# Programming in C/C++
# Exercises set eight: Overloading

Christiaan Steenkist
Jaime Betancor Valado
Remco Bos

November 10, 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:

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

```
#ifndef DEMO_H
#define DEMO_H

#include <iostream>
#include <string>

using namespace std;

class Demo
{
    string d_name;

    public:
        Demo(string name)
        :
            d_name(name)
        {
        }
        Demo(Demo const &other)
        :
            d_name(other.d_name + " (copy) ")
        {
        }
        ~Demo()
        {
        }
        void fun()
        {
            Demo toThrow("'local object'");
            cout << "Throwing object " << d_name
        << "by value"<< "\n";
            throw toThrow;
        }
        void hello()
        {
            cout << "Hello by " << d_name << "\n";
        }
};

#endif
```

Listing 2: main.cc

```
#include "demo.h"

int main()
{
    Demo demo("'main object'");
    try
    {
        demo.fun();
    }
    // Code below was commented/uncommented for
    //  each situation, i.e. catch 1 was used for
    //  catching by value, catch 2 for cathcing by
    //  reference.
    catch (Demo d) // 1
    {
        cout << "Caught exception by value\n";
        d.hello();
    }
    catch (Demo &d) // 2
    {
        cout << "Caught exception by reference\n";
        d.hello();
    }
}
```

## 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

```cpp
#ifndef INCLUDED_STRINGS_
#define INCLUDED_STRINGS_

#include <iosfwd>

// All the public member functions (bar constructors)
// fulfill the exception guarantees so the class
// as a whole fulfills the exception guarantees.

class Strings
{
  size_t d_size = 0;
  size_t d_capacity = 1;
  // now a double *
  std::string **d_str;

  public:
    Strings();

    Strings(int argc, char *argv[]);
    Strings(char **environLike);

    ~Strings();

    size_t size() const;
    size_t capacity() const;
    // for const-objects
    std::string const &at(size_t idx) const;
    // for non-const objects
    std::string &at(size_t idx);

    // add another element
    void add(std::string const &next);

    void resize(size_t newSize);
    void reserve(size_t newCapacity);

  private:
    // private backdoor
    std::string &safeAt(size_t idx) const;
    // to store the next str.
    std::string **storageArea();
    void destroy();
    // also deletes allocated strings
    void destroy(size_t start, size_t end);
    // to d_capacity
    std::string **enlarged();
    std::string **rawPointers(size_t nPointers);
};

// potentially dangerous practice:
// inline accessors
inline size_t Strings::size() const
{
  return d_size;
}

inline size_t Strings::capacity() const
{
  return d_capacity;
}

inline std::string const &Strings::at(
  size_t idx) const
{
  return safeAt(idx);
}

inline std::string &Strings::at(size_t idx)
```

```
{
  return safeAt(idx);
}


#endif
```

Listing 4: add.cc

```
#include "strings.ih"

// Basic: tmp is deleted after an allocation exception
//          from "new string(next)".
// Strong: Capacity is rolled back if
//           "new string(next)" fails but not
//           "storageArea()".
// Nothrow: This function does not throw any
//            exceptions.

void Strings::add(string const &next)
{
  string **tmp = 0;
  size_t oldCapacity = d_capacity;
  try
  {
    tmp = storageArea();

    tmp[d_size] = new string(next);
  }
  catch (bad_alloc &ba)
  {
    delete[] tmp;
    d_capacity = oldCapacity;
    cerr << "(Strings) Unable to add string:"
      << " \"memory allocation failed\".\n";
    return;
  }

  // destroy old memory if new storageArea
  if (tmp != d_str) // was allocated
  {
    // destroy the old string * array
    destroy();
    d_str = tmp;
  }

  ++d_size;
}
```

Listing 5: destroy2.cc

```
#include "strings.ih"

void Strings::destroy(size_t start, size_t end)
{
  for (size_t index = start; index != end; ++index)
    delete d_str[index];
    delete[] d_str;
}
```

