- In this project, we'll write a simulation like this one. Suppose that a chain of big-box retail stores is considering a rearrangement of the way that customers go through the check-out process, in hopes of processing customers more efficiently, helping them spend less time waiting in line, and keeping their business from being lost to more efficient competitors. To aid in that analysis, you'll build a program that will simulate two different arrangements of lines and registers, generating a log of the movement of customers throughout the simulation and, at the end, overall statistics.
- Along the way, you'll implement a doubly-linked list with head and tail pointers, then use it as the basis for a /*queue*/, which you'll then use in your simulation. This will give you additional practice with the kind of thinking and techniques you need to implement data structures, while also building your experience with some new areas of C++ that you may not have used a lot previously, such as templates, public and private inheritance, and move semantics.

H2 The first step: Implementing the necessary data structures

- Your first step is to implement the necessary data structures to support your application. Since we're building a simulation that's fundamentally centered around people standing in line, a queue implementation is what we need here. This part will be tested and graded separately — and you can obtain a substantial proportion of the correctness score for this project for submitting just this part, even if you don't build the application around it. But the implementation will need to be painstakingly complete and correct; we'll be doing the kind of testing we did in Project 0 when we grade it.
- In the *__core__* directory within your project directory, you'll find two files that you'll be most interested in:
- *__Queue.hpp__*, which contains a class template *__Queue<ValueType>__*. As luck would have it, I'm actually giving you a full implementation of this, which you'll need to leave as-is. (You can add things to it, if you'd like, but you can't make any changes to existing parts of its public interface, because we'll be running unit tests that need to be able to compile and run.) Now, the fact that you're being given the full implementation of this isn't quite the good news that it sounds like: All it does is inherit its implementation from a base class, which you'll have to implement.

- *DoublyLinkedList.hpp*, which contains a class template *DoublyLinkedList<ValueType>*. The public interface of this one is designed – and, like *Queue*, you'll need to leave the existing parts of the public interface as-is, though you can add anything else you'd like – but the implementation is mainly stubs, sufficient to make the unit tests compile and run, but not to make them pass. *DoublyLinkedList* forms the basis of the *Queue* implementation, but also includes a fair amount of additional functionality that will not be required by *Queue*.
- In the *gtest* directory, you'll find a collection of sanity-checking unit tests for both *Queue* and *DoublyLinkedList*, which demonstrates how each of the member functions are expected to behave and interact with one another. The depth of the testing is fairly light – while the sanity-checking tests touch all of the basic functionality, there are a lot of interesting cases missing, so you'll want to be sure that you write your own *DoublyLinkedList* tests, as well. Note, too, that your best bet is not to change any of the sanity-checking unit tests – other than perhaps commenting some of them out and then bringing them back later, if you need to run some of them in the absence of the others – because they're a fundamental way to demonstrate that your implementation will compile against our unit tests, so we'll be able to test it.
- For this part, any code you write needs to be in the *core* directory. There's a good chance you won't be making changes to any files except *DoublyLinkedList.hpp*, but even if you choose to add files, you would add them in the *core* directory. Avoid the temptation to create a *DoublyLinkedList.cpp*, though; templates are generally written entirely in header files. (See the Templates notes if you need a refresher on why.)

Features of your DoublyLinkedList class template

- Your DoublyLinkedList class template must implement all of the public member functions that have been declared within it already, which include all of the following:
- A constructor that initializes a list to be empty.
- Copy and move constructors that initialize lists from existing ones.
- Copy and move assignment operators that change existing lists to be like other existing ones.
- A destructor that cleans up the memory owned by the list.
- Adding and remove values at either end.
- Accessing the values at either end.
- Tracking and reporting its size.
- Support for two kinds of /*iterators*/: one that allows the contents of the list to be both accessed and modified, and another that allows them to be accessed but not modified. (See the section titled /*Iterators*/ below for more information about this requirement.)

