Exceptional Programming

for not so exceptional programmers

Tazio Ceri

October 19, 2016

Welcome to C, Qt or golang!

- Error handling via return values;
- possible when you have at least one "out of band" value;
- better when you have an "errorcode" out parameter;
- you cannot always use constructors properly;
- then, you will need an isvalid() method!
- In golang, there is an extra return value;
- in C, there is a global errno variable.

Not so happy days in Posix C

```
int sock = socket ( PF_INET, SOCK_STREAM, IPPROTO_TCP );
if (sock < 0) { printf(strerror(errno)); exit ( 1 ); }
...
if (bind (sock, (struct sockaddr*) &sa, sizeof(sa)) < 0)
{
    printf(strerror(errno));
    exit ( 1 );
}
if (listen( socket, 5 ) < 0)
{
    printf(strerror(errno));
    exit ( 1 );
}
```

A more general implementation in C

```
int ceiling( int p, int q, int* ok)
{
    if ( q == 0 ) { *ok = 0; return 0; }
    else *ok = 1;
    return ( p - 1 ) / q + 1;
}
int valid;
int c = ceiling(5, 3, &valid);
if ( valid ) { ... }
```

Not only if... also extra parameters!

Mandatory checking error base in golang

```
func Open(name string) (file *File, err error)

f, err := os.Open("filename.ext")
if err != nil { log.Fatal(err) }
```

It's easier in languages where there are multiple return values. This is the basic error handling management, there is also a Panic/Defer/Recover mechanism that's similar to exceptions.

Defensive programming and not so useful constructors

```
bool connect( const QUrl& url ) {
    if (!url.isValid()) {
        qDebug(QString("Invalid URL: %1").arg(url.toString()));
        return false;
    }
    .... // Getting a connection
    return true;
}
```

Checking object's validity is a direct consequence of having unuseful constructors.
Other solutions can be:

- named constructors;
- builders or factories.

Unhappyness and failure

Error handling pollutes standard program execution!

- *increased execution time*;
- *decreased readability*;
- *slow error propagation*, when you can not manage locally the error;
- *omnipresent and pervasive* validity check, that will often go wrong;
- *hard work* just to pop up the program to exit or...
- *reduced reuse* by calling exit(1) directly from your code.

Flow my errors, the policeman said

- Donald Knuth suggested keeping goto for this purpose. This is often done in C, for local errors.
- Joe Armstrong (Erlang): Repeat after me "Errors should be handled out of band in a parallel process - they are not part of the main app".

Exception guarantees

- No throw guarantee (failure transparency);
- strong guarantee (commit or rollback guarantee);
- weak gauarantee (no leak guarantee);
- death guarantee!

No throw guarantee

- Good for code that can't fail!
- In C++ you get this with the **noexcept** cv-qualifier;
- it allows optimization;
- it semplifies everything;
- use it always when you can!
- Remember that POD never throws.

Strong guarantee

- Transactional semantics;
- no side effect when exception is thrown;
- so in the end, everything has the same value as before.

Weak guarantee

- No resource leak will occur, in the exceptional case too;
- invariants are also preserved;
- objects involved may have changed their value.

Death guarantee

- Nothing is guaranteed;
- every man for himself!
- Except the author: he's fired.

Return codes and exception comparison

Exceptions

- Faster in normal case;
- slower in error case;
- impossible to ignore;
- somewhat harder to use.
- boring!

Return codes

- Slower in normal case;
- faster in error case;
- easy to ignore;
- obvious to use;
- boring!

Welcome to the modern C++ world: throw

```
int ceil( int num, int den )
{
    if ( den == 0 )
        throw std::logic_error("divide by zero");
    return ( num - 1 ) / den + 1;
}
void listen_exp( int sock, int backlog )
{
    int r = listen(sock, backlog);
    if ( r < 0 ) throw errno;
}
```

Throwing gone wrong

```
int boomer()
{
    bool wrong{false};
    int code = 0;
    code = calculate( & wrong );
    if ( wrong )
        throw &code; // Ooops
}
```

Throwing pointers is ok, but...

Throwing gone wrong

```
int boomer()
{
    bool wrong{false};
    int code = 0;
    code = calculate( & wrong );
    if ( wrong )
        throw &code; // Ooops
}
```

Throwing pointers is ok, but... the pointed object must not go out of scope!

