# PIC 10B Section 1 - Homework # 9 (due Friday, June 7, by 6 pm)

**You should upload only a single .h file and submit it to CCLE before the due date/time! Your work will otherwise not be considered for grading. Do not submit a zipped up folder or any other type of file besides .h.**
**Be sure you upload the file with the precise name requested of you and that it matches the name you used in your editor environment otherwise there may be linker and other errors when your homeworks are compiled on a different machine. Also be sure your code compiles and runs on Visual Studio 2017.**

At the end of this work you should submit a header file **Cache.h** with a templated class **cache** and nested class **iterator**, and a templated free-function **print**, i.e., not a member function, that can print cache objects! The header file should include all relevant definitions for the program to run.

**Do not submit a main routine!**

## A CACHE CLASS: "SHORT TERM MEMORY"

You will have a lot of freedom in this assignment in your design. You could manage your own heap memory directly (more painful, but good practice), use different container classes (also good practice so you get familiar with the C++ Standard Library), etc. The main skills being learned in this assignment are work with templates.

This **cache** class should be templated based on the data type it stores, the number of items it can "remember", i.e., store, and a means of comparing two elements. The idea is that it can be told to store items, but if it needs to remember something and it is already full then it will have to "forget" the oldest item. It can be told to remove items, in which case the oldest item will be removed.

There is one caveat here: it will store the **high** and **low** values of whatever data it stores. So even if those values are forgotten by the short term memory, they may still be remembered long term.

The class has **iterator**s that can go from the most recent value back as far as there are values stored.

For example, suppose the class can store **3 int** values and we sort them based on ">" so that 4 is considered higher than 7 (instead of sorting based on "<" where 4 is indeed lower than 7).

After inserting the items 2, 4, 6, 10, 8, it only stores the numbers 8, 10, 6 (from most recent to most distant - only 3 numbers stored); in addition the high value would be 2 and the low value would be 10. *If you are confused, read the instructions again about the data structure having its own comparison system.*

Here are the specific requirements; everything should be in the **pic10b** namespace:

**cache** should be a templated class with...

- three template parameters: a data type **T**, a number of elements it can store **N**, and a means of defining ordering **ord** that is by default **std::less<T>**;
- a default constructor that can also accept a value of type **ord** and set the member comparison with that;
- a **destructor** (unless the compiler generated destructor works);
- a **copy constructor** (unless the default copy mechanism generated by the compiler works);
- a **move constructor** (unless the default move mechanism generated by the compiler works);
- **assignment operators** (unless the default assignments generated by the compiler work);
- an **insert** member function, **overloaded by whether its input is an lvalue or rvalue**, that accepts something of type **T** and adds it, possibly at the cost of removing the oldest stored object;
- a templated **emplace** function that behaves as **insert** but can accept an arbitrary number of arguments of arbitrary type to construct an object and add it;
- a **pop** member function that removes the oldest element;
- a **begin** member function returning a **const_iterator** to the most recent element;
- an **end** member function returning a **const_iterator** beyond the most distant element (the usual past-the-end idea);

- a **size** member function to count the number of elements (not counting the long-term storage);
- a **get_low** member function to retrieve the lowest value; and
- a **get_high** member function to retrieve the highest value.

You should also throw exceptions with appropriate messages should the user do something logically incorrect like calling **pop** when there are no memories, etc.

The **const_iterator** class should...

- allow for **prefix** ++ and **postfix** ++;
- include the **dereferencing** operator (without allowing element mutations); and
- include the **arrow operator** (without allowing element mutations).

There should be a **print** that prints a **cache** object of any type, printing all the elements separated by spaces to the console. *This function should not be variadic: it should accept a single memory object and print all of its elements. Period.*

You may assume that the objects **T** have default constructors (but if you worked more directly with memory like with allocators this would not be necessary).

An example of the desired output for the code below is provided.

