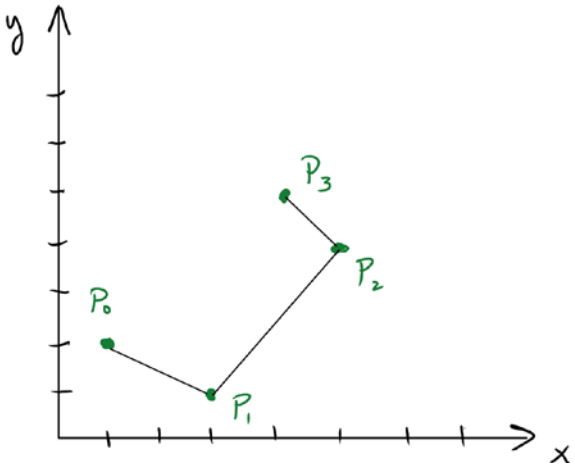
Homework 2: Creating a polyline ADT
ECE 2574, Spring 2018
Due before 5:00 pm on February 23

Honor Code: You must work *independently* on this assignment. You may copy and/or modify any code that you see in the textbook, and you may copy/modify any code provided to you by the instructor or GTA. Referring to other code is not permitted. Please review the statement in the syllabus.

Objectives: The main goals of this assignment are to gain experience with the design of Abstract Data Types and with linked lists. You will also reinforce important concepts related to object-oriented design, testing and debugging of code, and C++ in general.

Notice: This is a major assignment. It will require a lot more time and effort than Homework 1. You will work with linked lists, which means that you must write code using pointers and dynamic memory management. You need to write a copy constructor that does not produce shallow copies of objects. You will continue to use templates, along with inheritance and an abstract base class. The assignment also requires you to overload operators. These are not introductory-level topics.

Background: You must implement an ADT that supports geometric objects known as "polylines". A polyline is a connected series of line segments, usually specified as a sequence of 2-dimensional (2D) endpoints of those line segments. Mathematically, a polyline of n points can be represented as $(P_0, P_1, \ldots, P_{n-1})$, where P_i represents the coordinates (x_i , y_i) of a 2D point. An example is ((1, 2), (3, 1), (5, 4), (4, 5)), which is a sequence of $n = 4$ points that determine 3 line segments, as shown in the diagram below.



You can find lots of examples of polylines online, because they are common in applications involving computer graphics, computer-aided design (CAD), geographical information systems (GIS), and other disciplines.

Before you begin: Download the starter code, provided in file `hw2_starter.zip`. This zip archive contains several files similar to those that you received for previous assignments. Here is a summary of files that are provided to you:

`CMakeLists.txt`: Do not change this file, and do not submit it. This file is used by CMake to create a VS project named `hw2_test`. It creates other VS projects also, but `hw2_test` is the one that you will work with do develop code.

`catch.hpp`: Do not change this file, and do not submit it. It is needed by the Catch utility. (The Inginious auto-grader appears to be sensitive to particular versions of this file.)

`Node.h` and `Node.cpp`: Do not change these files, and do not submit them. They specify nodes to be used in linked lists by your polyline ADT.

`PolyInterface.h`: Do not change this file, and do not submit it. It contains the abstract base class `PolyInterface`. It also provides descriptions of all public methods that you need to implement.

`Poly.h`: Do not change this file, and do not submit it. It contains a specification of the derived class `Poly`.

`Poly.cpp`: Write code in this file to implement the methods that are described in `PolyInterface.h` and `Poly.h`. Submit this file as part of your solution.

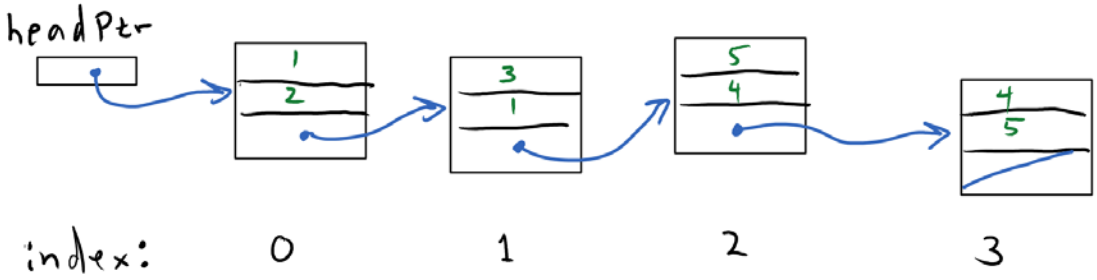
`hw2_test.cpp`: Write code in this file to test the methods for your polyline ADT. Submit this file as part of your solution. This file uses the Catch utility to create a main() function, and to implement test cases of your code.

`hw2_main.cpp`: This file is for your (optional) use only. Do not submit this file. It is here in case you would like to create your own main() function and write client code to experiment with your polyline ADT.

Because of the way that templates are handled, the implementation files `Poly.cpp` and `Node.cpp` must not be associated as Source Files within Visual Studio; instead, they will show up automatically under External Dependencies.

Project specification: Working from the starter code, develop and test an ADT named `Poly` that uses a linked list to contain one polyline. Write code to implement methods as described by comment blocks within the `*.h` files. Develop tests for your code using the Catch utility.

