone
  - strong error safety guarantee
- It's hard because something went wrong ...

Exception Safety Guarantees

- No-throw guarantee
  - "We don't use exceptions here"
  - Neither do our libraries? Neither does C++ runtime…?
    - Do you allocate memory? Use dynamic casts on references?
  - What happens if something goes wrong? See the alternatives…
- No guarantees
  - "We have no idea what we are doing"
  - What happens if something goes wrong?
- If an exception is thrown, the state of the program is undefined

Exception Safety Guarantees

- Basic exception safety guarantee
  - If the exception is thrown, the program remains in a defined state
  - Defined state: some actions are possible, with known results
  - On an object that throws, some member functions may be called (possibly only the destructor)
  - Defined state is not always a very useful state
  - Termination is a well-defined state (see “not very useful” above)
- Strong exception safety guarantee
  - Any action either succeeds or has no effect (“commit or rollback”)
  - This is very hard to do, may be expensive, sometimes impossible
  - More useful in a narrow context, like a library: any member function either succeeds or leaves the object unchanged
  - Still very expensive – STL does not provide such guarantee

Practical exception safety guarantee

- Some operations may provide strong guarantee
  - STL push_back() – either succeeds, or throws and the container remains unchanged
- Most operations provide basic guarantee
  - Basic invariants are maintained (at least destructors can be called on all objects)
  - No resources are permanently leaked (memory, locks, threads,…)
    - “permanently” – you may need to destroy some objects first
  - Total cleanup on throw is not required, but there must be a way to free resources (usually an object owns them and remains valid and accessible to the caller)
- Some operations provide no-throw guarantee
  - std::swap and user-defined swap – very important!
  - All destructors

Handle Errors, not Exceptions

- Exception safety guarantees are just a particular case of error handling guarantees
- If the program attempts something and there is an error:
  — Errors cannot happen (similar to “we don’t use exceptions” but not quite as easy)
  — Anything is possible (still “we have no idea what we’re doing”)
  — The program remains in a defined state and leaks no resources (basic guarantee again)
  — Every operation will either succeed or be undone completely (strong guarantee)
- Whether errors are handled by exceptions or not, some guarantees of a defined state must be given

Guidelines

- Know and specify error handling guarantees

Exceptions are Just a Tool

```
Widget* make_widget(Log* l) {
    l->open();
    Widget* w = NULL;
    try { w = new Widget(); }
       catch{ l->close();
       throw;
    }
    try { l->record(w); }
       catch{ delete w;
       l->close();
       throw;
    }
```

- This looks a lot like error codes...

Nothing Exceptional

```
Widget* make_widget(Log* l) {
    if (l->open() < 0) return NULL;
    Widget* w = NULL;
    w = new Widget();
    if (!w) {
        l->close();
        return NULL;
    }
    if (l->record(w) < 0) {
        delete w; l->close();
        return NULL;
    }
```

- This looks a lot like try-catch blocks ...

Errors Not Exceptions

- Think about handling errors, ignore the details

```
Widget* make_widget(Log* l) {
    l->open();
    if (error) return;
    Widget* w = new Widget();
    if (error) { l->close(); return; }
    l->record(w);
    if (error) { delete w; l->close(); return; }
```

Errors Not Exceptions

- In a complex operation, all steps which succeeded before an error must be undone
  - —or at least left in a well-defined state

```
Widget* make_widget(Log* l) {
    l->open();
    on_error: { l->close(); }       // From this point on
    Widget* w = new Widget();
    on_error: { delete w; }         // From this point on
    l->record(w);
```

Errors Not Exceptions

- We need an action which is performed automatically when we leave the scope
  - for any reason (throw, return, break)

```
Widget* make_widget(Log* l) {
    l->open();
    on_error: { l->close(); }
    Widget* w = new Widget();
    on_error: { delete w; }
    …
}
```

- Local objects are destructed when leaving scope
- What about l->close() when no errors happened?