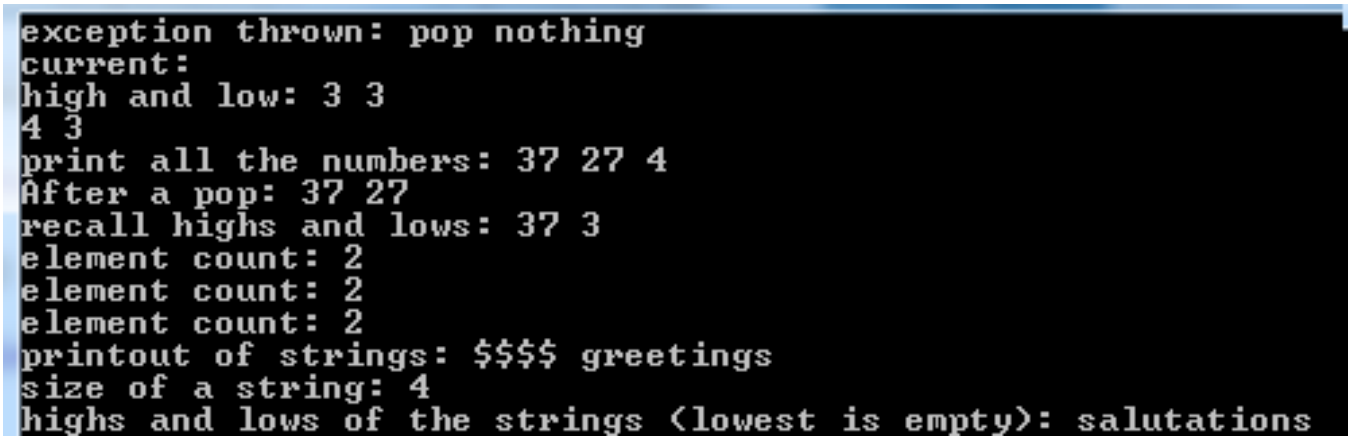
```
#include "Cache.h"
#include<iostream>
#include<string>

bool length_comp(const std::string& left, const std::string& right) {
  return left.size() < right.size();
}

int main() {

  pic10b::cache<int, 3> intCache; // can store up to 3 values of int plus high and low
  try { // can we pop with nothing there?
    intCache.pop();
  }
  catch (const std::exception& e) { // apparently not...
    std::cerr << "exception thrown: " << e.what() << '\n';
```

```
}

std::cout << "current: ";
for (int i : intCache) { // empty so does nothing
  std::cout << i << " ";
}
std::cout << '\n';

intCache.insert(3); // insert 3 and look for high and low
std::cout << "high and low: " << intCache.get_high() <<
  " " << intCache.get_low() << '\n';
intCache.insert(4); // insert 4

for (int i : intCache) { // print all the numbers
std::cout << i << " ";
}
std::cout << '\n';

// insert two more numbers, will pop the 3
intCache.insert(27);
intCache.insert(37);

std::cout << "print all the numbers: ";
for (auto itr = intCache.begin(),
  past_end = intCache.end(); itr != past_end; ++itr) {
  std::cout << *itr << " ";
}
std::cout << '\n';

// pop the most distant (4)
intCache.pop();
std::cout << "After a pop: ";
for (auto itr = intCache.begin(), past_end = intCache.end();
  itr != past_end; ++itr) { // print all the numbers
    std::cout << *itr << " ";
}
std::cout << '\n';
std::cout << "recall highs and lows: " << intCache.get_high() <<
  " " << intCache.get_low() << '\n';

auto intCache2 = intCache; // copy constructor
std::cout << "element count: " << intCache2.size() << '\n';
auto intCache3 = std::move(intCache); // move constructor
std::cout << "element count: " << intCache3.size() << '\n';
```

```
  intCache = intCache3; // assignment operator
  std::cout << "element count: " << intCache.size() << '\n';

  pic10b::cache<std::string, 2, bool(*)(const std::string&, const std::string&)>
    stringCache(length_comp);

  stringCache.emplace(); // emtpy string added
  stringCache.insert("hey");
  stringCache.insert("howdy");
  stringCache.insert("salutations");
  stringCache.insert("greetings");
  stringCache.pop(); // now only stores "greetings"
  stringCache.emplace(4, '$'); // also stores "$$$$"

  std::cout << "printout of strings: ";
  print(stringCache); // prints the cache
  std::cout << '\n';

  std::cout << "size of a string: ";

  // in a world if auto didn't exist...
  pic10b::cache<std::string, 2,
    bool(*)(const std::string&, const std::string&)>::const_iterator its =
    stringCache.begin();
  std::cout << its->size() << '\n'; // can use arrow operator!

  std::cout << "highs and lows of the strings (lowest is empty): "
    << stringCache.get_high() << " " << stringCache.get_low() << '\n';

  std::cin.get();

  return 0;
}
```

**Some initial guidance to get you on your way...**

1. Write the class in a more concrete ordering first: just use **operator**< instead of a more abstract operator. Then worry later about that third template argument with a default value. You'll only need to change a few lines of code once you've got this first part working.
2. If you use the C++ Standard data structures, you should not have to write a destructor, a copy constructor, a move constructor, or even assignment operators!




3. You might not need to write a separate iterator class:

4. Be sure to draw diagrams! No matter what implementation you take, you'll want nice diagrams!

5. Be very careful if a **size_t** value is 0 and you decrease its value... you want to avoid that happening; if your logic requires something like that, treat the 0 case separately.

6. You can save yourself some grief and define everything within the namespace and everything within the first interface where it appears, i.e. fully define **cache::iterator** inside of the **cache** interface, don't define it outside, etc. You are still responsible to know how to define template nested classes/functions outside the interface, however.

7. Define the **operator!=** as a member within the **memory::iterator** class! When you work with templates, the compiler and linker will take any chance they can to attack you and you are less vulnerable to attack if things are tightly organized and kept together (even if the code itself looks worse).

8. You are allowed to make use of the C++ Standard containers if you want. Recall that many of those data structure are themselves built upon simpler data structures.