Programming Assignment 3
Nefarious Plots

Time due: 11:00 PM, Thursday, October 31

Introduction

You have been hired by Twisted Plots to write a program allowing an artist to draw lines to produce amazing computer graphics. The software works by having the artist type a series of drawing commands that dictate what lines it should plot on the screen.

The display grid of the system is comprised of "pixels" and is 20 pixels high by 30 pixels wide (pixel stands for "picture element"). The pixel in the upper left corner of the grid is considered to be at row 1, column 1. The pixel in the bottom right corner of the grid is at row 20, column 30. This image shows the grid and illustrates several coordinates:

```
           11111111112222222223
  123456789012345678901234567890
 +------------------------------+
1|A                            C|    A is at (1,1)
2|  B                           |    B is at (2,3)
3|                              |    C is at (1,30)
4|                              |    D is at (7,11)
5|                              |    E is at (19,28)
6|                              |    F is at (20,1)
7|          D                   |    G is at (20,30)
8|                              |
9|                              |
10|                             |
11|                             |
12|                             |
13|                             |
14|                             |
15|                             |
16|                             |
17|                             |
18|                             |
19|                          E  |
20|F                           G|
 +------------------------------+
```

The program you will eventually write will allow a user to type a series of drawing commands and see the results in the display grid. We know that this may be the most complex program that many of you will have written in your programming careers to date, so rather than our just detailing the visible behavior of the program in this spec and leaving you to come up with a program design, we will make it easier for you by decomposing the problem for you. We will specify some functions that you must implement in your solution. In addition to testing your program as a whole, we will also test each of those functions separately. That way, if your program doesn't work overall but you correctly implemented some of the functions, you'll earn some credit.

You'll work on this program in three phases:

1. Learn how to build a multifile project and to use the little graphics library that we've provided to help you with this project.
2. Write a function to plot a line in the display grid.
3. Write a function that takes a drawing command string and calls the line plotting function appropriately.

Phase 1: Learn how to use the graphics library

Do the Project 3 warmup to learn how to build a multifile program and use the graphics library. Make sure you can build and run your program under both g31 and either Visual C++ or clang++.

Phase 2: Write the line plotting function

You **must** implement the following function, using the exact function name, parameters types, and return type shown in this specification. (The parameter *names* may be different if you wish.) The function must not use any global variables other than the four constants `HORIZ`, `VERT`, `BG`, and `FG` defined below. In addition to testing your program as a whole, we will also test this line plotting function function separately. That way, if your program doesn't work overall but you correctly implemented this function, you'll still get some credit. All of the code you write must be in the file `plot.cpp`.

You may write functions in addition to the one listed here. While we won't test those additional functions separately, their use may help you structure your program more readably and avoid duplicating code.

This function must not cause anything to be written to `cout`. (Thus, it doesn't call the `draw` function.) Your function may, however, use the `setChar` function from the graphics library to plot characters in the grid. Your function might be called with a specification of a line that doesn't lie entirely in the grid. In that case it must not plot any characters of that line, and certainly should never ask `setChar` to plot a character outside the grid, since attempting to do so causes `setChar` to terminate the program.

Here is the function you must implement:

```
const int HORIZ = 0;
const int VERT = 1;

const int FG = 0;
const int BG = 1;

bool plotLine(int r, int c, int distance, int dir, char plotChar, int fgbg);
```

This function sets characters in the grid to form a line segment, one of whose endpoints is (r,c). The `dir` parameter should be either `HORIZ`, indicating a horizontal line, or `VERT`, indicating a vertical line. If the `distance` parameter is positive, the line extends rightward or downward from (r,c), depending on whether the line is horizontal or vertical, respectively; if `distance` is negative, the line extends leftward or upward from (r,c); if `distance` is zero, the "line" consists only of the point (r,c). The absolute value of `distance` is the distance from (r,c) to the other endpoint of the line.

The `plotChar` parameter is the character used to form the line. The `fgbg` parameter should be either `FG` or `BG`. The value `FG` indicates that the line should be plotted in the foreground: At every position in the line, the plot character replaces whatever character is in the grid at that point. The value `BG` indicates the line is to be plotted in the background: At every position in the line, if the character in the grid at that point is a space character (`' '`), then the plot character replaces it; otherwise, the character at that position is left unchanged.

