Assignment 8

Simple Cryptography

README due: 9:00 PM, Tuesday, April 16, 2002 (*via* `cs70submit`)

Code Due: 9:00 PM, Thursday, April 18, 2002

The primary purpose of this assignment is to learn how lists can be used to build more complex data structures.

1 Scenario

You've realized that your mailbox archives contain a good deal of information that you would like to keep private. There are a number of cryptographic programs that might be able to help you keep your old email secure, but these techniques have two fundamental drawbacks:

1. They work too well—if you forget your passphrase, you may *never* be able to recover your data
2. You didn't write them and have no idea how they work

Your email may be private, but it isn't a matter of national security. So you would like to use an encryption algorithm that would cause a bored hacker to choose an easier target, but nothing *too* complex. Thus you have decided to investigate simple ciphers that can be broken without massive effort, and are beginning by looking at the *Vignère* cipher.[1]

You've also been thinking about string types because your program is going to be manipulating strings of characters. While reading up on *basic_string<char>* (which is the real type for which *string* is an alias), you discovered that the C++ standard is actually very vague about exactly what the complexity of various string operations should be. To be sure that you'll always have an efficient *string* implementation, you've decided to write your own.

2 The Vignère Cipher

The Vignère cipher is named after Blaise de Vignère, although it is really a corruption of a much more secure cipher he invented in 1585. The Vignère cipher is based on the

1. The Vignère cipher is fairly straightforward to break, except in one special case. If the key is at least as long as the message, the key is a truly random string of characters rather than an English word or phrase, and the key is never used again, the Vignère method reduces to something called a "one-time pad", which is the *only* provably secure encryption method.

earlier *Caesar cipher*,[2] which rotates letters through the alphabet. For example, a Caesar rotation of one letter would replace "A" by "B", "B" by "C", and so forth; wrapping around to replace "Z" by "A". A rotation of two letters would replace "A" by "C", "B" by "D", and so forth.

Caesar used a constant rotation for his encoding, which made decryption quite simple. The variation named after Vignère modified the scheme by varying the rotation for each successive letter of the message. For example, the first letter might be rotated by 14 positions, the second by 5, and so forth. The pattern of rotation is controlled by a *key*, which is expressed as a simple word or phrase. The key is repeated as necessary to make it match up with the message.

For example, suppose our key is `ALPHA` and we wish to encrypt the message `NOW IS THE TIME FOR CS FUN`. Ignoring the spaces, we can write the message and the key lined up as follows:

```
NOWISTHETIMEFORCSFUN
ALPHAALPHAALPHAALPHA
```

We will let "A" represent rotation by zero positions; "B", rotation by one position; and so forth. Thus the ciphertext for the message can be written directly below it:

```
NOWISTHETIMEFORCSFUN
ALPHAALPHAALPHAALPHA
--------------------
NZLPSTSTAIMPUVRCDUBN
```

It turns out that cryptographers traditionally break messages into 5-character groups to make things a bit easier to work with, so the pencil-and-paper method of doing the above would generate

```
NOWIS THETI MEFOR CSFUN
ALPHA ALPHA ALPHA ALPHA
-----------------------
NZLPS TSTAI MPUVR CDUBN
```

When the message is decrypted, the recipient must figure out where the blanks and punctuation should be.

We will take advantage of the computer's flexibility by implementing a slight variation on the Vignère cipher. Instead of encrypting a 26-letter alphabet, we will add support for blanks and encrypt 27 symbols: A through Z, plus blanks. We will keep the five-character grouping feature, however, so that we can't produce blanks in the encrypted output. Instead, our output alphabet will use a hyphen (-) to represent the 27th character. The message above, when encrypted through the sample solution with the key `ALPHA` and our 27-symbol alphabet thus becomes

2. The Caesar cipher is said to have been used by Julius Caesar to communicate with his army.

```
NOW IS THE TIME FOR CS FUN
ALPHAALPHAALPHAALPHAALPHAAL
---------------------------
NZKGISKHOE-DXTE-QCY-CCOMUNK
```

As you can see, we are still representing the 27th letter as blanks in the unencrypted text (the *plaintext*), but we are using the hyphen in the encrypted text (the *ciphertext*). Notice that there is a trailing blank in the plain text—this blank comes from the newline that appears at the end of any well-formed Unix file. A pencil-and-paper cryptographer would write the final message as `NZKGI SKHOE -DXTE -QCY- CCOMU NK`.