- Of course, partial credit is available for a partial implementation, though some features are going to be more fundamental than others. For example, if we can't construct a list, there's not much else we can do to test it; if the only thing missing is the move constructors or the iterators, there's still a lot we can test. Of paramount importance is that our unit tests can compile. Be sure your code compiles against all of the sanity-checking unit tests, even if they don't all pass; it we can't compile our tests, we can't run them, and we're not going to be willing or able to spend time debugging your code to make them compile.

Limitations

- When you work on *__Queue__* and *__DoublyLinkedList__*, you are not permitted to use any part of the C++ Standard Library. Do not submit a version of *__DoublyLinkedList.hpp__*, *__Queue.hpp__* (or any file that they include) that includes any C++ Standard Library headers. (This includes things like adding a *__print()__* member function that requires *__<iostream>__*. Remove anything like that – including any *__#include__* of any standard library header file! – before you submit your work.)
- The nodes in your *__DoublyLinkedList__* must remain completely private. There should be no part of the public interface that exposes the notion of *__Node__*, because the nodes in a linked list are an implementation detail; exposing nodes to code outside of *__DoublyLinkedList__* invites code that breaks the list's fundamental assumptions, such as introducing cycles or losing links to nodes and causing memory leaks. This is why the provided version of *__DoublyLinkedList__* has its *__Node__* struct declared as a private member of *__DoublyLinkedList__*, why there are iterators (to provide a way for code outside of *__DoublyLinkedList__* to iterate through a list without manually twiddling pointers), and why the iterator classes are also declared within *__DoublyLinkedList__*; that way iterators can see the nodes, but no code outside of *__DoublyLinkedList__* can.

The second step: Writing the application

- In the *__app__* directory, you'll write the rest of your code, a program that performs the following simulation. The format of the input and output are described in detail and need to be followed carefully; spelling, capitalization, and spacing are all relevant and must be correct for full credit. Your simulator will read all of its input from *__std::cin__* and write all of its output to *__std::cout__*. While you may want to use the technique of */redirection/* to use files for this purpose (see below), your simulator will always use *__std::cin__* and *__std::cout__*.

What are we simulating?

- In our hypothetical big-box retail store, customers shop and choose the merchandise they want to buy. Each customer then proceeds to get into a /*line*/ to wait to be checked out. When there is an available cashier, the customer goes to the /*register*/ where that cashier is waiting. When the cashier is finished checking the customer out, the customer exits the register and is considered finished.
- What we're interested in doing is tracking these movements: when customers enter lines, exit lines and enter registers, and finally exit registers. We'll then report various statistics at the conclusion of the simulation, to summarize the overall outcome.

Arrangements of lines and registers

- There are two different arrangements of lines and registers that our simulation will support.
- 1 One or more registers, each with its own separate line. A customer in a particular line will only ever proceed to the corresponding register.
  2 One or more registers, with one shared line feeding customers to all of them.

The input

- First of all, you may freely assume that the input given to your simulation will match the description below. It may obviously be different than what's shown here, but it will always follow all of the rules here. Your program is free to do anything you'd like – up to and including crashing – in the case that the input is invalid.
- The simulator's input begins with what we'll call the /*setup section*/, which specifies the parameters that control how the simulation will run. The setup section looks like this – the italiczed portions are included here for descriptive purposes, but are not included in the actual input.

```
4     /the length of the simulation, in minutes/
3     /the number of registers/
3     /the maximum line length, beyond which customers be lost/
M     /S for a single line, M for multiple lines (one for each register)/
40    /register #1 takes 40 seconds to process a customer/
50    /register #2 takes 50 seconds to process a customer/
30    /register #3 takes 30 seconds to process a customer/
```

There are a few notes to be aware of:

