The Exception Situation

Patrice Roy
Patrice.Roy@USherbrooke.ca
Université de Sherbrooke

CppCon 2016

Who am I?

- Father of five (four girls, one boy), ages 21 to 3
- Feeds and cleans up after a varying number of animals
- Used to write military flight simulator code, among other things
- Full-time teacher since1998
- Works a lot with game programmers
- Incidentally, WG21 and WG23 member
- Involved in SG14

How we will proceed...

- Exceptions have been a part of C++ for a long time now
  - They are not going away
- They allow programmers to concentrate on the meaningful parts of their code and treat the things that happen infrequently
  - …as… well, exceptional situations
- …to be dealt with when and where the context makes it reasonable or useful.

How we will proceed...

- On the other hand, some significant parts of the C++ programming community either dislike this mechanism or outright reject it, for a number of reasons
- Work in SG14 has raised performance issues in some cases
- Some dislike the additional execution paths introduced in programs that rely on exceptions
- Some programmers raised issues with respect to exceptions and tooling, integration with older codebases, writing robust generic code, etc.

How we will proceed...

- This talk will be neither for not against exceptions
- It will present a perspective on cases where they make sense, cases where they are less appropriate, alternative disappointment handling techniques presented along with client code in order to show how the various approaches influence the way code is written
- Performance measurements will be given along the way
- If we have enough time, some creative uses of exceptions will also be presented in order to spark ideas and discussions in the room

Expressing Disappointment

- Let's take a simplistic example

```
//
// Computes and returns the
// quotient num / denom. The
// remainder is discarded
//
int integral_div(int num, int denom);
```

- Let's call this signature the function's natural interface
- Ideally, we expect to pass it two arguments and get the result of its computation

Expressing Disappointment

- Let's take a simplistic example

```
//
// Computes and returns the
// quotient num / denom. The
// remainder is discarded
//
int integral_div(int num, int denom);
```

- What to do if `denom==0`?
- It's atypical, sure, but…

Expressing Disappointment

```
//
// Computes and returns the
// quotient num / denom. The
// remainder is discarded
//
// Precondition: denom != 0
//
int integral_div(int num, int denom);
```

- Now, the burden's on the shoulders of the caller, in a sense
- We'll suppose we want to validate this precondition (we could also claim it's « Garbage in, Garbage out »)

Expressing Disappointment

- Not an option: print out something… Ugh!

```
int integral_div(int num, int denom) {
    if (!denom)
        cerr << "Divide by zero" << endl;
    // world might collapse into
    // a singularity right there
    return num / denom;
}
```

- Beginners will do such things, before they grasp how functions interact with one another… >sigh!<
- We'll return to the « printing » question later

Expressing Disappointment

- Closing the program

```
int integral_div(int num, int denom) {
    if (!denom) {
        // abort(), terminate(), etc.
        exit(-1); // radical
    }
    return num / denom;
}
```

- A bit radical, but has merit
  - Division occurs only if it makes sense

Expressing Disappointment

- Asserting

```
int integral_div(int num, int denom) {
    assert(denom && "divide by zero");
    return num / denom;
}
```

- A bit radical, but also has merit
- Division occurs only if it makes sense
- Requires thorough testing, since `assert()` tends to disappear in so-called « Release » builds

Expressing Disappointment

- With client code

```
int main() {
    int n, d;
    if (cin >> n >> d)
        cout << integral_div(n,d) << endl;
}
```

- If `d==0`, this will terminate the program explicitly or through some integral division trap
- Preserves the function's natural interface

Expressing Disappointment

- Changing the signature
- One option is adding an output success code argument

```
int integral_div
    (int num, int denom, bool &ok) {
    if (!denom) {
        ok = false;
        return -1; // arbitrary
    }
    ok = true;
    return num / denom;
}
```

Expressing Disappointment

- Client code looks like this

```
int main() {
    int n, d;
    if (cin >> n >> d) {
        bool ok = true;
        int res = integral_div(n,d,ok);
        if (ok)
            cout << res << endl;
    }
}
```

- It's more involved, to say the least

Expressing Disappointment

- Changing the signature
- Another option is adding an output result argument and returning a success code:

```
bool integral_div
    (int num, int denom, int &res) {
    if (!denom)
       return false;
    res = num / denom;
    return true;
}
```

