Course Programming in C^{++}

Exercise List 5

Deadline: 08.04.2017

This is the last exercise about basic object building. We will implement trees using pointers. Using pointers gives rise to the question what should be done when trees are copied or assigned. Copying the complete tree is expensive, and in most cases unnecessary. It is much more efficient to copy only the pointer, because this can be done in constant time. Unfortunately, if one does this, one obtains pointers that share an object, and which have equal ownership. When a pointer goes out of scope, we don't know if we can destroy the object, because there may other pointers that still use it. Since C^{++} does not have *garbage collection*, we have to solve this problem by ourselves. It can be solved with *reference counting*: To every tree node, we add a counter of type **size_t** that counts how many pointers point to the node. When we perform a lazy copy (only copying the pointer), we increase the reference counter of the node whose pointer is being copied. In a destructor, we decrease the reference counter by one. When the reference counter becomes zero, we destroy the node.

1. Download the files `tree.h, tree.cpp, main.cpp, Makefile` from the course homepage. File `tree.h` contains two class definitions. `struct trnode` is used only internally by `tree`, and it is finished, so you don't need (and are not allowed) to add methods to it.

   Add your own `string` class of List 4 to the project.

2. Write the copy constructor, copying assignment, R-value assignment, and the destructor of `tree`. None of these operators is complicated.

   The copy constructor should copy the pointer, and increase the reference pointer in the `trnode` that the pointer points to. The destructor should decrease the reference counter. If it becomes zero, it should `delete` the `trnode`. There is no need to do anything more, because `delete` automatically calls the destructors of the subtrees.

   R-value assignment can be implemented by a simple exchange. The other assignment can be defined through R-value assignment.

   There is no need to define an R-value copy constructor because it cannot be made more efficient than the standard copy-constructor.

   You can also define copy assignment directly, but it is harder, because you have to consider self and subtree assignment.

3. Next, you can implement

```
const string& functor( ) const;
const tree& operator[] ( size_t i ) const;
size_t nrsubtrees( ) const;
```

   `operator[]` should not touch reference counters, because it returns only a reference, and references don't own.

4. At this point, it is easy to implement `std::ostream& operator << ( std::ostream& , const tree& )`. There is no need to make it a friend, because you can use the methods of 3.

5. We also want to implement non-const access methods

```
string& functor( );
tree& operator[] ( size_t i );
```

   We have to be very careful because of possible sharing. If we write `tree t1 = t2; t2. functor( ) = "hallo";`, then also `t1` will change, if we are not careful.

   The solution is to implement a method `ensure_not_shared( )`, that ensures that the `trnode` that we are using, is used only by us. If its reference counter equals one, it does nothing. Otherwise, it needs to make a copy.

   Once we have `ensure_not_shared( )`, implementation of `functor()` and `operator[] ( size_t i )` is easy.

6. Implement a function:

```
   tree subst( const tree& t,
               const string& var, const tree& val );
```

   It returns the tree that is obtained when every occurrence of `var`, that does not have subtrees, is replaced by `val`. Function `subst` should be not a member of `tree`.

7. The solution in task 5 (for allowing non-const access) is very unsatisfactory. If one writes for example:

```
tree t1 = ... ;
tree t2 = t1;
std::cout << t2. functor( ) << "\n";
```

   the compiler will use the non-const version of `functor( )`, which means that `t2` (the top node) will be copied without reason.

   Even worse, function **subst** will always copy the complete tree, even when no replacements are made.

   In order to see the effect, add a method

```
size_t getaddress( ) const
{
   return reinterpret_cast< size_t > ( pntr );
}
```

In order to solve the problem, implement methods:

```
   replacesubtree( size_t i, tree t );
      // Replace i-th subtree.
   replacefunctor( const string& f );
      // Replace the functor.
```

The functions must call `ensure_not_shared( )` only when it is certain that the tree will be changed.

8. Rewrite function **subst**, using **replacefunctor** and **replacesubtree**.

   Verify, by means of **getaddress** that the new version of **subst** does not make any unnecessary copies. (Only nodes that are on the path to a replaced functor should be copied.)

   Now, we finally have an implementation that reaches the original goal of avoiding deep copies as much as possible.

9. Check, using `valgrind`, and some code in which every function/method that you wrote is used, that there are no memory problems. Make sure that both R-value and copying assignment are used in your code, and that there is some sharing in it.