The function returns `true` if it succeeds, and `false` otherwise. The function succeeds if and only if all of these conditions hold:

- The `dir` value is either `HORIZ` or `VERT` (i.e., either 0 or 1).
- The `fgbg` value is either `FG` or `BG` (i.e., either 0 or 1).
- Every position of the line to be plotted must be within the grid.
- The `plotChar` must be a character for which the library function `isprint` returns a value that tests as true (i.e. `if (isprint(`*`that character`*`))` would select the true branch of the if statement). `isprint` is declared in the header `<cctype>`, and returns an integer that tests as true for any "printable" character (letters, digits, punctuation, and the space character), and an integer that tests as false for any other character.

If the function does not succeed (i.e., at least one of the above conditions doesn't hold), then it must not modify any characters in the grid.

Here is an example:

```
clearGrid();
if ( ! plotLine(14, 8, 3, HORIZ, '*', FG))   // first call
   cout << "1) Plotting failed when it shouldn't have!" << endl;
if ( ! plotLine(15, 10, -2, VERT, '@', FG))  // second call
   cout << "2) Plotting failed when it shouldn't have!" << endl;
if ( ! plotLine(13, 8, 3, HORIZ, '#', BG))   // third call
   cout << "3) Plotting failed when it shouldn't have!" << endl;
if (plotLine(13, 29, 3, HORIZ, 'X', FG))     // fourth call
   cout << "4) Plotting succeeded when it shouldn't have!" << endl;
draw();
```

The first call sets grid positions (14, 8), (14, 9), (14, 10), and (14, 11) to `*` characters. Since the function returns true, the first failure message is not printed.

The second call sets grid positions (15, 10), (14, 10), and (13, 10) to `@` characters. Since the function returns true, the second failure message is not printed. Notice that the `*` character at (14, 10) is replaced by the `@` because of the `FG` parameter.

The third call sets grid positions (13, 8), (13, 9), and (13, 11) to `#` characters. Since the function returns true, the third failure message is not printed. Notice that the `@` character at (13, 10) is not replaced by the `#` because of the `BG` parameter. The characters at the other coordinates of the line were space characters, so they were replaced.

The fourth call does not change anything in the grid, because it's asking to set (13, 29), (13, 30), (13, 31), and (13, 32), the last two of which lie outside the grid. Since the function returns false, the fourth message is not printed.

Here's another example, one that uses the `assert` facility of the standard C++ library. If you #include the header `<cassert>` you can call `assert` in the following manner:

```
assert(some boolean expression);
```

During execution, if the expression is true, nothing happens and execution continues normally; if it is false, a diagnostic message is written telling you the text and location of the failed assertion, and the program is terminated. Here is the example:

```
clearGrid();
assert(plotLine(1, 1, 0, HORIZ, 'H', FG));
assert(plotLine(1, 2, 0, HORIZ, 'i', FG));
assert(plotLine(1, 3, 0, HORIZ, '!', FG));
draw();  //  displays  Hi!  in the top row of the grid
assert(plotLine(1, 3, 0, HORIZ, ' ', FG));
draw();  //  displays  Hi   in the top row of the grid
assert(plotLine(1, 1, 10, HORIZ, ' ', BG));
draw();  //  displays  Hi   in the top row of the grid
assert( ! plotLine(1, 1, 10, HORIZ, '\n', FG));
draw();  //  displays  Hi   in the top row of the grid
```

Writing space characters in the foreground replace characters with spaces. Writing space characters in the background has no visible effect: Non-space characters are unchanged, and space characters are replaced by space characters, which doesn't effectively change them either. The last assertion asserts that calling `plotLine` with a newline character returns false (since newline is not a character for which `isprint` returns a value that tests as true), and the final call to `draw` shows that no grid characters were changed as a result of that last call to `plotLine`.

Phase 3: Write the command string interpreter

The final version of the program will repeatedly:

1. Prompt the user for a *command string*, which is a line of text containing a sequence of *plotting commands*.
2. Read a command string.
3. If the command string is the empty string, terminate the program. Otherwise, for each plotting command in the command string, perform the action specified by that plotting command.
4. After all the plotting commands in the command string have been performed, draw the resulting grid.

