**CIS2275 C++ Programming II NAME:**

**Inheritance, polymorphism, C++/CLI, Program 10 (The Last Program worth 200 points)**

**and four Enigma classes Due Date: Thursday, April 28 by 11:30am**

**Grade Breakdown: 200 points**

**20 pt You participate in running test cases in lab on other students’ files**

**Your program must be running in order for this portion of the score.**

**20 pts Your P10 is constructed correctly (as specified) and in its own .h and .cpp files.**

**20 pts Your user interface is simple to use. You follow program specs as described here.**

**60 pt (20 pts each) Successful encoding and decoding of each of your 3 techniques**

**20 pts Enigma base class array used correctly for polymorphism**

**30 pts Your project is correctly executed using a Windows Form and C++/CLI.**

**30 pts Other program specs met**

**Grading Note: During class, on 4/28, if your program is working, you will encode messages, then decode other student’s messages to test your program. DO NOT expect to finish your program during this lab period.**

**Turn In Requirements:**

1. **5 pts Name your Visual C++ 2015 project LastnameP10, such as NelsonP10.**
2. **5 pts Upload your project to Visual Studio Online.**
3. **5 pts If you upload to Blackboard, print out the \*.h and \*.cpp files, staple this page to the front of your printed source code when you turn it in for grading.**
4. **5 pts If you upload to Blackboard, remove BOTH debug folders and the .sdf file from your project before uploading it to Blackboard.**

**Program Requirements:**

1. **3 pts Write your name, email address and file name at the top of your source code in a comment.**
2. **5 pts Your main function should have cout statements that write “header” information to the screen. The header info includes your name, course and program information, as well as a 1-2 line description of the program.**
3. **5 pts. Use good C++ programming style and formatting for your program. Use appropriate comments to explain what you are doing.**

For this program, you will write a program that has three Enigma derived classes that perform string encoding and decoding. Your messages are written to a user-specified text-based file. The type of encoding and key are written into the file with the coded message. The program will use polymorphism and an array of parent class pointers in order to access the correct child or derived class.

You should be able to start with your Form from Program 9. You only have to add the choice of Enigma encoding/decoding type and you message summary will include the Enigma type. Use the same Write and Read file dialogs, with the addition of the type of encoding (PrimeShiftEnigma, ShiftyEnigma or the Dailey Enigma).

The form will control the flow of the program. You will need to make objects of all your Enigma classes on the form, and I will show you where to put the polymorphism processing code on the form.

Your form1.h will have an array of Enigma pointers, sized to 3 and one of each of the Enigma objects.

Enigma \*pE [3];

PrimeShiftEnigma pse;

ShiftyEngima se;

Dailey de;

pE[0] = &pse; //Prime Shift object address is in 0

pE[1] = &se; //Shifty object address is in 1

pE[2] = &de; //Dailey object address is in 2

In order to have easy access to the 3 different child classes, we use polymorphism and this base-class pointer technique. You will need to have some logic built into your program so that when you know what encoding or decoding scheme is used, you set an integer value for the appropriate one. For example,

The flow of the code will go something like this:

//determine which encoding/decoding scheme is used

//use an if/else if block of code with your logic

//to assign appropriate value into an integer index value, to either 0, 1, 2

//then call the appropriate methods

//This code will be in your event handlers:

pE[index]->setMessage( message);

string codedMessage = pE[index]->getCodedMessage();

int key = pE[index]->getKey();

etc.

You **will not** have three blocks of code calling three different sub class objects. Your program must have the logic to assign the appropriate value into the index, and that index that is used to access the array element.

All three of your encoding/decoding classes use our Enigma class as its parent. All three will use the Enigma’s key-shift encoder on all messages before performing the derived class encoding method.

**Prime Shift encoding scheme**: Each character in the message (after it has been key-shifted) is checked to see if it is a prime number. If it is a prime number, then check if the key is a prime. If the key is a prime, shift to the next prime. If the key is even, shift to the second prime up, else shift to the third prime up. If the number is not a prime, do nothing.

You must be sure that the shift wraps the values and keeps the character range between 32-126. If the next (or previous) prime is out of range, wrap it to the next relevant prime. The numeric values are then converted to a char and that string is the encoded message.

For example, if the character’s value is 37, it is replaced with 41. If the character is 113, the next prime is 127, but that is out of range. So it is wrapped back to 37 (the first prime in our range).

**Shifty encoding scheme**: Each character in the message (after it has been key-shifted) is converted to its ASCII value. The value is subtracted from 1000, resulting in a triplet of numbers. Each digit of the triplet is then converted to the symbolic value on the keyboard above the numbers 1 – 0. The string of symbols is the encoded message.

For example, if the character’s value is 37, it is subtracted from 1000, resulting in a triplet of 963. The corresponding keyboard characters are (^#.

**Dailey encoding scheme:** In this encoding scheme, we are going to insert random characters between the characters between the characters in the codedMessage. For each character in the codedMessage, randomly generate a number between 32 and 126 (inclusive), then convert it to a char. Insert that character after the character location you are at. The codedMessage will now be twice as long as before. Then find a new number, int num = key%4 + 2. Split up the codedMessage into sub-strings num characters long. If there is not an even number, the last section will be as long as it is. For each sub-string, reverse it and place it back into the codedMessage where it was, only now it’s reversed.

**OutPut File:** The encoded message, key, and coding scheme go into the text-based file in the following format:

First line: encoded message **|** PrimeShift = 0

Second line: enigma key value  **|** Shifty = 1

Third line: a number that represents the type of encoding: use these value **|** Dailey = 2