Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2015-11-30

Iterating a template list

Aim

In this laboration you will learn how to create templates and overload operators. This way you can make your class cooperate with the C++ language and STL.

Reading instructions

- Templates
- Operator overloading
  - Comparison operator `!=`
  - Preincrement operator `++`
  - Dereference (content) operator `*`
  - Stream output operator `<<`

Assignment: Iterating a template list

In this laboration you will extent the given code for a list. The list is similar to the list you created in a previous laboration. You will however notice the absence of some features, most notably it is missing all ways to access inserted elements.

In your previous list you may have solved this with a member function to retrieve a certain position in the list. This work well when you need access to one specific element. But using it to step through (iterate) the entire list is not efficient, since you will start over from the beginning in each call. It would be far more efficient if you could remember the current position to avoid restarting.

Another drawback of your previous solution is its lack of flexibility. It is coded to store one type of elements only. In reality you may want to use it with several different types in the same program. In this laboration you will solve both problems by first making the given list class a template, and then make it iterable.

Implement a class to keep track of the current position in the list. This class should be an inner class of the list, similar to the Link class in the given code. It should be possible to call the member function `List::begin` to get an instance of your class referring to the beginning of the list. Similar, `List::end` should give an instance of your class representing when you have reached (just past) the end of the list. The only way to get an initial and valid reference to one position in the list should be by calling `List::begin`. This should then be incrementable to get to subsequent positions, and comparable to determine when you reached the end.

Requirement: Your final code should compile and work for the following example code. Note that auto here may be substituted for your inner class.

```
#include "list.h"

List<int> list;
// The list is filled with data here...
// ...
for ( auto it = list.begin(); it != list.end(); ++it)
{
  cout << *it << endl;
}

List<std::string> word_list;
// The list is filled with data here...
// ...
// Once the above work, try this
for ( auto s : word_list )
{
  cout << s << endl;
}

// Printing entire list (as above but shorter...)
cout << list << endl;
cout << word_list << endl;
```

Code example 1: Step through a list

Hint: Template code can not be compiled until instantiated. Thus you should *include* it rather than compile it. A neat trick is to change the extension from `.cc` to `.tcc` and include this file just before the closing guard in the corresponding header file.

Hint: A private constructor will prevent any code but friends from creating instances of a class.

Hint: Read the sections about *Template friend operators*[1] and *The typename disambiguator*[2].

[1] `http://en.cppreference.com/w/cpp/language/friend`
[2] `http://en.cppreference.com/w/cpp/language/dependent_name`