- When we talk about the /*length of the simulation*/, we don't actually intend for the simulation to take that long to run. The goal is for the simulation to give a quick answer to the question of "What would a day in my store look like if we arranged things like this?"
- We'll say that each register has a /*register number*/ and that they are numbered consecutively starting from 1. Similarly, lines will have a /*line number*/ and they're also numbered consecutively starting from 1.
- The simulation length is given in minutes, while the processing times for each register are given in seconds.
  After reading the setup section of the input, your simulator will have what it needs to set things up and get started. From there, the rest of the input specifies /*customer arrivals*/. Each line in the customer arrival section of the input consists of two numbers: a positive number of customers and the time that these customers arrival. (Time in our simulation is always measured in terms of the number of seconds since the simulation started.) You can assume that the time associated with each line describing an arriving of customers will be greater than the time associated with the previous one.
- The input will always end with a line consisting of the word *__END__*. That doesn't mean that the simulation should end immediately; it just means that there are no more customer arrivals.

H3 The movement of customers through the simulation

So that we can all agree on the proper output of the simulation, we'll need to agree on the precise details of how customers move through the simulation. In the interest of keeping things simple, we'll take some liberties with reality – customers won't always do the smartest thing, we'll ignore how long it takes for customers to physically move around, and so on. Also, all actions are considered to have happened at discrete times measured in an integer number of seconds since the start of the simulation; something might happen at time 10 or time 11, but never at time 10.5.

- In any given second of simulation time, customer arrivals are always considered before customers are moved into and out of registers.
- When /*n*/ customers arrive at a particular time, we assume that they're separate – they each have a shopping basket and are interested in engaging in a separate transaction. Each of them has a decision to make and they make them one right after the other:

    - The customer chooses the shortest line and enters it. Note that this is based only on how many customers are in each line; the presence or absence of a customer at any registers is not considered. When there is a tie (i.e., two lines are equally the shortest), the customer will always choose the line with the smallest line number (e.g., if lines 3 and 7 are equally the shortest, the customer will enter line 3).
    - If all of the lines are the maximum length specified in the setup section, the customer is not willing to wait, and instead leaves the simulation immediately. That customer is considered to be /*lost*/. (This is a crude representation of a store being busy enough that it drives away customers.)
- Whenever a register is unoccupied, a customer /*immediately*/ moves from the corresponding line and into the register. That customer will stay for the appropriate number of seconds (as determined by the processing time for that register, specified in the setup section). At that time, the customer will leave the register and will /*immediately*/ be replaced by another.
- For the sake of simplicity, we'll assume that customers will never move from one line to another once they've entered a line, nor will a customer ever enter a register from any line other than the one that corresponds to it.

The output

The output of your simulator consists of two sections:

- The /*log*/, which indicates each time an "interesting" event occurs. The log begins with the word *__LOG__* alone on a line, followed by one line of output for each event. Each line of output describing an event consists of an integer simulation time (the number of seconds since the simulation started), a space, and then a description of the event. The following kinds of events are required to be logged:
    - The simulation started
    - The simulation ended
    - A customer entered a line, in which case we want to know which line number and how long the line is now (including the new customer)
    - A customer exited a line, in which case we want to know which line number and how many seconds the customer waited in that line
    - A customer entered a register, in which case we want to know which register number
    - A customer exited a register, in which case we want to know which register number – there's no need to see how long they waited, as this is always the same for a given register
    - A customer has been lost (i.e., they left without waiting in line because all lines were maximum length)

- The “stats” section. This section begins with a blank line (to separate it from the log visually), followed by the word *__STATS__* alone on a line, followed by these statistics:
  - How many customers entered a line during the simulation
  - How many customers exited a line during the simulation
  - How many customers exited a register during the simulation — we don’t show many customers entered a register, because every customer who exits a line immediately enters a register
  - The average wait time, in simulation seconds, for customers who exited a line. We only care about how long they waited in line, and we only measure this for customers who exited a line; customers still remaining in line at the end of the simulation are not included. Display this value to two digits after the decimal point.
  - How many customers are still left in line at the end of the simulation (i.e., they’ve entered a line but not exited it yet)
  - How many customers are still left at a register at the end of the simulation (i.e., they’ve entered a register but not exited it yet)
  - How many customers were lost during the simulation

H3 Example input and output

A complete example of the simulator’s input and expected output (for that input) are provided in the *__examples__* directory within your project directory in files called *__sample.in__* (the input) and *__sample.out__* (the output). Your goal is to match this format precisely; spelling, capitalization, and spacing are all relevant, so take some time to study the example and make sure you recognize the little details within it.

