PIC 10C Section 1 - Homework # 8 (due Sunday, May 22, at 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 renders the correct output on g++ with -O3 optimization on the C++20 Standard!!!

SPECIALIZAING A DEQUE FOR BOOLS

Similar to the **std::vector<bool>** optimization built into the C++ Standad, this homework focuses upon doing the same thing for a deque. For this homework, the **only header files you are allowed to use** are:

- **memory**
- **climits**
- **stdexcept**
- **type_traits**

You will submit **specialize_deque.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. **Beware of memory leaks!!!** You may find these files helpful in detecting basic memory leaks. The files may also help you to ensure you fully optimized your code. In the example below, there are **6** new allocations (and **6** deletes).

You can assume the code will be compiled with a **makefile** of the form:

```
hw8: HW8.cpp specialize_deque.h
    g++-8 -std=c++2a -O3 HW8.cpp -o hw8
```

for a **HW8.cpp** that could differ from the example given!

Within the **pic10c** namespace, you will write:

- a **deque** class, templated by a type **T** that has

  - a member **data** of type **std::unique_ptr<T[]>** to store the data;
  - members **cap** and **sz** to track the capacity and size of the deque;
  - members **left** and **right** to track the "leftmost" and "rightmost" indices of data within the dynamic array of data;
  - a defacult constructor to begin storing nothing, taking no heap memory, initializing the members appropriately;
  - **push_front** and **push_back** functions to append values at the beginning/end of the deque;
  - **pop_front** and **pop_back** functions to remove the frontmost/backmost element (exceptions should be thrown if the container is already empty);
  - **begin** and **end** functions, overloaded on const, to return iterator objects;
  - a **swap** member function to swap the contents of the implicit parameter with the explicit parameter;
  - **size** and **capacity** functions to return the size and capacity;
  - a subscript operator, overloaded on const, to retrieve the element at a given index;
  - **copy/move constructors** and **assignmnent operators**; and
  - a **destructor**.

  The iterators must be ~~input iterators~~ **forward iterators.**

- a **deque<bool>** specialization that has all the same features as the base template but optimizes for space so that within a single byte, many **bool**s can be stored. As part of the optimization, there should be nested **(const_)proxy** classes that:

  - can be converted to **bool**;
  - can be assigned-to from a **bool** when not **const_proxy**

  These classes should be returned from the subscript and dereferencing operators. Iterators of this specialized class do not need to have **operator->**.

- a **sum** function accepting a **deque<T>** when **T** is a numeric type (floating point or integral type other than bool) to return the sum of the values. It should be a compiler error to use this function on deques whose type **T** does not satisfy these conditions.

As far as implementation goes:

- With the first item inserted, the capacity should be made to 1 (in the base template) or **CHAR_BIT** (1 full byte) in the specialization.

- Every time the capacity is about to be exceeded, it should double in size before elements are moved over.
- **Reallocation should only occur if every space has been used**: you should allow the data to "wrap" around the left/right so that every space is used as fully as possible before more space is required*.

The deque is full when the size and capacity are equal.

A test case, code and output, are provided below.

```
#include "specialize_deque.h"
#include <iostream>

int main() {
  pic10c::deque<int> d;
  d.push_back(4);
  d.push_back(5);
  d.push_front(3);

  for (auto i : d) {
    std::cout << i << ' ';
  }
  std::cout << '\n';

  d.push_front(2);
  d.push_front(0);
  d.push_back(7);
  d[2] = 1;
  d[d.size() - 1] = 6;
  d.pop_back();
  d.pop_front();

  for (auto i : d) {
    std::cout << i << ' ';
  }
  std::cout << '\n';

  std::cout << "sum: " << sum(d) << '\n';

  pic10c::deque<bool> b;

  b.push_back(true);
  b.push_back(true);
  b.push_back(true);
  b.push_back(true);
  b.push_front(false);

  for (auto i = b.begin(), e = b.end(); i != e; ++i) {
    std::cout << *i << ' ';
  }
  std::cout << '\n';

  b.push_front(false);
  b.push_front(false);
  b.push_front(false);
  b.push_back(true);
  b.pop_back();
  b.push_front(true);
  b.push_back(false);
  b[b.size() - 2] = false;

  for (auto i = b.begin(), e = b.end(); i != e; ++i) {
    std::cout << *i << ' ';
  }
  std::cout << '\n';

  // sum(b); // <-- COMPILER ERROR

  return 0;
}
```

The output is:

```
3 4 5
2 1 4 5
sum: 12
0 1 1 1 1
1 0 0 0 0 1 1 1 0 0
```

Some hints...

- Start with the base template and ignore the specialization at first.
- The wrapping around business is the “hard part" but it’s simple if you write a couple of helper functions. If you know the index of your front and you know the size of the memory you have allocated, you should be able to compute a mapping from logical index **i** to the offset in the buffer **offset(i)**.
- You may also want to write a **get_right(i)** and **get_left(i)** to compute the next position of the front or back when a new item is added or one is taken away. For example, if **right** is the buffer offset for the back position, **get_right(right)** would be **right+1** if that fits in the buffer and is otherwise **0**.

- When new memory needs to be allocated, you can't just move the values over blindly. You should place them in the new chunk of memory in logical order, not the order in the old buffer.

- The iterators themselves can work adequately with a logical index **i** and a pointer to their owner container, **container**. Deferencing then amounts to working with **container->operator[](offset(i))**.

- The specialization logic is quite similar but from a logical index **i**, this translates to a position in a much bigger buffer (since each byte holds **CHAR_BIT** bits) so it is necessary to use integer division and modular arithmetic to compute the byte being accessed and then its corresponding bit.

- The bit manipulations and conversions can be handled as done in the class.