Expressing Disappointment

- Client code looks like this

```
int main() {
    int n, d;
    if (cin >> n >> d) {
        int res;
        if (integral_div(n,d,res))
            cout << res << endl;
    }
}
```

- Better, but only slightly
- Easy to neglect the return code
- C++17 **[[nodiscard]]** could help, but signature remains distinct from the original, intended one

Expressing Disappointment

- Changing the return type only
- One option is using `pair<int,bool>`, representing both a value and a success code

```
pair<bool,int> integral_div
    (int num, int denom) {
    if (!denom)
        return make_pair(false, -1);
    return make_pair(true, num / denom);
}
```

- This is close to the style found in the Go programming language
- Can generalize through `tuple`

Expressing Disappointment

- Client code looks like this

```
int main() {
    int n, d;
    if (cin >> n >> d) {
        auto res = integral_div(n,d);
        if (res.first)
            cout << res.second << endl;
    }
}
```

- Not bad

Expressing Disappointment

- Client code with C++17 looks like

```
int main() {
    if (int n, d; cin >> n >> d)
        if (auto [ok,res] = integral_div(n,d); ok)
            cout << res << endl;
}
```

- A bit better
  - More local

Expressing Disappointment

- Changing the return type only
- Another option is using `optional`, which might contain a result

```
optional<int>
    integral_div(int num, int denom) {
    if (!denom) return {};
    return { num / denom };
}
```

Expressing Disappointment

- Client code looks like this

```
int main() {
    int n, d;
    if (cin >> n >> d) {
        auto res = integral_div(n,d);
        if (res)
            cout << res.value() << endl;
    }
}
```

- Not bad

Expressing Disappointment

- Client code in C++17 looks like this

```
int main() {
    if (int n, d; cin >> n >> d)
        if (auto res = integral_div(n,d); res)
            cout << res.value() << endl;
}
```

- Not bad
  - More local, again

Expressing Disappointment

- Changing the return type only
- Yet another option is `expected<T,E>` which contains either a result of type `T` or an alternative of type `E`

```
class divide_by_zero{};
expected<int,divide_by_zero>
    integral_div(int num, int denom) {
    if (!denom)
        return { divide_by_zero{} };
    return { num / denom };
}
```

- This code is speculative (there's no `std::expected`)

Expressing Disappointment

- Client code looks like this

```
int main() {
    int n, d;
    if (cin >> n >> d) {
        auto res = integral_div(n,d);
        if (res)
            cout << res.value() << endl;
    }
}
```

- Not bad, like `optional`…
  - …but then, `expected` can be seen as a generalization of `optional`
- The « alternative », `E`, type of `expected<T,E>` could be a `std::error_code`

Expressing Disappointment

- Client code with C++17 looks like this

```
int main() {
    if (int n, d; cin >> n >> d)
        if (auto res = integral_div(n,d); res)
            cout << res.value() << endl;
}
```

- Again, like `optional`

Expressing Disappointment

- Throwing an exception

```
class divide_by_zero {};
int integral_division(int num, int denom) {
    if (!denom)
        throw divide_by_zero{};
    return num / denom;
}
```

Expressing Disappointment

- With client code:

```
int main() {
    int n, d;
    if (cin >> n >> d)
        cout << integral_div(n, d) << endl;
}
```

Expressing Disappointment

- With client code:

```
int main() {
    int n, d;
    if (cin >> n >> d)
        cout << integral_div(n, d) << endl;
}
```

- Exactly like the original version
- No change to the original interface

Expressing Disappointment

- With client code:

```
int main() {
    int n, d;
    if (cin >> n >> d)
        cout << integral_div(n, d) << endl;
}
```

- Exactly like the original version
- No change to the original interface
- Wait, no `try`? No `catch`?

Expressing Disappointment

- With C++17 client code:

```
int main() {
    if (int n, d; cin >> n >> d)
        cout << integral_div(n, d) << endl;
}
```

- Exactly like the original version (enhanced for C++17)
- No change to the original interface
- Wait, no `try`? No `catch`?

Expressing Disappointment

- In most cases, there's no point in catching what's been thrown
  - You catch when there are things you should do
  - Otherwise, just don't!
- Even if you catch, you don't have to handle the exception
  - You handle it if you know how to
  - Otherwise, just re-throw!