3 High-Level Interface

Your encryption program will prompt the user for a password, and then encode or decode a message contained in a file. There are some very specific requirements for how the program operates (see Section 4.1), designed so that your functions could conceivably be moved into a different program (e.g., a mail reader).

The program will follow a fairly standard Unix command-line interface style. There will be two ways to invoke your program, depending on whether you are encrypting or decrypting information.

3.1 Encryption

To encrypt, a user would type

```
unix% ./vignere -e file
```

or

```
unix% ./vignere -e -g 5 file
```

or

```
unix% ./vignere -g 5 -e file
```

The `-e` flag indicates that you want to encrypt `file` and write the result to `cout`. The optional `-g` switch specifies a grouping factor (i.e., generate code groups of 5 characters at a time). Your program should not assume that switches appear in any particular order.[3]

It turns out (try running `ps auxw` or `ps -ef`, depending on the machine you are using) that it is not a good idea to give passwords on the command line. Instead, your program should prompt for a password on `cerr`, using the string `Password:␣` (i.e., with

3. There is more information in Section 6.1 on how to process command-line arguments.

a trailing blank, and no newline), and read a password from cin (see Section 6.3). The password should be a word or phrase, terminated by a newline character.

In a real encryption program, you would turn off character echoing on the terminal so that nobody could look over the user's shoulder and get the password, but that's a bit of a pain for a CS 70 assignment, so your program should ignore that detail. The actual password will be somewhat modified from what the user types by converting lowercase to uppercase and changing nonalphabetic characters to blanks (the same rules will be applied to the message to be encrypted).

If the -g switch is not given, the output of the program should be a single line. If -g appears, the output should be divided into groups of the specified number of characters, with each group separated by a blank. When groups are being generated, your program should never generate an output line longer than 70 characters (unless the argument to -g is itself greater than 70). There should be no blanks at the end of the output lines.

Because we are working with a 27-character alphabet, your program will not be able to deal with lowercase characters and punctuation. Therefore, lowercase characters should be converted to uppercase, and all non-alphabetic characters should be treated as blanks (the same rules will apply to the password). For example, the string

```
CS 70 is really, really FUN!
```

would be encrypted exactly the same as if it read as follows (the line of dots is there to help you see all the blanks):

```
CS    IS REALLY  REALLY FUN
..........................
```

Because blanks are used for grouping, they cannot be part of your output alphabet. Instead, the meaningful part of the output of an encryption run should consist of the characters A through Z and the hyphen (-). The hyphen can either precede A or follow Z (the choice is yours, and the results should be equivalent in either case).

3.2 Decryption

Decryption is simpler than encryption. There is only one way to decrypt:

```
unix% ./vignere -d file
```

The specified `file` should be some previous output of your program. As with encryption, you should prompt the user for the password. The decrypted message should be written to cout. Since all formatting and punctuation have been lost, you should write the output as a single long line, and let the user worry about figuring it out.

3.3 Checking Arguments

Your program should verify that its arguments are correct. This task includes ensuring that exactly one of `-d` and `-e` is specified; that `-g` is not given with `-d`; that `-g` has an argument; that the name of a file to be encrypted or decrypted is given on the command line; and that no illegal (undefined) switches are given. If any of these rules are violated, you should print the following usage message on cerr:

```
Usage: ./vignere { -e [-g n] | -d } file
```

4 Required Techniques and Data

For this assignment, you will be required to use a number of data structures, data elements, and techniques that are not a direct consequence of the external interface requirements (see Section 3). The purpose of these extra restrictions is for you to gain experience with a number of important C++ data structures and techniques.

4.1 Required Techniques

There are two obvious ways in which a simple encryption program might work. Both ways assume that you already have the password stored internally. The first approach is to read a single character at a time from the input file, encrypt it, and write it to the output.

The second approach is to read the entire input file into a giant internal string. After reading the input, you can iterate through the string, encrypt each character, and store the encrypted version back into the string (modifying the character in place). Finally, you can write the encrypted string to the output.

You must use the second approach in this assignment, partly because it will give you more practice in using iterators, and partly because it will make your code cleaner.

An important design detail is that you should *not* insert grouping characters as you encrypt. Instead, you should implement the `-g` switch as part of your output routine.

Your program must store both the password and the string to be encrypted using the chunky string class described in Section 4.2.

