Exercises for the Lecture Series

"Object-Oriented Programming for Scientific Computing"

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

To be handed in on 05. 05. 2015 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 extend this implementation using the concept of `const` and clean encapsulation.

1. Until now, the class `Node` only had `public` members, but to prevent accidental modification the `Node* next` pointer should be `private`.
   - Change the implementation accordingly so that `Node* next` becomes `private`.
   - How can you now grant the class `List` access to the `private next` pointer?
2. Add the `const` keyword to the class `List` in the places where it is appropriate.
   - Which methods should or should not be `const`?
   - What is a good choice for the parameters and return values?
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 SHARED_PTR UND 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 useful if the list would carry out its memory management itself, you decide to modify your implementation.

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, it should always be possible to convert from non-`const` to `const` ...
- Why doesn't this 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