Lecture 7 - Exercises

Introduction

Exercise 1 - Ensuring garbage collection on a dynamically allocated std::string

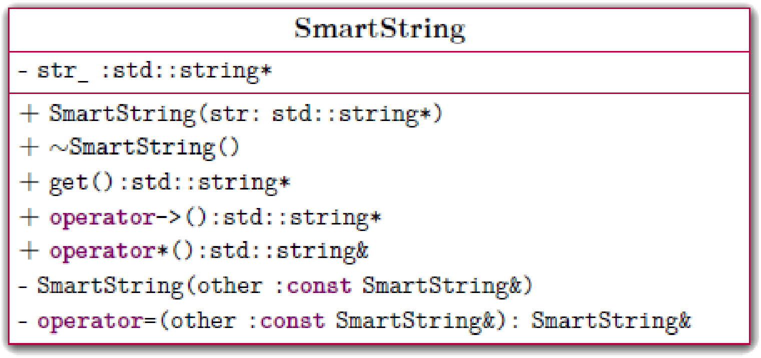

SmartString
- str_ :std::string*
+ SmartString(str: std::string*)
+ ~SmartString()
+ get():std::string*
+ operator->():std::string*
+ operator*():std::string&
- SmartString(other :const SmartString&)
- operator=(other :const SmartString&): SmartString&

Figure 1.2 from the exercise description

From the given UML class (above), we've created a header file and an implementation. The full code can be found in the collapses under this.

Implementation for SmartString.hppImplementation for SmartString.hpp

```
#ifndef SMARTSTRING_H
#define SMARTSTRING_H

#include <string>
#include <iostream>
using namespace std;

class SmartString
{
public:
    SmartString(string* str);
    string* get();
    string* operator->();
    string& operator*();
    ~SmartString();

private:
    string* str_;
    SmartString(const SmartString& other);
    SmartString& operator=(const SmartString other);
};

#endif
```

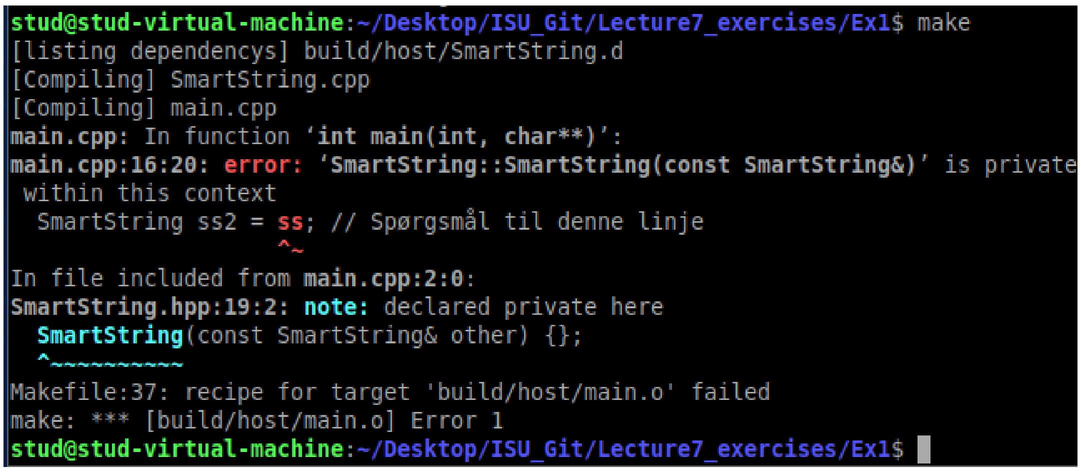
Implementation for SmartString.cppImplementation for SmartString.cpp

```
#include "SmartString.hpp"

using namespace std;
```

```
SmartString::SmartString(string* str) : str_(str) {}

string* SmartString::get()
{
    return str_;
}

string* SmartString::operator->()
{
    return str_;
}

string& SmartString::operator*()
{
    return *str_;
}

SmartString::~SmartString()
{
    delete str_;
}
```