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel

Exceptionally Simple

```
struct LogGuard {
    Log* l;
    LogGuard(Log* l) :l(l) { l->open(); }
    ~LogGuard() { l->close(); }
}
template<class T> struct ScopedPtr {
    T* p_;
    ScopedPtr (T* p): p_(p) {}
    ~ ScopedPtr () { delete p_;}
}
```

- C++11 provides unique_ptr<T> - use it
  - I want to show what's happening inside

Exceptionally Simple

```
Widget* make_widget(Log* l) {
    LogGuard LG(l);
        // If anything happens log will close
    ScopedPtr <Widget> w = new Widget();
        // If anything happens w will be
        // deleted
    l->record(w);
    return w;
} // Log is closed automatically – yay!
  // w is deleted automatically - bummer
```

Exceptionally Resourceful

- We have converted the error handling problem to a resource management problem
  - Standard resource management technique in C++ is RAII – resource acquisition is initialization
- Resources which are acquired in a scope must be released upon leaving the scope
  - unless they are still owned by someone else
- C++ has many ways to manage resources and their ownership

Exceptionally Simple

```
Widget* make_widget(Log* l) {
    LogGuard LG(l);
       // If anything happens log will close
    ScopedPtr <Widget> w = new Widget();
       // If anything happens w will be
       // deleted
    l->record(w);
       // w should not be deleted now
       // we need a way to release ownership
    return w; // w is ScopedPtr not Widget*
} // Log is closed automatically – yay!
```

Manual Release

```
template<class T> struct ScopedPtr {
    T* p_;
    ScopedPtr (T* p): p_(p) {}
    ~ ScopedPtr () {
        delete p_;
    }
    T* release() {
        T* tmp = p_;
        p_ = NULL;
        return tmp;
    }
}
```

Exceptionally Simple

```
Widget* make_widget(Log* l) {
    LogGuard LG(l);
        // If anything happens log will close
    ScopedPtr <Widget> w = new Widget();
        // If anything happens w will be
        // deleted
    l->record(w);
    return w.release(); // No errors after this!
} // Log is closed automatically
```

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel
- Use RAII to automate resource cleanup

Exceptional Automation

- Using resource management consistently makes programming even easier
  - reference counting or ownership transfer
- unique_ptr<Widget> make_widget(Log* l) {

```
unique_ptr<Widget> make_widget(Log* l) {
        LogGuard LG(l);
        unique_ptr<Widget> w = new Widget();
        l->record(w);
        return w;               // pointer moved out of w
}
```

- Before C++11 you could use autoptr (and it actually did what you wanted)
- Where are the try-catch blocks?

Do or Don't Do, There Is No Try

- We are handling errors not exceptions
  - exceptions are just a tool, we could use error codes
- Error handling is mostly about making sure that resources are not leaked and program invariants are maintained
  - these actions must be taken even without errors
  - this must be done in normal control flow
  - not error-only control flow like catch blocks or error tests
- But at some point errors must be handled too

Reasons to Try (and Catch)

- Stop exceptions from propagating
  - must be done somewhere (or the program terminates)
- Process the exception, propagate the error
  - throw different exception or return an error code
    - useful when adding exceptions to “legacy” code
- Modify exception and re-throw it
  - special case, rarely used
- Clean up the try block after the error
  - NOT a reason to catch
  - use automatic scope-specific cleanup instead

Reasons to Try (and Catch)

- Actions which must be done in error channel only should be done in catch blocks
  - Stop exceptions from propagating
  - Process the exception, propagate the error
  - Modify exception and re-throw it
- Actions which must be done in error and normal channels should not be done in try blocks
  - Clean up the try block after the error

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel
- Use RAII to automate resource cleanup
- Use catch blocks only when necessary

Unnecessary Error Handling

- Manual error cleanup
- Not taking advantage of language features available for error handling
- Bad organization of error objects