Imagine a multicolor pen that you can use to draw lines, except that instead depositing inks of different colors onto paper as you move it, it places characters in a grid as you move it. Just as a real pen at any particular time has a color that it's set to and a position on the paper, we will consider our virtual pen to have a *current character* that it plots with, a *current mode* (either foreground mode or background mode), and a *current position* in the grid.

When the program starts, the current character is `*` and the current mode is foreground mode. This can be changed by a plotting command described below. For each command string, the current position is set to row 1, column 1 of the grid, and then the plotting commands in the command string are performed. Plotting commands in the command string will cause the current position to change as the virtual pen moves as it plots draw lines.

Here is an example of a command string, one that contains five plotting commands:

```
h12V3H-1B@v-3
```

Each plotting command starts with a single letter indicating the action to perform (e.g, `H` or `h` to draw a horizontal line). (The action is the same whether the letter is upper or lower case.) Following the letter is the argument to that command. (The argument for each command are described later.) Arguments that are numbers are written with a possible minus sign character (`-`) followed by exactly one or two digits. The command string above contains the five plotting commands:

```
h12
V3
H-1
B@
v-3
```

The first plotting command above consists of the letter `h` and the argument 12. Starting from the current position, initially (1, 1), it plots a horizontal line of the current character, initially `*`, in the current mode, initially foreground mode. Because the argument is positive, the line is plotted in the rightward direction. The line ends 12 positions to the right of where it started, and that becomes the new current position. So the effect of this plotting command is to set positions (1, 1), (1, 2), (1, 3), ..., (1, 12), and (1, 13) of the grid to `*` and the current position to (1, 13).

The second plotting command above consists of the letter `V` and the argument 3. Starting from the current position, (1, 13), this plots a vertical line of `*` downward. The effect of this plotting command is to set positions (1, 13), (2, 13), (3, 13), and (4, 13) of the grid to `*` and the current position to (4, 13).

The third plotting command above consists of the letter `H` and the argument -1. Starting from the current position, (4, 13), this plots a horizontal line of `*` leftward. The effect of this plotting command is to set positions (4, 13) and (4, 12) of the grid to `*` and the current position to (4, 12).

The fourth plotting command above consists of the letter `B` and the argument `@`. This plotting command does not plot any characters or change the current position. Instead, it changes the current mode to background mode and the current character to `@`.

The fifth plotting command above consists of the letter `v` and the argument -3. Starting from the current position, (4, 12), this plots a vertical line of `@`, the current character, upward in background mode, the current mode. In background mode, only space characters in the grid are replaced by the current character. The effect of this command is to leave position (4, 12) unchanged (since it contains a `*`), set (3, 12) and (2, 12) to `@` (since they contain spaces), and leave (1, 12) unchanged (since it contains a `*` placed there by the first plotting command). The current position is now (1, 12); the virtual pen is there even though the character there was unchanged.

After all the plotting commands in the command string above are performed, the grid is drawn, showing this on the screen:

```
          11111111112222222223
 123456789012345678901234567890
+------------------------------+
1|*************                 |
2|            @*                |
3|            @*                |
4|            **                |
5|                              |
6|                              |
7|                              |
8|                              |
9|                              |
10|                             |
11|                             |
12|                             |
13|                             |
14|                             |
15|                             |
16|                             |
17|                             |
18|                             |
19|                             |
20|                             |
 +------------------------------+
```

Suppose the next command string the user enters is

```
v2b h12fHh1fih0
```

The current character is still `@` and the current mode is still background mode; those always carry over from the previous command string. However, at the start of every command string, the current position is (1, 1).

The seven plotting commands in this command string and their effects are

```
v2
b      (the b is followed by a space character)
h12
fH
h1
fi
h0
```

The `v2` leaves (1, 1) unchanged, since we're still in background mode, and sets (2, 1) and (3, 1) to the current character, `@`, making the current position (3, 1). The second command puts us in background mode with the space character as the current character, and the third command makes the current position (3, 13). Notice that setting the current character to a space character in background mode lets you move the virtual pen without changing any characters in the grid, rather like lifting a real pen off the paper to move it without drawing anything.

