Exceptions and Error Handling ¶ Δ

Contents of this section:

FAQ Why use exceptions?

What good can using exceptions do for me? The basic answer is: Using exceptions for error handling makes your code simpler, cleaner, and less likely to miss errors. But what's wrong with "good old `errno` and `if`-statements"? The basic answer is: Using those, your error handling and your normal code are closely intertwined. That way, your code gets messy and it becomes hard to ensure that you have dealt with all errors (think "spaghetti code" or a "rat's nest of tests").

First of all there are things that just can't be done right without exceptions. Consider an error detected in a constructor; how do you report the error? You throw an exception. That's the basis of RAII (Resource Acquisition Is Initialization), which is the basis of some of the most effective modern C++ design techniques: A constructor's job is to establish the invariants for the class (create the environment in which the member functions are to run) and that often requires the acquisition of resources, such as memory, locks, files, sockets, etc.

Imagine that we did not have exceptions, how would you deal with an error detected in a constructor? Remember that constructors are often invoked to initialize/construct objects in variables:

```
vector<double> v(100000);   // needs to allocate memory
ofstream os("myfile");      // needs to open a file
```

The `vector` or `ofstream` (output file stream) constructor could either set the variable into a "bad" state (as `ifstream` does by

default) so that every subsequent operation fails. That's not ideal. For example, in the case of `ofstream`, your output simply disappears if you forget to check that the open operation succeeded. For most classes that results are worse. At least, we would have to write:

```
vector<double> v(100000);   // needs to allocate memory
if (v.bad()) { /* handle error */ } // vector doesn't actually ha
ofstream os("myfile");      // needs to open a file
if (os.bad())  { /* handle error */ }
```

That's an extra test per object (to write, to remember or forget). This gets really messy for classes composed of several objects, especially if those sub-objects depend on each other. For more information see The C++ Programming Language section 8.3, Chapter 14, and Appendix E or the (more academic) paper Exception safety: Concepts and techniques .

So writing constructors can be tricky without exceptions, but what about plain old functions? We can either return an error code or set a non-local variable (e.g., `errno`). Setting a global variable doesn't work too well unless you test it immediately (or some other function might have re-set it). Don't even think of that technique if you might have multiple threads accessing the global variable. The trouble with return values are that choosing the error return value can require cleverness and can be impossible:

```
double d = my_sqrt(-1);     // return -1 in case of error
if (d == -1) { /* handle error */ }
int x = my_negate(INT_MIN); // Duh?
```

There is no possible value for `my_negate()` to return: Every possible `int` is the correct answer for some `int` and there is no correct answer for the most negative number in the twos-complement representation. In such cases, we would need to return pairs of values (and as usual remember to test) See Stroustrup's Beginning programming book for more examples and explanations.

Common objections to the use of exceptions:

- *"But exceptions are expensive!"* Not really. Modern C++ implementations reduce the overhead of using exceptions to a

few percent (say, 3%) and that's compared to *no error handling*. Writing code with error-return codes and tests is not free either. As a rule of thumb, exception handling is extremely cheap when you don't throw an exception. It costs nothing on some implementations. All the cost is incurred when you throw an exception: that is, "normal code" is faster than code using error-return codes and tests. You incur cost only when you have an error.

- *"But in JSF++ Stroustrup himself bans exceptions outright!"* JSF++ is for hard-real time and safety-critical applications (flight control software). If a computation takes too long someone may die. For that reason, we have to guarantee response times, and we can't – with the current level of tool support – do that for exceptions. In that context, even free store allocation is banned! Actually, the JSF++ recommendations for error handling simulate the use of exceptions in anticipation of the day where we have the tools to do things right, i.e. using exceptions.
- *"But throwing an exception from a constructor invoked by `new` causes a memory leak!"* Nonsense! That's an old-wives' tale caused by a bug in one compiler – and that bug was immediately fixed over a decade ago.

FAQ How do I use exceptions?

See The C++ Programming Language section 8.3, Chapter 14, and Appendix E . The appendix focuses on techniques for writing exception-safe code in demanding applications, and is not written for novices.

In C++, exceptions are used to signal errors that cannot be handled locally, such as the failure to acquire a resource in a constructor. For example:

```
class VectorInSpecialMemory {
    int sz;
    int* elem;
public:
    VectorInSpecialMemory(int s)
        : sz(s)
        , elem(AllocateInSpecialMemory(s))
    {
        if (elem == nullptr)
            throw std::bad_alloc();
    }
    ...
};
```

Do not use exceptions as simply another way to return a value from a function. Most users assume – as the language definition encourages them to – that ** *exception-handling code is error-handling code* **, and implementations are optimized to reflect that assumption.

A key technique is resource acquisition is initialization (sometimes abbreviated to RAII), which uses classes with destructors to impose order on resource management. For example:

```
void fct(string s)
{
    File_handle f(s,"r");   // File_handle's constructor opens th
    // use f
} // here File_handle's destructor closes the file
```

If the "use f" part of fct() throws an exception, the destructor is still invoked and the file is properly closed. This contrasts to the common unsafe usage:

```
void old_fct(const char* s)
{
    FILE* f = fopen(s,"r"); // open the file named "s"
    // use f
    fclose(f);  // close the file
}
```

If the "use `f`" part of `old_fct` throws an exception – or simply does a return – the file isn't closed. In C programs, `longjmp()` is an additional hazard.

FAQ What shouldn't I use exceptions for?

C++ exceptions are designed to support **error handling**.

- Use `throw` only to signal an error (which means specifically that the function couldn't do what it advertised, and establish its postconditions).
- Use `catch` only to specify error handling actions when you know you can handle an error (possibly by translating it to another type and rethrowing an exception of that type, such as catching a `bad_alloc` and rethrowing a `no_space_for_file_buffers`).
- **Do not** use `throw` to indicate a coding error in usage of a function. Use assert or other mechanism to either send the

process into a debugger or to crash the process and collect the crash dump for the developer to debug.

