

Errors and Exception Handling

Many programs have more code to handle error condition than to solve the problem for which the program was written! There are many different classes of errors that can occur:

1. User input errors (e.g., mistyped input, wrong filename given, wrong mouse button pressed, etc.)
2. Device errors (e.g., network goes down, disk crashes, modem not turned on)
3. System resource limitations (e.g., disk is full, heap memory exhausted, file does not exist)
4. Component failures (e.g., accessing beyond the end of an array)

The `java.net.Socket` class is a good example. The class implements an object-oriented wrapper onto Unix sockets for networking (using Java native functions), which is a system level interface. However, you don't need to know all the low-level details of sockets when implementing a network client application (like an SMTP client), but you do need to be made aware of error conditions that may arise within the class library. These include I/O errors and network errors, such as unknown host names, aborted network connections, etc.

A class library can sometimes handle errors internally, and do something sensible. Other times, the user of the class must be notified of the error. Exceptions provide a *structured* way to communicate error information across a class or procedure abstraction boundary.

Many programs are written that do not perform much error checking, including student programs! Programmers often assume that the program is always given the correct input, there will be no device errors, system resources are always available, and component failures either can't happen because "I wrote the code", or if they do occur, you just call "abort." For many types of applications, these assumptions hold true because the programs are never "stressed". However, this assumes you have complete control over the development of the application and how it is used. Imagine the flight control computer on a Boeing 747 having been developed in this manner, and calls "abort" when something goes wrong--that will not be a very satisfying experience!

As implementors of OO class libraries, you cannot always predict how someone else might use (or misuse!) your classes to build an application. The fact is that we have to program defensively, which is especially true when developing class libraries that other, perhaps more naive, programmers will use to develop their programs.

Assertion checking using the "assert" macro in C/C++ is fine, but usually results in a call to `abort` the program. This is not very helpful or satisfying to the user. Case in point, Netscape Communicator 4.5 frequently "aborts".

There are various techniques used by programmers to cope with errors. The following are common in C/C++.

1. Return an integer error code. Most operations return some value, so pick a value not in the normal domain of return values (e.g., -1)

```
int result = read(buf, size);
if (result < 0) // failed
  switch (errno) { // use global errno to decipher cause of failure
    case ERR1:
      ...
    case ERR2:
      ...
    default: abort();
  }
else
  // ok
```

The caller then has the responsibility to detect the error, decipher what really happened, then do something reasonable. Notice that in C/C++, "errno" is a global variable that is set when a system error occurs. When you have multiple threads executing, a global `errno` variable is a BIG problem. What you often see is that the caller doesn't want to be bothered with error checking, so the problem is just passed back up the call chain until something catches it, or the program is killed unexpectedly.

```
if (result < 0) // uh-oh, something bad happened
  return result; // so, just pass the problem back up the call chain
```

C++ presents some unique and subtle problems. For example, how do you indicate an error in a constructor?

```
Foo::Foo(int size) { char* c = new char[size]; }
```

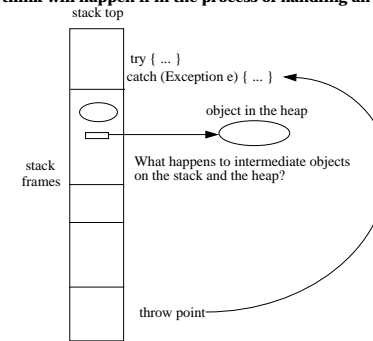
Q: What happens if the call to the Java or C++ "new" operator fails to allocate storage from the heap?

Exceptions and the Stack

When an exception is thrown, the stack is "unwound" and intermediate stack frames are deallocated as part of throwing the exception. Keep in mind that any object references in nested scopes between where the exception is "thrown" (or raised) and where it is "caught" are lost, and the objects they pointed to may become garbage (if not other scopes hold references to them). In Java, this is not a problem, because the garbage collector will collect them, eventually. However, in C++, when you throw an exception, you have to be sure to explicitly deallocate any heap allocated objects that may be referenced in intervening stack frames, otherwise you end up with a space leak.