The `fH` puts us in foreground mode with `H` as the current character, and the `h1` puts the `H` in (3, 13) and (3, 14); we'll overwrite the second one when the `fi` sets the current character to `i` and the `h0` sets (3, 14) to `i`, leaving the current position as (3, 14). After it performs the plotting commands in the command string, the program draws the grid, showing

```
          11111111112222222223
 12345678901234567890123456789 0
+------------------------------+
1|*************                |
2|@           @*                |
3|@           @Hi               |
4|            **                |
5|                              |
6|                              |
7|                              |
8|                              |
9|                              |
10|                             |
11|                             |
12|                             |
13|                             |
14|                             |
15|                             |
16|                             |
17|                             |
18|                             |
19|                             |
20|                             |
  +------------------------------+
```

If the next command string the user enters is

```
cv14
```

the program interprets it as consisting of the two plotting commands

```
c
v14
```

The `c` command clears the entire grid (resulting in every character of the grid being a space character) and sets the current position to (1, 1), the current character to `*`, and the current mode to foreground mode. The `v14` plots a vertical line. The grid is then drawn, showing

```
          11111111112222222223
 123456789012345678901234567890
 +------------------------------+
1|*                             |
2|*                             |
3|*                             |
4|*                             |
5|*                             |
6|*                             |
7|*                             |
8|*                             |
9|*                             |
10|*                            |
11|*                            |
12|*                            |
13|*                            |
14|*                            |
15|*                            |
16|                             |
17|                             |
18|                             |
19|                             |
20|                             |
 +------------------------------+
```

Syntax for the plotting commands

A command string is a sequence of one or more plotting commands. There must be no characters between its component plotting commands, and plotting commands must not contain any characters other than those specified by the definition of their syntax below. For example, `H25H-10` is a valid command string, but these are not:

```
H25,H-10     (Comma not allowed between plotting commands.)
H25 H-10     (Space not allowed between plotting commands.)
H+25H-10     (Plus not allowed in H command.)
```

List of required plotting commands

Your program must support the plotting commands listed below.

Horizontal Line command

This command consists of an upper or lower case `H` immediately followed by an argument in one of the following forms:

- one digit character
- two digit characters
- a minus sign followed by one digit character
- a minus sign followed by two digit characters

This command sets characters in the grid to form a horizontal line segment, one of whose endpoints is the current position. If the command does not contain a minus sign, the line extends rightward from the current position; if the command contains a minus sign, the line extends leftward. The one or two digits in the command indicate the distance from the current position to the other endpoint of the line. Notice that the plotting commands `H3` and `H03` have the same effect. If the one or two digits are `0` or `00`, the "line" consists only of the current position.

If the current mode is foreground mode, the character in the grid at every position in the line is replaced by the current character. If the current mode is background mode, only the space characters in the grid that are at positions in the line are replaced by the current character. After this plotting command is performed, the current position is the position of the other endpoint of the line; unless the digits in the command were `0` or `00`, this will be a different position from what it was before performing the command.

If performing this command would attempt to plot a character outside the grid, then this command must not modify any characters in the grid and must not change the current position. For example, if the current position is (13, 29), you would not plot any characters for the plotting command `H3`, and the current position would remain (13, 29). Notice that the command `H76` is a syntactically valid plotting command, but performing this command could never result in any characters being plotted.

Vertical Line command

This command consists of an upper or lower case `V` immediately followed by an argument in one of the same four forms as for the `H` command.

This command sets characters in the grid to form a vertical line segment, one of whose endpoints is the current position. If the command does not contain a minus sign, the line extends downward from the current position; if the command contains a minus sign, the line extends upward. The one or two digits in the command indicate the distance from the current position to the other endpoint of the line. Notice that the plotting commands `V3` and `V03` have the same effect. If the one or two digits are `0` or `00`, the "line" consists only of the current position.

If the current mode is foreground mode, the character in the grid at every position in the line is replaced by the current character. If the current mode is background mode, only the space characters in the grid that are at positions in the line are replaced by the current character. After this plotting command is performed, the current position is the position of the other endpoint of the line; unless the digits in the command were `0` or `00`, this will be a different position from what it was before performing the command.

If performing this command would attempt to plot a character outside the grid, then this command must not modify any characters in the grid and must not change the current position. For example, if the current position is (13, 29), you would not plot any characters for the plotting command `V8`, and the current position would remain (13, 29). Notice that the command `V76` is a syntactically valid plotting command, but performing this command could never result in any characters being plotted.

Foreground command

This command consists of an upper or lower case `F` immediately followed by one character. That character must be one for which the library function `isprint` returns a value that tests as true.