- **Do not** use `throw` if you discover unexpected violation of an invariant of your component, use assert or other mechanism to terminate the program. Throwing an exception will not cure memory corruption and may lead to further corruption of important user data.

There are other uses of exceptions – popular in other languages – but not idiomatic in C++ and deliberately not supported well by C++ implementations (those implementations are optimized based on the assumption that exceptions are used for error handling).

In particular, do not use exceptions for control flow. `throw` is not simply an alternative way of returning a value from a function (similar to `return`). Doing so will be slow and will confuse most C++ programmers who are rightly used to seeing exceptions used only for error handling. Similarly, `throw` is not a good way of getting out of a loop.

FAQ What are some ways `try` / `catch` / `throw` can improve software quality?

By eliminating one of the reasons for `if` statements.

The commonly used alternative to `try` / `catch` / `throw` is to return a *return code* (sometimes called an *error code*) that the caller explicitly tests via some conditional statement such as `if`. For example, `printf()`, `scanf()` and `malloc()` work this way: the caller is supposed to test the return value to see if the function succeeded.

Although the return code technique is sometimes the most appropriate error handling technique, there are some nasty side effects to adding unnecessary `if` statements:

- **Degrade quality:** It is well known that conditional statements are approximately ten times more likely to contain errors than any other kind of statement. So all other things being equal, if you can eliminate conditionals / conditional statements from your code, you will likely have more robust code.

- **Slow down time-to-market:** Since conditional statements are branch points which are related to the number of test cases that are needed for white-box testing, unnecessary conditional statements increase the amount of time that needs to be devoted to testing. Basically if you don't exercise every branch point, there will be instructions in your code that will *never* have been executed under test conditions until they are seen by your users/customers. That's bad.
- **Increase development cost:** Bug finding, bug fixing, and testing are all increased by unnecessary control flow complexity.

So compared to error reporting via return-codes and `if`, using `try` / `catch` / `throw` is likely to result in code that has fewer bugs, is less expensive to develop, and has faster time-to-market. Of course if your organization doesn't have any experiential knowledge of `try` / `catch` / `throw`, you might want to use it on a toy project first just to make sure you know what you're doing — you should always get used to a weapon on the firing range before you bring it to the front lines of a shooting war.

FAQ I'm still not convinced: a 4-line code snippet shows that return-codes aren't any worse than exceptions; why should I therefore use exceptions on an application that is orders of magnitude larger?

Because exceptions scale better than return-codes.

The problem with a 4-line example is that it has only 4 lines. Give it 4,000 lines and you'll see the difference.

Here's a classic 4-line example, first with exceptions:

```
try {
  f();
  // ...
} catch (std::exception& e) {
  // ...code that handles the error...
}
```

Here's the same example, this time using return-codes (`rc` stands for "return code"):

```
int rc = f();
```

```
if (rc == 0) {
  // ...
} else {
  // ...code that handles the error...
}
```

People point to those "toy" examples and say, "Exceptions don't improve coding or testing or maintenance cost in that; why should I therefore use them in a 'real' project?"

Reason: exceptions help you with real-world applications. You won't likely see much if any benefit on a toy example.

In the real world, the code that *detects* a problem must typically propagate error information back to a different function that will *handle* the problem. This "error propagation" often needs to go through dozens of functions — `f1()` calls `f2()` calls `f3()`, etc., and a problem is discovered way down in `f10()` (or `f100()`). The information about the problem needs to get propagated all the way back to `f1()`, because only `f1()` has enough context to actually know what should be done about the problem. In an interactive app, `f1()` is typically up near the main event loop, but no matter what, the code that detects the problem often isn't the same as the code that handles the problem, and the error information needs to get propagated through all the stack frames in between.

Exceptions make it easy to do this "error propagation":

```
void f1()
{
  try {
    // ...
    f2();
    // ...
  } catch (some_exception& e) {
    // ...code that handles the error...
  }
}

void f2() { ...; f3(); ...; }
void f3() { ...; f4(); ...; }
void f4() { ...; f5(); ...; }
void f5() { ...; f6(); ...; }
void f6() { ...; f7(); ...; }
void f7() { ...; f8(); ...; }
void f8() { ...; f9(); ...; }
void f9() { ...; f10(); ...; }

void f10()
{
  // ...
  if ( /*...some error condition...*/ )
    throw some_exception();
  // ...
}
```

Only the code that detects the error, `f10()`, and the code that handles the error, `f1()`, have any clutter.

However using return-codes forces "error propagation clutter" into all the functions in between those two. Here is the equivalent code that uses return codes:

```
int f1()
{
  // ...
  int rc = f2();
  if (rc == 0) {
    // ...
  } else {
    // ...code that handles the error...
  }
}

int f2()
{
  // ...
  int rc = f3();
  if (rc != 0)
    return rc;
  // ...
  return 0;
}

int f3()
{
  // ...
  int rc = f4();
  if (rc != 0)
    return rc;
  // ...
  return 0;
}

int f4()
{
  // ...
  int rc = f5();
  if (rc != 0)
    return rc;
  // ...
  return 0;
}

int f5()
{
  // ...
  int rc = f6();
  if (rc != 0)
    return rc;
  // ...
  return 0;
}

int f6()
{
  // ...
  int rc = f7();
  if (rc != 0)
    return rc;
  // ...
  return 0;
}

int f7()
{
```

```
  // ...
  int rc = f8();
  if (rc != 0)
    return rc;
  // ...
  return 0;
}

int f8()
{
  // ...
  int rc = f9();
  if (rc != 0)
    return rc;
  // ...
  return 0;
}

int f9()
{
  // ...
  int rc = f10();
  if (rc != 0)
    return rc;
  // ...
  return 0;
}

int f10()
{
  // ...
  if (...some error condition...)
    return some_nonzero_error_code;
  // ...
  return 0;
}
```

The return-code solution "spreads out" the error logic. Functions `f2()` through `f9()` have explicit, hand-written code related to propagating the error condition back up to `f1()`. That is badness:

- It clutters functions `f2()` through `f9()` with extra decision logic — the most common cause of bugs.
- It increases the bulk of the code.
- It clouds the simplicity of the programming logic in functions `f2()` through `f9()`.
- It requires the return value to perform two distinct duties — functions `f2()` through `f10()` will need to handle both *"my function succeeded and the result is `xxx`"* and *"my function failed and the error information is `yyy`"*. When the types of `xxx` and `yyy` differ, you sometimes need extra by-reference parameters to propagate both the "successful" and "unsuccessful" cases to the caller.