H3 Limitations

You must use your own *__Queue__* class template to implement the concept of a line of customers. Beyond that, you are free to use any part of the C++ Standard Library you wish in your simulator, even though it’s forbidden in your *__DoublyLinkedList__* and *__Queue__* implementations. For example, you may prefer to use *__std::vector__* to store the registers or the lines (though you can certainly use *__DoublyLinkedList__* if you prefer).

H2 What is private inheritance?

- The provided *__Queue__* class template is quite short, even though it’s also quite capable. You won’t need to write any code in *__Queue__* to make it work completely and correctly. That’s because it inherits its implementation from the *__DoublyLinkedList__* class template. In general, for any type *__T__*, the type *__Queue<T>__* inherits from the type *__DoublyLinkedList<T>__*. As you would expect, then, *__Queue__* objects have all of the member variables and member functions that *__DoublyLinkedList__* objects have, which means the entire implementation of *__DoublyLinkedList__* is inherently a part of *__Queue__*.

- It's not entirely crazy that *__Queue__* inherits from *__DoublyLinkedList__*, because you could certainly say that a queue is a linked list, if that's how you wanted to implement one. A queue is a linked list that provides more limited abilities; it does less than a linked list does, rather than doing more. And, in fact, we'd like to keep it that way. If we have a queue-shaped problem, we probably only want to be able to perform queue-specific operations like enqueue, dequeue, and front, with anything else likely to be a bug in our program, anyway. What we really want is for *__Queue__* to inherit from *__DoublyLinkedList__*, but for that inheritance relationship to be a secret; only *__Queue__* knows about it, so code outside of *__Queue__* can't do things like call *__removeFromEnd__* or create iterators that can insert elements in the middle somewhere.
- In C++, /*private inheritance*/ is how you achieve this kind of goal. Private inheritance is what it sounds like: It's an inheritance relationship that only the inheriting class knows about. Suppose you start a class declaration in C++ like this:
  class Y : private X
- What this means is that *__Y__* inherits from *__X__*, but that the only code in the program that can make use of that relationship is *__Y__*. A *__Y__* object will have all of the member variables and member functions of *__X__*, but they won't be accessible outside of *__Y__*. Inside of the *__Y__* class, it would be possible to point an *__X__** to a *__Y__* object, but this wouldn't be possible outside of the *__Y__* class.
- Why a feature like this is useful is that it allows a class to inherit an implementation without inheriting an interface. This way, a *__Queue__* can be a *__DoublyLinkedList__*, but no one else knows it; no one can treat a *__Queue__* like a *__DoublyLinkedList__* except *__Queue__*. So if a *__Queue__* member function wants to call *__addToEnd()__* or *__removeFromFront()__* on the *__this__* object, it can do that; but code outside of the *__Queue__* class won't be able to do it.
- There are two ways to avoid bugs. One is being careful not to do things you shouldn't do. The other is to /*make it impossible*/ to do the things you shouldn't do. The latter is something to strive for, and static type checking in C++ is a useful tool to help you get there, with private inheritance one of the many techniques available to express the features and constraints in your design.

Iterators

- An /*iterator*/ is an abstraction for a position within a data structure, the goal of which is to allow the values in a data structure to be accessed (and potentially modified) without relying on the details of how the data structure is implemented. Given an iterator, an algorithm can obtain all of the values in the data structure without manipulating pointers, understanding the data structure's layout, or sometimes without even being aware of what kind of data structure it is. When you can separate an algorithm from the details of the data structures it operates on, it becomes possible to write the algorithm generally; linear search is the same algorithm no matter what kind of data structure you're searching, as long as there's a way to say "Give me the next thing I haven't seen" and ask "Is there anything left that I haven't seen?" Iterators provide those kinds of fundamental capabilities.
- For more background on this topic, see the section titled /Iterators. Our iterators aren't quite the same as the ones in the standard library, but their purpose is the same.