Unnecessary Error Handling

- Manual error cleanup
  - deleting raw pointers, closing files, etc
  - Explicitly reclaiming non-memory resources
- Not taking advantage of language features available for error handling
- Bad organization of error objects

Resizing a Buffer

```
template <class T> Vector<T>::resize() {
    T* new_buf=allocate(2*size);
    try {
        uninitialized_array_copy(buf,buf+size,new_buf);
    } catch(...) {
        deallocate(new_buf);
        throw;
    }
    for(int i = size-1; i >= 0; i--) (buf+i)->~T();
    deallocate(buf);
    buf=new_buf; size *= 2;
}
```

- If resizing fails, the whole operation is undone

Copying Data

```
template <class T> void uninitialized_array_copy(
    T* begin, T* end, T* dest) {
     T* start = dest;
     try {
          while(begin != end)
             new(dest++) T(*begin++);
     } catch(...) {
          while(start != dest)
             start++->~T( );
          throw;
     }
}
```

- If copying fails, previous copies are destroyed

Resizing a Buffer, no Catch

```
template<class T> struct Buffer {
    T* p;
    Buffer(size_t s): p(allocate(s)) {}
    ~Buffer() { deallocate(p); }
    void swap(T* q)  noexcept(true)
        { std::swap(q,p); } // swap() does not throw!
}
template <class T> Vector<T>::resize() {
    Buffer<T> new_buf(2*size);
    uninitialized_array_copy(buf,buf+size,new_buf.p);
    for(int i = size-1; i >= 0; i--) (buf+i)->~T();
    new_buf.swap(buf);
    size *= 2;
}
```

Copying Data, No Catch

```
template<class T> struct CopyGuard {
    T* b; T*& e; bool release;
    CopyGuard(T* b, T*& e): b(b),e(e),release(false){}
    ~CopyGuard() {
        if (!release) while(b!=e) b++->~T();
    }
    void success() { release = false; }
}
template <class T> void uninitialized_array_copy(
   T* begin, T* end, T* dest) {
    CopyGuard<T> CG(dest, dest);
    while(begin != end) new(dest++) T(*begin++);
    CG.success();
}
```

Unnecessary Error Handling

- Manual error cleanup
- Not taking advantage of language features available for error handling
  - Return codes require more explicit error handling than exceptions
    - when exceptions are used correctly
- Bad organization of error objects

Exceptions vs Error Codes Again

- Using return codes:

```
Widget* make_widget() {
    if (create_widget()==NULL) return NULL;
}
Widget* process_widget() {
    if (make_widget()==NULL) return NULL;
}
```

- Errors must be propagated manually

Exceptions vs Error Codes Again

- Using exceptions poorly:

```
Widget* make_widget() {
    try { create_widget(); }
    catch(...) { ...handle error... throw; }
}
Widget* process_widget() {
    try { make_widget(); }
    catch(...) { ...handle error... throw; }
}
```

Exceptions vs Error Codes Again

- Using exceptions well:

```
Widget* make_widget() {
    ScopeGuard SG(...);
    create_widget();
}
Widget* process_widget() {
    ScopeGuard SG(...);
    make_widget();
}
```

- Exceptions continue to propagate until caught

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel
- Use RAII to automate resource cleanup
- Use catch blocks only when necessary
- Prefer exceptions to error codes

Unnecessary Error Handling

- Manual error cleanup
- Not taking advantage of language features available for error handling
- Bad organization of error objects
  - error objects could be exceptions or codes
  - different error objects in different subsystems
  - inconsistent error objects
  - error objects must be designed for the whole system
    - may be unavoidable if the system is composed from subsystems with different error handling

Error Codes and System Design

- Subsystem 1: database

```
Database<Record>::store( Record r ) {
    if ( file.fail() ) throw DB_IO_problem();
}
```

- Subsystem 2: music jukebox

```
Jukebox::add_song( SongRecord s ) {
    try { database.store(s); }
    catch(DB_IO_problem e)
       { throw Jukebox_IO_problem(); }
}
```