Test

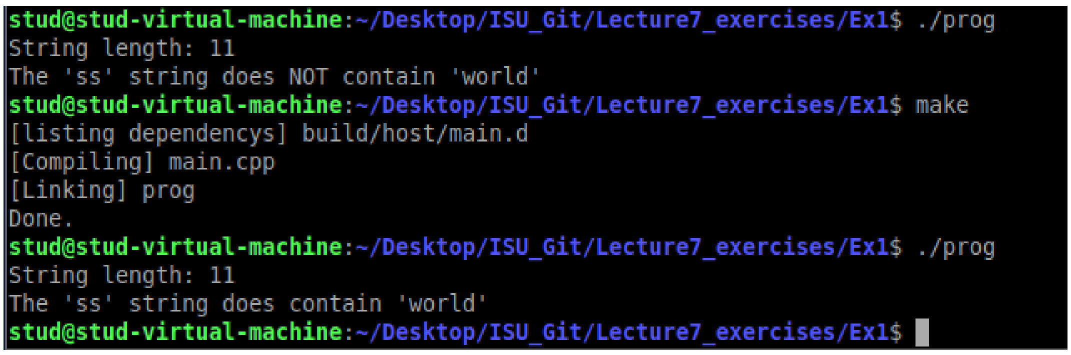
To test the program, the given main.cpp (Listing 1.1) was used. However, since the copy constructor wasn't implemented in this exercise, the terminal shows the following input:



```
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture7_exercises/Ex1$ make
[listing dependencys] build/host/SmartString.d
[Compiling] SmartString.cpp
[Compiling] main.cpp
main.cpp: In function ‘int main(int, char**)’:
main.cpp:16:20: error: ‘SmartString::SmartString(const SmartString&)’ is private
 within this context
  SmartString ss2 = ss; // Spørgsmål til denne linje
                    ^~
In file included from main.cpp:2:0:
SmartString.hpp:19:2: note: declared private here
  SmartString(const SmartString& other) {};
  ^~~~~~~~~~~
Makefile:37: recipe for target 'build/host/main.o' failed
make: *** [build/host/main.o] Error 1
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture7_exercises/Ex1$
```

The terminal output with SmartString ss2 = ss in the main.cpp

If we however comment the copy constructor SmartString ss2 = ss, we'll instead get the following output:



```
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture7_exercises/Ex1$ ./prog
String length: 11
The 'ss' string does NOT contain 'world'
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture7_exercises/Ex1$ make
[listing dependencys] build/host/main.d
[Compiling] main.cpp
[Linking] prog
Done.
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture7_exercises/Ex1$ ./prog
String length: 11
The 'ss' string does contain 'world'
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture7_exercises/Ex1$
```

First tested for a string without the word "world", followed by the one that does contain it.

Questions

Why must the copy constructor and assignment operator be private with no implementation? -And what is the consequence when these are private?

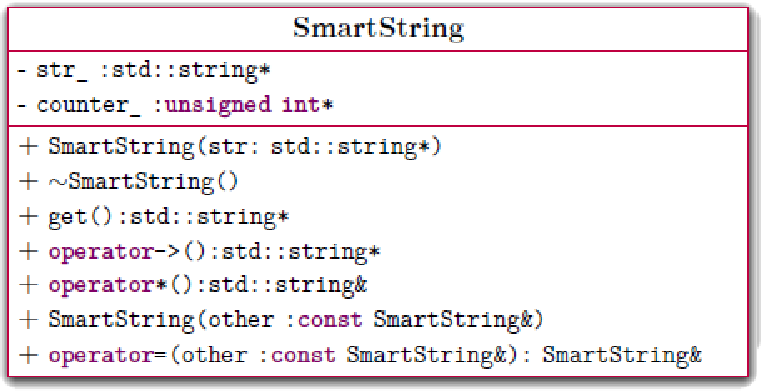
The copy constructor and assignment operator has to be private to guarantee the users of the program won't be able to use them. If the users were able to use them, one couldn't assure a full clean-up.

What exactly does the operator->() do?

The operator -> () returns the object to which the pointer is pointing. In the main.cpp we use it to get the length of the ss string.

Exercise 2 - The Counted Pointer

In this exercise we're given a similar UML as in exercise 1, but with minor changes.



```
SmartString
- str_ :std::string*
- counter_ :unsigned int*
+ SmartString(str: std::string*)
+ ~SmartString()
+ get():std::string*
+ operator->():std::string*
+ operator*():std::string&
+ SmartString(other :const SmartString&)
+ operator=(other :const SmartString&): SmartString&
```

Figure 2.1 from the hand out.

We used the .hpp and .cpp files from exercise 1, but made a few alterations.

