Exceptions and Exception Handling

Exceptions provide a different way of handling those "less than ideal" situations which crop up in your program. When writing reusable code, there are times when your classes will encounter situations where they don't know what to do, and can't continue. For example, what should you do if you run out of memory when trying to dynamically allocate storage? Or, what should you do if the file you're trying to open is locked or does not exist?

Many of these questions cannot be answered with a single answer. For one program, the correct answer might be "just terminate the program". However, if your program is controlling an air traffic control center or an airplane, that isn't a very good solution. Instead, what you really need is the ability to have your library code simply report that it could not continue for some reason and have the application code deal with this condition appropriately.

> The Eckel text doesn't even discuss exceptions in the first volume, so you Eckel fans will have to look elsewhere for your exception handling reading. The Deitel text does cover exceptions, though; all of Chapter 13 is devoted to it.
>
> Don't be concerned with exceptions and exception handling beyond what is discussed in this handout. Exceptions are a wonderful programmatic concept, but C++'s implementation of them is far from perfect. For that reason, I don't require you to use exceptions for any of the code you write. Most compilers—even the heralded g++ compiler consider by many to be the most faithful to the ANSI standard—has trouble getting exceptions right. And even the standard, in order to be backward compatible with ANSI C code, is somewhat lax in how programmers deal with exceptions and exception handling. The concept is useful, but C++ isn't exactly the best language for teaching them. Bottom line—this handout provided the technical information and is here for completeness purposes, not because Jerry the lecturer thinks you should be using them. CS193D is a C++ course, so you're entitled to all of it, even the bad parts. =)

Traditional Error Handling Methods

Handling errors is a very necessary, though very messy and difficult aspect of programming. Traditional methods of handling errors usually involve returning an error code from every function, either through a return value, an extra reference parameter, or a global variable such as `errno`.

There are several problems with these traditional methods of error reporting. One problem is that checking for an error condition after *every* function call practically doubles the amount of code you have to write. Another problem is that you may forget to check for an error condition, or otherwise choose to ignore them. Propagating errors

upward from function to function can become messy as well, because you are constrained to returning one kind of error condition from a function.

C++ Exceptions

C++ attempts to alleviate these problems through the use of exceptions. Exceptions do not have to be handled on each function call. If a series of function calls deal with one aspect of a program, you can handle the error conditions in one location. Suppose you have called a function A which calls function B which calls function C, and C reports that there is an error. You can handle the exception in A — C will return straight to A if B does not handle that error type. You can't ignore an exception altogether — if you don't handle it, your program will terminate. This is actually a good thing because it **forces** you to handle failures and other unexpected conditions. You can also report different types of exceptions from a single function. Overall, exceptions are more flexible than traditional techniques of handling errors.

Exception Types

As mentioned before, you can report several types of exceptions, even from within a single function. You can define a separate exception handler for each of these exception types. The type of an exception is the can be any type in C++—it can be a built-in type, such as an `int` or a `char *`, or it can be a user-defined type. Often, you will define a class which encapsulates the information relevant to the exception. If, for instance, you had an out-of-range exception when subscripting an instance of an `IntArray`, you may want to have an exception class which encapsulates the valid range of the array as well as the index requested.

Throwing Exceptions

Reporting an exception in C++ is called **throwing** an exception. You do this by using the `throw` keyword along with an object or variable you want to throw. You can throw an exception of any type. To throw a string exception, you could do this:

```
int IntArray::operator[](int index) const
{
   if (index < 0 || index >= numElems)
      throw "Out of Bounds";

   return this->elems[index];
}
```

Better yet, you might create a special `IndexOutOfBounds` class and throw a temporary instance of it. The benefit here is that you have a type specific to the nature of the exception in this scenario.

```
int IntArray::operator[](int index) const
{
   if (index < 0 || index >= numElems) {
      throw IndexOutOfBounds(index, numElems);
```

```
    }

    return this->elems[index];
}
```

Catching Exceptions

Catching exceptions is the C++ lingo for handling exceptions. If you don't catch an exception which has been thrown, your program will terminate. So if you don't want your program to terminate, you had better catch all exceptions which are thrown. The basic control structure for exception handling in C++ is the try/catch block. A `try/catch` block which handles `IndexOutOfBounds` exceptions might look like this:

```
try {
   // code that might throw an IndexOutOfBounds
} catch (IndexOutOfBounds iobe) {
   // code designed to handle any exception of type IndexOutOfBounds
}
```

Any exceptions thrown within the `try` block will be handled by a matching exception handler in the `catch` block. You can catch different types of exceptions within one `try/catch` block, and you can catch exceptions of any type by using an ellipsis `(...)` as the exception type. This is also called a catch-all handler, and is demonstrated below:

```
try {
   // Call some code here...
} catch (char* str) {
   // Handle character string exceptions here...
} catch (IndexOutOfBounds& boundsError) {
   // Handle IndexOutOfBounds exceptions here...
} catch (...) {
   // Handle all other exceptions here...
}
```

The order in which you place your catch blocks is important. The first matching exception handler will be invoked, so if you placed the catch-all exception handler first, no other exception handler would be called.