The point of throwing an exception is usually that it is caught and some corrective action can be taken or a clean "shutdown" procedure can occur--something more graceful than abort.

Q: What do you think will happen if in the process of handling an exception, another exception occurs?



2. In C/C++, you can define special "error handler" functions that will be called when an error occurs. For example, the new operator in C++ defines a default "new error handler" that is called when a request for memory allocation fails (see the <new.h> include file for your favorite compiler)

```
typedef void (*VFP)(); // define a void function pointer type with no args

extern void _default_new_handler();
VFP new_handler = &_default_new_handler;

void* operator new(size_t s)
{
  void *p;
  while ((p = malloc(s)) == 0) // malloc fails when virtual memory exhausted
    if (_new_handler) // if error handler set, call it and try again
      (*_new_handler)(); // call new handler via pointer to function
  else
    return NULL;
}
return p;
```

In C++, you can define your own new handler to replace the default one supplied by the compiler run-time:

```
extern VFP set_new_handler(VFP func);

void my_new_handler()
{
  int err = try_to_extend_virtual_memory(); // e.g., call sbrk() under Unix
  if (err == OK) return; // got some
  else abort(); // give up and die
}

VFP old = set_new_handler(&my_new_handler);
```

Throwing an Exception on the Occurrence of an Error

If we have an **exception handling** facility, we could **throw** an exception. Of course, there would have to be some code somewhere that would **catch** the exception. The way this is done in Java (and ANSI C++) is using a combination of a lexical **try block** and an one or more lexical **catch blocks** that act as exception handlers. An exception is raised by using the **throw** statement. e.g.:

```
void some_func() throws SomethingBadHappenedException
{
    int result_code = do_something_that_might_fail();
    if (result_code == OK)
        ... // hurray! it worked!
    else
        throw new SomethingBadHappenedException("HELP!");
}
```

The **throw** statement causes a transfer of control **up the call chain** to the "nearest" catch block for the type of object thrown (in the case above, we throw a String object. In this case, the type of the object thrown is a String. For C programmers, a try block is like a "setjmp" and a throw to a catch block is like a "longjmp". However, setjmp/longjmp don't allow you to pass an object up the stack as part of "unwinding" the stack.

Typically, it is a good idea to implement an *exception class hierarchy* of error types and throw an object that represents all the information known about the error **at the time it occurred**. Then, a catch block higher up in the call chain can interface with the exception object to find out more about what went wrong, instead of just passing an opaque error value (like -1) that is not very informative. For example, provide as much info as possible:

```
throw SomethingBadHappenedException("some_func", result_code, "failed to ...");
```

Where SomethingBadHappenedException is but one of many subclasses of the **java.lang.Exception** class:

```
class SomethingBadHappenedException extends Exception { ... }
```

This way, you can always write a "catch all" block, as: `catch (Exception e) { ... }`

The Reader Thread

```
class Reader extends Thread {
    private Client client;

    public Reader(Client c) {
        super("Client Reader"); // pass identifying string to Thread(String) constructor
        this.client = c;
    }

    public void run() {
        BufferedReader in = null;
        String line;
        try {
            in = new BufferedReader(new InputStreamReader(client.socket.getInputStream()));
            while (true) {
                line = in.readLine();
                if (line == null) {
                    System.out.println("server closed connection");
                    break;
                }
                System.out.println(line);
            }
        }
        catch (IOException e) {
            System.err.println("Reader Failed: " + e);
        }
        finally try {
            if (in != null) in.close();
        }
        catch (IOException e) { System.err.println(e); System.exit(0); }
    }
}
```