Catching

```
int catcher()
{
  try
  {
    launcher();
  }
  catch ( int err ) { /* by value */ }
  catch ( my_exception* ex ) { /* by pointer - no! */}
  catch ( std::invalid_argument& first ) { /* by ref */ }
  catch ( const std::exception& last ) { /* const ref */}
  //catch ( somewhat&& what ) { /* Error! */ }
  catch ( ... ) { /* all */ }
}
```

Basic guidelines

- throw by value;
- catch by eventually const reference;
- the first match gets selected;
- catching order matters because Liskov's principle applies;
- therefore the catch all clause must be last.
- Liskov's principle applies also when catching by value,
- so beware of slicing!!

Challenge: managing thrown exception in constructor

```
struct File {
    File( const char* n ) { ...
        if ( ! exists(n) ) throw file_dont_exist(n);
    }
};
class FileWriter {
    File* to_open;
public:
    FileWriter(const char* name): to_open( new File( name ) ) {}
};
```

Can you make the FileWriter's constructor respect the no-throw guarantee? Can you map the thrown exception to another?

Solution: managing thrown exception in constructor

FileWriter's constructor has no no-throw implementation: a constructor must fail if construction of subobject fails.

```
FileWriter(const char* name) try : to_open( new File( name ) )
{
   ... constructor's body ...
}
catch ( ... ) { /* manage every exception here, mapping to another one*/ }
```

Function try blocks are necessary when dealing with exceptions thrown by initialization list. They are sometimes used with destructors, too. They can be used with every function, but they are not necessary to manage exceptions.

noexcept

- Use it as a function qualifier whenever you can have the no-throw guarantee;
- enables optimizations;
- it's an unevaluated context;
- it can be used to select different implementations via TMP;
- it accept a boolean argument;
- **noexcept(false)** means that the function throws
- **noexcept(true)** is implicit for destructors;
- noexcept it's not used for overloading;
- it's not enforced at compile time;
- if an exception pops up a noexcept function, std::terminate is called.

Dynamic exception specification

- Deprecated: don't use this feature!
- it has a huge overhead;
- the only real usage was throw(), now replaced by noexcept
- it's not enforced at compile time;
- when its condition is violated, it calls std::unexpected.

```
void deprecated_function()
    throws(std::logic_error, incompetence_detected)
{ ... }
```

Termination handlers

```
[[noreturn]] void terminate () noexcept {
  if( auto exc = std::current_exception() ) {
    try { rethrow_exception( exc ); }// recognize the type
    catch( std::exception const& exc ) { ... }
    catch( ... ) { ... }
  }

  std::exit( EXIT_FAILURE );
}

set_terminate( terminate ); // global
```

Termination handlers

- abort() or exit(failure) are not really avoidable;
- used for fancy logging;
- used for memory dump routines;
- not thread safe!

Destructor and exceptions - short version

- Never throw from a destructor!

Destructor and exceptions - long version part 1

- There are techniques and use cases to throw from destructors;
- nevertheless if your destructor throws from inside a container, there is no guarantee!
- Moreover, destructors are called during stack unwinding,
- that is: when an exception has already occurred.
- When the second exception gets thrown, std::terminate is called.

Destructor and exceptions - long version part 2

```
~ScopedResourceManager() noexcept(false) {
  if ( not std::uncaught_exception() ) {
     //code that may throw
  }
  else {
     // ignore every exception
  }
}
```

C++98 and C++14 solution - but has a problem. uncaught_exception returns a bool!

Destructor and exceptions - long version part 3

```
class ScopedResourceManager { int n{0}; ... };
ScopedResourceManager() { n =  std::uncaught_exceptions(); }
~ScopedResourceManager() noexcept(false) {
  try { /* do everything that may throw */ }
  catch ( ... ) {
    if ( n == std::uncaught_exceptions() - 1 ) {
       exception_ptr ep = std::current_exception();
       rethrow_exception( ep );
    } else { /* manage everything here */ }
  }
}
```

C++17! Alexandrescu asked it to Sutter, see N4152.

Nested exceptions

```
template<class T> void throw_with_nested( T&& t );
template<class E> void rethrow_if_nested( const E& e );
class nested_exception;
```

These utilities help you to rethrow and nest exceptions, when needed. A bit Java-style approach!

Zero cost exceptions

- The Linux and Win64 implementation;
- almost zero cost when no exception is thrown;
- almost means that some optimization can't be done;
- exception table is in another area of memory, no information on the stack;
- exception table may be large and slow down shared object loading;
- exception table is slow to access when exception are thrown;
- gcc uses an external library to implement stack unwinding.

Setjmp/Longjmp Exception Handling

- Win32 implementation;
- uses less memory but there is overhead when exceptions are not thrown;
- stack changes significantly.