Implementation for SmartString.hppImplementation for SmartString.hpp

```
#ifndef SMARTSTRING_H
#define SMARTSTRING_H

#include <string>
#include <iostream>
using namespace std;

class SmartString
{
public:
    SmartString(string* str);
    string* get();
    string* operator->();
    string& operator*();
    SmartString(const SmartString& other);
    SmartString& operator=(const SmartString other);
    ~SmartString();

    unsigned int* getCount(); // Added for testing purpose. Described in "Test"

private:
    string* str_;
    unsigned int* counter_;
};

#endif
```

Implementation for SmartString.cppImplementation for SmartString.cpp

```
#include "SmartString.hpp"

using namespace std;

SmartString::SmartString(string* str) : str_(str), counter_(new unsigned int)
{
    (*counter_) = 1;
}

string* SmartString::get()
{
    return str_;
}

string* SmartString::operator->()
{
    return str_;
}

string& SmartString::operator*()
{
    return *str_;
}

SmartString::SmartString(const SmartString& other) : str_(other.str_), counter_(other.counter_)
{
    (*counter_)++;
}

SmartString& SmartString::operator=(const SmartString other)
{
    if (this != &other)
    {
        (*counter_)--;
        counter_ = other.counter_;
        str_ = other.str_;
        (*counter_)++;
    }

}

SmartString::~SmartString()
{
    (*counter_)--;

    if ((*counter_) == 0)
    {
        delete str_;
        delete counter_;
    }
}

    //Added getCount for visual testing purpose
unsigned int *SmartString::getCount()
{
    return counter_;
}
```

Test

For our test, we've created a getCount() function which returns the counter_. This makes it possible to give a more visual test output in the terminal.

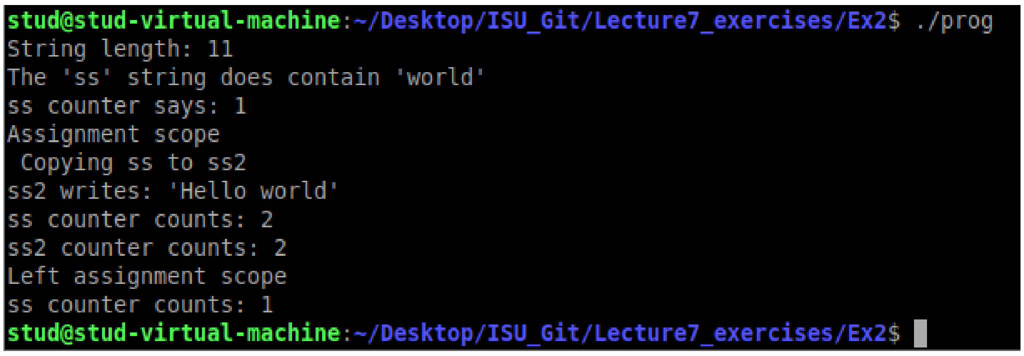
The implementation for getCount():

```
unsigned int *SmartString::getCount()
{
    return counter_;
}
```

The test program itself is created

main.cpp for exercise 2main.cpp for exercise 2

```
#include <iostream>
#include "SmartString.hpp"

using namespace std;

int main(int argc, char* argv[])
{
    SmartString ss(new string("Hello world"));
    cout << "String length: " << ss->length() << endl;

    if (ss->find("world") != string::npos)
        cout << "The 'ss' string does contain 'world'" << endl;
    else
        cout << "The 'ss' string does NOT contain 'world'" << endl;

    cout << "ss counter says: " << *ss.getCount() << endl;

    {
            cout << "Assignment scope \n Copying ss to ss2" << endl;
            SmartString ss2 = ss; // Copy constructor

            cout << "ss2 writes: '" << *ss2 << "'" <<endl;
            cout << "ss counter counts: " << *ss.getCount() << endl;
            cout << "ss2 counter counts: " << *ss2.getCount() << endl;
    }

    cout << "Left assignment scope" << endl;
    cout << "ss counter counts: " << *ss.getCount() << endl;

}
```



```
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture7_exercises/Ex2$ ./prog
String length: 11
The 'ss' string does contain 'world'
ss counter says: 1
Assignment scope
 Copying ss to ss2
ss2 writes: 'Hello world'
ss counter counts: 2
ss2 counter counts: 2
Left assignment scope
ss counter counts: 1
stud@stud-virtual-machine:~/Desktop/ISU_Git/Lecture7_exercises/Ex2$ 
```

The terminal output after running the test

Questions

Why must the counter be dynamically allocated? Where should this happen?

