Course C^{++}

Exercise List 5

Date: 03.04.2014

This is the last exercise about basic data structure building. Trees are always implemented by pointer structures. This gives rise to the question what should be done when trees are assigned. Copying the tree is costly, because it requires a full tree traversal. It is much nicer to copy only the pointer, because this can be done in constant time. Unfortunately, this results in two pointers pointing at the same tree, and we don't know when the tree can be cleaned up. Some languages (Java) have built-in *garbage collection*, which means that the run time environment detects automatically when structures in heap memory are not reachable anymore from the program, and cleans them up automatically. C^{++} doesn't have this, so we need to solve this problem by ourselves. The solution is to use *reference counting*. To every tree node, we add an unsigned integer that counts how often the node is used. When we do a lazy copy (copying only a pointer), we increase the reference counter of the node that is being copied. In a destructor, we decrease the reference counter by one, until it becomes zero. Only when its reference counter becomes zero, the node is really destroyed.

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.

   The user should use only `class tree`.

2. Write the copy constructor, copying assignment, Rvalue 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 two.

   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 the compiler will automatically call the destructors of the subtrees.

   Rvalue assignment can be a simple exchange. The other assignment can be defined through Rvalue assignment.

   All these functions are simple enough to define in the class, so that they will be inlined.

3. Next, you can implement

```
const std::string& functor( ) const;
const tree& operator [] ( unsigned int i ) const;
unsigned int nrsubtrees( ) const;
```

`operator []` should not touch reference counters, because it returns only a reference, not a full copy that would be able to keep a tree alive.

4. At this point, it should be easy to implement `std::ostream& operator << ( std::ostream& , const tree& )`, using the methods of the previous task. There is no need to make it a friend.

5. Now, we would also want to implement the non-const methods

   ```
   std::string& functor( );
   tree& operator[] ( unsigned int );
   ```

   We have to be very careful because of possible sharing. If we write `tree t1 = t2; t1. functor( ) = "hallo";`, then also `t2` 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. Don't forget to decrease the reference counter in the other `trnode`!

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

6. At this point, we have a complete implementation of **tree**, and we can make what we want, for example a differentiation program. But I think it is enough for today, so let's stop with a simple substitution function:

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

   It returns the tree that is obtained when every occurrence of `var`, without subtrees, is replaced by `val`. Function subst is not a member function of **tree**.

7. Check, using the `top` command and some loop in which every method is used, that there are no memory leaks. Check carefully. Make sure that `ensure_not_shared` is used. Make sure that Rvalue assignment is used.

   ```
   tree a = tree( "a" );
   tree b = tree( "b" );
   tree c = tree( "c" );
   tree d = tree( "f", { a, b, c } );
      // {a,b,c} will be converted into an std::vector,
   ```

```
   // since the constructor requires a reference, it will
   // be a temporary, and hence an Rvalue.

// You can also write:
std::vector< tree > arguments = { a, b, c };
std::cout << tree( "f", std::move( arguments )) << "\n";
std::cout << arguments. size( ) << "\n";
   // Will be 0.
```

In the next week, I want to do a little more with `tree`, so implement it carefully.