4.2 Required *ChunkyString* Class

Your solution to this assignment *must* make use of a string class that stores a string as a linked list (using the STL's *list* class) of fixed-size "chunks", each of which is up to twelve characters long. Such a class is a compromise between storing the string as an array (which makes inserting characters expensive) and storing it as a linked list of single characters (which wastes memory).

Your string class, which must be called *ChunkyString*, should be defined as a class with two private data members. The first data member is of type *list*<*Chunk*>, where *Chunk* is a private inner class defined with the following data members:

```
class Chunk {
    ⋮
    private:
        unsigned int    length_;
        char            value_[CHUNKSIZE];
};
```

where CHUNKSIZE is a constant giving the maximum number of characters in each chunk (i.e., 12). The second data member is an integer holding the size of the string.

In this representation, a short string can fit in a single list element (one chunk). If the string is too large to fit into a single chunk, you create a second piece and then tie it together with the first, using the linked list as the underlying structure.

4.2.1 Operations Provided by the *ChunkyString* Class

To the outside world, your string class should work just like an implementation of a restricted version of the C++ *string* class (C++'s specification for the *string* type includes a myriad of useful member functions—thankfully, you won't be implementing all of them). Note that a user of your class should not be able to tell, based on the interface, that the strings are stored in pieces instead of as a single array of characters. This requirement has two implications:

1. You should support many of the "standard" string operations
2. Your private data should be *private*—invisible to the outside world

The exact subset of string operations you choose to support is up to you, and is somewhat dependent on what operations you need to implement to get your program to work. Your string class should not provide any operation that does not exist in the C++ *string* class.[4] The following operations are required:

- A type, *iterator*, with the usual STL iterator operations
- A default constructor
- A copy constructor and assignment operator
- A destructor that leaves the object in a sane (empty) state

4. You are allowed one specific exception to this rule: you can, if you choose, provide type conversion functions to allow *string*s to be converted to and from *ChunkyString*s (and you may provide either implicit conversion operators, or explicit (named) conversion functions). You may not use such type conversions to provide easy implementations of any *ChunkyString* operations, however.

- A push_back function that adds a single character to the end of the string
- A size function
- An insert function, where s.insert(i, c) inserts character c into the string s in front of the character that iterator i points to, and returns an iterator pointing to the newly inserted character
- An erase function, where s.erase(i) erases the character that iterator i points to, and returns an iterator pointing to the character that came after the one that was deleted
- A begin function that returns *iterator* pointing to the first character of the string
- An end function that returns *iterator* pointing "past the end" of the string
- A nonmember function **operator<<** that prints a *ChunkyString*

4.2.2 Complexity of *ChunkyString* Operations

All of the above functions listed in Section 4.2.1 should be implemented with the minimum complexity possible for this representation. In the list of functions above, the default constructor, push_back, insert, erase, and size should all be $\mathrm{O}(1)$. The copy constructor, assignment operator, and stream output are inherently $\mathrm{O}(n)$, where n is the size of the string, and should be implemented that way. (String concatenation is also $\mathrm{O}(n)$, if you choose to implement it.)

4.2.3 Substitutability of *string* for *ChunkyString*

For the operations it implements, the *ChunkyString* class should work just like the *string* class. While debugging your other code, you can use a **typedef** to make *ChunkyString* an alias for the C++ *string* class and thereby eliminate one possible source of bugs.

4.3 Required *VignereCipher* Class

The *VignereCipher* class must implement encoding and decoding of strings of characters. It is the only part of your code that is *required* to use the *ChunkyString* class.

4.3.1 Operations Provided by the *VignereCipher* Class

Your *VignereCipher* class must provide the following operations:

- A constructor, VignereCipher(***const*** *ChunkyString&* passphrase, bool encrypt), that accepts a passphrase and a boolean indicating whether it will be encrypting or decrypting text (true means encrypt)

- A function, *void* process(*ChunkyString&* text), which accepts text in the crypto alphabet (A-Z plus hyphen) and modifies that text so that it is encrypted or decrypted as necessary

Coding Stages

You are strongly advised to conduct your work according to the following plan, which contains some additional filename and compilation requirements.

C1. Draw a diagram of the *ChunkyString* data structure. Begin mulling how the iterator will work. The iterator will be tricky, as will insert and erase, so give yourself time to think it through. (You do not need to hand in this diagram, but it will help you to complete the assignment.)