Example

- Suppose you have this class synopsis for a ***naïve*** resizable generic array

```
template <class T>
  class Array {
    T *elems; // pointer to first element
    std::size_t nelems, // number of elements
    std::size_t cap; // capacity
  public:
    std::size_t size() const noexcept {
      return nelems;
    }
    std::size_t capacity() const noexcept {
      return nelems;
    }
    // ...
  };
```

Example

```
template <class T>
  class Array {
    // ...
    using iterator = T*;
    using const_iterator = const T*;
    iterator begin() noexcept { return elems; }
    const_iterator begin() const noexcept { return elems; }
    iterator end() noexcept {
      return begin() + size();
    }
    const_iterator end() const noexcept {
      return begin() + size();
    }
    // ...
};
```

Example

```
template <class T>
  class Array {
    // ...
    bool full() const noexcept {
      return size() == capacity();
    }
    void push_back(const T &arg) {
      if (full())
        grow();
      elems[size()] = arg;
      ++nelems;
    }
    // ...
  };
```

Example

```
template <class T>
  class Array {
    // ...
  private:
    void grow() { // naïve
      std::size_t new_cap = capacity()?
        capacity() * 2 : 64;
      auto p = new T[new_cap];
      std::copy(begin(), end(), p);
      delete [] elems;
      elems = p;
      cap = new_cap;
    }
    // ...
  };
```

Example

```
// ...
void grow() { // naïve
  std::size_t new_cap = capacity()?
    capacity() * 2 : 64;
  auto p = new T[new_cap];
  std::copy(begin(), end(), p);
  delete [] elems;
  elems = p;
  cap = new_cap;
}
// ...
```

Example

```
// ...
void grow() { // naïve
  std::size_t new_cap = capacity()?
    capacity() * 2 : 64;
  auto p = new T[new_cap];
  std::copy(begin(), end(), p);
  delete [] elems;
  elems = p;
  cap = new_cap;
}
// ...
```

Example

```
// ...
void grow() { // naïve
  std::size_t new_cap = capacity()?
    capacity() * 2 : 64;
  auto p = new T[new_cap];
  std::copy(begin(), end(), p);
  delete [] elems;
  elems = p;
  cap = new_cap;
}
// ...
```

Example

```
// ...
void grow() { // naïve
  std::size_t new_cap = capacity()? capacity() * 2 : 64;
  auto p = new T[new_cap];
  try {
     std::copy(begin(), end(), p);
  } catch (...) {
     delete [] p;
     throw;
  }
  delete [] elems;
  elems = p;
  cap = new_cap;
}
// ...
```

Example

```
// ...
void grow() { // naïve
  std::size_t new_cap =
    capacity()? capacity() * 2 : 64;
  std::unique_ptr<T[]> p {
    new T[new_cap]
  }; // RAII
  std::copy(begin(), end(), &p[0]);
  delete [] elems;
  elems = p.release();
  cap = new_cap;
}
// ...
```

Expressing Disappointment

- With RAII approaches, you rarely need to resort to explicit exception handling
  - RAII is soooo nice sometimes!

Expressing Disappointment

- You do need to know **[except.terminate]** :

« In some situations exception handling must be abandoned for less subtle error handling techniques. [Note:

These situations are:

(1.1) — when the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception (15.1), calls a function that exits via an exception, or (1.2) — when the exception handling mechanism cannot find a handler for a thrown exception (15.3), [...]

In such cases, std::terminate() is called (18.8.4). In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before std::terminate() is called. […] »

Expressing Disappointment

- You do need to know **[except.terminate]** :

« […] In the situation where the search for a handler (15.3) encounters the outermost block of a function with a noexcept-specification that does not allow the exception (15.4), it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before std::terminate() is called. In all other situations, the stack shall not be unwound before std::terminate() is called. An implementation is not permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to std::terminate(). »

Expressing Disappointment

- Summary:
  - Just printing a message: normally not an option
    - Leaves the program in the dark!
  - Terminating / asserting: can make sense
    - Actually used in some SG14 communities
    - Want quick exit, maybe log error, quick restart
    - Situation should not happen: fix it offline, publish update

Expressing Disappointment

- Summary (continued):
  - Altering the signature
    - C-like solution, seen in many (ageing) APIs
    - Requires explicit validity checks in client code
    - Discipline!
    - Often aided by macros, e.g.: `if(SUCCEEDED(hr)) ...`

```
if ((cdb = cdbw_open()) == NULL)
    err(EXIT_FAILURE, "cdbw_open");

if (cdbw_put(cdb, key, strlen(key), val, strlen(val)) == -1)
    err(EXIT_FAILURE, "cdbw_put");

if (cdbw_output(cdb, fd, "my-cdb", NULL) == -1)
    err(EXIT_FAILURE, "cdbw_output");
```

