Exceptions

shan@iha.dk
V1.2

Introduction

In this exercise you will be trying out how exceptions work and what to do with them. There will be guarantees to be met as well as mistakes to be found. You will be altering a C-style program to utilize exceptions. Finally you will be discussing as to whether exceptions is the choice for error handling or not...

Exercise 1 Safety Guarantees

Exercise 1.1 Code inspecting

Consider each of these 4 different code snippets. What do you reckon that they provide as a guarantee? You have to be specific and relate to the different functionalities that actually are in use.

This means that some functionalities may provide one type of guarantee whereas others may provide another.

When designing code you should also strive to ensure that although your code *insides* do as promised, interfaces should promote proper use as well - consider if any of the interfaces fail - e.g. cannot uphold guarantees. Also note any mistakes you may find in your quest.

Use reasoning to substantiate your answer!

Do note that small subtle tricks may have been employed. Do not let yourselves be fooled! :-D.

Exercise 1.1.1 Snippet 1

Listing 1.1: C++ Snippet 1 - Guarantees

```
1  class Test { /* Some code */ };
2
3  template<typename T, int N>
4  class MyArray
5  {
6  public:
7    T& operator[](size_t i)
8    {
9      return data_[i];
10   }
11
12 private:
13    T   data_[N];
14 };
15
16 /* Using */
17 void f()
18 {
19   MyArray<Test, 10> my;
20
21   Test t;
22
23   my[5] = t;
24 }
```

Exercise 1.1.2 Snippet 2

Listing 1.2: C++ Snippet 2 - Guarantees

```
1  class Test { /* Some code */ };
2
3  template<typename T>
4  class MyVector
5  {
6  public:
7    MyVector(size_t capacity)
8    : capacity_(capacity), count_(0), data_(new T[capacity])
9    {}
10
11   bool full() const { return (count_ == capacity_); }
12
13   void push_back(const T& oneMore)
14   {
15     if(full())
16     {
17       capacity_ *= 2;
18       T* newData = new T[capacity_];
19
20       std::copy(data_, data_+count_, newData);
21       std::swap(data_, newData);
22       delete[] newData;
23     }
24
25     data_[count_] = oneMore;
26     ++count_;
27   }
28 private:
29    size_t   capacity_;
30    size_t   count_;
31    T*       data_;
32 };
33
34 /* Using */
35 void f()
36 {
37   MyVector<Test> my(20);
38
39   Test t;
40
41   my.push_back(t);
42 }
```

Exercise 1.1.3 Snippet 3

Listing 1.3: C++ Snippet 3 - Guarantees

```
1  class String
2  {
3  public:
4    String() : s_(nullptr){}
5
6    String(const char* s) : s_(new char[strlen(s)+1])
7    {
8      std::strcpy(s_, s);
9    }
10
11   String(const String& other)
12   : s_(new char[strlen(other.s_)+1])
13   {
14     std::strcpy(s_, other.s_);
15   }
16
17   String& operator=(const String& other)
18   {
19     delete[] s_;
20     s_ = new char[strlen(other.s_)+1];
21     std::strcpy(s_, other.s_);
22     return *this;
23   }
24
25   ~String()
26   {
27     delete[] s_;
28   }
29 private:
30   char* s_;
31 };
32
33 /* Using */
34 void f()
35 {
36   String s("Hello world");
37
38   String aCopy(s);
39
40   s = "Hello girls";
41 }
```

Exercise 1.1.4 Snippet 4

Listing 1.4: C++ Snippet 4 - Guarantees

```
1  class DataSet
2  {
3  public:
4    DataSet(Key* key, Blob* blob)
5    : key_(key), blob_(blob)
6    {
7      if(!key->isValid())
8        throw InvalidKey(key->id());
9    }
10
11   void overWrite(const Key* key, const Blob* blob)
12   {
13     *key_  = *key;
14     *blob_ = *blob;
15   }
16
17   ~DataSet()
18   {
19     delete key_;
20     delete blob_;
21   }
22 private:
23   Key*  key_;
24   Blob* blob_;
25 };
26
27 /* Using */
28 void f()
29 {
30   DataSet ds(new Key, new Blob);
31
32   {
33     Key k(getKeyValue());
34     Blob b(fetchDBBlobByKey(k));
35
36     ds.overWrite(&k, &b);
37   }
38 }
```

Exercise 1.2 Providing the strong guarantee

Consider each of the snippets above; upgrade the one/all so that it/they provide the strong guarantee.

For each of them reflect upon whether it is a sound idea to perform such an *upgrade*?