Executing this command sets the mode to foreground mode and the current character to the character that follows the upper or lower case `F`.

Background command

This command consists of an upper or lower case `B` immediately followed by one character. That character must be one for which the library function `isprint` returns a value that tests as true.

Executing this command sets the mode to background mode and the current character to the character that follows the upper or lower case `B`.

Clear command

This command consists of an upper or lower case `C`.

This command sets every character of the grid to be a space character and sets the current position to (1, 1), the current character to `*`, and the current mode to foreground mode.

Error reporting

If a command string has invalid syntax, your program must print this error message:

```
Syntax error at position n
```

where *n* is the position in the string of the error, considering position 1 to be the first character of the string, as a normal human non-programmer artist reading the message would. (Note that this is unlike the C++ convention, which call the first character of a string position 0.) If a command string has invalid syntax, report only the leftmost syntax error encountered in a left-to-right traversal of the command string. If the error occurs because the command string ends but a character is expected, the position number will be one more than the length of the string. Here are some kinds of syntax errors that may occur:

- Where a plotting command is expected to begin, a character that is not a valid command letter appears, e.g., `Q` or `6` or `%` instead of `H`, `h`, `V`, `v`, etc.
- Extra characters appear, such as a space between a plotting command letter `H` and its numeric argument.
- Missing or extra arguments.
- A numeric argument that does not have exactly one or two digits.
- A non-printable character (i.e., one for which `isprint` returns a value that tests as false) as the argument to the `F` or `B` command.

For example, here are the messages to write for these command strings (we've annotated some of the messages; you don't write the annotations, of course):

```
F#H+25H?V3! Syntax error at position 4   leftmost syntax error
B@H         Syntax error at position 4   expecting - or digit after H
C12         Syntax error at position 2   C is one command; 1 can't start a command
Q3V4#       Syntax error at position 1
V03C H123#  Syntax error at position 5
H18H-123#   Syntax error at position 8   H-12 is one command; 3 can't start a command
H5H-1-2     Syntax error at position 6   H-1 is one command; - can't start a command
FH8         Syntax error at position 3   FH is one command; 8 can't start a command
```

If a command string has valid syntax, but performing a plotting command would attempt to plot a character outside the grid, your program must print this error message:

```
Cannot perform command at position n
```

where *n* is the position in the string of the letter that starts the earliest occurring command that would make this attempt, considering position 1 to be the first character of the string. For example, the command string `H28V10H5V86` would cause you to write

```
Cannot perform command at position 7
```

because when the current position is (11, 29), the `H5` would not succeed, and the position of that `H` is 7 when considering the string to start at position 1. Notice that the `V86` plotting command is syntactically valid, but would also not succeed; no message is written about it in this case, because the unsuccessful `H5` occurred earlier in the string, and you are to write only one error message for a given command string. Notice also that for the command string `V86F` your program must print `Syntax error at position 5`, since the command string does not have valid syntax; problematic performances are reported only for syntactically valid strings.

The main routine

The main routine of the program you turn in must clear the grid, and then repeatedly prompt the user for a command string by writing "`Enter a command string: `", read a line with the command string, and if the string is not empty, call the `performCommands` function to parse the command string and perform the plotting commands. If `performCommands` returns an error indication, your main routine must write an error message; otherwise, your main routine draws the grid in response to that command string. An empty command string causes your main routine to exit normally.

This main routine satisfies this requirement if you implement `performCommands` as specified below.

```
int main()
{
    setSize(20, 30);
    char currentChar = '*';
    int currentMode = FG;
    for (;;)
    {
        cout << "Enter a command string: ";
        string cmd;
        getline(cin, cmd);
        if (cmd == "")
            break;
        int position;
        int status = performCommands(cmd, currentChar, currentMode, position);
        switch (status)
        {
          case 0:
            draw();
            break;
          case 1:
            cout << "Syntax error at position " << position+1 << endl;
            break;
          case 2:
            cout << "Cannot perform command at position " << position+1 << endl;
            break;
          default:
              // It should be impossible to get here.
            cerr << "performCommands returned " << status << "!" << endl;
        }
    }
}
```

The `performCommands` function

You must implement this function:

```
int performCommands(string commandString, char& plotChar, int& mode, int& badPos)
```

This function parses the command string, performing the indicated plotting commands by calling the `plotLine` function described in Phase 2. The current position starts at (1, 1), the current character starts off as `plotChar`, and the current mode starts as `mode`. If the command string is syntactically valid, and all plotting commands are performed successfully, this function returns 0 when the plotting actions are completed and does not modify the `badPos` parameter; `plotChar` will be the (possibly changed) current character and `mode` will be the (possibly changed) current mode resulting from performing the commands.

