Expect the Exceptional

Before we get to the really cool stuff, we need to talk about exceptions, the last major C++ construct that isn't covered in the 106/7/L track, since many of you haven't really seen them. They're also a good way to get you thinking about designing effective classes. After all, things go wrong in code, even things that aren't the programmer's fault, and so we need to know how to deal with them.

Even if you haven't seen them in C++, you may have encountered exceptions before: Java forces them down your throat with exception specifications, Python has them, and ML-based languages love them. And yeah, they're in C++ too.

Introduction: Why Exceptions are a Good Thing (TM)

Consider this rather normal C-esque code fragment:

```
FILE * f = fopen("path/to/my/file","r");
char buff[100];
int i;
fscanf(f,"%d %100s",&i,&buff);
// Do stuff with i and buff
fclose(f);
```

What's wrong here? If you said, "He doesn't check to make sure the FILE*'s non-null!", you're half right. If you said, "and fscanf's return value too!", then you got the other half. This poor innocent developer could be running all over memory, doing untold damage to his[1] application, potentially corrupting other data files permanently, and possibly even formatting his hard drive![2] Instead the developer is supposed to check every possible error condition. So, what's the proper code?

```
char buff[100];
int i;
FILE * f = fopen("path/to/my/file","r");
if(!f)
{
  // Report error here
  // Do I return an error code? A null pointer? Abort?
}
// ...
if(fscanf(f,"%d %100s",&i,&buff)<2)
{
  // Again, what do we do here?
  // And we have to remember to close the FILE*, otherwise
```

1 I promise to try to alternate his's and her's across handouts. This is a "his" handout.

2 Referencing an uninitialized variable in C++ is considered "undefined behavior." This means that the compiler is free to generate code that does anything , including formatting one's hard drive. My theoretical C++ compiler does exactly that, or maybe it swears at you through the sound card. I can't decide which. Doing both would just be cruel.

```
  // we leak an OS resource. That's really bad.
  // Now what? same thing? Abort? Log and return?
}
// Do stuff with i and buff
fclose(f);
```

This code is correct [assuming we put in the `fclose(f)`], but it's long and it's awfully inconvenient. And it's less efficient. That's 2 extra if-checks! Minor yes, but what if you're deep inside an inner loop? That's no good. And then, if we decide to just return a null pointer or an error code, then client code will have to check for them too! And let's be honest, that's a lot of checking for two errors that really don't happen that often. (But yes, we need to check for them.)

RAII: Resource-Acquisition-Is-Initialization

So let's tackle this problem starting with that `FILE *`. If you recall, there were two problems with it. The first was opening it, and the second was making sure we closed it. Luckily, C++ has an idiom just for this purpose, and it's called RAII, which is just a glorified way of saying, "use constructors and destructors to manage your resources." So, let's make the most naïve wrapper class around a `FILE *` ever:

```
class file
{
private:
     FILE * f_;
     // Why are these private
     // and declared, but not implemented at all?
     file(const file &);
     file & operator=(const file &);
public:
     file(const char * path, const char * mode)
     : f_(fopen(path,mode))
     {
          if(!f_)
            // but what goes here?
     }
     ~file()
     {
          fclose(f_); // Easy enough.
     }
     FILE * get() {return f_;}
};
```

So, what do we do about this nasty business about the file failing to open in the constructor. We can't really return an error code: constructors don't have a return signature. We could two-stage the construction or set a flag, but that solves nothing since the user would still have to check for an error condition. (This is a great failure, by the way, of `std::fstream`, but I digress...) The answer, as you may have discerned from the beginning of this handout, is an exception:

```
class file
```

```
{
private:
     //...
public:

     class exception: public std::exception
     {
     private:
          char mesg_[BIG_SIZE];
     public:
          exception(const char * path)
          {
               // We need to avoid allocations in
               // exceptions for reasons that we'll discuss
               // later...
               strcat(mesg,"Couldn't open ");
               strncat(mesg,BIG_SIZE-15,path);
          }

          // Ignore the "throw ()" for now.
          const char * what() const throw ()
          {
            return mesg_;
          }
     };

     file(const char * path, const char * mode)
     : f_(fopen(path,mode))
     {
          if(!f_)
            throw exception(path);
     }
     // ...
};
```

Yeah, that's a bit messy, though that's in part from my trying to save space. But take a look at the client's code:

```
char buff[100];
int i;
file f("path/to/my/file","r");
if(fscanf(f.get(),"%d %100s",&i,&buff)<2)
{
  // What do we do here?
  // We don't have to close the file, that's for sure!
}
// We could let f go out of scope here by enclosing it in
// braces at this point, thus closing the file as soon as
// we're done. I won't go into that.
```

```
// Do stuff with i and buff
```

It's as short and easy as the original unsafe example! Since this bit of code probably wouldn't know what to do in the case of a file-opening's failure, it's probably best to pass on that failure to a higher caller, so that they can handle this problem, either through aborting, logging, or whatever makes sense to the application at a higher level.

