This project involves a system that is capable of assembling and testing simple circuits composed of standard digital logic gates. There are many good sources of information on logic gates; a Google search will turn up a number of high-level overviews. Recall that the Boolean values true and false are commonly represented in digital environments by the constants 1 and 0, respectively. A logic gate (for the purposes of this assignment) is simply an entity that accepts one or more Boolean values as inputs and emits a single Boolean value in response.

For our purposes, only the following logic gates are of interest:

**not gate:** accepts a single input $x$ and emits $\neg x$

<table>
<thead>
<tr>
<th>$x$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**and gate:** accepts two inputs $x$ and $y$ and emits $x \land y$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**or gate:** accepts two inputs $x$ and $y$ and emits $x \lor y$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**nand gate:** accepts two inputs $x$ and $y$ and emits $\neg (x \land y)$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**xor gate:** accepts two inputs $x$ and $y$ and emits $(\neg x \land y) \lor (x \land \neg y)$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

We also need one gate that does not perform a logical function, but serves a necessary purpose in circuit construction:

**splitter:** accepts a single input $x$ and emits $x$ to each of two outputs

<table>
<thead>
<tr>
<th>$x$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that each of the two-input gates is symmetric with respect to its input values. That is, if you reverse the values supplied through the two input wires, the resulting output will remain the same.
Gates are connected by wires. (To be picky, the diagrams shown above include input and output wires.) The only restriction on how a collection of gates may be connected is that, to be usable, each gate must be connected to as many input and output wires as called for by its type.

A circuit is a collection of connected wires and gates, and has one or more input wires and, for our purposes, exactly one output wire. For simplicity, each of our circuits will include a single emitter (described below) that serves as an interface to the circuit.

For convenience, it is useful to provide a simple device that collects all of the input wires for the entire circuit into a single controller, which can be used to send any desired signal through any desired wire. I call such a device an emitter. An emitter has no input wires, and can be associated with an arbitrary number of output wires (each of which is presumably also serving as an input wire for a gate in a circuit).

Here is an example of a circuit, called a canceller. What would its state table (truth table) look like?

\[
\begin{array}{ccc}
  x & y & \text{Output} \\
  0 & 0 & ?? \\
  0 & 1 & ?? \\
  1 & 0 & ?? \\
  1 & 1 & ?? \\
\end{array}
\]

Note also that every circuit represents a Boolean expression. The canceller circuit above represents \( x \land \neg y \).

Some additional circuit examples will be posted on the website for your entertainment.

As you should have already inferred from the descriptions of the various logic gates, the primary design focus of this assignment will depend upon the fact that there are various kinds of logic gates, and there are natural inheritance relationships among them. You must implement a set of classes, all within a single inheritance hierarchy, to represent the different types of gates described above. You will also need a class to represent wires (a wire isn't really a gate).

Polymorphism comes into play because a gate should behave properly when it receives a signal from a wire, even though the wire will have no idea what kind of gate it's connected to. To simplify checking that you have achieved true polymorphism, you are forbidden to use pointers of any derived gate type when assembling circuits.

As you should infer from the description above, a secondary focus of this assignment is again associations among dynamically created objects. The system, described so far, contains no associations except those from wire to gate and from gate to wire. Your design and implementation should make these associations as flexible and natural as possible; it is particularly important that your design places the correct responsibilities on each class in the system.

As with any system in which dynamic allocation is used, it is important that you manage memory effectively. Do not create unnecessary objects, and be sure that all memory that is allocated dynamically is deallocated as well. You will also need some dynamic data structures to keep track of the various wires and gates that are created. You may use any truly dynamic structure for this purpose; the use of templates, whether STL or homegrown, is expressly allowed and encouraged.

We will release portions of an implementation for your use. The code will be supplied as is, with no warranty. It will be largely undocumented, so you will have to analyze it in order to use it safely. It will also deliberately omit some features, which you will have to complete to render the code fully functional. And, of course, the code will not work at all without your implementations of the classes mentioned earlier. You may use as much, or as little, of that code as you like. You may modify it, or ignore it.
Building Circuits

Your program must be able to build a circuit from gates and wires, and test the circuit by generating a state table for it.

Where will the circuit components (gates and wires) come from? In a good design, there will be a factory object that is responsible for supplying parts on demand. Client code would simply ask the factory to make a wire, or a gate of a specific type, and the factory would supply one. The supplied code will include a sample factory class, called Fabricator, which illustrates one approach to this. A Fabricator object actually retains ownership of the requested part, storing it in an internal database. A client must first ask the Fabricator to create the desired component, and then ask the Fabricator for access to the component (in order to connect it to other components). For safety, the Fabricator only gives the client a constant pointer to the component.

Who will actually assemble the components that make up a circuit? That should be handled by another object. In this project, you will be supplied with instructions for building a circuit via an input file. For example, here are the instructions for assembling the canceller circuit shown earlier:

```
Canceller ; name of the circuit
order wire In01 ; parts list (to request from Fabricator)
order wire In02
order wire W01
order wire Out01
order notgate NOT01
order andgate AND01
input In01 ; specify which wires are circuit inputs
input In02
output Out01 ; and which is the circuit output
wire In01 to NOT01 ; connect wire In01 to gate NOT01
gate NOT01 to W01 ; connect gate NOT01 to wire W01
wire W01 to AND01
wire In02 to AND01
gate AND01 to Out01 ; finished with assembly
done
```

