Programming in C/C++
Exercises set one: Exceptions

Christiaan Steenkist
Jaime Betancor Valado
Remco Bos

November 17, 2016

Exercise 1, catching and throwing references

Exercise description

There are 3 parts to this exercise:

- Show that exception catchers catching objects result in additional copies of thrown objects, compared to exception catchers catching references to objects.
- Also show that when throwing objects or references copies of the (referred to) objects are thrown.
- Also answer the question whether 'throw;' results in throwing the currently available exception or a copy of that exception.

1a, Throwing by value, catching by value

Throwing object 'main object' by value. Caught exception by value.

```
1 // 2 copies are found
2 Hello by 'local object' (copy)  (copy)
```

1b, Throwing by value catching by reference

Throwing object 'main object' by value. Caught exception by reference.

```
1 // 1 copy is found (answered part 2)
2 Hello by 'local object' (copy)
```

Part 2

The '(copy)' is appended by the copy constructor, so atleast 1 copy is made by throwing an object.

Part 3

'Throw' throws the original exception. An exception is rethrown when it is not caught yet in the present try-block level, then the exception will be retrown to a higher level until it is caught. That means that the exception is handled and will be inactivated.

Code listings

Listing 1: demo.h

```
1  #ifndef DEMO_H
2  #define DEMO_H
3
4  #include <iostream>
5  #include <string>
6
7  using namespace std;
8
9  class Demo
10 {
11     string d_name;
12
13     public:
14         Demo(string name)
15         :
16             d_name(name)
17         {
18         }
19         Demo(Demo const &other)
20         :
21             d_name(other.d_name + " (copy) ")
22         {
23         }
24         ~Demo()
25         {
```

```
26          }
27          void fun()
28          {
29              Demo toThrow("'local object'");
30              cout << "Throwing object " << d_name
31                  << "by value"<< "\n";
32              throw toThrow;
33          }
34          void hello()
35          {
36              cout << "Hello by " << d_name << "\n";
37          }
38  };
39
40  #endif
```

Listing 2: main.cc

```
1   #include "demo.h"
2
3   int main()
4   {
5       Demo demo("'main object'");
6       try
7       {
8           demo.fun();
9       }
10      // Code below was commented/uncommented for
11      //  each situation, i.e. catch 1 was used for
12      //  catching by value, catch 2 for cathcing by
13      //  reference.
14      catch (Demo d) // 1
15      {
16          cout << "Caught exception by value\n";
17          d.hello();
18      }
19      catch (Demo &d) // 2
20      {
21          cout << "Caught exception by reference\n";
22          d.hello();
23      }
```

```
24 }
```

Exercise 2, delete[]

We designed a simple class that can only be constructed 4 times and then causes an exception.

Explanation

First memory is allocated for 10 objects. When the exception is reached during construction of objects, the construction of further objects is terminated. This results in the destructors being called of the objects that were alreaedy constructed.

Output

```
1 Constructor called
2 Constructor called
3 Constructor called
4 Constructor called
5 max. number of objects is reached
6 Destructor called
7 Destructor called
8 Destructor called
9 Destructor called
```

Code listing

Listing 3: maxfour.h

```
1  #ifndef MAXFOUR_H
2  #define MAXFOUR_H
3
4  #include <iostream>
5
6  using namespace std;
7
8  class MaxFour
9  {
10     public:
11         MaxFour();
12         ~MaxFour();
```

```
13 };
14
15 #endif
```

Listing 4: constructor.cc

```
1  #include "maxfour.h"
2  #include <array>
3
4  MaxFour::MaxFour()
5  {
6      static int constNum = 0;
7      ++constNum;
8      if (constNum > 4)
9      {
10         throw 0;
11     }
12     cerr << "Constructor called\n";
13 }
```

Listing 5: destructor.cc

```
1 #include "maxfour.h"
2
3 MaxFour::~MaxFour()
4 {
5     cerr << "Destructor called\n";
6 }
```

Listing 6: main.cc

```
1  #include "maxfour.h"
2  #include <array>
3
4  int main()
5  {
6      try
7      {
8          MaxFour *objArr = new MaxFour[10];
9          delete[] objArr;
10     }
11     catch(...)
```

```
12      {
13          cerr << "max. number of objects is reached \n"
    ;
14      }
15  }
```

Exercise 3, exceptions in the `Strings` class

Exception handling has been put into the `Strings` class. Generally bad allocations are handled by the class itself. The constructor can still throw bad allocation exceptions in case there is not enough memory to create a strings class.

