Universidad
Carlos III de Madrid

Industrial Informatics I Lab Session

Bachelor in Industrial Electronics and Automation

2016-2017

Lab 8

Teaching staff :

Juan Carlos Gonzalez Victores

Mohamed Abderrahim

Lab 8 – Exceptions and error handling

An exception is an abnormal event that breaks the normal execution of a program. You can know what exactly caused the error based on the information accompanied by it, which indicates the error type. If not treated properly, an error can interrupt the thread.

In C++ the exception handling system revolves around three keywords: `try`, `catch` and `throw`. Overall, we can say that the set of instructions that handle exceptions is enclosed in the `try` block; when an error occurs within the `try` block, an exception is thrown using the keyword `throw`; and finally the exception is captured and processed from within the `catch` block. It should be emphasized that exceptions are nothing but a variable of a data type, and as such they must be defined, declared and operated.

The general form of `try-catch` is as follows:

```
try {
      // try block
}
catch (type1 e) {
      // catch block where is processing type1 exceptions
}
catch (type2 e) {
      // catch block where is processing type2 exceptions
}
...
```

As you can see it is normal to have several nested `catch` blocks for different types of exceptions.

The `try` block can be of any size: from a few instructions long up to as big as covering the entire `main` function.

An exception is caught by the first `catch` block that has the same data type as the exception. If no exceptions occur in a `try` block, then the subsequent `catch` blocks are skipped, and the program continues normally.

An exception can be raised by using the command `throw exception`, where `exception` could be of any data type. For this exception to be caught, it has to be launched from within the `try` block, either directly or indirectly through a function call, from where this exception is launched. If an exception is not handled, it results in abnormal termination of the program.

Look at the following simple example demonstrating this:

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
      cout << "Learning exceptions in C++" << endl;
      try {
            cout << “Inside try block” << endl;
            throw 99;
            cout << “This is not executed” << endl;
      }
      catch(int i) {
            cout << "Catched exception with value: " << i << endl;
      }
```

```
	cout << "END" << endl;

	return 0;
}
```

Note that once an exception is thrown in the `try` block, it immediately ends the execution of the `try` block and the control is passed to the corresponding `catch` block.

Generally the code contained within the `catch` block is meant to try to solve the error that has occurred or, if not possible, terminate the execution of the program in an orderly manner.

If an exception is not handled from within the function that generated the exception then it is forwarded to the function that invoked it, and so on until it finds a handler that handles the exception or until the application terminates abnormal.

Exercise

Write a program containing a function that accepts two numbers as its parameter, performs the division operation on these numbers, and returns the results as its return value. In the event that the divisor is the number zero, an exception must be generated that will be captured in the `main` function, and the exception must be handled by asking the user to re-enter a non-zero divisor, until the division operation could be performed.

```
#include <iostream>
#include <cstring> //para el strcpy

using namespace std;

class MyException {
	public:
		char mesage[80];
		int number;
		MyException() {
			*mesage = 0; number = 0;
		}
		MyException(const char *msg, int n) {
			strcpy(this->mesage, msg);
			this->number = n;
		}
};

int main(int argc, char *argv[]) {
	int n;
	try {
		cout << "Insert a positive number: " << endl;
		cin >> n;
		if(n < 0) {
			throw MyException("Negative number!", n);
		}
	}
	catch(MyException e) {
		cerr << "ERROR: " << e.mesage << endl;
		cerr << "value = " << e.number << endl;
	}

	return 0;
```

```
}
```

An exception may be of any type, including user defined data type [i.e. your own class]. In fact, usually most exceptions are some sort of class, as this allows you to include extra information about the error for better error handling or for providing further details about the error.

The order of the `catch` blocks can be very important when dealing with exceptions that include a parent class type and one or more inherited class types. The `catch` block in-charge of handling exceptions of the parent class type can also capture exceptions of its inherited class types. Therefore, if you want to catch exceptions of inherited class types and the parent type in different catch blocks, then the catch blocks of the inherited class types must be placed above the parent class type. Otherwise, the exception handler of the parent class type will capture all the inherited class type exceptions as well.

1. Capturing all exceptions

The following method is used to capture all kinds of exceptions, and treat it the same way:

```
catch (...) {
      // processing exception
}
```

A very common use of this is as a default exception handler. Placing this as the last `catch` block, in a series of `catch` blocks, allows for generally handling those exceptions which are not required to be handled explicitly.

2. Exception specifications for functions

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a `throw` suffix to the function declaration:

```
type_returned funtion_name(parameters_list) throw (type_list) {
      // function body
}
```

3. Relaunching exceptions

An exception may be re-launched only from within its handler, and this is done with the expression `throw`; with no arguments. This causes the exception to be forwarded to its external level (Another `try-catch` block in the layer above). Finally remember that the key in handling exceptions is to provide an orderly way to handle errors and this involves correcting the situation that caused the error wherever possible.

Exercise

Modify the previous 'Division by 0' program such that now the generated exception from within the sub-function is re-launched, and the exception is handled from within the `main` function.

Exercise

Continue the Computer Simulator project by implementing error handling via exceptions. Now let's create a class hierarchy of exceptions that allow us to handle possible errors that may arise, clearly, elegantly and in a controlled manner.

Create a parent class `ComputerException` that will have a single attribute message of type `string message`, which the rest of its inherited class will inherit. Now create classes `InputException`, `ProcessorException` and `OutputException`, inheriting from the class `ComputerException`, which will handle the errors in the respective devices. For example, if we have `Keyboard` as our Input type, then class `InputException` will handle potential errors that may arise when opening a file, reading it, etc.

Errors that arise during the execution of the program should be captured and the program should be able to recover from an error wherever possible. For example, if a file cannot be opened to read data as the file may not exist, then it should be possible to provide a different file name and/or file path to the program. In the event that the system cannot recover from an error, the program must ends in an orderly manner and with a clear message indicating that the computer is part of the error.

A possible design is as follows: