C++ Exceptions

Peter Chapin

CIS-3012, C++ Programming

Vermont State University

Reporting Errors

- Traditionally (i.e., in C) errors are reported by returning “funny values”

```
int ch;

ch = getc( in_file );
if( ch == EOF ) {
    // No more data (“end-of-file”), or error.
    // Note that EOF could mean an error occurred (use ferror(in_file) to check)
}
```

- In the code above, `ch` must be `int` because EOF is outside the range of all possible characters. *Many programs get this wrong!*

This Requires Handing Back Error Codes

- For example:

```
int f( ) {
    int rc = NO_ERROR;
    if (error_detected) { rc = ERROR; }
    else { … }
    return rc;
}

int g( ) {
    int rc = NO_ERROR;
    if( f( ) == ERROR ) { rc = ERROR; }
    else { … }
    return rc;
}
```

- Every call of `f` must be checked for an error code.
- If `f` fails, then g has failed, so…
- … (possibly different) error codes must be handed back from g
- <u>Error codes must be propagated manually up the call chain</u>

The Big Picture

- In general, the place where an error is detected does not know how to handle that error
  - Error detected in a general-purpose, third-party library does not know the application’s requirements
- Instead, information about the error must be propagated back on the call chain to where it can be handled
  - At a higher level in the program’s logic, the context of the error is better understood, so decisions about handling can be done there

What’s Wrong?

- Propagating error codes is tedious
  - Many programmers don’t bother to check for them, resulting in programs that behave poorly when an error occurs
  - It doesn’t help that some errors are very rare (semi-justifying the lack of a check)
- It clutters the main logic
  - Programs are harder to read when error handling logic is mingled with “normal” program flow
- It’s complicated to propagate non-trivial information
  - Returning a single integer is easy. What if more information was needed?

Enter Exceptions

- For example:

```
void f( ) {
    if (error_detected) { throw 42; }
    return;
}

void g( ) {
    f( );  // Call f and ignore error
}

void h( ) {
    try {
        g( );
    }
    catch( int error_code ) { ... }
}
```

- When an error is detected, throw an *exception object* (of any type)
- Exception object <u>propagates automatically</u> (by the runtime system)
- High level function installs handler with `try/catch` construct

Exceptions…

- … are intended to propagate error information across long distances
  - That is, over several layers of calls
  - If you find yourself throwing an exception and then catching it immediately (a couple of lines later), something is wrong
- … are intended to de-clutter program logic by pushing error handling to the side
  - Try to put exception handlers at the bottom of the functions that have them
  - You don’t want to look at error handling when trying to understand the basic operation of a function

Overhead?

- In C++ throwing an exception is expensive (i.e., time consuming)
  - Compilers optimize the "main line" where an exception is not thrown
- The runtime system *unwinds* the program stack looking for the right handler, calling destructors as it goes, etc.
  - Thus, there can be a long delay between a throw and when the handler starts executing.
- ***<u>Only Use Exceptions For Exceptional Things</u>***
  - Errors that "cannot happen" (assuming your program is right)
  - Unusual errors that rarely happen

Overused?

- Many programs (especially in other languages) overuse exceptions
  - Sometimes this is the fault of the programmer
  - Often it is the fault of the language or library being used, requiring exceptions when it doesn’t make sense to use them.
- The C++ community has considered these questions careful

Poster Child: Failure to Open a File

- It is normally better to report failure to open a file by returning an error code and *not* by way of exception
    - Failure to open a file is not exceptional. It happens all the time, e.g., when the file does not exist
    - Handling such a failure is often a local matter, so propagating error information over a long distance isn't needed, manipulating error codes is fine
    - If necessary, the error code can be translated into an exception:

```
std::ifstream input_file{ “config.txt” };
if( !input_file )
    throw BadConfiguration( “missing configuration file” );
```

Any Type?

- In C++, exception objects can be any type
    - Primitive, scalar types: integers, characters, floating point numbers
    - Pointer types (e.g., pointing at a complex, dynamically allocated object)
    - Class types (e.g., standard library strings, vectors, maps, etc.)
    - Class types (e.g., classes of the programmer’s own design)
- Unlike Java, C++ exception types need not be derived from any particular base class or be part of any particular class hierarchy
    - Although the standard library provides a class hierarchy that can be used
- Exception types can use multiple inheritance for some special effects

Try Block Syntax

```
try {
    // If an exception is thrown by this code, examine the handlers below
}
catch( int ex ) {
    // Handle a thrown integer. The exception object is named ex in this scope.
}
catch( const std::string &ex ) {
    // Handle a thrown standard string
}
catch( const std::map<std::string, std::string> &ex ) {
    // Handle a thrown map from strings to strings
}
catch( ... ) {
    // Handle everything else
}
```

More on Exception Syntax

- If no handlers match, the exception continues to propagate
- The handlers are checked in order, so if an earlier handler matches, later handers won’t be considered
  - This mostly matters when using OOP techniques
- The “catch all” handler must appear at the end if it is used at all
  - Otherwise, no other handlers would ever be matched
- Catching by reference (to const) prevents the exception object from being unnecessarily copied
- Giving a name to the exception object is optional

Finally?