The first line contains the name of the circuit, a tab character, and then possibly spaces and a comment. Each remaining line contains one assembly instruction. Each assembly instruction consists of one or more tokens, each of which will be followed by a tab character. The tab after the final command token may optionally be followed by spaces and a comment.

```
order<tab>[wire | notgate | andgate | nandgate | orgate | xorgate | splitter]<tab><name><tab>
    Ask the factory to make a wire, or gate of the specified type, and give it the specified name. There shouldn't be any way for this operation to fail unless the type specifier is incorrect, which will not happen with any official test data.

input<tab><name><tab>
    Do whatever is required to make the named wire an input to the circuit being assembled. This cannot fail unless there is no wire in the inventory with the given name; in that case the assembler should log an error and abandon building the circuit (although we do not intend to test this).

output<tab><name><tab>
    Do whatever is required to make the named wire an output from the circuit being assembled. This cannot fail unless there is no wire in the inventory with the given name; in that case the assembler should log an error and abandon building the circuit (although we do not intend to test this).

wire<tab><wire name><tab>to<tab><gate name><tab>
    Do whatever is required to connect the named wire as an input to the named gate. This cannot fail unless one of the named components does not exist, or one of them doesn't have room for another connection. In those cases, the assembler should log an error and abandon building the circuit (although we do not intend to test this either).
```
gate<tab><gate name><tab>to<tab><wire name><tab>
Do whatever is required to connect the named wire as the output from the named gate. This cannot fail unless one of
the named components does not exist, or one of them doesn't have room for another connection. In those cases, the
assembler should log an error and abandon building the circuit (although we do not intend to test this either).

done<tab>
The circuit is complete. Do whatever cleanup or final assembly may be necessary (possibly none), and give the
completed circuit to the client who requested it.

The assembly script is guaranteed to be syntactically correct. There will always be an order command for each necessary
component, the correct number of input commands, exactly one output command, and exactly one done command. Note
that there may be logical errors, such as naming a non-existent component or attempting to attach one (end of a) wire to more
than one gate. Your program should deal with such issues gracefully.

Who is responsible for testing the circuit? Again, this should be carried out by an object which is capable of taking any valid
circuit, feed it all possible combinations of input values, and create a state table like the ones shown earlier. Note the tester
doesn't have to actually determine if the results are correct; that's for a human to evaluate.

Invocation
As before, your program will take the name of the script file from the command line, and write all of its output to standard
output for redirection, something like:

```
Gates <assembly script>  >   <log>
```

If the specified assembly script file does not exist, the program should print an appropriate error message and either exit or
prompt the user for a correction. You should follow the guidelines given in the previous project for the log (e.g., echoing and
numbering commands). Your program will read the assembly script, construct the corresponding circuit, generate a complete
state table for it, log the state table, and exit.

Design and implementation requirements
The discussion above suggests that there are a number of system-level components that make use of the gates and wires. One
possible (probably incomplete) design of the upper-level system could resemble:
There may be additional components implied by the diagram; for example, some aggregation and association relationships will imply a container is needed (and some will not). You are not bound by this sample design, but you should carefully consider the logical relationships it implies.

There are some explicit requirements, in addition to those on the Coding Style page of the course website:

- All of the gate types must belong to a single inheritance hierarchy.
- Give careful consideration to how you “factor” the commonality of gates into base types.
- The base gate class must be abstract.
- The destructors for the gates and wires must be instrumented to log the destruction of each object by name (you did notice each of those object is given a name when it is created).
- The factory class is allowed to use pointers of the derived gate types, but everywhere else in your code you must only use a base gate pointer (if the target is a kind of gate). Note well that includes the implementations of the wire class, all of the gate classes, and the circuit and emitter if you implement one.
- No class is allowed to store its type as a data member, or to provide a function that could be used to report the object's type to a client. The use of RTTI is strictly forbidden. Violations will be penalized massively.

Evaluation:

Shortly before the due date for the project, we will announce which TA will be grading your project. You will schedule a demo with your assigned TA. The procedure for scheduling your demo will be announced later. At the demo, you will perform a build, and run your program on the demo test data, which we will provide to the TAs. The TA will evaluate the correctness of your results. In addition, the TA will evaluate your project for good internal documentation and software engineering practice.

Note that the evaluation of your project will depend substantially on the quality of your code and documentation.

What to turn in and how:

Submit a zip'd archive, including all of your C++ source code files, to the Curator System. Be sure to read the submission guidelines on the course website. Submit only what is called for. Submit nothing else. In particular, do not submit any temporary files created by the compiler and/or linker, aside from those listed on the website. Instructions for submitting to the Curator are given in the Student Guide at the Curator website: http://www.cs.vt.edu/curator/. Be sure to follow those instructions carefully. You will submit your assignments via the URL:

http://eags.cs.vt.edu:8080/curator/

You will be allowed to submit your solution up to three times. Your latest submission will be evaluated unless you specify otherwise at your demo session.

You will schedule an evaluation demo with one of the TAs. The TA assignments and appointment calendar will be announced later. Failure to schedule a demo with the correct TA during the demo period, you will receive a score of zero for this project.
Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you must include the following pledge statement in the header comment for the file containing main():

```
// On my honor:
//
// - I have not discussed the C++ language code in my program with
//   anyone other than my instructor or the teaching assistants
//   assigned to this course.
//
// - I have not used C++ language code obtained from another student,
//   or any other unauthorized source, either modified or unmodified.
//
// - If any C++ language code or documentation used in my program
//   was obtained from another source, such as a text book or course
//   notes, that has been clearly noted with a proper citation in
//   the comments of my program.
//
// - I have not designed this program in such a way as to defeat or
//   interfere with the normal operation of the Curator System.
//
// - I have neither given nor received unauthorized aid in the
//   completion of this assignment.
//
// <Student Name>  <PID>
```

Failure to include this pledge in a submission is a violation of the Honor Code.