Example Threaded Network ReaderWriter Class with Exceptions

It is often useful to write a network client that contains two threads: a Reader thread and a Writer thread. The Reader thread listens for data from the server and sends it to the console or a file. The Writer thread takes input from the user or a file, and sends it to the server.

```
public class Client {
    Socket socket;
    private Thread reader, writer;

    public Client (String host, int port) {
        try {
            socket = new Socket(host, port); // Socket constructor connects to host on port
            reader = new Reader(this);
            writer = new Writer(this);
            // give the reader higher priority
            reader.setPriority(6); // reader has priority over the write
            writer.setPriority(5);
            reader.start(); // start reader thread (i.e., invoke Reader.run(this=reader)
            writer.start(); // start writer thread (i.e., invoke Writer.run(this=writer)
        }
        catch (Exception e) { // catch the most general type of exception
            System.err.println(e); // IOException or UnknownHostException?
        }
    }

    // main just creates the client object, which starts the reader/writer threads
    public static void main(String[] args) {
        try { new Client(args[0], Integer.parseInt(args[1])); } // host & port from cmdline
        catch (NumberFormatException e) { System.err.println(e); System.exit(-1); }
    }
}
```

The Writer Thread

```
class Writer extends Thread {
    private Client client;

    public Writer(Client c) {
        super("Client Writer");
        this.client = c;
    }

    public void run() {
        BufferedReader in = null;
        PrintWriter out = null;
        try {
            String line;
            in = new BufferedReader(new InputStreamReader(System.in));
            out = new PrintWriter(client.socket.getOutputStream());
            while (true) {
                line = in.readLine();
                if (line == null) {
                    break;
                }
                out.println(line);
            }
        }
        catch (IOException e) {
            System.err.println("Writer: " + e);
        }
        finally try { if (out != null) out.close(); }
        catch (IOException e) { System.err.println(e); System.exit(0); }
    }
}
```

Comparison: Using Exceptions in C++

```
template<class T, int SIZE> class Vector {
    T _vec[SIZE];
    int _n;
public:
    class RangeError { // "nested C++ class" defining a Vector specific exception type
    public:
        int index;
        Vector<T, SIZE>* self
        Range(Vector<T,SIZE*> s, int i) : self(s), index(i) {}
    };
    ...
    T& operator[](int i) throw(RangeError) {
        if (0 <= i && i < sz)
            return _vec[i]
        else
            throw RangeError(this, i);
    }
    ...
}

Vector<int, 100> x;

try { // try block
    iterate_over(x);
}
catch (Vector<int, 100>::RangeError& r) { cerr << "bad index" << r.index << '\n'; }
```

Exceptions Should Be Exceptional

Consider the following:

1. Should you raise an exception when you iterate beyond the end of an array or vector?
2. Should you raise an exception when you reach the end of a file?
3. Should you raise an exception when you pop an empty stack?

In general, use exceptions to cope with unpredictable events and not as a way to cause control to jump around in your program to all sorts of handler routines. Java does not have a goto statement, but you could abuse the exception handling mechanism and implement a non-local goto mechanism. THIS IS A VERY BAD IDEA.

This style of event processing is better done using other mechanisms, such as "call backs", which are commonly found in window managers and network servers.

Q: What about retrying a method that raised an exception that was caught? When is this okay?

```
bool success = FALSE;
while (!success)
{
    try {
        .... // code that might raise an exception
        success = TRUE;
    }

    catch (SomeException e) {
        ... // code to correct the problem leading to the exception and try again,
        // or possibly re-throw the exception, or abort.
    }
    // fall out and try again
}
```

Comparison: Using Exceptions in C++

You must arrange for heap allocated objects to be explicitly destructed. How can you do this? Well, you could program so that you always catch ANY thrown exception, and then cleanup

```
g()
{
    Stack<int> stack;
    String *s = new String();
    ...
    try {
        f(); // assume f throws some exception
        ...
        delete s;
    }

    catch (...) { // use of ellipses in catch expression in C++ means "catch all"
        delete s; // have to be sure to delete heap objects, since we have no gc
        throw; // rethrow caught exception to outer scope
    }
}
```

Summary of Design Hints for Using Exceptions

1. Catch only what you can handle.
2. Exceptions are for exceptional circumstances.
3. Don't rely on exceptions if you can test and return a meaningful error code.
4. Don't throw an exception if you can continue to execute safely.
5. Let library users choose how they want to handle exceptions (i.e., write subclasses of the class Exception).
6. Release any resources that might be held unnecessarily during exception processing.
7. Implement a finally clause to do any final cleanup.