Listing 6: enlarged.cc

```
#include "strings.ih"

// Basic: Nothing is allocated if rawPointers fails.
// Strong: Nothing is changed if the exception occurs.

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 7: 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.
// Nothrow: This function throws no exceptions.

void Strings::reserve(size_t nextCapacity)
{
  if (d_capacity < nextCapacity)
  {
    size_t oldCapacity = d_capacity;
    while (d_capacity < nextCapacity)
      d_capacity <<= 1;

    try
    {
      d_str = enlarged();
    }
    catch (bad_alloc &ba)
    {
      d_capacity = oldCapacity;
      cerr << "(Strings) Unable to increase size:"
        << " \"memory allocation failed\".\n";
    }
  }
}
```

Listing 8: resize.cc

```
#include "strings.ih"

// Basic: If reserve fails then nothing is changed
//          so there are no leaks.
//         If the initializing of an empty string fails
//          then all the new empty strings will be
//          deleted along with the resized allocation.

// Strong: If reserve fails then the requested
//           enlarging can not be performed
//           and so resize stops.
//          If the initializing of the empty strings
//           fails then the whole class is rolled back.

// Nothrow: This function throws no exceptions.

void Strings::resize(size_t newSize)
{
  string **oldStr = d_str;
  size_t oldSize = d_size;
  size_t oldCapacity = d_capacity;

  // make sure there's enough memory
  reserve(newSize);
  if (d_capacity < newSize)
  {
    cerr << "(Strings) Could not resize:"
      << " \"reserve failed\".";
    return;
  }

  // enlarging? initialize new strings
  if (d_size > newSize)
  {
    try
    {
      for (; d_size != newSize; ++d_size)
        d_str[d_size] = new string;
    }
    catch (bad_alloc &ba)
    {
      destroy(oldSize, d_size);
      d_str = oldStr;
      d_capacity = oldCapacity;
      d_size = oldSize;
      cerr << "(Strings) Unable to increase size:"
        << " \"Memory allocation failed.\"\n";
    }
  }
  // shrinking? remove excess strings
  else if (newSize < d_size)
  {
    for (; d_size-- != newSize; )
      delete d_str[d_size];
  }
}
```

Listing 9: storagearea.cc

```
#include "strings.ih"

// Basic: This function does not cause any leaks
// Strong: This function rolls back the capacity
//            if enlarged fails.

string **Strings::storageArea()
{
  // enough room?
  if (d_size + 1 < d_capacity)
    // return the current memory block
    return d_str;

  try
  {
    // double the capacity
    d_capacity <<= 1;
    // return ptr to the enlarged space
    return enlarged();
  }
  catch(bad_alloc &ba)
  {
    d_capacity >>= 1;
    throw;
  }
}
```

Listing 10: strings1.cc

```
#include "strings.ih"

// If the construction fails none of the
// exception guarantees are applicable.
// The caller of the constructor will
// want to know that it failed.

Strings::Strings()
try :
  d_str(rawPointers(1))
{}
catch (bad_alloc &ba)
{
  cerr << "(Strings) Unable to create strings object:"
    << " \"memory allocation failed\".\n";
  throw;
}
```

Listing 11: strings2.cc

```
#include "strings.ih"

// If the construction fails none of the
// exception guarantees are applicable.
// The caller of the constructor will
// want to know that it failed.

Strings::Strings(int argc, char *argv[])
try :
  Strings()
{
  for (size_t begin = 0, end = argc; begin != end;
      ++begin)
    add(argv[begin]);
}
catch (bad_alloc &ba)
{
  throw;
}
```

Listing 12: strings3.cc

```
#include "strings.ih"

// If the construction fails none of the
// exception guarantees are applicable.
// The caller of the constructor will
// want to know that it failed.

Strings::Strings(char **environLike)
try :
  Strings()
{
  while (*environLike)
    add(*environLike++);
}
catch (bad_alloc &ba)
{
  throw;
}
```