- Error codes can suffer from the same problem

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel
- Use RAII to automate resource cleanup
- Use catch blocks only when necessary
- Prefer exceptions to error codes
- Consider error objects part of system design

Error Codes and System Re-Design

- Subsystem 1: database (new)

```
Database<Record>::store( Record r ) {
    if ( file.fail() ) throw DB_IO_problem();
}
```

- Subsystem 2: music jukebox (old)

```
int Jukebox::add_song( SongRecord s ) {
    try { database.store(s); }
    catch(DB_IO_problem e)
        { return Jukebox_IO_problem; }
}
```

- Errors are repackaged at system boundaries

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel
- Use RAII to automate resource cleanup
- Use catch blocks only when necessary
- Prefer exceptions to error codes
- Consider error objects part of system design
- Handle/convert errors at subsystem boundaries

Becoming Exceptional

- Exceptions make error handling easier
- Most of the time we work on existing projects not start new projects
- Many existing projects do not use exceptions
  - most of those that do try too hard and catch too much
- Rewriting the whole system to use exceptions may be impossible
- How to start using exceptions?

Becoming Exceptional

- How to start using exceptions?

Becoming Exceptional

- How to start using exceptions?
  - That's not the goal, the goal is to handle errors

```
char* read_record(const char* filename){
    int fd = open(filename, O_RDONLY);
    char buffer[1024];
    int n = read(fd, buffer, 1024);
    if (n != 1024) return NULL;
    ...
    close(fd);
```

- Using RAII to close the file is more important than a particular error handling technique

Becoming Exceptional

- How to start using exceptions?
  - That's not the goal, the goal is to handle errors well

```
char* read_record(const char* filename){
    int fd = open(filename, O_RDONLY);
    char buffer[1024];
    int n = read(fd, buffer, 1024);
    if(n != 1024) return NULL;
    ...
    close(fd); // has error code too
```

- But don't neglect the advantages of exceptions over error codes either

Becoming Exceptional

- How to start using exceptions for handling errors?
- Maintain defined state, avoid leaking resources
  - use RAII for automatic resource cleanup
  - this technique does not depend on using exceptions
- Use exceptions to handle errors in new subsystems
- Convert errors at subsystem boundaries
  - catch exceptions before returning control to the old system, return error codes as that system expects

Legacy and 3rd party code

- Code from different sources may use different error handling approaches
- Convert error handling on the interfaces if necessary
- Know and maintain error handling guarantees

- Challenges:
- Error handling is not always visible in the interfaces
- Error handling is usually the worst documented part

- There is no substitute and no shortcuts for knowing what “defined state” is and what’s guaranteed

Legacy and 3rd party client code

- "Integrating 3rd party code" usually means using someone else's libraries to build applications
- What if the situation is reversed: you are building a library to serve a variety of clients?
- Convert errors at subsystem boundaries
  - Need something to convert
- Consider designing a library with multiple error-handling mechanisms

```
void Transmogrify(Doohickey& d, int* error_code=NULL);
```

or

```
void Transmogrify(Doohickey& d);
void Transmogrify(Doohickey& d, int& error_code) noexcept;
```

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel
- Use RAII to automate resource cleanup
- Use catch blocks only when necessary
- Prefer exceptions to error codes
- Consider error objects part of system design
- Handle/convert errors at subsystem boundaries
- Convert legacy code to exceptions gradually

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel
- Use RAII to automate resource cleanup
- Use catch blocks only when necessary
- Prefer exceptions to error codes
- Consider error objects part of system design
- Handle/convert errors at subsystem boundaries
- Convert legacy code to exceptions gradually