Exercise 1.3 MyVector template class that provides the strong guarantee

Use the below interface to implement the class template `MyVector<>` and make each and every function provide the "strong guarantee".

Discuss:

- Some may be hampered performance by this, which and why
- Some are not affected performance wise, which and why

Listing 1.5: The class Message

```
1  template <typename T>
2  class MyVector
3  {
4  public:
5    explicit MyVector(int capacity = 10);
6
7    MyVector(const MyVector& other);
8
9    MyVector& operator=(const MyVector& other);
10
11   ~MyVector();
12
13   int size() const;
14
15   T& back();
16
17   void push_back( const T& t );
18   void pop_back();
19
20   void insert(const T& t, size_t n);
21
22   T* begin();
23   T* end();
24
25   T& operator []( size_t i );
26 private:
27   T* data_; /* Contains the actual elements - data on the heap */
28 };
```

Exercise 2 Exception handling

In this exercise you will be set the task of transforming a program utilizing the classic C approach for dealing with errors to the C++ way using exceptions.

Basically its about a log system, where one can write a simple log entry. A simple file is used as storage and for every added entry the log file is opened, the entry is written and the file closed.

Exercise 2.1 Pure inspection

Start by inspecting the code in `Transformation`[1] and take note of how errors are indicated as well as how one can determine the cause.

Exercise 2.2 Testing the program

Compile and run the program, does it work as expected?

What happens if we make the file `Exception.log` read-only?

Exercise 2.3 Transformation

At this point we commence the transformation from return codes to exception handling.

The exception that you are to throw upon error inherits from `std::runtime_error` found in `<stdexcept>`. Prior to writing it do consider the various error states that you need to deal with.

The exception class should have an interface like the below:

Listing 2.1: The class Message

```
1  class LogFileException  : public std::runtime_error
2  {
3  public:
4    enum EState { ??? };
5
6    explicit LogFileException( EState state, const std::string& str = "" )
7    : runtime_error( str ), state_( state ) {}
8
9    EState getState() { return state_; }
10 private:
11   EState  state_;
12 };
```

Part of the transformation process imposes changes to instances variables as well as return values - dont forget.

Obviously this extends to the using party as well, since all the error checking can be exchanged for some try-catches.

Things to consider:

- Why must the constructor be `explicit`
- What changes does exceptions impress onto the code and design as opposed to the original 'C' error handling code.
- This is a somewhat simple setup, but consider whether you believe this to be a prudent approach using exceptions. Discuss and exemplify your position on the matter.

Exercise 3 When to use exceptions

Inspect each of those snippets below and reason why you *should* or *should not* use exceptions. *IF* you disagree with the chosen implementation then rewrite it and discuss how your changes are an improvement.

[1] Found in the directory as this document.

Exercise 3.1 Snippet 1

Listing 3.1: The class Message

```
1  int16_t pollPressureValve()
2  {
3    if(PORT6IO_STATUS & DATA_RDY)
4      return PORT6IO_DATA;
5    else
6      throw DataNotRdy;
7  }
8
9  /* Using */
10 void f()
11 {
12   try
13   {
14     int16_t value = pollPressureValve();
15     ctrlSpeed = process(value);
16     writeToLog(ctrlSpeed, value);
17   }
18   catch(DataNotRdy& dnr)
19   {
20     /* Do stuff */
21   }
22
23 }
```

Exercise 3.2 Snippet 2

Imagine the following snippet run on a hard realtime system controlling a heliocopter.

Listing 3.2: The class Message

```
1  GPSPos getHelioGPSPos()
2  {
3    /* ... */
4  }
5
6  Altitude getHelioAltitude()
7  {
8
9    Altitude alt(/* get sensor data */);
10   if(!valid(alt))
11     throw CensorError("Altitude error");
12
13     /* ... */
14 }
15
16 Attitude getHelioAttitude()
17 {}
18
19 SetPos calcSpeedAdjustment(Altitude alt, Attitude att)
```

```
20 {
21   /* Do some calcs... */
22
23   if(!valid(newPos))
24     throw ExtremeDeviation("New position is invalid");
25 }
26
27
28 int main()
29 {
30   for(;;)
31   {
32     try
33     {
34       for(;;)
35       {
36         Altitude alt = getHelioAltitude();
37         Attitude att = getHelioAttitude();
38         SetPos sp = calcSpeedAdjustment(atl, att);
39
40       }
41     }
42     catch(CensorError&)
43     {
44       RecalibrateSensors();
45     }
46     catch(...)
47     {
48       EmergencyLand();
49       ResetHW();
50     }
51   }
52 }
```