Code listing

Listing 7: strings.h

```
1  #ifndef INCLUDED_STRINGS_
2  #define INCLUDED_STRINGS_
3
4  #include <iosfwd>
5
6  // All public member functions have the basic
7  //  guarantee.
8
9  // All public member functions ahve the strong
10 //    guarantee.
11
12 // Out of all the public member functions only
13 //  the constructors, reserve and resize function
14 //  do not ahve the nothrow guarantee.
15
16 class Strings
17 {
18   size_t d_size = 0;
19   size_t d_capacity = 1;
20   // now a double *
21   std::string **d_str;
22
23   public:
24     Strings();
25
```

```
26      Strings(int argc, char *argv[]);
27      Strings(char **environLike);
28
29      ~Strings();
30
31      size_t size() const;
32      size_t capacity() const;
33      // for const-objects
34      std::string const &at(size_t idx) const;
35      // for non-const objects
36      std::string &at(size_t idx);
37
38      // add another element
39      void add(std::string const &next);
40
41      void resize(size_t newSize);
42      void reserve(size_t newCapacity);
43
44    private:
45      // private backdoor
46      std::string &safeAt(size_t idx) const;
47      // to store the next str.
48      std::string **storageArea();
49      void destroy();
50      // also deletes allocated strings
51      void destroy(size_t start, size_t end);
52      // to d_capacity
53      std::string **enlarged();
54      std::string **rawPointers(size_t nPointers);
55
56      // to roll back
57      void reroll(string **oldStr, size_t oldCapacity,
58        size_t oldSize)
59  };
60
61  // potentially dangerous practice:
62  // inline accessors
63  inline size_t Strings::size() const
64  {
65    return d_size;
```

```
66  }
67
68  inline size_t Strings::capacity() const
69  {
70    return d_capacity;
71  }
72
73  inline std::string const &Strings::at(
74    size_t idx) const
75  {
76    return safeAt(idx);
77  }
78
79  inline std::string &Strings::at(size_t idx)
80  {
81    return safeAt(idx);
82  }
83
84
85  #endif
```

Listing 8: add.cc

```
1   #include "strings.ih"
2
3   // Basic: tmp is deleted after an allocation exception
4   //          from "new string(next)".
5   // Strong: Capacity is rolled back if
6   //           "new string(next)" fails but not
7   //           "storageArea()".
8   // Nothrow: This function does not throw any
9   //            exceptions.
10
11  void Strings::add(string const &next)
12  {
13    string **tmp = 0;
14    size_t oldCapacity = d_capacity;
15    try
16    {
17      tmp = storageArea();
18
```

```
19      tmp[d_size] = new string(next);
20    }
21    catch (bad_alloc &ba)
22    {
23      delete[] tmp;
24      d_capacity = oldCapacity;
25      cerr << "(Strings) Unable to add string:"
26        << " \"memory allocation failed\".\n";
27      return;
28    }
29
30    // destroy old memory if new storageArea
31    if (tmp != d_str) // was allocated
32    {
33      // destroy the old string * array
34      destroy();
35      d_str = tmp;
36    }
37
38    ++d_size;
39  }
```

Listing 9: destroy2.cc

```
1  #include "strings.ih"
2
3  void Strings::destroy(size_t start, size_t end)
4  {
5    for (size_t index = start; index != end; ++index)
6      delete d_str[index];
7      delete[] d_str;
8  }
```

Listing 10: enlarged.cc

```
1  #include "strings.ih"
2
3  // Basic: Nothing is allocated if rawPointers fails.
4  // Strong: Nothing is changed if the exception occurs.
5
6  string **Strings::enlarged()
```

```
{
  string **ret = 0;
  // new block, doubling the # pointers
  try
  {
    ret = rawPointers(d_capacity);
  }
  catch (bad_alloc &ba)
  {
    throw;
  }

  // copy the existing pointers
  for (size_t idx = 0; idx != d_size; ++idx)
    ret[idx] = d_str[idx];

  return ret;
}
```

Listing 11: reroll.cc

```
#include "strings.ih"

void Strings::reroll(string **oldStr, size_t
   oldCapacity,
  size_t oldSize)
{
  d_str = oldStr;
  d_capacity = oldCapacity;
  d_size = oldSize;
}
```

Listing 12: reserve.cc

```
#include "strings.ih"

// Basic: If "enlarged()" fails then there
//         are no allocations or leaks.
// Strong: The capcity is rolled back if
//          enlarging fails.

```

```
8  // This function may throw bad_alloc.
9
10 void Strings::reserve(size_t nextCapacity)
11 {
12   if (d_capacity < nextCapacity)
13   {
14     size_t oldCapacity = d_capacity;
15     while (d_capacity < nextCapacity)
16       d_capacity <<= 1;
17
18     try
19     {
20       d_str = enlarged();
21     }
22     catch (bad_alloc &ba)
23     {
24       d_capacity = oldCapacity;
25       string message = "(Strings) Unable to increase";
26       message += " size: \"memory allocation";
27       message += " failed\".\n";
28       throw bad_alloc(message);
29     }
30   }
31 }
```