If you look very narrowly at `f1()` and `f10()` in the above examples, exceptions won't give you much of an improvement. But

if you instead open your eyes to the big picture, you will see a substantial difference in all the functions in between.

Conclusion: one of the benefits of exception handling is a cleaner, simpler way to propagate error information back to the caller that can handle the error. Another benefit is your function doesn't need extra machinery to propagate both the "successful" and "unsuccessful" cases back to the caller. Toy examples often don't emphasize either error propagation or handling of the two-return-types problem, therefore they don't represent Real World code.

FAQ How do exceptions simplify my function return type and parameter types?

When you use return codes, you often need two or more distinct return values: one to indicate that the function succeeded and to give the computed result, and another to propagate the error information back to the caller. If there are, say, 5 ways the function could fail, you could need as many as 6 different return values: the "successful computation" return value, and a possibly different package of bits for each of the 5 error cases.

Let's simplify it down to two cases:

- "I succeeded and the result is `xxx`."
- "I failed and the error information is `yyy`."

Let's work a simple example: we would like to create a `Number` class that supports the four arithmetic operations: add, subtract, multiply and divide. This is an obvious place for overloaded operators, so let's define them:

```
class Number {
public:
  friend Number operator+ (const Number& x, const Number& y);
  friend Number operator- (const Number& x, const Number& y);
  friend Number operator* (const Number& x, const Number& y);
  friend Number operator/ (const Number& x, const Number& y);
  // ...
};
```

It's very easy to use:

```
void f(Number x, Number y)
{
  // ...
  Number sum  = x + y;
```

```
  Number diff = x - y;
  Number prod = x * y;
  Number quot = x / y;
  // ...
}
```

But then we have a problem: handling errors. Adding numbers could cause overflow, dividing could cause divide-by-zero or underflow, etc. Whoops. How can we report both the "I succeeded and the result is `xxx`" as well as "I failed and the error information is `yyy`"?

If we use exceptions, it's easy. Think of exceptions as a separate return type that gets used only when needed. So we just define all the exceptions and throw them when needed:

```
void f(Number x, Number y)
{
  try {
    // ...
    Number sum  = x + y;
    Number diff = x - y;
    Number prod = x * y;
    Number quot = x / y;
    // ...
  }
  catch (Number::Overflow& exception) {
    // ...code that handles overflow...
  }
  catch (Number::Underflow& exception) {
    // ...code that handles underflow...
  }
  catch (Number::DivideByZero& exception) {
    // ...code that handles divide-by-zero...
  }
}
```

But if we use return codes instead of exceptions, life gets hard and messy. When you can't shove both the "good" number *and* the error information (including details about what went wrong) inside the `Number` object, you will probably end up using extra by-reference parameters to handle one of the two cases: either "I succeeded" or "I failed" or both. Without loss of generality, I will handle the computed result via a normal return value and the "I failed" case via a by-reference parameter, but you can just as easily do the opposite. Here's the result:

```
class Number {
public:
  enum ReturnCode {
    Success,
    Overflow,
    Underflow,
    DivideByZero
  };

  Number add(const Number& y, ReturnCode& rc) const;
  Number sub(const Number& y, ReturnCode& rc) const;
  Number mul(const Number& y, ReturnCode& rc) const;
```

```
  Number div(const Number& y, ReturnCode& rc) const;
  // ...
};
```

Now here's how to use it — this code is equivalent to the above:

```
int f(Number x, Number y)
{
  // ...

  Number::ReturnCode rc;
  Number sum = x.add(y, rc);
  if (rc == Number::Overflow) {
    // ...code that handles overflow...
    return -1;
  } else if (rc == Number::Underflow) {
    // ...code that handles underflow...
    return -1;
  } else if (rc == Number::DivideByZero) {
    // ...code that handles divide-by-zero...
    return -1;
  }

  Number diff = x.sub(y, rc);
  if (rc == Number::Overflow) {
    // ...code that handles overflow...
    return -1;
  } else if (rc == Number::Underflow) {
    // ...code that handles underflow...
    return -1;
  } else if (rc == Number::DivideByZero) {
    // ...code that handles divide-by-zero...
    return -1;
  }

  Number prod = x.mul(y, rc);
  if (rc == Number::Overflow) {
    // ...code that handles overflow...
    return -1;
  } else if (rc == Number::Underflow) {
    // ...code that handles underflow...
    return -1;
  } else if (rc == Number::DivideByZero) {
    // ...code that handles divide-by-zero...
    return -1;
  }

  Number quot = x.div(y, rc);
  if (rc == Number::Overflow) {
    // ...code that handles overflow...
    return -1;
  } else if (rc == Number::Underflow) {
    // ...code that handles underflow...
    return -1;
  } else if (rc == Number::DivideByZero) {
    // ...code that handles divide-by-zero...
    return -1;
  }

  // ...
}
```

The point of this is that you normally have to muck up the interface of functions that use return codes, particularly if there is more error information to propagate back to the caller. For example, if there are 5 error conditions and the "error information" requires different data structures, you might end up with a fairly messy function interface.