Source: https://twitter.com/vzverovich/status/734776187998769152

Expressing Disappointment

- Summary (continued):
  - Enriching the return type (`pair`, `optional`, `expected`)
    - Reasonable
    - Requires explicit validity checks in client code
    - Discipline!
  - Throwing an exception
    - No change to interface
    - Client code has to insert `try ... catch` blocks if action required

What about Efficiency?

- Which options can be `constexpr`-ed?
  - Printing is out ☺
  - `terminate()`, `exit()` and `abort()` are not `constexpr`
  - The version that asserts can be rewritten as:

```
constexpr int integral_div(int num, int denom) {
  return assert(denom != 0), num / denom;
}
```

What about Efficiency?

- Which options can be `constexpr`-ed? (continued)
  - The versions with different signatures cannot be made `constexpr` with C++14, as they modify a by-ref argument

```
constexpr int
    integral_div(int num, int denom, bool &ok) {
    if (!denom) {
        ok = false;
        return -1; // arbitrary
    }
    ok = true;
    return num / denom;
}
```

What about Efficiency?

- Which options can be `constexpr`-ed? (continued)
  - With the versions with different return types, `constexpr` requires a literal return type
    - Ok with `pair`

```
constexpr pair<int,bool>
  integral_div(int num, int denom) {
    return !denom ?
      make_pair(-1,false) :
      make_pair(num/denom, true);
}
```

    - Not Ok with `optional` or `expected` due to non-trivial destructors)

What about Efficiency?

- Which options can be `constexpr`-ed? (continued)
  - With the version that throws, `constexpr` is Ok, even with C++11
  - When the throwing path is taken, computation is resolved at run-time

```
constexpr int integral_div(int num, int denom) {
    return !denom? throw divide_by_zero{}
                 : num / denom;
}
```

Exceptions – Purpose

- **The location where a problem is detected is rarely an appropriate location to handle it**
- In the `integral_div()` case, it's one thing to detect that a divide by zero would occur. It's another thing entirely to know what to do about it
  - Printing a message *might* be Ok
  - So could displaying a modal warning window
  - So could starting an emergency reactor shutdown sequence

Exceptions – Purpose

- **Error handling tends to pollute the normal code path**
- Can be seen from exception handling's origins
  - Donald Knuth suggested keeping `goto` for this purpose
  - Joe Armstrong: « Repeat after me "Errors should be handled out of band in a parallel process - they are not part of the main app" »

- Knuth: Donald E. Knuth. 1974. Structured Programming with go to Statements. *ACM Comput. Surv.* 6, 4 (December 1974), 261-301. DOI=http://dx.doi.org/10.1145/356635.356640
- Armstrong: https://twitter.com/joeerl/status/7402457403966545 92 (see also http://erlang.org/download/armstrong_thesis_2003.pdf)

Exceptions – Purpose

- Error handling tends to pollute the normal code path

Exceptions – Purpose

```
int main() {
   HRESULT hr = CoInitializeEx(0, COINIT_APARTMENTTHREADED);
   if (FAILED(hr)) {
      cerr << "CoInitializeEx()" << endl;
      return -1;
   }
   IUnknown *pUnk;
   hr = CoCreateInstance(CLSID_SumFact, 0, CLSCTX_INPROC_SERVER, IID_IUnknown, reinterpret_cast<void **> (&pUnk));
   if (FAILED(hr)) {
      cerr << "CoCreateInstance():" << hr << endl;
      CoUninitialize();
      return -2;
   }
   ISum *pSum;
   hr = pUnk->QueryInterface(IID_ISum, reinterpret_cast<void **>(&pSum));
   pUnk->Release();
   if (FAILED(hr)) {
      cerr << « Missing interface" << endl;
      CoUninitialize() ;
      return -3;
   }
   int result;
   hr = pSum->Sum(2, 3, &result);
   if (SUCCEEDED(hr))
      cout << result << endl;
   pSum->Release();
   CoUninitialize();
}
```

