Experiment 2 – Exceptions in Java and C++

Exceptions are central to the C++ as well as the Java programming paradigm. They provide a powerful structured mechanism for dealing with errors that is far more flexible than the traditional Unix/C model of using a function's return value to indicate success or failure.

The following paragraphs were taken from Sun's online Java tutorial:

> *The Java programming language uses exceptions to provide error-handling capabilities for its programs. An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.*
>
> *When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.*
>
> *After a method throws an exception, [...] the runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. [...] The exception handler chosen is said to catch the exception.*
>
> If no exception handler is found the program terminates.

For further information please find help in:

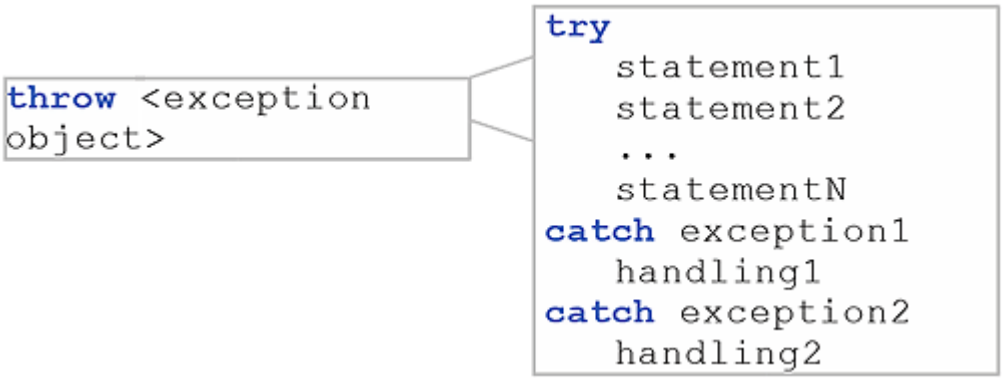
http://java.sun.com/docs/books/tutorial/essential/exceptions/

Contents

Basic Components

- `try` - block of statements, where exceptions can occur and be caught
- `catch` – block of statements to handle exceptions, that occurred in corresponding try block

- `throw` – keyword to throw exception



Exceptions in Java

In Java exceptions are instances of corresponding exception classes and can inherit from `java.lang.Exception` (must at least extend `java.lang.Throwable`).

„Non-runtime-exceptions“ are part of methods signature.

```
void foo (...) throws IOException, FileNotFoundException
```

In Java there are, unlike C++, no (real) destructors for cleaning up.

Java – finally

`finally` - for cleaning up after exception may have occurred

```
FileInputStream f = new FileInputStream (filename);
try{
     // do something with f
     throw new Exception ();
}finally{
     f.close ();
}
```

`finally` is always executed even if exception has occured.

Java – by Example

```
import java.io.*;

(1)     public class Example {
(2)        private static RandomAccessFile file;
(3)        private static String test;
(4)
(5)        public static void main(String[] args) {
(6)           try {
(7)               file = new RandomAccessFile(new
          File("./textfile"),"r");
(8)               try {
(9)                   test = file.readUTF();
```

```
(10)              }
(11)              finally {
(12)                  file.close();
(13)              }
(14)          } catch(FileNotFoundException fnfe){
(15)              System.out.println("File not found.");
(16)          } catch(EOFException e1){
(17)              System.out.println("Read failed: End of file
          has been reached.");
(18)          } catch(UTFDataFormatException e2){
(19)              System.out.println("Read failed: wrong data
          format.");
(20)          } catch(IOException e3){
(21)              System.out.println("Read failed: I/O
          problem.");
(22)          } catch(Exception e){
(23)              System.out.println("Read failed: unknown
          Problem.");
(24)          }
(25)       }
(26)    }
```

It is important to note that the order of catch-statements is crucial. The most general exception must be caught at last.

Exceptions in C++

In C++ an exception object can be any type (also **int**, **float** and so on...).

```
try{
     // do something
     throw 10;
} catch (int i){
     cout << „caught “ << i << endl;
}
```

In C++ exceptions should be caught by reference, except for basic types like **int**.

Resource Acquisition Is Initialization (RAII)

C++ has no **finally** clause like Java. But you can use the destructor to free resources even in the presence of exceptions. That technique is called “resource acquisition is initialization” (RAII) or scoped objects.

```
class File {
     public:
          File (...) {
               f = fopen (...);
               ...
          }
          ~File() {
```

```
            if (NULL != f)
                fclose (f);
        }
};
...
try {
    File file (...);
    // do something
        // destructor of file will be called
    // and file will be closed
} catch (...) {
    ...
}
```

Object `file` is called scoped object, because it lives only in the scope of **`try`** (between the brackets). The compiler ensures that if the execution leaves a scope all destructors of objects that live only in this scope are executed. A scope can be left normal control flow or by an exception. That's why scoped objects are nice way to wrap resources in C++. You get the clean up nearly for free.