- C++ try blocks do *not* have a finally clause as, for example, Java does
  - The purpose of `finally` is to execute code either when the try block exits normally OR when an exception is thrown.
  - C++ uses destructors of classes (RAI) to do this
- *When an exception propagates through a block,* ***the destructors of all fully constructed objects are executed***
  - … thus, resources are reclaimed regardless of if the code exits normally or exits by way of an exception
  - … <u>provided</u> all resource reclamation code is inside destructors
  - *Smart pointers are very helpful in this respect!*

Multiple Threads?

- In a multi-threaded program, an exception thrown in one thread has no impact on other threads
  - <u>Every thread as its own call stack</u>. Only the stack being used by the throwing thread is unwound.
  - Even if the same handler code is executed simultaneously by two threads, it doesn’t cause a problem because every thread has its own stack and its own exception object

Simple Ideas

- Instead of returning an integer error code, why not throw the error code (i.e., throw the integer instead of returning it)?
  - Useful if the handler is “far away” from the point where the error is detected
- Throw a human-readable string containing an error message
  - Useful to humans, less useful to other parts of the program unless the string encodes relevant data about the exception in an easy-to-parse way
- Throw a structure containing information about the error so the handler has all the information it needs to fully handle the problem

Example (In General Purpose Library)

```
struct ConfigError {
    std::string     file_name;
    FileCoordinates file_coordinates;
    std::string     config_key;
    std::string     config_value;
};

// Throw an instance of the structure initialized as indicated.
// Here we are reporting a problem on line 10, column 43 of “config.txt”: the
// key SILLY_KEY has a bad value: 1.0.
//
// Note the explicit constructor call for class FileCoordinates.

throw ConfigError{ “config.txt”, FileCoordinates(10, 43), “SILLY_KEY”, “1.0” };
```

Now, The Handler (In Application Code)

```
catch( const ConfigError &ex ) {
    // Ignore errors in SILLY_KEY
    if( ex.config_key != “SILLY_KEY” ) {
        ostringstream formatter;
        formatter << “FILE: ” << ex.file_name
        formatter << “, LINE: ” << ex.file_coordinates.line
        formatter << “, COLUMN: ” << ex.file_coordinates.column
        DialogBox( window_handle, formatter.str( ).c_str( ), DLG_ALERT, DLG_OK );
    }
}
```

Exceptions and Inheritance

```
// First, we define an inheritance hierarchy

// For brevity, I elide the contents of these classes.
// In a real application they would have interesting constructors and internals

class Animal { … }
class Reptile : public Animal { … }
class Mammal : public Animal { … }
class Cat : public Mammal { … }
class Dog : public Mammal { … }
```

Exceptions and Inheritance

```
// Examples of handlers

catch( const Cat &ex ) {
    // Handle thrown cats
}
catch( const Reptile &ex ) {
    // Handle thrown reptiles of any kind
    // Meaning all classes derived from Reptile
    // Calling virtual methods on ex will dispatch in the usual way
}
catch( const Animal &ex ) {
    // Handle all remaining animals
    ex.make_noise( );    // Bark? Squeal? Trumpet? Cluck?
}
```

Exceptions and Inheritance

```
// A more compelling example

class NetworkError { … }
class ClientError : public NetworkError { … }
class ServerError : public NetworkError { … }
class AddressError : public NetworkError { … }
class IPv4AddressError : public AddressError { … }
class IPv6AddressError : public AddressError { … }
```

Standard Exceptions

- The standard library has a hierarchy of exceptions already
- It is widely used, but optional
- You can hook into the standard hierarchy by deriving your exception classes from it
  - This allows code that catches the standard classes to also catch your exceptions without knowing anything about your exceptions

Meet The Family

- `#include <stdexcept>`

```
namespace std {
    class exception { … };
    class logic_error             : public exception { … };
        class domain_error        : public logic_error { … };
        class invalid_argument    : public logic_error { … };
        class length_error        : public logic_error { … };
        class out_of_range        : public logic_error { … };
    class runtime_error           : public exception { … };
        class range_error.        : public runtime_error { … };
        class overflow_error      : public runtime_error { … };
        class underflow_error     : public runtime_error { … };
}
```

What?

