

# Programming in C/C++

## Exercises set eight: Overloading

Christiaan Steenkist  
Jaime Betancor Valado  
Remco Bos

November 9, 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.

#### Part 1, Throwing by value, catching by value

Throwing object 'main object' by value Caught exception by value Hello by 'local object' (copy) (copy) // 2 copies are found

-Throwing by value catching by reference:

```
1 Throwing object 'main object' by value
2 Caught exception by reference
3 Hello by 'local object' (copy) // 1 copy is found (
  answered part 2)
```

## 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 catching 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 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 3: `strings.h`

```
1  #ifndef INCLUDED_STRINGS_
2  #define INCLUDED_STRINGS_
3
4  #include <iosfwd>
5
6  // All the public member functions (bar constructors)
7  // fulfill the exception guarantees so the class
8  // as a whole fulfills the exception guarantees.
9
10 class Strings
11 {
12     size_t d_size = 0;
13     size_t d_capacity = 1;
14     // now a double *
15     std::string **d_str;
16
17     public:
18         Strings();
19
20         Strings(int argc, char *argv[]);
21         Strings(char **environLike);
22
23         ~Strings();
24
25         size_t size() const;
26         size_t capacity() const;
27         // for const-objects
28         std::string const &at(size_t idx) const;
29         // for non-const objects
```

```

30     std::string &at(size_t idx);
31
32     // add another element
33     void add(std::string const &next);
34
35     void resize(size_t newSize);
36     void reserve(size_t newCapacity);
37
38 private:
39     // private backdoor
40     std::string &safeAt(size_t idx) const;
41     // to store the next str.
42     std::string **storageArea();
43     void destroy();
44     // also deletes allocated strings
45     void destroy(size_t start, size_t end);
46     // to d_capacity
47     std::string **enlarged();
48     std::string **rawPointers(size_t nPointers);
49 };
50
51 // potentially dangerous practice:
52 // inline accessors
53 inline size_t Strings::size() const
54 {
55     return d_size;
56 }
57
58 inline size_t Strings::capacity() const
59 {
60     return d_capacity;
61 }
62
63 inline std::string const &Strings::at(
64     size_t idx) const
65 {
66     return safeAt(idx);
67 }
68
69 inline std::string &Strings::at(size_t idx)

```

```

70 {
71     return safeAt(idx);
72 }
73
74
75 #endif

```

Listing 4: 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 5: 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 6: 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()
7  {
8      string **ret = 0;
9      // new block, doubling the # pointers
10     try
11     {
12         ret = rawPointers(d_capacity);
13     }
14     catch (bad_alloc &ba)
15     {
16         throw;
17     }
18
19     // copy the existing pointers
20     for (size_t idx = 0; idx != d_size; ++idx)

```

```

21     ret[idx] = d_str[idx];
22
23     return ret;
24 }

```

Listing 7: reserve.cc

```

1  #include "strings.ih"
2
3  // Basic: If "enlarged()" fails then there
4  //         are no allocations or leaks.
5  // Strong: The capacity is rolled back if
6  //         enlarging fails.
7  // Nothrow: This function throws no exceptions.
8
9  void Strings::reserve(size_t nextCapacity)
10 {
11     if (d_capacity < nextCapacity)
12     {
13         size_t oldCapacity = d_capacity;
14         while (d_capacity < nextCapacity)
15             d_capacity <<= 1;
16
17         try
18         {
19             d_str = enlarged();
20         }
21         catch (bad_alloc &ba)
22         {
23             d_capacity = oldCapacity;
24             cerr << "(Strings) Unable to increase size:"
25                  << " \"memory allocation failed\".\n";
26         }
27     }
28 }

```

Listing 8: 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 // Nothrow: This function throws no exceptions.
16
17 void Strings::resize(size_t newSize)
18 {
19     string **oldStr = d_str;
20     size_t oldSize = d_size;
21     size_t oldCapacity = d_capacity;
22
23     // make sure there's enough memory
24     reserve(newSize);
25     if (d_capacity < newSize)
26     {
27         cerr << "(Strings) Could not resize:"
28             << " \"reserve failed\".";
29         return;
30     }
31
32     // enlarging? initialize new strings
33     if (d_size > newSize)
34     {
35         try
36         {
37             for (; d_size != newSize; ++d_size)
38                 d_str[d_size] = new string;
39         }
40         catch (bad_alloc &ba)
41         {
42             destroy(oldSize, d_size);
43             d_str = oldStr;

```

```

44         d_capacity = oldCapacity;
45         d_size = oldSize;
46         cerr << "(Strings) Unable to increase size:"
47             << " \"Memory allocation failed.\\\"\\n";
48     }
49 }
50 // shrinking? remove excess strings
51 else if (newSize < d_size)
52 {
53     for (; d_size-- != newSize; )
54         delete d_str[d_size];
55 }
56 }

```

Listing 9: 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 10: strings1.cc

```
1 #include "strings.ih"
2
3 // If the construction fails none of the
4 // exception guarantees are applicable.
5 // The caller of the constructor will
6 // want to know that it failed.
7
8 Strings::Strings()
9 try :
10     d_str(rawPointers(1))
11 {}
12 catch (bad_alloc &ba)
13 {
14     cerr << "(Strings) Unable to create strings object:"
15         << " \"memory allocation failed\".\n";
16     throw;
17 }
```

Listing 11: strings2.cc

```
1 #include "strings.ih"
2
3 // If the construction fails none of the
4 // exception guarantees are applicable.
5 // The caller of the constructor will
6 // want to know that it failed.
7
8 Strings::Strings(int argc, char *argv[])
9 try :
10     Strings()
11 {
12     for (size_t begin = 0, end = argc; begin != end;
13         ++begin)
14         add(argv[begin]);
15 }
16 catch (bad_alloc &ba)
17 {
```

```
18     throw;
19 }
```

Listing 12: strings3.cc

```
1  #include "strings.ih"
2
3  // If the construction fails none of the
4  // exception guarantees are applicable.
5  // The caller of the constructor will
6  // want to know that it failed.
7
8  Strings::Strings(char **environLike)
9  try :
10     Strings()
11     {
12         while (*environLike)
13             add(*environLike++);
14     }
15 catch (bad_alloc &ba)
16     {
17         throw;
18     }
```