None of this clutter happens with exceptions. Exceptions can be thought of as a separate return value, as if the function automatically "grows" new return types and return values based on what the function can throw.

Note: Please don't write me saying that you propose using return codes and storing the error information in a namespace-scope, global or static variable, such as `Number::lastError()`. That isn't thread-safe. Even if you don't have multiple threads today, you rarely want to permanently prevent anyone in the future from using your class with multiple threads. Certainly if you do, you should write lots and lots of BIG UGLY COMMENTS warning future programmers that your code is not thread-safe, and that it probably can't be made thread-safe without a substantial rewrite.

FAQ What does it mean that exceptions separate the "good path" (or "happy path") from the "bad path"?

It's another benefit of exceptions over return-codes.

The "good path" (sometimes called the "happy path") is the control-flow path that happens when everything goes well — when there are no problems.

The "bad path" (or "error path") is the path that control-flow takes when something goes wrong — when there is a problem.

Exceptions, when done right, separate the happy path from the error path.

Here is a simple example: function `f()` is suppoesd to call functions `g()`, `h()`, `i()` and `j()`, in sequence, as shown below. If any of those fail with a "foo" or "bar" error, `f()` is to handle the error immediately then return successfully. If any other error occurs, `f()` is to propagate the error information back to the caller.

Here is the code if exceptions are used:

```
void f()  // Using exceptions
{
  try {
    GResult gg = g();
    HResult hh = h();
```

```
    IResult ii = i();
    JResult jj = j();
    // ...
  }
  catch (FooError& e) {
    // ...code that handles "foo" errors...
  }
  catch (BarError& e) {
    // ...code that handles "bar" errors...
  }
}
```

The “good” path and the “bad” path are cleanly separated. The “good” (or “happy”) path is the body of the `try` block — you can read that linearly, and if there are no errors, control flows in a simplistic path through those lines. The “bad” path is the body of the `catch` block and the body of any matching `catch` blocks in any caller.

Using return codes instead of exception clutters this to the point where it is difficult to see the relatively simple algorithm. The “good” (“happy”) and “bad” paths are hopelessly intermixed:

```
int f()  // Using return-codes
{
  int rc;  // "rc" stands for "return code"

  GResult gg = g(rc);
  if (rc == FooError) {
    // ...code that handles "foo" errors...
  } else if (rc == BarError) {
    // ...code that handles "bar" errors...
  } else if (rc != Success) {
    return rc;
  }

  HResult hh = h(rc);
  if (rc == FooError) {
    // ...code that handles "foo" errors...
  } else if (rc == BarError) {
    // ...code that handles "bar" errors...
  } else if (rc != Success) {
    return rc;
  }

  IResult ii = i(rc);
  if (rc == FooError) {
    // ...code that handles "foo" errors...
  } else if (rc == BarError) {
    // ...code that handles "bar" errors...
  } else if (rc != Success) {
    return rc;
  }

  JResult jj = j(rc);
  if (rc == FooError) {
    // ...code that handles "foo" errors...
  } else if (rc == BarError) {
    // ...code that handles "bar" errors...
  } else if (rc != Success) {
    return rc;
  }

  // ...

  return Success;
}
```

By intermixing the good/happy path with the bad/error path, it's harder to see what the code is supposed to do. Contrast that with the version that used exceptions, which is almost self-documenting — the basic functionality is very obvious.

FAQ I'm interpreting the previous FAQs as saying exception handling is easy and simple; did I get it right?

No! Wrong! Stop! Go back! Do not collect $200.

The message isn't that exception handling is easy and simple. The message is that exception handling is worth it. The benefits outweigh the costs.

Here are some of the costs:

- **Exception handling is not a free lunch**. It requires discipline and rigor. To understand those disciplines, you really should read the rest of the FAQ and/or one of the excellent books on the subject.
- **Exception handling is not a panacea**. If you work with a team that is sloppy and undisciplined, your team will likely have problems no matter whether they use exceptions or return codes. Incompetent carpenters do bad work even if they use a good hammer.
- **Exception handling is not one-size-fits-all**. Even when you have decided to use exceptions rather than return codes, that doesn't mean you use them for everything. This is part of the discipline: you need to know when a condition should be reported via return-code and when it should be reported via an exception.
- **Exception handling is a convenient whipping boy**. If you work with people who blame their tools, beware of suggesting exceptions (or anything else that is new, for that matter). People whose ego is so fragile that they need to blame someone or something else for their screw-ups will invariably blame whatever "new" technology was used. Of course, ideally you will work with people who are emotionally capable of learning and growing: with them, you can make all sorts of suggestions, because those sorts of people will find a way to make it work, and you'll have fun in the process.

Fortunately there is plenty of wisdom and insight on the proper use of exceptions. Exception handling is not new. The industry as a whole has seen many millions of lines of code and many person-centuries of effort using exceptions. The jury has returned its verdict: exceptions *can* be used properly, and when they *are* used properly, they improve code.

Learn how.

FAQ Exception handling seems to make my life more difficult; that *must* mean exception handling itself is bad; clearly *I'm* not the problem, right??

You *absolutely* might be the problem!

The C++ exception handling mechanism can be powerful and useful, but if you have the wrong mindset, the result can be a mess. It's a tool; use it properly and it will help you; but don't blame the tool if you use it improperly.

If you're getting bad results, for instance, if your code seems unnecessarily convoluted or overly cluttered with `try` blocks, you might be suffering from a wrong mindset. This FAQ gives you a list of some of those wrong mindsets.

Warning: do *not* be simplistic about these "wrong mindsets." They are guidelines and ways of thinking, not hard and fast rules. Sometimes you will do the exact opposite of what they recommend — do *not* write me about some situation that is an exception (no pun intended) to one or more of them — I *guarantee* that there are exceptions. That's not the point.