- All the standard exception classes have a constructor taking a reference to a `const std::string`
- All the standard exception classes have a `what` method that returns that string (as a `const char *`)
- The intent is for this string to contain a (potentially) human-readable message describing the exception
- It could also be used to encode other information if desired

Logic vs Runtime

- A *logic error* is an error in the program itself. In theory they can be prevented by the programmer.
    - In other words, <u>a logic error is a program bug</u> that was detected by the program as it runs
- A *runtime error* is an error that arises from the environment in which the program executes. It is not the program's fault.
    - It could be argued that runtime errors should not be reported as exceptions at all, but if the error is obscure enough it might still make sense to use an exception

Logic Errors

- `domain_error`
  - An argument to a function is outside the function’s domain (e.g., negative values to a square root function)
- `invalid_argument`
  - An argument to a function is invalid. Similar to `domain_error`, except used for non-mathematical functions (e.g., bad string format)
- `length_error`
  - To “report an attempt to produce an object whose length exceeds its maximum allowable size.”
- `out_of_range`
  - To “report an argument value not in its expected range.”

Runtime Errors

- `range_error`
  - To “report range errors in internal computations.” Here *range* is used in a set-theoretic way to mean the computed result of a function is not in the allowed set of possibilities, e.g., a string can’t be made with the right format
- `overflow_error`
  - To “report an arithmetic overflow error.” Although an arithmetic overflow is a kind of range error, this exception is intended to be used when numeric computations go beyond the allowed range of their type
- `underflow_error`
  - To “report an arithmetic underflow error.”

Example

- An exception to throw if a function/method is not implemented:

```
class NotImplemented : public std::logic_error {
public:
    explicit NotImplemented( const std::string &message ) :
        std::logic_error( message )
    { }
};
```

  - Derived from `logic_error` since calling an unimplemented function is a bug
  - Constructor takes a message string that is used to initialize the base class, becoming the *what* message for the exception object
  - No other data members or methods in this example. <u>There could be others!</u>

```
throw NotImplemented( "f( int, const string & ) not implemented" );
```

Other Standard Exceptions

- The C++ standard defines other exceptions derived from `std::exception` for various special purposes. For example:
- `std::bad_alloc` is thrown when new fails to find memory
  - This is rare: virtual memory means programs don’t usually run out
  - … and if they do, the OS will have bigger problems!
- The `bad_alloc` exception can still be thrown when…
  - … you are programming for a highly constrained (embedded) system without virtual memory support
  - … the process has a resource quota imposed on it that limits its memory usage

Exception Specifications?

- Java allows you to declare which exceptions a method might throw
  - Compiler verifies (statically) that all declared exceptions of called methods are either handled or declared to be thrown
  - The idea was to ensure that no exceptions will go unnoticed/unhandled
- Problems:
  - Not all exceptions are “checked”, and they can be thrown even without the specification. Thus, the feature doesn’t really guarantee that there are no unhandled exceptions
  - Sometimes, because of the logic of your program, you know an exception won’t be thrown, yet you are forced to handle an exception that can’t happen or declare that you will throw an exception you never will

Controversial

- For the reasons stated earlier, Java exception specifications are controversial
  - Some believe they are a mistake in the design of Java
  - Some believe they are useful tools to help make more reliable programs
  - Perhaps both are right?
- Most modern languages do not have exception specifications
  - For example, Scala does not, despite targeting the JVM

What about C++?

- C++ 1998 had a form of exception specifications
- It was even more controversial than Java’s
  - It did not provide much help with creating more reliable programs
  - C++ needed to be compatible with legacy (pre-1998) code and with C, which limited the design options for exception specifications
- It basically didn’t work, and it caused more problems than it solved
- A “best practice” quickly arrived: <u>*Don’t Use Exception Specifications*</u>!
- Deprecated in C++ 2011. Completely removed in C++ 2017
  - A rare case of the language actually getting *smaller*! (*gasp*)

With One (um) Exception…

- General exception specifications have questionable utility…
- … but it is useful to know if any exceptions will be thrown at all
- C++ 2011 (and beyond) allows you to declare `noexcept` functions

```
void swap_ints( int &x, int &y ) noexcept
{
    int temp = x;
    x = y;
    y = temp;
}
```

A promise that this function will not throw

Copying integers does not throw. Promise kept!

What’s the Point?

- Better documentation. To evaluate exception safety, you need to know what does *not* throw
  - Expressions involving primitive types
  - (Raw) pointer manipulation
  - Accessing memory (might cause UB, i.e., out-of-bounds access)
  - Calling any function marked as `noexcept` (tools can potentially help)
- Better optimization.
  - Generating code to deal with exceptions is complicated for the compiler
  - Functions not marked as `noexcept` are assumed to maybe throw
  - Thus, `noexcept` makes it possible for the compiler to simplify the code

What If You Lie?

- For example:

```
void f( ) noexcept
{
    throw std::runtime_error( “BWHAHAHA!!” );
}
```

- *The program terminates at once!*
- This might seem harsh, but…
  - … the program is violating its contracts
  - This is an internal matter; a bug to be caught during testing
- <u>It is *not* a compile-time error!</u>