C2. Create a `Makefile` for your code. (You will need to keep your `Makefile` up to date as you make changes.) In this assignment, you must specify specify `g++3` as the C++ compiler (in the `CXX` macro of your `Makefile`) rather than `g++`.

C3. Create the *VignereCipher* class. You *must* use the filenames `vignerecipher.hpp` and `vignerecipher.cpp` because this class will be tested separately. Your preliminary testing can define *ChunkyString* as an alias for the *string* class (using a **typedef**).

C4. Create the *ChunkyString* class, but do not implement insert or erase (write stub functions that do nothing). You *must* use the filenames `chunkystring.hpp` and `chunkystring.cpp` because this class will be tested separately. In addition, you *must* use `CHUNKYSTRING_HPP_INCLUDED` as the preprocessor variable in your include guard for `chunkystring.hpp`.

C5. Implement the main driver function, which must be compiled to an executable `vignere`. Include option processing, but ignore the grouping factor set by the `-g` switch. Test your program.

C6. Implement grouped output (`-g` and test your program again.

C7. Complete the *ChunkyString* class by implementing insert and erase.

C8. Clearly describe your design in your `README` file. If you have to make changes to the design after the submission deadline for the `README`, describe those in a `NEWS` file.

Note that your code must produce exactly the same output as the provided sample outputs.

5 Testing

`hw08-input1.txt` is a trivial sample input file. The corresponding output file, when encrypted using the `-g 5` flag and the passphrase `cs fun`, is `hw08-encrypt1.txt`. If that output file is fed back into the decryption routine, it generates a slightly modified version of the original, `hw08-decrypt1.txt`. Note that the decrypted version has a blank at the end (visible only if you use an editor or `cat -vet` to examine it), as discussed in Section 2.

Encrypting a different input file (`hw08-input2.txt`) with the passphrase

```
My roommate never studies, why should I?
```

but no `-g` switch produces a single, long output line (`hw08-encrypt2.txt`). The decrypted version of the file, `hw08-decrypt2.txt`, demonstrates that a certain amount of (presumably) useful information has been lost.

Finally, to make the assignment interesting, `hw08-encrypt-secret.txt` is an encrypted file for you to decode once your program is working correctly. The file was encrypted with the passphrase

```
Everything is Oll Korrect!
```

6 Useful Techniques

This section covers a number of techniques that you will need to use to implement your program.

6.1 Processing Command-Line Arguments

When a C or C++ program is invoked under Unix, you can give it one or more *arguments* (parameters) on the command line. The system preprocesses these arguments for you and makes them available to your `main` program as function parameters. You should declare your `main` program as follows:

```
int main(int argc, char* argv[])
{
    ⋮
}
```

The parameter `argc` gives the number of arguments that appeared on the command line, *including* the name of the program itself. So an invocation such as `./vignere` would produce an `argc` of 1, whereas `./vignere x y z` will set `argc` to 4.

The parameter `argv` is a primitive array of pointers to characters—that is, an array of C-style strings. `argv[0]` is always the name of the program as you invoked it—for

example, `./vignere`. Similarly, `argv[1]` is the first argument, expressed as a C-style string; `argv[2]` is the second argument; and so forth. For convenience, `argv[argc]` is guaranteed to be a `NULL` pointer. *It is illegal to refer to* `argv[i]` *when* `i` *is greater than* `argc`.

If dealing with primitive arrays and C-style strings seems cumbersome to you, you can trivially create a variable `args` of type *vector<string>* containing the program's arguments using the following variable declaration (in `main`):

```
vector<string> args(argv, argv+argc);
```

6.1.1 Phases of Command-Line Processing

In most cases, Unix programs handle their command-line arguments in two phases. In the first phase, the *options* are sorted out and recorded by setting various variables (often Boolean values). In the second phase, remaining non-option arguments are processed in the manner specified by the options.

First Phase of Command-Line Processing As mentioned above, in the first phase we process the *options* or *flags*, which typically begin with a dash (`-`). Some options (but not all) also require a *parameter* immediately after the flag. Because options can appear in any order, option processing is usually done in a loop similar to the following:

```
while (there are more arguments left)
    if (the next argument begins with a dash)
        process that argument
    else
        break;
```

The "process that argument" section is the interesting part of the code. There are two typical approaches: use a **switch** statement based on the second character of the option, or an **if**/**else if** sequence to detect which option has been specified and to handle it.