Additional Reading

- Herb Sutter, "Exceptional C++", Addison-Wesley 1999
- Herb Sutter, "More Exceptional C++", Addison-Wesley 2002
- Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley 2004, Appendix E (Standard-Library Exception Safety), http://www.research.att.com/~bs/3rd.html
- Bjarne Stroustrup, "Programming with Exceptions", 2001, http://www.informit.com/articles/article.aspx?p=21084
- Herb Sutter, "Exception Safety and Exception Specifications: Are They Worth It?", http://www.gotw.ca/gotw/082.htm
- Mark Radford, "Designing C++ Interfaces - Exception Safety", 2001, http://accu.org/index.php/journals/444
- Herb Sutter, "Unmanaged Pointers in C++: Parameter Evaluation, auto_ptr, and Exception Safety", DDJ 12.01.2002, http://www.ddj.com/cpp/184403851

Additional Reading

- Herb Sutter, "Exception-Safe Function Calls", http://www.gotw.ca/gotw/056.htm
- Andrei Alexandrescu and Petru Marginean, "Generic: Change the Way You Write Exception-Safe Code - Forever", DDJ 12.01.2000
- David Abrahams, "Lessons Learned from Specifying Exception-Safety for the C++ Standard Library", http://www.boost.org/community/exception_safety.html
- Stephen Dewhurst, "C++ Gotchas", Addison-Wesley 2002, "Gotcha 64: Throwing String Literals"
- Stephen Dewhurst, "Flexible Memory Management in C++", Software Development West 2008
- Marshall Cline, "Exceptions and error handling", http://www.parashift.com/c++-faq-lite/exceptions.html (Exceptions FAQ)

Additional Reading

- Jon Kalb, http://exceptionsafecode.com/
- David Abrahams, “Exception Safety in Generic Components”, http://www.boost.org/community/exception_safety.html
- Scott Meyers, “Declare functions as noexcept if possible”, http://scottmeyers.blogspot.com/2014/03/declare-functions-noexcept-whenever.html
- Andrzej Krzemieński, “noexcept – what for?”, https://akrzemi1.wordpress.com/2014/04/24/noexcept-what-for/
- TS Filesystem library, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4099.html - supports both exceptions and error codes

Guidelines

- Know and specify error handling guarantees
- Handle normal cleanup in normal channel
- Use RAII to automate resource cleanup
- Use catch blocks only when necessary
- Prefer exceptions to error codes
- Consider error objects part of system design
- Handle/convert errors at subsystem boundaries
- Convert legacy code to exceptions gradually
- Bonus guideline

QUESTIONS?

www.mentor.com

Bonus Nightmare Material

Bonus Nightmare Material

- C++ now supports threads
  - In practice, C++ supported threads for a long time
- Most Unix/GCC systems use POSIX Threads (PThreads)
  - pthread_create(), pthread_join(), pthread_lock(), …
- What about pthread_cancel()?
  - If a thread is terminated by pthread_cancel(), are destructors called?

Bonus Nightmare Material

- GLibC implements pthread_cancel() as an exception
- Destructors are called as the stack is unwound – yay!
- You can catch it – yay! yay..?

```
try {
  wait_on_socket();
} catch (…) {
  cout << “We just killed pthread_cancel()” << endl;
}
```

- In practice, “cancel” exception is special and the program will abort

Bonus Nightmare Material

- GLibC implements pthread_cancel() as an exception
- Destructors are called as the stack is unwound – yay!
- You can really catch it – yay!

```
try {
  wait_on_socket();
} catch (abi::__forced_unwind&) {
  throw; // Sorry, didn't mean to mess with pthread_cancel
} catch (...) {
  cout << "Something else is wrong" << endl;
}
```

- You probably want to #ifdef that for GLibC systems

Bonus Nightmare Material

- GLibC implements pthread_cancel() as an exception
- Destructors are called as the stack is unwound – yay!
- You can really catch it – yay!
- Any cancellation point can throw now – oh #^&$*!
  - That's most system calls…
  - Here goes the noexcept() guarantee
  - Aren't destructors noexcept()? Yes, they are…
  - Destructors can close files (and often do)
  - close() is a cancellation point…
- Bonus guideline – avoid pthread_cancel()