The difference between Iterator and ConstIterator

- Our implementation of *__DoublyLinkedList__* draws a distinction between *__Iterators__* and *__ConstIterators__*. That distinction mirrors a similar distinction in the containers in the C++ Standard Library such as *__std::vector__* and *__std::list__*. For our purposes, the distinction is this:
- An *__Iterator__* has the ability to access and modify the contents of our list. This includes moving forward and backward, accessing and changing the current value, inserting values before or after the current value, and removing the current value.
- A *__ConstIterator__* has some of these abilities, but not all of them. What it promises to holds constant is the contents of the data structure, so it only provides features that don't introduce change. They can be moved forward and backward, and they can access the current value, but they can't insert, remove, or change existing values.
- Note, too, that it's possible to have a *__const Iterator__* (i.e., a variable of the type *__Iterator__* that is itself marked as *__const__*), but this is a different thing altogether. What a *__const Iterator__* holds constant is the iterator (i.e., you can't move it), but it makes no promise about the contents of the list. This is similar, in concept, to the way that the *__const__* specifier interacts with pointer types, in that *__const__* can be used to protect the pointer, the value the pointer points to, or both.

How our iterators are different from the standard ones

- The key difference between our iterators and the ones in the C++ Standard Library is our design avoids the need for the deep use of a C++ called feature called /*operator overloading*/, in which the meaning of operators is redefined for objects of classes you write; this is not something that's commonly been seen in depth by students coming into ICS 46, even if it's been touched on. Fundamentally, our iterators are pretty similar to the /*bidirectional iterators*/ in the standard library, with member functions taking the place of the operator overloads, along with some slight differences in semantics. So, if you understand how to use the standard iterators, you'll find using and testing your own iterators to be pretty similar.

Redirection (or, How to avoid re-typing the input every time you run your program)

- When you use the ***run*** script to execute your program – and, generally, by default in Linux – the standard input and standard output are connected to the shell window where you executed the program. The program's input is typed into that shell window; the program's output is written to that shell window.
- However, you can also use a technique called /*redirection*/ to connect the standard input and/or standard output to other devices – most importantly for us, to files! – instead of the default. For example, the contents of an existing file may be redirected into the standard input, so that your program will read its input from the file instead of from the shell window; similarly, the standard output can be redirected into a file, meaning that all of the output printed to ***std::cout*** will be saved into that file, rather than being displayed in the shell window.
  The typical mechanism for redirection is to use the < and > operators on the command line, like this:
  SomeProgramYouWantToRun <FileContainingStdInput.txt >FileToStoreStdOutput.txt
- You don't have to use both; you can use only < (in which case the standard input is read from a file, but the standard output is written to the shell window) or only > (in which case the standard input is read from the shell window, but the standard output is written to a file). It's up to you.

Using redirection to avoid re-typing test input

- This can be a handy technique to use in your testing, to avoid the problem of having to re-type the program's input every time you run it. If you used the *__project2__* project template – and you should have! – when creating your project, you should see a directory called *__examples__* in your project directory; in the *__examples__* directory, you should see a file *__sample.in__*, which contains the sample input from this project write-up. If you want to run your program using the same input, issue this command (from your project directory):

  ./*run* <*examples*/sample.in
- The effect is that the contents of the *__sample.in__* file in the *__examples__* directory will be redirected into your program's standard input, in lieu of you having to type it. You might also want to write other test files for yourself in that same *__examples__* directory, then use a similar technique to redirect them into the standard input of your program. You can avoid a lot of re-typing this way!

  Similarly, you can use redirection to send the output of your program to a file.

  ./*run* <*examples*/sample.in >my.out
- That would create a file called *__my.out__* in your project directory containing the output of your program. You could then compare it to the example output using the *__diff__* command, which is a handy way to quickly see if there are any differences.

  diff my.out examples/sample.out