Here are some "wrong exception-handling mindsets" in no apparent order:

- **The return-codes mindset:** This causes programmers to clutter their code with gobs of `try` blocks. Basically they think of a `throw` as a glorified return code, and a `try/catch` as a glorified "if the return code indicates an error" test, and they put one of

these `try` blocks around just about every function that can `throw`.

- **The Java mindset:** In Java, non-memory resources are reclaimed via explicit `try/finally` blocks. When this mindset is used in C++, it results in a large number of unnecessary `try` blocks, which, compared with RAII, clutters the code and makes the logic harder to follow. Essentially the code swaps back and forth between the "good path" and the "bad path" (the latter meaning the path taken during an exception). With RAII, the code is mostly optimistic — it's all the "good path," and the cleanup code is buried in destructors of the resource-owning objects. This also helps reduce the cost of code reviews and unit-testing, since these "resource-owning objects" can be validated in isolation (with explicit `try/catch` blocks, each copy must be unit-tested and inspected individually; they cannot be handled as a group).
- **Organizing the exception classes around the physical thrower rather than the logical reason for the throw:** For example, in a banking app, suppose any of five subsystems might throw an exception when the customer has insufficient funds. The right approach is to throw an exception representing the *reason* for the throw, e.g., an "insufficient funds exception"; the wrong mindset is for each subsystem to throw a subsystem-specific exception. For example, the `Foo` subsystem might throw objects of class `FooException`, the `Bar` subsystem might throw objects of class `BarException`, etc. This often leads to extra `try/catch` blocks, e.g., to catch a `FooException`, repackage it into a `BarException`, then throw the latter. In general, exception classes should represent the problem, not the chunk of code that noticed the problem.
- **Using the bits / data within an exception object to differentiate different categories of errors:** Suppose the `Foo` subsystem in our banking app throws exceptions for bad account numbers, for attempting to liquidate an illiquid asset, and for insufficient funds. When these three logically distinct kinds of errors are represented by the same exception class, the catchers need to say `if` to figure out what the problem really was. If your code wants to handle only bad account numbers, you need to `catch` the master exception class, then use `if` to determine whether it is

one you really want to handle, and if not, to rethrow it. In general, the preferred approach is for the error condition's logical category to get encoded into the *type* of the exception object, not into the *data* of the exception object.

- **Designing exception classes on a subsystem by subsystem basis:** In the bad old days, the specific meaning of any given return-code was local to a given function or API. Just because one function uses the return-code of 3 to mean "success," it was still perfectly acceptable for another function to use 3 to mean something entirely different, e.g., "failed due to out of memory." Consistency has always been *preferred*, but often that didn't happen because it didn't *need* to happen. People coming with that mentality often treat C++ exception-handling the same way: they assume exception classes can be localized to a subsystem. That causes no end of grief, e.g., lots of extra `try` blocks to `catch` then `throw` a repackaged variant of the same exception. In large systems, exception hierarchies *must* be designed with a system-wide mindset. Exception classes cross subsystem boundaries — they are part of the intellectual glue that holds the architecture together.
- **Use of raw (as opposed to smart) pointers:** This is actually just a special case of non-RAII coding, but I'm calling it out because it is so common. The result of using raw pointers is, as above, lots of extra `try/catch` blocks whose only purpose in life is to `delete` an object then re-`throw` the exception.
- **Confusing logical errors with runtime situations:** For example, suppose you have a function `f(Foo* p)` that must never be called with nullptr. However you discover that somebody somewhere is sometimes passing nullptr anyway. There are two possibilities: either they are passing nullptr because they got bad data from an external user (for example, the user forgot to fill in a field and that ultimately resulted in a nullptr) or they just plain made a mistake in their own code. In the former case, you should throw an exception since it is a runtime situation (i.e., something you can't detect by a careful code-review; it is not a bug). In the latter case, you should definitely fix the bug in the caller's code. You can still add some code to write a message in the log-file if it ever happens again, and you can even throw an exception if it ever happens again, but you must not merely change the code

within `f(Foo* p)`; you must, *must*, *MUST* fix the code in the caller(s) of `f(Foo* p)`.

There are other "wrong exception-handling mindsets," but hopefully those will help you out. And remember: don't take those as hard and fast rules. They are guidelines, and there are exceptions to each.

FAQ I have too many try blocks; what can I do about it?

You might have the *mindset* of return codes even though you are using the *syntax* of `try/catch/throw`. For instance, you might put a try block around just about every call:

```
void myCode()
{
  try {
    foo();
  }
  catch (FooException& e) {
    // ...
  }

  try {
    bar();
  }
  catch (BarException& e) {
    // ...
  }

  try {
    baz();
  }
  catch (BazException& e) {
    // ...
  }
}
```

Although this uses the `try/catch/throw` syntax, the overall structure is very similar to the way things are done with return codes, and the consequent software development/test/maintenance costs are basically the same as they were for return codes. In other words, this approach doesn't buy you much over using return codes. In general, it is bad form.

One way out is to ask yourself this question for each try block: "Why am I using a try block here?" There are several possible answers:

- Your answer might be, "So I can actually handle the exception. My catch clause deals with the error and continues execution without throwing any additional exceptions. My caller never knows that the exception occurred. My catch clause does not

throw any exceptions and it does not return any error-codes." In that case, you leave the try block as-is — it is probably good.

- Your answer might be, "So I can have a catch clause that does *blah blah blah*, after which I will rethrow the exception." In this case, consider changing the try block into an object whose destructor does *blah blah blah*. For instance, if you have a try block whose catch clause closes a file then rethrows the exception, consider replacing the whole thing with a `File` object whose destructor closes the file. This is commonly called RAII.
- Your answer might be, "So I can repackage the exception: I catch a `XyzException`, extract the details, then throw a `PqrException`." When that happens, consider a better hierarchy of exception objects that doesn't require this catch/repackage/rethrow idea. This often involves broadening the meaning of `XyzException`, though obviously you shouldn't go too far.
- There are other answers as well, but the above are some common ones that I've seen.