Listing 13: resize.cc

```
1  #include "strings.ih"
2
3  // Basic: If reserve fails then nothing is changed
4  //         so there are no leaks.
5  //        If the initializing of an empty string fails
6  //         then all the new empty strings will be
7  //         deleted along with the resized allocation.
8
9  // Strong: If reserve fails then the requested
10 //          enlarging can not be performed
11 //          and so resize stops.
12 //         If the initializing of the empty strings
13 //          fails then the whole class is rolled back.
14
```

```
15 // This function may throw a bad_alloc from reserve
16 //  or a bad_alloc from enlarging.
17
18 void Strings::resize(size_t newSize)
19 {
20   string **oldStr = d_str;
21   size_t oldSize = d_size;
22   size_t oldCapacity = d_capacity;
23
24   // make sure there's enough memory
25   reserve(newSize);
26
27   // enlarging? initialize new strings
28   if (d_size > newSize)
29   {
30     for (; d_size != newSize; ++d_size)
31     {
32       try
33       {
34         d_str[d_size] = new string;
35       }
36       catch (bad_alloc &ba)
37       {
38         destroy(oldSize, d_size);
39         reroll(oldStr, oldCapacity, oldSize);
40
41         string message = "(Strings) Unable to";
42         message += " increase size: \"memory";
43         message += " allocation failed\".\n";
44         throw bad_alloc(message);
45       }
46   }
47   // shrinking? remove excess strings
48   else if (newSize < d_size)
49   {
50     for (; d_size-- != newSize; )
51       delete d_str[d_size];
52   }
53 }
```

Listing 14: storagearea.cc

```
1  #include "strings.ih"
2
3  // Basic: This function does not cause any leaks
4  // Strong: This function rolls back the capacity
5  //          if enlarged fails.
6
7  string **Strings::storageArea()
8  {
9    // enough room?
10   if (d_size + 1 < d_capacity)
11     // return the current memory block
12     return d_str;
13
14   try
15   {
16     // double the capacity
17     d_capacity <<= 1;
18     // return ptr to the enlarged space
19     return enlarged();
20   }
21   catch(bad_alloc &ba)
22   {
23     d_capacity >>= 1;
24     throw;
25   }
26 }
```

Listing 15: strings1.cc

```
1  #include "strings.ih"
2
3  // Basic: If the constructor fails then
4  //         the allocations are not finished
5  //         so there are no leaks.
6  // NOT Strong: There is no state to roll back to.
7  // NOT Nothrow: This constructor throws.
8
9  // This function may throw bad_alloc.
10
```

```
11 Strings::Strings()
12   d_str(rawPointers(1))
13 {}
```

Listing 16: strings2.cc

```
 1 #include "strings.ih"
 2
 3 // Basic: If the constructor fails then
 4 //         the allocations are not finished
 5 //         so there are no leaks.
 6 // NOT Strong: There is no state to roll back to.
 7 // NOT Nothrow: This constructor throws.
 8
 9 // This function may throw bad_alloc.
10
11 Strings::Strings(int argc, char *argv[])
12   Strings()
13 {
14   for (size_t begin = 0, end = argc; begin != end;
15       ++begin)
16     add(argv[begin]);
17 }
```

Listing 17: strings3.cc

```
 1 #include "strings.ih"
 2
 3 // Basic: If the constructor fails then
 4 //         the allocations are not finished
 5 //         so there are no leaks.
 6 // NOT Strong: There is no state to roll back to.
 7 // NOT Nothrow: This constructor throws.
 8
 9 // This function may throw bad_alloc.
10
11 Strings::Strings(char **environLike)
12   Strings()
13 {
14   while (*environLike)
15     add(*environLike++);
16 }
```

Exercise 4, calling it quits

We had to find a way to quit no matter what but still call all the destructors. To do this we made a very small custom class that is thrown so that no catch can intercept it.

Code listing

Listing 18: example.cc

```
#include <iostream>
#include <string>

using namespace std;

//Exclusive class to exit the program
class Quiter {};

int main()
try
// nesting deeper
{
  // nesting deeper
  {
    // nesting deeper
    {
      // statements of deepest nested level
      try
      {
        Quiter x;
        throw x;
        // This terminates the program,
        // the exception is caught by the
        // same-level catch and then
        // rethrown to upper levels
        // until one catch (Quiter) is
        // founded.
      }
      catch(...)
      {
```

```
31            throw;
32          }
33        }
34      }
35  }
36  catch (Quiter &quit)
37  {
38    cout << "Ending of program" << endl;
39  }
```

Exercise 6, throws

Here we have different throws that might come from exercise 6.

No.	Statement	Reason
1.	X x;	Default constructor could throw an exception
2.	cout <<x;	Output operator could throw an exception
3.	cout <<x;	If x is a type member data of X, the call of X::operator int() could throw an exception
4.	*xp = new X(x);	Copy constructor of x could throw an exception
5.	*xp = new X(x);	Operator new could throw an exception
6.	*xp = new X(x);	The overloaded operator new from X could throw an exception
7.	cout <<(x + *xp);	If X has an operator conversor to a class Y, x could throw an exception
8.	cout <<(x + *xp);	If X has an operator conversor to a class Y, *xp could throw an exception
9.	cout <<(x + *xp);	If X has an operator conversor to a class Y, operator + itself could throw an exception
10.	cout <<(x + *xp);	The same as point 3 but with (x + *xp)
11.	cout <<(x + *xp);	Operator + return an object that shortly after is destroyed, the destroyer could throw an exception
12.	delete xp;	Overloaded operator delete could throw an exception
13.	delete xp;	Destructor of xp could throw an exception
14.	}	Reaching the end of the block the destructor of x is called, so again an exception could be thrown