Note: Dynamic allocated objects (allocated by **`new`**) are no scoped objects! They live on the heap and their lifetime is not bounded by any scope. The destructor of a dynamic object is only by an explicit **`delete`**.

Tasks – Exceptions in C++ and Java

Today's tasks are about exceptions in C++ and Java. You have to improve the robustness of your rcopy. The second part is about finding an exception-related bug in Java.

Task 1 – Exceptions in `rcopy` (C/C++)

Where and how can you use exceptions in your **`rcopy`** application? Extend the C Version of rcopy with exceptions (of course now in C++). Do not reuse your C++ version from task 1. Write C++ wrapper classes for your C solution! Help: http://www.cplusplus.com/doc/tutorial/

Details:

- wrapper class for **File** with:
  - `open`, `seek`, `read`, `write`, `close`, `size` ...
  - destructor should call `close` if necessary
- wrapper class for memory realm: **Buffer**
  - constructor allocates: `Buffer b (1024);`
  - method for reversing the buffers content
  - destructor deallocates memory
  - class File has to work together with class Buffer

Also take care of exception handling. You should at least catch exceptions and provide some error message. For doing so you should replace the standard error codes:

```
class Exception {
int error_code;
...
private:
Exception (int e): error_code (e) {}
...
};

#include <cerrno>
...
if (-1 == fseek (...))
throw Exception (errno);
```

Do not forget cleaning up and freeing resources with the help of scoped objects.

```
File f;
try{
     f.open (...);
     // do something
} catch (const Exception& e){
     // do some recovery
} // f.~File () is called for local
  // variables if their block is done
  // no matter if an exceptions is
  // thrown or not
```

Make your rcopy robust against *all* kinds of user or operation system errors. That means – check all return values for operating system errors (mostly I/O errors) and command line arguments for user errors.

Stick to our files and interfaces that we provide via SubVersion. You may add methods (e.g. for file size or reversing the buffers content), but you are not allowed to change the signature of any method specified by us.

Task 2 – Exceptions in `MyStack` (Java)

You shall find the bug in the class `MyStack`[1].

Capabilities of `MyStack` are:

- storing arbitrary objects on a stack
- reallocating memory if necessary
- returning the number of elements on the stack
- returning a string representation of the stack contents

You should implement your own main method. Find the included bug by testing the methods of `MyStack`. The bug may cause an error, that leaves the `MyStack` object in an inconsistent state. Find and fix the bug the implementation!

Hint: Think about invariants of our stack implementation and how to break them with the existing methods. Keep in mind: the current task is about exceptions. So the bug may be related to exceptions somehow.

Please mind that there is no method to get the last element or remove elements from the stack. This

1 You with your initial checkout of the tasks repository.

is **NOT** the bug. It is just to keep the example as simple as possible.

Hints

Here are a few hints for task 2:

- look for RAII in your favorite C++ book
  e.g. Scott Meyers: "*More Effective C++*"
  or the Internet
  e.g. http://www.gamedev.net/reference/articles/article953.asp
- destructor must not throw exceptions
  Scott Meyers: "*Effective C++*"
  e.g.: https://www.securecoding.cert.org/confluence/display/cplusplus/ERR33-CPP.+Destructors+must+be+exception-safe
- check the return values of function calls – do not solely rely on **`ferror`** (except for **`fread`**)
  (using **`ferror`** is usually fine, but you might fail our fault injection tests)
- the function **`stat`** the file size among others
- throw an exception if your API is miss-used (e.g. reading from an already closed file)
- prevent undefined behavior (see **`fclose`**)

Organizational Remarks

Deadline

Solve this week's experiment at home or in the lab.

Please be aware that there is a deadline for this task. You can find the exact date published on our website. If you do not hand in your solution until the specified day we will regard your solution as missing so you will not receive your certificate.

Directory Structure

We require a specific directory structure for your solutions.

```
2_exception/
     rcopy/
          File.cc
          File.h
          Buffer.cc
          Buffer.h
          Exception.cc
          Exception.h
          rcopy.cc
          Makefile
     mystack/
          mystack.java
```

Some hints to help us to speed up the checking of your solution:

The header files, we provide you, contain already the expected class definitions. You are allowed to add new members. But you must implement (in the corresponding `*.cc` files) and use the methods we specified. And you are not allowed to change signature of the provided methods. It is not necessary to add new source code files. But if you do so, adapt the `Makefile` to include them into the build process.

When fixing the bug in `MyStack`, do not change the signature of existing methods. It will not be necessary. You can add a class with an `main` method for testing, we will not use it for our checking.

Using SubVersion (svn)

Use SubVersion to commit your solution. Once one of your solution's directories was checked in an automatic test routine will be started. To do so it is necessary that you delete an existing file („DoNotTest“) from svn. Otherwise, as long as this file exists there, your solution will not be regarded. You may create a new „DoNotTest“.