Main point is to ask "Why?". If you discover the *reason* you're doing it, you might find that there are better ways to achieve your goal.

Having said all this, there are, unfortunately, some people who have the return-code-mindset burned so deeply into their psyche that they just can't seem to see any alternatives. If that is you, there is still hope: get a mentor. If you *see* it done right, you'll probably get it. Style is sometimes caught, not just taught.

FAQ Can I throw an exception from a constructor? From a destructor?

- For constructors, yes: You should throw an exception from a constructor whenever you cannot properly initialize (construct) an object. There is no really satisfactory alternative to exiting a constructor by a `throw`. For more details, see here.
- For destructors, not really: You can throw an exception in a destructor, but that exception must not leave the destructor; if a destructor exits by emitting an exception, all kinds of bad things are likely to happen because the basic rules of the standard library

and the language itself will be violated. Don't do it. For more details, see here.

For examples and detailed explanations, see Appendix E of TC++PL3e .

There is a caveat: Exceptions can't be used for some hard-real time projects. For example, see the JSF air vehicle C++ coding standards .

FAQ How can I handle a constructor that fails?

Throw an exception.

Constructors don't have a return type, so it's not possible to use return codes. The best way to signal constructor failure is therefore to throw an exception. If you don't have the option of using exceptions, the "least bad" work-around is to put the object into a "zombie" state by setting an internal status bit so the object acts sort of like it's dead even though it is technically still alive.

The idea of a "zombie" object has a lot of down-side. You need to add a query ("inspector") member function to check this "zombie" bit so users of your class can find out if their object is truly alive, or if it's a zombie (i.e., a "living dead" object), and just about every place you construct one of your objects (including within a larger object or an array of objects) you need to check that status flag via an `if` statement. You'll also want to add an `if` to your other member functions: if the object is a zombie, do a no-op or perhaps something more obnoxious.

In practice the "zombie" thing gets pretty ugly. Certainly you should prefer exceptions over zombie objects, but if you do not have the option of using exceptions, zombie objects might be the "least bad" alternative.

Note: if a constructor finishes by throwing an exception, the memory associated with the object itself is cleaned up — there is no memory leak. For example:

```
void f()
{
  X x;            // If X::X() throws, the memory for x itself will
```

```
  Y* p = new Y();  // If Y::Y() throws, the memory for *p itself will
}
```

There is some fine print on this topic, so you need to keep reading. Specifically you need to know how to prevent memory leaks if the constructor itself allocates memory, and you also need to be aware of what happens if you use "placement" `new` rather than the ordinary `new` used in the sample code above.

FAQ How can I handle a destructor that fails?

Write a message to a log-file. Terminate the process. Or call Aunt Tilda. But do *not* throw an exception!

Here's why (buckle your seat-belts):

The C++ rule is that you must never throw an exception from a destructor that is being called during the "stack unwinding" process of another exception. For example, if someone says `throw Foo()`, the stack will be unwound so all the stack frames between the

```
    throw Foo()
```

and the

```
  }
  catch (Foo e)
  {
```

will get popped. This is called *stack unwinding*.

During stack unwinding, all the local objects in all those stack frames are destructed. If one of *those* destructors throws an exception (say it throws a `Bar` object), the C++ runtime system is in a no-win situation: should it ignore the `Bar` and end up in the

```
  }
  catch (Foo e)
  {
```

where it was originally headed? Should it ignore the `Foo` and look for a

```
  }
  catch (Bar e)
  {
```

handler? There is no good answer — either choice loses information.

So the C++ language guarantees that it will call `terminate()` at this point, and `terminate()` kills the process. Bang you're dead.

The easy way to prevent this is *never throw an exception from a destructor*. But if you really want to be clever, you can say *never throw an exception from a destructor while processing another exception*. But in this second case, you're in a difficult situation: the destructor itself needs code to handle both throwing an exception and doing "something else", and the caller has no guarantees as to what might happen when the destructor detects an error (it might throw an exception, it might do "something else"). So the whole solution is harder to write. So the easy thing to do is *always* do "something else". That is, *never throw an exception from a destructor*.

Of course the word *never* should be "in quotes" since there is always some situation somewhere where the rule won't hold. But certainly at least 99% of the time this is a good rule of thumb.

FAQ How should I handle resources if my constructors may throw exceptions?

Every data member inside your object should clean up its own mess.

If a constructor throws an exception, the object's destructor is *not* run. If your object has already done something that needs to be undone (such as allocating some memory, opening a file, or locking a semaphore), this "stuff that needs to be undone" *must* be remembered by a data member inside the object.

For example, rather than allocating memory into a raw `Fred*` data member, put the allocated memory into a "smart pointer" member object, and the destructor of this smart pointer will `delete` the `Fred` object when the smart pointer dies. The template `std::unique_ptr` is an example of such as "smart pointer." You can also write your own reference counting smart pointer. You can also use smart pointers to "point" to disk records or objects on other machines.

By the way, if you think your `Fred` class is going to be allocated into a smart pointer, be nice to your users and create a `typedef` within your `Fred` class:

```
#include <memory>

class Fred {
public:
  typedef std::unique_ptr<Fred> Ptr;
  // ...
};
```

That `typedef` simplifies the syntax of all the code that uses your objects: your users can say `Fred::Ptr` instead of `std::unique_ptr<Fred>`:

```
#include "Fred.h"

void f(std::unique_ptr<Fred> p);  // explicit but verbose
void f(Fred::Ptr             p);  // simpler

void g()
{
  std::unique_ptr<Fred> p1( new Fred() );  // explicit but verbose
  Fred::Ptr             p2( new Fred() );  // simpler
  // ...
}
```

FAQ How do I change the string-length of an array of `char` to prevent memory leaks even if/when someone throws an exception?