Lippincott's function

```
status_code lippincott() {
  try { throw; }
  catch (const MyException1&) { return ERROR1; }
  catch (const MyException2&) { return ERROR2; }
  catch (...) { return UNKNOWN; }
}
extern "C" status_code func() {
   try { return do_func(); }
   catch (...) { return lippincott(); }
}
```

Useful to map exceptions to return codes, for example when interfacing with C. This function can be used only in *catch* blocks or std::terminate() will be called.

The mother of exception safety: RAII

- We all should know this idiom;
- take a resource, and only one resource, in a constructor;
- free that resource in the destructor;
- don't throw out a destructor unless you know what you are doing;
- if you are listening to me, you clearly don't know enough.

The father of exception safety: POD

- You must use no-throw data to implement one of the three exception guarantee;
- plain old data never throws;
- remember that pointers are POD!

Exercise

```
f( std::unique_ptr<T1>{ new T1 },
   std::unique_ptr<T2>{ new T2 } );
```

What exception guarantee does it have? T1 and T2 have not a noexcept constructor - nor f is noexcept!

Beware of reordering!

A perfectly legal execution order

- Allocate space for T1;
- allocate space for T2;
- call constructor for T1;
- call constructor for T2;
- construct 'T1' unique_ptr;
- construct 'T2' unique_ptr;
- f call.

No guarantee is given!

It has to do with throwing constructors, not allocation problems: if a constructor throws it's guaranteed only deallocation of its own space. Use **make_unique** to solve!

Copy problem

```
template <class T> class vector {
    T* data;
    std::size_t s;
public:
    operator=( const vector<T>& o ) {
       delete[] data;
       s = o.s;
       data = new T[s];
       for ( int i = 0; i < s, ++i ) data[i] = o.data[i];
    }
};
```

What if T's copy throws? Can we assure a strong guarantee?

Copy and swap idiom

```
vector<T>& vector<T>::operator=( const vector<T>& o )
{
   vector<T> temp( o );
   std::swap( *this, temp );
   //delete[] data;
   //s = temp.s;
   //data = temp.data;
   //temp.data == nullptr;
}
```

swap is noexcept if move construction / assignment is noexcept.

Guideline

Important

Move construction and move assignment should be noexcept.

- That makes swap work.
- So STL can select swap implementation to move assignement;
- instead of falling back to copy to guarantee the strong one!

‘

Basically all non-const operation on a container should be carefully designed. COW data structures, like those in functional languages, have inherently strong exception guarantee.

Question: how many paths?

```
String EvaluateSalaryAndReturnName( Employee e )
{
   if( e.Title() == "CEO" || e.Salary() > 100000 )
      cout << e.First() << " " << e.Last() << " is overpaid" << endl;
   return e.First() + " " + e.Last();
}
```

- Different orders of evaluating function parameters are ignored, and failed destructors are ignored.
- Called functions are considered atomic.
- To count as a different execution path, an execution path must be made up of a unique sequence of function calls performed and exited in the same way.

Answer!

- There are 3 "normal" paths.
- Exceptional paths are 20.
- In three lines!

Expection

Usage of exception in "normal" flow!

- terribly slow;
- unmantainable;
- it proves a serious design illness.

Log and throw

- Your catch blocks always log;
- and often they throw again the exception,
- because they can't handle the error there.

Log flood

All your catch blocks are log multipliers! Good luck figuring out the problem from the log's ocean.

Catchonite

- Code pollution of catches everywhere;
- performance hit;
- logging or ignoring exception is tempting;
- boring to write and mantain.
- Sometimes needed anyway.

Catch and ignore

- Good solution to turn problems into undebuggable ones;
- information get lost, nothing to do.
- Avoid to write temporarly empty catch blocks!

Catch and rethrow losing information

- Catch an exception to throw something else;
- without wrapping the original one!
- Losing information, not a good idea.

Old mindset for error handling

- Check all return value and deal with problems;
- identify all functions that throw;
- think about what to do when they do;
- use dynamic exception specification to help compiler and users;
- try/catch to control execution flow.

New good mindset

- Think about structure;
- think about invariants;
- use SOLID principles even if you are not doing real OOP.

Java's checked exception

```
public void weirdo() throws A1, A2 { ... }
```

Pro or con?

Are exceptions and errors part of interface or they should be hidden?

Links

- The original Cargill's paper against exceptions
- Alexandrescu's Declarative control flow
- Funny implementation of exception in C with longjmp and setjmp
- C++ standard proposal n4152
- Exception Safe Code by Jon Kalb
- Boost Community :: error handling
- Boost Exception

Books

- Exceptional C++
- More Exceptional C++
- Exceptional C++ style