A linked list is a natural data structure for storing the sequence of points that constitute a polyline. The following diagram illustrates a linked list that represents the polyline shown on the previous page. Each node of the linked list represents one 2D point with three fields: one field for the *x* coordinate (shown at the top of each node in the diagram), one field for the *y* coordinate, and one field for a pointer to the next node.



The bottom of the diagram shows “index” values for these nodes. This terminology is not standard throughout the community, but is being introduced particularly for this assignment. A head node, when it exists, will always be at index 0. In this diagram, the tail node is at index 3. These index

values are not stored as part of the linked list. Their purpose is to identify particular nodes for some of the ADT's methods, such as `insert` and `remove`.

Your code will use templates to specify the type of the data values. Both data points (*x* and *y*) will be of the same type. Write your code to support templated types of `int` and `double`.

You may be wondering why we are going to the trouble of using class inheritance in this assignment. A primary reason is data structures other than linked lists can be used to implement polylines. (Matlab uses 2D matrix representations of polylines, for example.) The `PolyInterface` class could also be used to define several variations and extensions of polylines. For example, we could create a derived class that represents a polygon (by implicitly connecting the last point P_{n-1} to the first point P_0). Further, we could derive classes to represent convex vs. nonconvex polygons, etc. Thinking about practical applications, we could derive a class based on a polyline that represents a path on the road to be followed by a self-driving car. For this assignment, you do not need to derive any additional classes beyond `Poly`.

As you will see in the comments within the starter code, you must implement an `insert` method that accepts an index parameter to specify where a new node should be inserted. For the `insert` method, no nodes should be removed. Here are some examples to illustrate correct operation: Assume that the polyline ((1, 2), (3, 1), (5, 4), (4, 5)) exists, and assume that a new point (98, 99) is to be inserted.

After inserting at index 0, the result would be: ((98, 99), (1, 2), (3, 1), (5, 4), (4, 5))
After inserting at index 1, the result would be: ((1, 2), (98, 99), (3, 1), (5, 4), (4, 5))
After inserting at index 3, the result would be: ((1, 2), (3, 1), (5, 4), (98, 99) (4, 5))
After inserting at index 4, the result would be: ((1, 2), (3, 1), (5, 4), (4, 5), (98, 99))

Similarly, you must implement a `remove` method that accepts an index parameter to specify which node should be removed from the polyline. Here are some examples to illustrate correct operation: Again, assume that the polyline ((1, 2), (3, 1), (5, 4), (4, 5)) exists.

After removing the point at index 0, the result would be: ((3, 1), (5, 4), (4, 5))
After removing the point at index 1, the result would be: ((1, 2), (5, 4), (4, 5))
After removing the point at index 3, the result would be: ((1, 2), (3, 1), (5, 4))

You must overload the + operator so that it performs concatenation of two polylines. As an example of correct operation, suppose that `p` contains the sequence ((1, 1), (2, 2)) and `q` contains the sequence ((10, 20), (10, 21)). The result of `p+q` should be a newly created polyline that contains the sequence ((1, 1), (2, 2), (10, 20), (10, 21)). For this operator, it is acceptable for one or both polylines to be empty.

Similarly, you must overload the = operator so that it performs assignment of one polyline's values to another. For example, the statement `p=q` should cause `p` to contain a copy of the polyline stored in `q`. Interlude C6 of the textbook describes several things that you need to consider when writing this code.

Additional requirements: At the beginning of your test file `hw2_test.cpp`, place a comment block similar to the following:

```
////////////////////////////////////////////////////
// ECE 2574, Homework 2, Jane Doe ← your name here
//
// File name:   hw2_test.cpp
// Description: (Describe the purpose of this file)
// Date:        2/14/2018  ← completion date here
//
```

Add appropriate comments throughout all of the code that you modify or create.

Online testing: We will use the INGInious autograding system to determine part of your grade for this assignment. To use the autograder, place the following C++ source files into a single zip file: `Poly.cpp` and `hw2_test.cpp`. The zip archive must contain these files only, without any directories. A suggested file name for the zip archive is `hw2_`*name*`.zip`, where *name* is your family name.

Submit your zip file to the INGInious autograder at https://grader.ece.vt.edu. It will compile and run the tests that you have formulated in `hw2_test.cpp`, and the autograder will also run some "instructor" tests. A grade will be reported to you, proportional to the number of tests that have executed correctly.

You can submit to the autograder as many times as you like, but it will limit you to 4 submissions every hour. (This limit is to discourage people from using it as their only compiler and overloading the machine.) An advantage of autograding is that you will have peace of mind concerning the major portion of the grade for this assignment.

Submission to Canvas: After you are satisfied with your code based on your autograde results and addressing the requirements, upload the same a zip file (as described above) to Canvas at the HW2 assignment link. You do not need to submit any other files or directories.

Be careful to verify that you have uploaded the correct files to Canvas. After you have uploaded your zip file, it is suggested that you also download it from Canvas and verify that it is correct.

Grading: There are 100 points allocated to this assignment, and most of the grade will be determined by the autograder.

- Correctly submitting the required files to INGInious and to Canvas: 5 points
- Your tests compile: 0 points
- Your tests pass: 20 points (proportional)
- Instructor's tests compile with your code: 0 points
- Instructor's tests pass: 50 points (proportional)
- No memory leaks (checked by autograder using `valgrind`): 10 points
- Good coding style, and your tests are reasonably thorough: 15 points (assessed manually by GTA)

Please note that your most recent submission to INGInious determines the autograded portion of your grade.