If what you really want to do is work with strings, don't use an array of `char` in the first place, since arrays are evil. Instead use an object of some `string`-like class.

For example, suppose you want to get a copy of a string, fiddle with the copy, then append another string to the end of the fiddled copy. The array-of-`char` approach would look something like this:

```
void userCode(const char* s1, const char* s2)
{
  char* copy = new char[strlen(s1) + 1];    // make a copy
  strcpy(copy, s1);                         //   of s1...

  // use a try block to prevent memory leaks if we get an exception
  // note: we need the try block because we used a "dumb" char* above
  try {

    // ...code that fiddles with copy...

    char* copy2 = new char[strlen(copy) + strlen(s2) + 1];  // append
    strcpy(copy2, copy);                                    //   onto
    strcpy(copy2 + strlen(copy), s2);                       //     er
    delete[] copy;                                          //
    copy = copy2;

    // ...code that fiddles with copy again...
```

```
  }
  catch (...) {
    delete[] copy;   // we got an exception; prevent a memory leak
    throw;           // re-throw the current exception
  }

  delete[] copy;     // we did not get an exception; prevent a memory
}
```

Using `char*`s like this is tedious and error prone. Why not just use an object of some `string` class? Your compiler probably supplies a `string`-like class, and it's probably just as fast and certainly it's a lot simpler and safer than the `char*` code that you would have to write yourself. For example, if you're using the `std::string` class from the standardization committee, your code might look something like this:

```
#include <string>           // Let the compiler see std::string

void userCode(const std::string& s1, const std::string& s2)
{
  std::string copy = s1;    // make a copy of s1
  // ...code that fiddles with copy...
  copy += s2;               // append s2 onto the end of copy
  // ...code that fiddles with copy again...
}
```

The `char*` version requires you to write around three times more code than you would have to write with the `std::string` version. Most of the savings came from `std::string`'s automatic memory management: in the `std::string` version, we didn't need to write any code…

- to reallocate memory when we grow the string.
- to `delete[]` anything at the end of the function.
- to `catch` and re-`throw` any exceptions.

FAQ What should I throw?

C++, unlike just about every other language with exceptions, is very accomodating when it comes to what you can throw. In fact, you can throw anything you like. That begs the question then, what should you throw?

Generally, it's best to throw objects, not built-ins. If possible, you should throw instances of classes that derive (ultimately) from the `std::exception` class. By making your exception class inherit (ultimately) from the standard exception base-class, you are making

life easier for your users (they have the option of catching most things via `std::exception`), plus you are probably providing them with more information (such as the fact that your particular exception might be a refinement of `std::runtime_error` or whatever).

The most common practice is to throw a temporary:

```
#include <stdexcept>

class MyException : public std::runtime_error {
public:
  MyException() : std::runtime_error("MyException") { }
};

void f()
{
   // ...
   throw MyException();
}
```

Here, a temporary of type `MyException` is created and thrown. Class `MyException` inherits from class `std::runtime_error` which (ultimately) inherits from class `std::exception`.

FAQ What should I catch?

In keeping with the C++ tradition of "there's more than one way to do that" (translation: "give programmers options and tradeoffs so *they* can decide what's best for *them* in *their* situation"), C++ allows you a variety of options for catching.

- You can catch by value.
- You can catch by reference.
- You can catch by pointer.

In fact, you have all the flexibility that you have in declaring function parameters, and the rules for whether a particular exception matches (i.e., will be caught by) a particular catch clause are almost exactly the same as the rules for parameter compatibility when calling a function.

Given all this flexibility, how do you decide what to catch? Simple: unless there's a good reason not to, catch by reference. Avoid catching by value, since that causes a copy to be made and the copy can have different behavior from what was thrown. Only under very special circumstances should you catch by pointer.

FAQ But MFC seems to encourage the use of catch-by-pointer; should I do the same?

Depends. If you're using MFC and catching one of their exceptions, by all means, do it their way. Same goes for any framework: when in Rome, do as the Romans. Don't try to force a framework into your way of thinking, even if "your" way of thinking is "better." If you decide to use a framework, embrace *its* way of thinking — use the idioms that its authors expected you to use.

But if you're creating your own framework and/or a piece of the system that does not directly depend on MFC, then don't catch by pointer just because MFC does it that way. When you're *not* in Rome, you don't *necessarily* do as the Romans. In this case, you should *not*. Libraries like MFC predated the standardization of exception handling in the C++ language, and some of these libraries use a backwards-compatible form of exception handling that requires (or at least encourages) you to catch by pointer.

The problem with catching by pointer is that it's not clear who (if anyone) is responsible for deleting the pointed-to object. For example, consider the following:

```
MyException x;

void f()
{
  MyException y;

  try {
    switch ((rand() >> 8) % 3) {  // the ">> 8" (typically) improves
      case 0: throw new MyException;
      case 1: throw &x;
      case 2: throw &y;
    }
  }
  catch (MyException* p) {
    // should we delete p here or not???!?
  }
}
```

There are three basic problems here:

1. It might be tough to decide whether to `delete p` within the `catch` clause. For example, if object `x` is inaccessible to the scope of the `catch` clause, such as when it's buried in the private part of some class or is `static` within some other compilation unit, it might be tough to figure out what to do.

2. If you solve the first problem by consistently using `new` in the `throw` (and therefore consistently using `delete` in the `catch`), then exceptions always use the heap which can cause problems when the exception was thrown because the system was running low on memory.
3. If you solve the first problem by consistently *not* using `new` in the `throw` (and therefore consistently *not* using `delete` in the `catch`), then you probably won't be able to allocate your exception objects as locals (since then they might get destructed too early), in which case you'll have to worry about thread-safety, locks, semaphores, etc. (`static` objects are not intrinsically thread-safe).