Syntax and Mechanics of Exceptions

Before we finish this example, it's about time we talked about the mechanics of exceptions in C++, and then how they work in modern implementations, by which I mean g++.[3] To get us started, here's some code:

```
try
{
  file f("path/to/file","r");
  std::string("This string is really long to get around any"
  "optimizations the library might have for short ones.");
  struct i_always_throw
  {
     i_always_throw(){throw std::exception("see?");}
  } watch_me_throw; // an exception will be thrown here.
}
catch(std::exception & e)
{
  // Unless the file doesn't open, or the string throws (which
  // they might), this will print "see?".
  std::cerr << e.what() << std::endl;
}
catch(...) // this gets called if nothing else matches.
{          // We should never reach this point in our code.
  std::cerr << "Some weird exception. I'll rethrow..."
            << std::endl;
  throw; // rethrow the exception. Hopefully a higher level will
         // deal with it.
}
```

This is a bit dense, so let's go through it one step at a time. The first thing we have is a "`try`" block. This is a sign to the compiler that if something goes wrong (i.e. an exception is thrown), there are one or more "`catch`" blocks that need to be executed based on the type of exception that is thrown. Inside the try block we construct three objects, each of which can throw on exception from their constructors. The `file` object, as you may recall, can throw an exception of type `file::exception` which derives from `std::exception`. `std::string`'s constructor can throw an `std::bad_alloc`, (if the `new`'d buffer cannot get the room it needs), and it's also derived from `std::exception`. Finally, we have the user defined type that always throws from its constructor. Note that we don't have

3 There's the easy way of implementing exceptions in terms of C's `setjmp` and `longjmp`, but it is very obtuse as it incurs a rather massive runtime performance penalty for every try block.

to throw types derived from `std::exception`, or even class types at all. We could instead decide to have our exceptions be `int`'s, or `char *`'s, or any type you could want. However, since class types have very little overhead, can be derived from, and otherwise can carry a lot more information than primitive types, I—and people who write books about these things—suggest that you **prefer to throw class types.**

So what happens when the code throws? In the C++ model, the program crawls back along the stack calling the destructors for *objects whose constructors have completed.* This helps ensure that we deterministically free all resources that we allocated, except for raw unprotected calls to `new` or `malloc` and the like. (Your first assignment will involve creating a mechanism that helps with pointers.) The process continues until the program sees a catch block that matches the type of the exception thrown. The code for that block is then executed, and—assuming no one aborted the program—control resumes at the end of all the catch blocks after the destructor of the exception has been executed. Remember that if you throw by pointer, the compiler is only obligated to run the destructor of the pointer, which is a no-op, so in general you should **prefer to throw by value**.

And that brings us to the catch blocks. In the first one we catch any type derived from `std:exception.` Also, note that we catch by reference. (We would match derived types even if we didn't catch by reference.) Otherwise, we could potentially get object slicing from derived types, potentially leading to a problem not dissimilar to `delete`'ing pointers to bases that do not have `virtual` destructors. This leads me to my third emboldened point: **prefer to catch by reference**. We then log the exception to standard error, and continue on our merry way. In the second, we catch exceptions of type "`...`", which will match any type. However, the `...` erases all information we could have known about the type of the exception, so we can't do anything with it, except re-throw it or suppress it. The latter is a bad idea, since if you don't know what error was thrown, how do you know that you should suppress it.

That leaves two major questions: what happens if an exception doesn't get caught? That one's easy: the program terminates. What happens if two exceptions are active at the same time? That one's easy too: the program terminates, immediately, without doing any other cleanup. To quote a wonderfully terrible movie: "Game over, man! Game over!" (Note that for the most part, you can only get a second exception from destructors. And since having the program terminate is a Bad Thing TM if you didn't ask it to, you should **never let a destructor throw!** There are other reasons, which we'll get to later.)