Exceptions – Purpose

```
int main() {
   HRESULT hr = CoInitializeEx(0, COINIT_APARTMENTTHREADED);
   if (FAILED(hr)) {
      cerr << "CoInitializeEx()" << endl;
      return -1;
   }
   IUnknown *pUnk;
   hr = CoCreateInstance(CLSID_SumFact, 0, CLSCTX_INPROC_SERVER, IID_IUnknown, reinterpret_cast<void **> (&pUnk));
   if (FAILED(hr)) {
      cerr << "CoCreateInstance():" << hr << endl;
      CoUninitialize();
      return -2;
   }
   ISum *pSum;
   hr = pUnk->QueryInterface(IID_ISomme, reinterpret_cast<void **>(&pSum));
   pUnk->Release();
   if (FAILED(hr)) {
      cerr << « Missing interface" << endl;
      CoUninitialize() ;
      return -3;
   }
   int result;
   hr = pSum->Sum(2, 3, &result);
   if (SUCCEEDED(hr))
      cout << result << endl;
   pSum->Release();
   CoUninitialize();
}
```

Exceptions – Purpose

- **Error handling tends to pollute the normal code path**

```
int main() {
   HRESULT hr = CoInitializeEx(0, COINIT_APARTMENTTHREADED);
   IUnknown *pUnk;
   hr = CoCreateInstance(
      CLSID_SumFact, 0, CLSCTX_INPROC_SERVER,
      IID_IUnknown, reinterpret_cast<void **>(&pUnk)
   );
   ISum *pSum;
   hr = pUnk->QueryInterface(IID_ISum, reinterpret_cast<void **>(&pSum));
   pUnk->Release();
   int result;
   hr = pSum->Somme(2, 3, &result);
   cout << result << endl;
   pSum->Release();
   CoUninitialize();
}
```

Exceptions – Purpose

- **Error handling tends to pollute the normal code path**

```
int main() {
   HRESULT hr = CoInitializeEx(0, COINIT_APARTMENTTHREADED);
   IUnknown *pUnk;
   hr = CoCreateInstance(
      CLSID_SumFact, 0, CLSCTX_INPROC_SERVER,
      IID_IUnknown, reinterpret_cast<void **>(&pUnk)
   );
   ISum *pSum;
   hr = pUnk->QueryInterface(IID_ISum, reinterpret_cast<void **>(&pSum));
   pUnk->Release();
   int result;
   hr = pSum->Somme(2, 3, &result);
   cout << result << endl;
   pSum->Release();
   CoUninitialize();
}
```

- We can't always make it seem like errors will not occur (as is, this code is *very* dangerous)

Exceptions – The Pros

- Exceptions have lots of upsides
  - They don't affect the natural interface of functions
    - Unless one considers `noexcept`
  - They create an alternative code path for unusual situations
  - Said otherwise, they don't pollute the normal code path
  - They separate disappointment detection from disappointment handling
    - Handling typically requires knowledge of context
  - Can be used to signal disappointment from constructors
    - Constructors have no return value

Exceptions – The Cons

- Not everyone likes exceptions (!)
  - They create an alternative codepath for unusual situations
    - Also a « pro », depending on the perspective
  - They have non-zero cost
    - Mostly, but not only, for the `catch` blocks
  - They can be abused
    - Like many features on the language
  - They pose difficulty on the boundaries where non-exception-safe code lies (C code, tools written in other languages)
    - Lippincott functions are useful here
    - See https://www.youtube.com/watch?v=3ZO0V4Prefc for an excellent presentation on the topic

Exceptions – Costs

- I ran some microbenchmarks that were shown at the sG14 meeting during CppCon 2015
- A short summary:
  - Performance comparisons between exceptions and error handling
  - On my laptop
    - MSVC14 (Visual Studio 2015), x64, Release, /EHsc
    - Clang 3.7 with Microsoft CodeGen, x64, Release, -fexceptions
  - Online compilers
    - ideone.com compiled for C++14
    - coliru.stacked-crooked.com, g++ -std=c++14 -O2 -Wall -pedantic -pthread main.cpp && ./a.out

Exceptions – Costs

- Test A – Exception[less]-Cost
  - Generate 5'000'000 consective integers starting from -100
  - Shuffle them
  - Call a function for each integer i
  - Said function, if i<0:
    - returns false (without exceptions)
    - throws (with exceptions)
    - otherwise, performs some numerical computation
  - Idea: evaluate the relative costs each approach of relatively infrequent situations (probability: 0.00002)