When an option takes a parameter, there are a couple of tricky aspects to processing it. Perhaps the sneakiest involves the way that the parameter is consumed. Because the parameter is a separate argument, you need to get rid of it as part of processing the option. You also have to make sure that it's actually there. Typically, you increment the loop index inside the option-processing code.

For example,

```
for (int argNo = 1; argNo < argc; argNo++) {
    // see above
    // ...
        // processing for option "-g":
        ++argNo;
        if (argNo >= argc)
            // Parameter is missing: issue usage error.
        // Parameter for -g is now in argv[argNo]. Because we incremented
        // argNo just now, and will increment it again in the for statement,
        // the parameter for -g will not be examined to see if it looks like
        // an option.
}
```

The other tricky aspect involves converting the parameter into a usable form. In particular, you'll need to convert the grouping factor from a string to an integer. Fortunately, there are library routines to do that for you. If you have decided to keep the command-line arguments as C-style strings, you can use the function atoi (which is declared via #include <cstdlib>, the name means "Ascii to integer"). You can use atoi as follows:

```
int usefulThing = 0;                    // Default value is zero
    ⋮
if (some useful decision)
    usefulThing = atoi(argv[argNo]);
```

If, on the other hand, you prefer to do things in C++ style and are representing the arguments as a *vector* of *strings*, you can perform the conversion as follows:

```
#include <stringstream>
    ⋮
int usefulThing = 0;                    // Default value is zero
    ⋮
if (some useful decision) {
    istringstream in(args[argNo]);
    in >> usefulThing);
}
```

You may be wondering what you should do if the argument given by the user is not an integer. In this assignment, we will assume that the user never makes this mistake. If you were developing a more robust solution, it would be easy to add checking to the C++-style code (by using the usual mechanisms to check the error status of the stream), whereas the C-style atoi function provides no mechanism to return error

status. If you were using C-style strings, you would need to switch to using the strtol library function.

These techniques may sound complicated, but they are actually easy to write. You have already seen examples of option processing in assignments 5 and 7.

Second Phase of Command-Line Processing The second phase involves handling the *positional* arguments, which are those arguments whose purpose is identified by their position on the command line. For example, the cp (copy) command in Unix accepts two positional arguments: the name of the file to copy from and the name of the file to copy into. In the command

```
unix% cp -p foo bar
```

you are asking the program to copy `foo` to `bar` using the `-p` (preserve attributes) option.

For this assignment, there is only one positional argument: the name of the file to be encrypted or decrypted. If you choose to have a separate option-processing function, it probably makes more sense to process the positional arguments inside main, instead of inside the option-processing function.

6.2 Converting Characters to Standard Format

We have already discussed most of the high-level interface of your program. It should accept both the password and the message to be encrypted in mixed case. For both strings, it should convert lowercase characters to uppercase and convert nonalphabetic characters to blanks. The `<cctype>` header file defines a couple of functions that will be useful in this regard:

isalpha(*ch*) Returns true if the character *ch* is alphabetic ("A" to "Z" or "a" to "z") and false otherwise

toupper(*ch*) Returns the uppercase equivalent of *ch* if *ch* is alphabetic

6.2.1 Doing Arithmetic Operations on Characters

This assignment would be much easier if characters were encoded in a friendly fashion. For example, if the letter "A" was represented by the number "0", "B" by "1", and so forth, up to "Z" = "25", with "26" representing a hyphen, it would be relatively easy to write the encryption code. Unfortunately, "A" is decimal 65 (in Ascii). However, there is an easy way to solve this problem: do arithmetic on characters. As an example, you can convert a character ch to the "A" = "0" scheme with the following code:

```
char convertedCharacter = ch - 'A';
```

This trick will work *only* if ch is one of the uppercase letters ("A" through "Z"). It fails if ch is a lowercase letter, a blank, or some other special character.

In the same way, you can convert a number between "0" and "25" back to an uppercase letter with

```
ch = convertedCharacter + 'A';
```

Again, this code will only work if convertedCharacter is "0" through "25". If convertedCharacter is some other value, this code will not generate a valid letter.[5]

6.3 How to Read Input

Just as output is written to an *ostream* such as cout or cerr, input is read from an *istream* such as cin. Doing so for this assignment will require that you use several C++ I/O features. Most of these features are enabled when you #include <iostream>.

To read the password, you will need to read one character at a time from cin. This task can be performed with the code like the following:

```
char nextCharacter;
while (cin.get(nextCharacter))
    // ... do stuff with nextCharacter
```

In this code, the loop will exit when there are no more characters available on cin (i.e., EOF was hit). *Note that EOF is not the same as the end of a line.* Because the passphrase is only one line long, you must detect the end of the line yourself.

