PIC 10C Section 1 - Homework # 1 (due Friday, April 1, by 11:59 pm)

You should upload each .cpp and/or .h file separately and submit them 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 and .cpp.

Be sure you upload files with the precise name that 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 2022.

TEMPLATED MAXIMUM HEAP

This homework is a review of basic templates and a chance to implement the maximum heap data structure. In the end, you will submit **max_heap.h** and "Honesty.txt" as described in the syllabus. Failure to provide the Honesty file will automatically result in a zero.

Recall that a **maximum heap** is a data structure that always has the "largest" value at the top. This value can be popped off. We shall allow the user to specify a custom means of sorting the values.

For this homework, the **only header files you are allowed to use** are:

- **utility**
- **vector**
- **stdexcept**

The main purpose of this homework is to review prerequisite material. A visual representation and explanation for how to implement a maximum heap is given in the Data Structures notes from PIC 10B; relevant template information appears in the Template notes from PIC 10B. The Timer header file can be found here:
https://github.com/3k1m/cpp/blob/master/timing/Timer.h.

Please refer to the syllabus for how the work will be graded: note that more than half of the marks come from good coding practices and code documentation.

Here are the requirements of the class:

The **max_heap** class must:

- be defined within the **pic10c** namespace;
- be templated by the type of data it stores, **T**, and a comparison operator, **compare_type**, which should be **std::less<T>** by default;
- store a **std::vector<T>** called **values** and a **compare_type** called **pred** as member variables;
- have a single constructor with a single parameter such that if no arguments are provided, **pred** will be set to compare_type{} and otherwise uses the value provided as the value for **pred**;
- have an **insert** function running in $O(\log n)$ time that, when passed an object of type **T** (be careful about lvalues and rvalues for efficiency), places the object into the data structure;
- have a **size** function to return the number of elements in the structure;
- have a **top** function that returns the "maximum" value from the heap (as defined by **pred**); and
- have a **pop** function running in $O(\log n)$ time that, when called, removes the "maximum" value from the heap (as defined by **pred**), throwing a **std::logic_error** if the heap is empty at the call.

Your code should be free of memory leaks. Here is a memory leak checker you can use:
https://github.com/3k1m/cpp/tree/master/leaks.

A test case, code and output, are provided below. Note that inserting N items will incur a cost of $O(\sum_{k=1}^{N} \log k) = O(N \log N)$. Since $\log N$ grows so slowly, the ratio of times between inserting and removing $N1$ and $N2$ items should be $\approx N1/N2$. Note how when the number of insertions/removals goes up by a factor of 100, the time only changes by about the same factor. The precise times it takes to run on your computer could vary but the actual scaling of the times (being close to proportional to the number of insertions/removals) should match up.

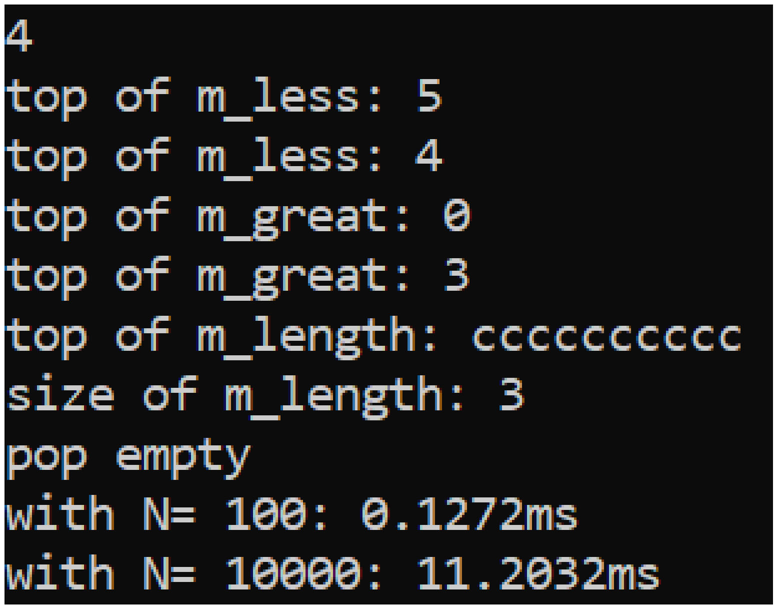
```
#include "max_heap.h"
#include "Timer.h"

#include <string>
```

```
#include <iostream>

int main() {
  // plain vanilla max heap, does things by <
  pic10c::max_heap<int> m_less;
  m_less.insert(3);
  m_less.insert(4);
  m_less.insert(5);
  m_less.insert(0);
  std::cout << m_less.size() << '\n';

  std::cout << "top of m_less: " << m_less.top() << '\n';
  m_less.pop();
  std::cout << "top of m_less: " << m_less.top() << '\n';

  // more fancy, replaces < with > so sorting is reversed
  pic10c::max_heap<int, std::greater<int>> m_great;
  m_great.insert(3);
  m_great.insert(4);
  m_great.insert(5);
  m_great.insert(0);

  std::cout << "top of m_great: " << m_great.top() << '\n';
  m_great.pop();
  std::cout << "top of m_great: " << m_great.top() << '\n';

  // lambda that will compare two strings by size
  // you'll need at least C++14 settings for this to compile!
  auto by_length = [](const auto& s1, const auto& s2) {
    return s1.size() < s2.size();
  };

  // this max heap will sort things by the length of the strings it stores
  pic10c::max_heap<std::string, decltype(by_length)> m_length{ by_length };
  m_length.insert("hello");
  m_length.insert("cccccccccc");
  m_length.insert(std::string{});


  std::cout << "top of m_length: " << m_length.top() << '\n';
  std::cout << "size of m_length: " << m_length.size() << '\n';

  pic10c::max_heap<int> uh_oh;
  try { // go ahead, try to do a pop
```

```
    uh_oh.pop();
  }
  // and we'll catch the exception if it comes up
  catch (const std::logic_error & E) {
    std::cout << E.what() << '\n';
  }

  // now we do time trials
  const size_t N1 = 100;
  const size_t N2 = 10000;

  pic10c::max_heap<size_t> s;

  simple_timer::timer<> t;

  for (size_t i = 0; i < N1; ++i) {
    s.insert(i);
  }

  while (s.size() > 0) {
    s.pop();
  }

  auto time1 = t.tock();
  std::cout << "with N= " << N1 << ": " << time1 << '\n';
  t.tick();


  for (size_t i = 0; i < N2; ++i) {
    s.insert(i);
  }

  while (s.size() > 0) {
    s.pop();
  }

  auto time2 = t.tock();
  std::cout << "with N= " << N2 << ": " << time2 << '\n';

  return 0;
}
```

A few hints...

1. If you are not familiar with templates, start by writing a maximum heap class for



```
4
top of m_less: 5
top of m_less: 4
top of m_great: 0
top of m_great: 3
top of m_length: cccccccccc
size of m_length: 3
pop empty
with N= 100: 0.1272ms
with N= 10000: 11.2032ms
```

ints making use of only < for comparisons (the 10B notes explain the maximum heap structure).

2. Beware the demotion cases when a parent has only a left child.

3. You should be using **size_t** for your indices so be careful about underflow and guard against it.

4. Be careful that the size gets updated when appropriate.

5. If your maximum heap works for **int**s with <, now make it a templated class only over the type **T** — basically you'll need to replace **int** with **T** after templating the class. Make sure it works with only the default < used.

6. Now generalize to an arbitrary sorting.