The allocation happens in the constructor. This is necessary since for every SmartString object a new counter is created as well.

How is the *copy constructor* to be implemented?

See implementation above in the code

How is the *assignment operator* to be implemented?

See implementation above in the code

What happens in the *destructor* and how should it be implemented?

The destructor decreases the counter until it's equal to 0. Then it enters the if statement and deletes counter_ and str_. The counter is there to make sure that no one is using them.

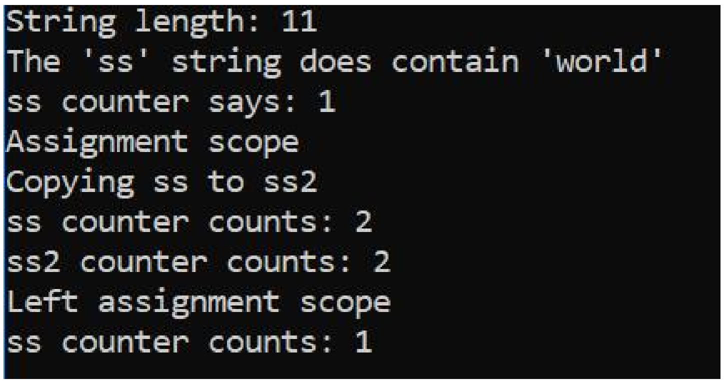
Exercise 4 - Discarding our solution in favor of boost::shared_ptr<>

Exercise 4.1 - Using boost:shared_ptr<>

In this example we have replaced our home made libary "SmartString" with the standard std::shared_ptr, which has the same basic functionality as the "boosted/shared_ptr".
We again see that after we leave the assignment scope of where in the "ss2" shared pointer is created, it is destroyed, and the "used_counter" goes from 2 down to 1: this is due to the destructor de-allocating the used memory.

Shared_ptr testShared_ptr test

```
#include "pch.h"
#include <iostream>

using namespace std;
int main(int argc, char* argv[])
{
    shared_ptr<string> ss(new string("Hello world"));
    cout << "String length: " << ss->length() << endl;

    if (ss->find("world") != string::npos)
        cout << "The 'ss' string does contain 'world'" << endl;
    else
        cout << "The 'ss' string does NOT contain 'world'" << endl;

    cout << "ss counter says: " << ss.use_count() << endl;
    {
        cout << "Assignment scope \nCopying ss to ss2" << endl;
        shared_ptr<string> ss2 = ss; // Copy constructor

        cout << "ss counter counts: " << ss.use_count() << endl;
        cout << "ss2 counter counts: " << ss2.use_count() << endl;
    }

    cout << "Left assignment scope" << endl;
    cout << "ss counter counts: " << ss.use_count() << endl;
}
```



```
String length: 11
The 'ss' string does contain 'world'
ss counter says: 1
Assignment scope
Copying ss to ss2
ss counter counts: 2
ss2 counter counts: 2
Left assignment scope
ss counter counts: 1
```

Exercise 5 - Resource Handling

What do you consider a *resource?*

A resource could be anything that aides our program to function: Disk space, memory and more. In our case, we are mostly concerned with our use of RAM.

In which situations do you foresee challenges with resources and how could they be handled?

If we have a program that is very big, we cannot for resource purposes allocate it all on the stack. This is due to the stacks very limited size, compared to the heap. And while the heap is limited in size, it is still much bigger and is only limited by virtual memory.

In particular regarding memory, when would you be able to eliminate the need for allocations? Or when is it a *must* that something is allocated on the heap?

If we have a function that allocates memory on the stack; and that memory is never used after that, we should allocate on the stack. It's faster and is handled by the CPU. But if we for example have a variable that we need to use later on, we can allocate it on the heap, since all threads share one heap. Also if we later need to resize the variable, this can only be done on heap allocated variables.

Filer

opg1_test.png	37,4 KB	2018-11-20	Mie Nielsen
ss_fejl.png	39,3 KB	2018-11-20	Mie Nielsen
opg2_test.png	20,1 KB	2018-11-20	Mie Nielsen
UML1.png	21,1 KB	2018-11-20	Mie Nielsen
UML2.png	22,7 KB	2018-11-20	Mie Nielsen
consoloutput.png	9,15 KB	2018-11-20	Martin Lundberg