Exercises for the Lecture Series

"Object-Oriented Programming for Scientific Computing"

Dr. Ole Klein
ole.klein@iwr.uni-heidelberg.de

To be handed in on 23. 05. 2018 before the lecture

Exercise 1 Linked List (const Correctness)

In the last exercise you programmed a linked list to practice the interaction of constructors, destructors and pointers. We now want to examine / extend this implementation using the concept of `const` and clean encapsulation.

1. The **`const`** keyword has been addded to the class `List` in the places where it is appropriate. Answer the following questions using this class as an example:
   - Which methods should or should not be **`const`**?
   - What is a good choice for the parameters and return values?
2. Add a **`const`** method `max` that returns the maximum of the stored values, as long as the list is non-empty. For efficiency, this value should be cached and only recomputed when it it requested and has become invalid in the meantime.
   - What effect does the keyword **`mutable`** have and why is it needed here?
   - What would be inefficient about updating the stored value every time the list is modified in some way?
3. A function for printing the list could look as follows:

   ```
   void printList (const List& list)
   {
       for (const Node* n = list.first(); n != 0; n = list.next(n))
           std::cout << n->value << std::endl;
   }
   ```

   Test your implementation with this function.
4. Write a free function

   ```
   void append (List& list1, const List& list2);
   ```

   which copies all the entries from `list2` and appends them to `list1`.

6 Points

Exercise 2 unique_ptr, shared_ptr and weak_ptr

After you have studied smart pointers in class, you reexamine your previous programs. You find a number of tasks that deal with linked lists. Since it would be very convenient if the list would carry out its memory management itself, you decide to modify your implementation after checking that the additional memory requirements of shared pointers are not an issue in your specific use cases. (Unique pointers would actually be more appropriate here, but would increase your workload for the third subtask.)

1. Convert the class of the previous exercise to smart pointers, i.e. swap all instances of `Node*` with `shared_ptr<Node>`.
2. What exactly happens when the list is deleted? In what order are the destructors called, and how is the memory released?
3. Please modify your implementation again to obtain a doubly linked list: each element should also point to its predecessor. What is problematic here? Can you avoid the problem using `weak_ptr<Node>`?

Your solution should be an answer to all posed problems at once, i.e. a doubly linked list that doesn't utilize raw pointers.

6 Points

Exercise 3 Pointer Puzzle

You may actually try this with your compiler.

Look at the following program:

```
void foo ( const int** );

int main()
{
    int** v = new int* [10];
    foo(v);

    return 0;
}
```

The compiler will exit with an error message, because you make a `const int**` out of the `int**`:

```
g++ test.cc -o test
test.cc: In function 'int main()':
test.cc:6: error: invalid conversion from
  'int**' to 'const int**'
test.cc:6: error:   initializing argument 1
  of 'void foo(const int**)'
```

- Actually, shouldn't it always be possible to convert from non-`const` to `const` ...
- Why doesn't this rule apply here?

Tip:

It's clear why the following program doesn't compile:

```
const int* bar ();

int main()
{
    int** v = new int* [10];
    v[0] = bar();

    return 0;
}
```

What is the relation between this program and the one above?

6 Points

Exercise 4 Constness

Here is a list of function prototypes, some variables and some assignments. Which expressions aren't allowed and why?

```
int foo ( const int& );
int bar ( int& );

int main()
{
  int i = 0;
  int& j = i;
  static const int f = i;
  int* const p = 0;
  p = &i;
  *p = f;
  const int& l = j;
  const int& k = f;
  foo ( j );
  bar ( l );
  foo ( k );
}
```

2 Points