

CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 19
November 8, 2011

Exceptions

Exceptions

Exceptions

An *exception* is an event that prevents normal continuation.

Exceptions may be due to program errors or data errors, but they may also be due to external events:

- ▶ File not found.
- ▶ Insufficient permissions.
- ▶ Network failure.
- ▶ Read error.
- ▶ Out of memory error.

How to respond to different kinds of exceptions is application-dependent.

Exception handling

When an exception occurs, a program has several options:

- ▶ Try again.
- ▶ Try something else.
- ▶ Give up.

Problem: Exceptions are often detected at a low level of the code.
Knowledge of how to respond resides at a higher level.

C-style solution using status returns

The C library functions generally report exceptions by returning status values or error codes.

Advantages: How to handle exception is delegated to the caller.

Disadvantages:

- ▶ Every caller must handle every possible exception.
- ▶ Exception-handling code becomes intermingled with the “normal” operation code, making program much more difficult to comprehend.

C++ exception mechanism

C++ exception mechanism is a means for a low-level routine to report an exception directly to a higher-level routine.

This separates exception-handling code from normal processing code.

An exception is reported using the keyword `throw`.

An exception is handled in a `catch` block.

Each routine in the chain between the reporter and the handler is exited cleanly, with all destructors called as expected.

Throwing an exception

`throw` is followed by an *exception* value.

Exceptions are usually objects of a user-defined exception type.

Example:

```
throw AgeError("Age can't be negative");
```

Exception class definition:

```
class AgeError {  
    string msg;  
public:  
    AgeError(string s) : msg(s) {}  
    ostream& printError(ostream& out) const { return out<< msg; }  
};
```


Catching an exception

A `try` region defines a section of code to be monitored for exceptions.

Immediately following are `catch` blocks for handling the exceptions.

```
try {  
    ... //run some code  
}  
catch (AgeError& aerr) {  
    // report error  
    cout<< "Age error: ";  
    aerr.printStackTrace( cout )<< cout;  
    // ... recover or abort  
}
```

The `catch` parameter should generally be a reference parameter as in this example.

What kind of object should an exception throw?

`catch` filters the kinds of exceptions it will catch according to the type of object thrown.

For this reason, **each kind of exception should throw its own type of object.**

That way, an exception handler appropriate to that kind of exception can catch it and process it appropriately.

While it may be tempting to throw a string that describes the error condition, it is difficult to process such an object except by printing it out and aborting (like `fatal()`).

Properly used, exceptions are much more powerful than that.

Standard exception class

The standard C++ library provides a polymorphic base class `std::exception` from which all exceptions thrown by components of the C++ Standard library are derived.

These are:

| exception | description |
|--------------------------------|--|
| <code>bad_alloc</code> | thrown by <code>new</code> on allocation failure |
| <code>bad_cast</code> | thrown by a failed <code>dynamic_cast</code> |
| <code>bad_exception</code> | thrown when an exception type doesn't match any catch |
| <code>bad_typeid</code> | thrown by <code>typeid</code> |
| <code>ios_base::failure</code> | thrown by functions in the <code>iostream</code> library |

(from <http://www.cplusplus.com/doc/tutorial/exceptions/>)

Catching standard exceptions

Class `std::exception` contains a virtual function

```
const char* what() const;
```

that is overridden in each derived exception class to provide a meaningful error message.

Because the base class is polymorphic, it is possible to write a single `catch` handler that will catch all derived exception objects.

Example:

```
catch (exception& e)
{
    cerr << "exception caught: " << e.what() << endl;
}
```

Deriving your own exception classes from `std::exception`

```
#include <iostream>
#include <exception>
using namespace std;
class myexception: public exception {
    virtual const char* what() const throw()
        { return "My exception happened"; }
} myex; // declares class and instantiates it
int main () {
    try {
        throw myex;
    }
    catch (exception& e) {
        cout << e.what() << endl;
    }
    return 0;
}
```

Multiple catch blocks

- ▶ Can have multiple `catch` blocks to catch different classes of exceptions.
- ▶ They are tried in order, so the more specific should come before the more general.
- ▶ Can have a “catch-all” block `catch (...)` that catches all exceptions. (This should be placed last.)

Rethrow

A `catch` block can do some processing and then optionally rethrow the exception or throw a new exception.

- ▶ One exception can cause multiple `catch` blocks to execute.
- ▶ To rethrow the same exception, use `throw;` with no argument.
- ▶ To throw a new exception, use `throw` as usual with an argument.

A subtle fact about rethrow

Rethrowing the current exception is not the same as throwing an exception with the same exception object.

`throw e;` always **copies** object `e` to special memory using the copy constructor for `e`'s class.

`throw;` **does not make another copy** of the exception object but instead uses the copy already in special memory.

This difference becomes apparent if the copy is not identical to the original (possible for a custom copy constructor), or if the copy constructor has side effects (such as printing output).

Example

```
1  #include <iostream>
2  using namespace std;
3  class MyException {
4  public:
5      MyException() {}
6      MyException( MyException& e ) {
7          cout << "Copy constructor called\n"; }
8      ~MyException() {}
9  } myex; // declares class and instantiates it
10 int main () {
11     try {
12         try { throw myex; }
13         catch (MyException& e) {
14             cout << "Exception caught by inner catch\n"; throw; }
15     }
16     catch (MyException& err) {
17         cout << "Exception caught by outer catch\n";
18     }
19     return 0;
20 }
```

Results

In the preceding example, the `throw myex` on line 12 causes a copy, but the `throw` on line 14 does not.

This produces the following output:

```
Copy constructor called  
Exception caught by inner catch  
Exception caught by outer catch
```

Throw restrictions

It is possible to specify that a function can only throw certain kinds of exceptions (or none at all).

This “feature” is regarded as a bad idea because the current semantics are not what one would expect.

It does not prevent the exceptions from being thrown; rather, it causes a run-time test to be inserted which calls `unexpected_exception()` when an exception is thrown that is not listed in the function’s throw specifier.

Uncaught exceptions: Ariane 5

Uncaught exceptions have led to spectacular disasters.

The European Space Agency's Ariane 5 Flight 501 was destroyed 40 seconds after takeoff (June 4, 1996). The US\$1 billion prototype rocket self-destructed due to a bug in the on-board guidance software. [\[Wikipedia\]](#)

This is not about a programming error.

It is about system-engineering and **design failures**.

The software did what it was designed to do and what it was agreed that it should do.

Uncaught exceptions: Ariane 5 (cont.)

Here's a summary of the events and its import for system engineering:

- ▶ A decision was made to leave a program running after launch, even though its results were not needed after launch.
- ▶ An overflow error happened in that calculation,
- ▶ An exception was thrown and, by design, was not caught.
- ▶ This caused the vehicle's active and backup inertial reference systems to shut down automatically.

As the result of the unanticipated failure mode and a diagnostic message erroneously treated as data, the guidance system ordered violent attitude correction. The ensuing disintegration of the over-stressed vehicle triggered the pyrotechnic destruction of the launcher and its payload.

Termination

There are various conditions under which the exception-handling mechanism can fail. Two such examples are:

- ▶ Exception not caught by any `catch` block.
- ▶ A destructor issues a `throw` during the stack-unwinding process.

When this happens, the function `terminate()` is called, which by default aborts the process.¹

This is a **bad thing** in production code.

Conclusion: All exceptions should be caught and dealt with explicitly.

¹It's behavior can be changed by the user.