The string to be encrypted must be read from a file whose name is given on the command line. Before you can read a named file, you must #include <fstream>. Then you must *open* the file, read it, and *close* it. In C++, doing so is easy: you open a file by creating an *ifstream* (for reading) or *ofstream* (for writing). The file is automatically closed when the associated *ifstream* or *ofstream* is destroyed.

Once you have created an *ifstream*, you can read from it just as if it were cin.

To make this explanation more concrete, suppose you want to read characters from a file named `myfile.txt`. You could write some code like

```
ifstream inputStream("myfile.txt");
if (!inputStream)
    // ... Oops, myfile.txt doesn't seem to be available!
char nextCharacter;
while (inputStream.get(nextCharacter))
    // ... do stuff with nextCharacter
```

5. This example code only works on computers that use Ascii or a similar encoding. For the purposes of this assignment, it's okay if your program only works on those computers. For a real project, of course, you would probably want to write more robust and portable code by abstracting knowledge of the character set into a separate function or using a library that already includes this knowledge.

Of course, in most cases you won't want to hardwire the filename into your code. (Even if you did, it counts as a "magic number", so you should define it as a ***const*** *string* rather than sticking it into the middle of your program.) Here is a very similar example:

```
bool readFile(const string& whichFile)
{
    ifstream inputStream(whichFile.c_str());
    if (!inputStream)
        return false;                    // Couldn't open the file
    char nextCharacter;
    while (inputStream.get(nextCharacter))
        // ...do stuff with nextCharacter
    return true;
}
```

In this case, the filename is stored in a ***const*** *string&* variable named whichFile, which is passed as a function argument. This function returns true if the file was successfully read. (It also doesn't return the string read, which makes it somewhat useless. Fixing that deficiency is left to you as an exercise.) The c_str member function of the *string* class converts a C++ *string* into a C-style *char**; this conversion is necessary because the *ifstream* constructor doesn't accept *string*s.

7 Tricky Stuff

As usual, there are parts of this assignment that are tricky. Here are a few:

- There is no law that says every "chunk" in your *ChunkyString* has to be 100% full. In some situations, you might find it useful to leave chunks partially empty, even in the middle of the list. *You will, however, lose points if you always store exactly one character per list node.*
- The string iterator is fairly hard to implement, but it has similarities to iterators we have covered in class. If you cannot make your iterator work correctly, make it at least not crash your program, and make your main program fall back to use the string class.
- Iterators don't always work very well in combination with ***const*** objects. The simplest solution is to just eliminate ***const*** if the compiler complains. A slightly cleaner solution is to "cast away the **const**" to get rid of the complaint.

  For example, if you had a ***const*** *ChunkyString* foo and wanted to create an iterator on it, you couldn't write

  ```
  ChunkyString::iterator i = foo.begin();
  ```

because the compiler would complain. To tell the compiler that you know best, use

```
ChunkyString::iterator i = (const_cast<ChunkyString&>(foo)).begin();
```

There are other solutions, but they are more complex.

We advise you not to make a *const_iterator* for your string class because of these issues.

- There are a few places where you will need to have two iterators (or an iterator and a loop index) running simultaneously but not in synchronization. One is when you are encrypting/decrypting; another is when you are generating output in groups. Debug these code sections carefully.
- Remember that you can use assignment to reset an iterator to the beginning.
- Your encryption/decryption function is required to work by modifying the text that is passed to it. This requirement forces you to write an iterator that allows the underlying object to be modified.
- You may find it easier to skip implementing the `-g` switch at first, and add it to your output routine later. Support for the `-g` switch is necessary to get full credit.
- When you first start testing, use the passphrase `A`. That will cause the output to be the same as the input, which will make your life much simpler. After you get that working, try `AA`, which should give the same result.
- Be sure to test your program with various settings of the `-g` switch, as well as without that flag.
- Be sure that your output routine limits line length to 70 characters when the `-g` switch is used (unless the argument to `-g` is greater than 70, of course).
- Be sure to test your program with illegal invocations, to verify that it produces a usage message in those cases.
- Be sure to test with a password that is longer than the input, one that is shorter, and one that is exactly the same length.
- You should also test both encryption and decryption with `/dev/null` (the empty file) as input. In both cases, the output should be a single empty line.