This isn't to say it's not possible to work through these issues. The point is simply this: if you catch by reference rather than by pointer, life is easier. Why make life hard when you don't have to?

The moral: avoid throwing pointer expressions, and avoid catching by pointer, *unless* you're using an existing library that "wants" you to do so.

FAQ What does `throw;` (without an exception object after the `throw` keyword) mean? Where would I use it?

You might see code that looks something like this:

```
class MyException {
public:
  // ...
  void addInfo(const std::string& info);
  // ...
};

void f()
{
  try {
    // ...
  }
  catch (MyException& e) {
    e.addInfo("f() failed");
    throw;
  }
}
```

In this example, the statement `throw;` means "re-throw the current exception." Here, a function caught an exception (by non-const reference), modified the exception (by adding information to it), and

then re-threw the exception. This idiom can be used to implement a simple form of stack-trace, by adding appropriate catch clauses in the important functions of your program.

Another re-throwing idiom is the "exception dispatcher":

```
void handleException()
{
  try {
    throw;
  }
  catch (MyException& e) {
    // ...code to handle MyException...
  }
  catch (YourException& e) {
    // ...code to handle YourException...
  }
}

void f()
{
  try {
    // ...something that might throw...
  }
  catch (...) {
    handleException();
  }
}
```

This idiom allows a single function (`handleException()`) to be re-used to handle exceptions in a number of other functions.

FAQ How do I throw polymorphically?

Sometimes people write code like:

```
class MyExceptionBase { };

class MyExceptionDerived : public MyExceptionBase { };

void f(MyExceptionBase& e)
{
  // ...
  throw e;
}

void g()
{
  MyExceptionDerived e;
  try {
    f(e);
  }
  catch (MyExceptionDerived& e) {
    // ...code to handle MyExceptionDerived...
  }
  catch (...) {
    // ...code to handle other exceptions...
  }
}
```

If you try this, you might be surprised at run-time when your `catch (...)` clause is entered, and not your `catch`

`(MyExceptionDerived&)` clause. This happens because you didn't throw polymorphically. In function `f()`, the statement `throw e;` throws an object with the same type as the *static* type of the expression `e`. In other words, it throws an instance of `MyExceptionBase`. The `throw` statement behaves as-if the thrown object is copied, as opposed to making a "virtual copy".

Fortunately it's relatively easy to correct:

```
class MyExceptionBase {
public:
  virtual void raise();
};

void MyExceptionBase::raise()
{ throw *this; }

class MyExceptionDerived : public MyExceptionBase {
public:
  virtual void raise();
};

void MyExceptionDerived::raise()
{ throw *this; }

void f(MyExceptionBase& e)
{
  // ...
  e.raise();
}

void g()
{
  MyExceptionDerived e;
  try {
    f(e);
  }
  catch (MyExceptionDerived& e) {
    // ...code to handle MyExceptionDerived...
  }
  catch (...) {
    // ...code to handle other exceptions...
  }
}
```

Note that the `throw` statement has been moved into a virtual function. The statement `e.raise()` will exhibit polymorphic behavior, since `raise()` is declared `virtual` and `e` was passed by reference. As before, the thrown object will be of the *static* type of the argument in the `throw` statement, but within `MyExceptionDerived::raise()`, that static type is `MyExceptionDerived`, not `MyExceptionBase`.

FAQ When I throw this object, how many times will it be copied?

Depends. Might be "zero."

Objects that are thrown must have a publicly accessible copy-constructor. The compiler is allowed to generate code that copies the thrown object any number of times, including zero. However even if the compiler never actually copies the thrown object, it must make sure the exception class's copy constructor exists and is accessible.

FAQ Why doesn't C++ provide a "finally" construct?

Because C++ supports an alternative that is almost always better: The "resource acquisition is initialization" technique. The basic idea is to represent a resource by a local object, so that the local object's destructor will release the resource. That way, the programmer cannot forget to release the resource. For example:

```
// wrap a raw C file handle and put the resource acquisition and
// in the C++ type's constructor and destructor, respectively
class File_handle {
    FILE* p;
public:
    File_handle(const char* n, const char* a)
        { p = fopen(n,a); if (p==0) throw Open_error(errno); }
    File_handle(FILE* pp)
        { p = pp; if (p==0) throw Open_error(errno); }

    ~File_handle() { fclose(p); }

    operator FILE*() { return p; }   // if desired

    // ...
};

// use File_handle: uses vastly outnumber the above code
void f(const char* fn)
{
    File_handle f(fn,"rw"); // open fn for reading and writing
    // use file through f
} // automatically destroy f here, calls fclose automatically wit
  // (even if there's an exception, so this is exception-safe by
```

In a system, in the worst case we need a "resource handle" class for each resource. However, we don't have to have a "finally" clause for each acquisition of a resource. In realistic systems, there are far more resource acquisitions than kinds of resources, so the "resource acquisition is initialization" technique leads to less code than use of a "finally" construct.

Also, have a look at the examples of resource management in Appendix E of TC++PL3e .

FAQ Why can't I resume after catching an exception?

In other words, why doesn't C++ provide a primitive for returning to the point from which an exception was thrown and continuing execution from there?

Basically, someone resuming from an exception handler can never be sure that the code after the point of throw was written to deal with the execution just continuing as if nothing had happened. An exception handler cannot know how much context to "get right" before resuming. To get such code right, the writer of the throw and the writer of the catch need intimate knowledge of each others code and context. This creates a complicated mutual dependency that wherever it has been allowed has led to serious maintenance problems.

Stroustrup seriously considered the possibility of allowing resumption when he designed the C++ exception handling mechanism and this issue was discussed in quite some detail during standardization. See the exception handling chapter of The Design and Evolution of C++ .

If you want to check to see if you can fix a problem before throwing an exception, call a function that checks and then throws only if the problem cannot be dealt with locally. A `new_handler` is an example of this.