Exceptions – Costs

- Test B – Exception[less]-Cost-Unwinding
  - 10'000 times, call a recursive function
  - Function generates a vector short strings (subject to SSO) and calls itself recursively. Does the same thing with a vector of « long » (non subject to SSO) strings
  - At a specific depth, function fails
    - returns std::string::npos (without exceptions)
    - throws (with exceptions)
    - otherwise, performs a computation on the size of the strings
  - Idea: evaluate the relative costs of multilevel stack unwinding and error management / exception handling

Exceptions – Costs

- Test C – Exception[less]-Cost-Stack-Unwinding-Vec
  - 10'000 times, call a recursive function
  - Function generates a vector<vector<char>> of elements, and calls itself recursively
  - At a specific depth, function fails
    - returns a constant (without exceptions)
    - throws (with exceptions)
    - otherwise, performs a computation on the size of the vectors
  - Idea: evaluate the relative costs of multilevel stack unwinding and error management / exception handling

Exceptions – Costs

- Test D – Exception[less]-redundant-check
  - 10'000 times, call a recursive function
  - Function generates a vector<vector<char>> of elements, and calls itself recursively
  - At a specific depth, function fails
    - returns a constant (without exceptions)
    - throws (with exceptions)
    - otherwise, performs a computation on the size of the vectors
  - Idea: evaluate the relative costs of multilevel stack unwinding and error management / exception handling

Exceptions – Costs

- Test E – Exception[less]-never occurs
  - 10'000 times, call a recursive function
  - Function generates a vector<vector<char>> of elements, and calls itself recursively
  - At a specific depth, function fails
    - returns a constant (without exceptions)
    - throws (with exceptions)
    - otherwise, performs a computation on the size of the vectors
  - Idea: evaluate the relative costs of multilevel stack unwinding and error management / exception handling

Exceptions – Costs

- For numbers and test code, see
  - https://drive.google.com/drive/folders/0B8Nts3Mlg8gDTUQtRjM4TVVTR1E

Exceptions and the Standard Library

- Does the Standard Library throw?
  - Not really, except (!) for `vector.at()` and a few odd cases
- Mostly, exceptions come from the types used in generic code
  - And from `std::bad_alloc`

Exceptions == Errors?

- Boost.Graph
- Neat trick by Caso Neri
- Throwing a function to manage errors?

What neat trick?

- Types like `any` do type erasure, but only exact type can be extracted
  - Makes it difficult to extract an object through its base class
  - At least in a type-safe manner
- The following `any_ptr` allows type-erasing an instance of a child class, then extracting it as an instance of its direct or indirect parent classes
  - … and it's type safe
  - … and it relies on exceptions!

What neat trick?

```
class any_ptr {
public:
  any_ptr(const any_ptr&) = delete;
  any_ptr& operator=(const any_ptr&) =
delete;
private:
  void *p_;
  void (*thrower_)(void *);
  void (*destroyer_)(void *);
  // ...
```

What neat trick?

```
// ...
template <class T>
  static void thrower(void *p) {
    throw static_cast<T*>(p);
  }
template <class T>
  static void destroyer(void *p) {
    delete static_cast<T*>(p);
  }
// ...
```

What neat trick?

```
// ...
public:
  template <class T>
    any_ptr(T *p)
      : p_{p},
        thrower_{thrower<T>},
        destroyer_{destroyer<T>}
    {
    }
// ...
```

What neat trick?

```
// ...
template <class T>
  T* cast_to() const {
    try {
      thrower_(p_);
    } catch (T *p) {
      return p;
    } catch(...) {
    }
    return nullptr;
  }
// ...
```

What neat trick?

```
    // ...
    ~any_ptr() {
        destroyer_(p_);
    }
};
```

Self-diagnosing exceptions?

```
using diag_t = void(*)(ostream&);
int f(int n) {
  if (n < 0)
    throw static_cast<diag_t>([]{
      cout << "Ouch!" << endl;
    });
  return n;
}
int main() {
  try {
    cout << f(-3) << endl;
  } catch(diag_t diagnosis) {
    diagnosis();
  }
}
```

- No, I don't recommend this, but I find it cute!

References

- Lawrence Crowl, Handling Disappointment in C++
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0157r0.html
- SG14 forum
  - https://groups.google.com/a/isocpp.org/forum/#!forum/sg14
- Numerous exchanges with game developers