And for those of you with Java experience, you may wonder, what about a "finally" construct? Nope. So what should you do if you want a piece of code to be deterministically called at the end of a scope, no matter what happens, even if exceptions are thrown? A destructor! And this little idiom has its own name: "scope guard." Say you want to log a `std::string` called `mesg` to `std::clog` every time you exited a function's scope? Here's the quick and dirty version:

```
struct scope_logger
{
   const std::string & mesg;
   ~scope_logger()
   {
      std::clog << mesg << std::endl;
   }
   scope_logger(std::string & mesg): mesg(mesg) {}
```

```
} my_scope_logger(message);
```

Not too bad, huh? Having to write your own every time might be a little tedious, and that's why there's Andrei Alexandrescu's Loki::ScopeGuard and a possible Boost library called Finally that make the process very easy. Look them up if you're interested. The latter really abuses the preprocessor and relies on a rather clever application of Godel Numbering(!), so it's really hard to read and even harder to implement.

Exception Specifications: Don't use these.

C++, like Java, also has exception specifications. Unlike in Java, these are enormously silly and don't gain you anything. (Well, too be honest, they're perhaps more useless in Java for reasons I can discuss.) For those that haven't seen Java exception specifications, in C++ they look like this:

```
void my_func() throws (my_exception1, my_exception2) {/*...*/}
```

The idea is that with throw specifications you promise client code that you will not throw any exceptions except the ones you listed. If you do try to throw a different one, the compiler translates the exception into an `std::bad_exception`, which essentially generates code to call `std::unexpected`, which by default calls `std::terminate`.[4] Unlike in Java, no checking is done to ensure that such exceptions don't get thrown, so instead the compiler basically puts try...catch blocks around your code, which doesn't gain you anything really. Some compilers even refuse to inline code with exception specifications, which means that exception specifications can cause slow downs. And, as we'll discuss, exception safety doesn't really depend on the types of exceptions thrown, and in reality this doesn't buy you anything over a well placed comment. Finally, think about the problems of writing exception specifications for template code. There is no way to add the exceptions that a user-defined type might throw to the exception specification, which means that static enforcement is either impossible or useless in templates.

Standard Exceptions

Let's see what we can do about that `fscanf` conundrum. I think it's pretty obvious at this point that I think we should throw an exception on failure. The question is, what kind? We could roll our own, but that doesn't seem necessary here. After all, a good programmer knows what to write, a great one knows what not to. And indeed, the standard library has a whole host of exceptions ready for us to use or to derive from. Let's look at (some of) them: (All are declared in <stdexcept> unless otherwise stated.)

- `exception` : general purpose, base of the tree.
  - `bad_alloc` : Thrown whenever `new` fails.
  - `logic_error` : Used when preconditions are violated. (I prefer asserts most of the time.)
    - `domain_error`, `invalid_argument`, `length_error`, `out_of_range`
  - `ios_base::failure` : For IO failures. (Declared in <ios>)
  - `runtime_error` : Used when postconditions cannot be met.
    - `range_error, overflow_error, underflow_error`
  - `bad_exception` : discussed above.

What do you think is best? (I'll let you decide this one. We'll discuss it in class too.)

4 You can change this behavior by calling `std::set_unexpected`. Or you can just not use exception specifications.

Efficiency, or When Exceptions Are Bad

Since C++ is a systems language, I'd be amiss not to talk about efficiency a bit. I've already claimed that code with exceptions on a good compiler is in the non-exceptional case considerably faster than the equivalent code without it. But when is it a bad idea to use exceptions instead of—say—error codes? As you may have guessed, the exceptional case is very slow, since the compiler writers tried very hard to optimize the normal execution path. So, when exceptions are common, your code will be slower, and so you might want to avoid exceptions in time-dependent code when exceptions are common. That is, **avoid using exceptions for circumstances that are not exceptional.**

Finally, in truly performance-sensitive conditions such as real time systems, the slow down caused by exceptions is almost never okay, since in those cases you need operations to take the same amount of time every time, exceptional circumstances not withstanding.

That's it for now. In the next handout (which will be from Stroustrup's book: again, good coders know what to write; great ones know what not to), we'll discuss what it means to be exception-safe, and then implement a (portion of) `std::vector<>` that fits that description as much as we can.

References

Jossutis, Nicolai. *The C++ Standard Library*. Addison-Wesley, 1999.
Stroustrup, Bjarne. *The C++ Programming Language*. 3rd Ed. Addison-Wesley, 1997.
Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, 2005.
Sutter, Herb. *Exceptional C++*. Addison-Wesley, 2000.
Sutter, Herb. *More Exceptional C++*. Addison-Wesley, 2002.