For reasons we haven't discussed yet, you should never throw and catch an actual object— always use a reference, a pointer, or a built-in as the exception type. You can choose to filter out some exceptions by handling them in a particular `try/catch` block and allowing an outer `try/catch` block to handle the ones you don't want to handle. This can be done by re-`throw`ing the exceptions you don't want to handle from the `catch` block.

Resource Management

Between the point where an exception is thrown and an exception is caught, all fully initialized direct objects which are local variables or parameters to functions are **destroyed** while the stack is unwound to the point where the exception is caught. For instance, consider the following code:

```
bool Flight::init(istream& instream) const throw(char*)
{
   Flight flight;
   Airport *a = new Airport("SFO", "San Francisco");
   ...
   throw "Out of Bounds";
   ...
}

try {
   if (flightList->contains(airport))
   ...
} catch(char* str) {
   // Handle the error
}
```

In this case, the destructor for the `Flight` object is called. However, the destructor for the `Airport` object is not, because it is not a direct object. Because of this, it is often wise to declare objects directly in the presence of exceptions.

Exception Specifications

One problem with traditional error handling methods is that it is difficult to know which errors a function might return. Although reading documentation on the function may help, good documentation may not exist, and even if it does, it is a pretty tedious task to check every function to see which errors it might return.

C++ allows you to specify which exception types a function will throw, or to specify that a function will throw no exceptions at all. This specification is provided as part of a function interface. For example, we might have a function `ReadAirportsFromFile` which might throw a `FileNotFoundException` exception. The prototype for this function would then need to be:

```
void ReadAirportFromFile(const char *filename) throw (FileNotFoundException);
```

Exception specifications can have any number of exception types, or even no exception types. Having no exception types in an exception specifier means that the function will not throw any exceptions. Class methods can have exception specifications:

```
class Stack {
   ...
   Stack();                      // can throw any exception type
   ~Stack() throw ();            // can't throw any types whatsoever
```

```
    int pop(void) throw(EmptyStackException);
    int peek(void) const throw (EmptyStackException);
    void push(int value) throw (FullStackException, bad_alloc);
    ...
};
```

The exception specification must be duplicated word for word when implementing a function or method or there will be a compile-time error (at least in theory according to the C++ standard). The compiler won't check the transitivity of exception specifications. For instance, the `pop` method above can call a function which throws exceptions other than those of type `EmptyStackException`. If you provide no exception specification, it means that the function can throw any type of exception. The compiler can't always check to see that a function doesn't `throw` an exception type that isn't specified, but it can catch many cases of this.

Efficiency Concerns

Throwing exceptions is typically very costly in terms of performance. It takes minimal overhead to use `try/catch` blocks, but the actual throwing of exceptions is very expensive. Because of this, you should only be throwing exceptions in situations where an unexpected event is occurring. Unexpected events are presumably rare enough that the performance penalty isn't what's important.

Throwing Exceptions in Constructors

Throwing exceptions within constructors is a good way to announce that the constructor could not do its job. Because a constructor has no explicit return type, there is not a very simple way to report error conditions unless we throw an exception. A lot of care should be taken to make sure that a constructor never leaves an object in a half-constructed state. Using automatic resource management as described earlier can help avoid this problem. It is generally unsafe to throw exceptions from within a copy constructor. This is because copy constructors often acquire and release resources. If an exception is thrown in the middle of the copy constructor, it could leave the copied object in an unpredictable state.

Throwing Exceptions in Destructors

Destructors should be exception-safe. This is because a destructor will be automatically called when an exception is thrown to de-allocate all objects which are on the stack between the exception throw and catch. If a destructor ends up throwing an exception, either directly or indirectly, the program will terminate. To avoid this problem, your destructor really needs to catch any exceptions which might be thrown by calls to helper methods and functions. The bottom line here: never allow an exception of any sort to be thrown from a destructor; the alternative opens up the possibility that a buggy program simply terminates without presenting any easily identifiable source for the bug.