If the command string is syntactically invalid, this function returns 1 after setting `badPos` to the position in the string of the leftmost syntax error, considering 0 to be the position of the first character of the command string. (As is often the case in programs, the start-at-0 convention used by this function internal to your program doesn't match the start-at-1 convention the main routine uses in presenting a message to the outside world. The main routine deals with that adjustment.)

If the command string is syntactically valid, but performing a plotting command would attempt to plot a character outside the grid, this function returns 2 after setting `badPos` to the position in the string of the letter that starts the earliest occurring command that would make this attempt, considering 0 to be the position of the first character of the command string.

If a command string has an error, it is your choice as to whether or not the plotting commands preceding the point of error are performed, and whether or not `plotChar` and `mode` are modified.

Although the main routine we provide will never call `performCommands` with an empty string, you must still handle that case correctly: The empty string is a syntactically valid string, and performing it performs zero plotting actions, all zero of them successfully.

You may write additional functions that `performCommands` calls to help it do its job. While we won't test those additional functions separately, their use might help you structure `performCommands` more readably and avoid duplicating code.

Neither the `performCommands` function nor any functions it calls may use any global variables whose values may be changed during execution. Global *constants*, like `MAXROWS` and `HORIZ`, are fine.

The `performCommands` function must not cause `setChar` to be called except as a result of `plotLine` being called. In other words, `performCommands` uses only `plotLine` do any character plotting; it must not bypass `plotLine` and try to plot characters by calling `setChar` directly.

Programming Guidelines

Your program must build successfully under both g31 and either Visual C++ or clang++. Try to ensure that your functions do something reasonable for at least a few test cases. That way, you can get some partial credit for a solution that does not meet the entire specification.

When you turn in your solution, none of the required functions except `main` may write any output to `cout`. (Of course, during development, you may have them write whatever you like (perhaps by calling `draw`) to help you debug.) If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. `cerr` is the standard error destination; items written to it by default go to the screen. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish. (Note that the `draw` functions write to `cout`, not `cerr`. If an expression passed to `assert` evaluates to false, a diagnostic message is written to `cerr`, not `cout`.)

You must not change the types of the parameters to `plotLine` and `performCommands`. In particular, you must not change the first two arguments of `plotLine` to be of type `int&` instead of `int`.

Except temporarily for your own testing purposes, do not make any changes whatsoever to the `grid.h` or `grid.cpp` files. You will turn in only `plot.cpp`, and we'll use our own versions of `grid.h` and `grid.cpp` when testing your code.

The correctness of your program must not depend on undefined program behavior. Your program could not, for example, assume anything about `c`'s value at the point indicated, or even whether or not the program crashes:

```
int main()
{
    string s = "Hello";
    char c = s[6];  // c's value is undefined
    …
```

Running under g31 may not normally detect *all* occurrences of undefined behavior, but if you build your program with

```
g31 -D_GLIBCXX_DEBUG -o plot plot.cpp grid.cpp
```

and run it for a test case that accesses a bad string position, the program will crash, which indicates you have a problem.

What to turn in

What you will turn in for this assignment is a zip file containing these two files and nothing more:

1. A text file named **plot.cpp** that contains the source code for your C++ program. Your source code should have helpful comments that tell the purpose of the major program segments and explain any tricky code.

2. A file named **report.docx** or **report.doc** (in Microsoft Word format) or **report.txt** (an ordinary text file) that contains:
    a. A brief description of notable obstacles you overcame.
    b. A description of the design of your program. You should use pseudocode in this description where it clarifies the presentation.
    c. A list of the test data that could be used to thoroughly test your program, along with the reason for each test. You don't have to include the results of the tests, but you must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.) If you use the `assert` style above for writing your test code, you can copy those `asserts`, along with a very brief comment about what it's testing for. Notice that most of this portion of your report can be written just after reading the requirements in this specification, before you even start designing your program.

Your zip file must *not* contain the `grid.h` and `grid.cpp` files that we supply. When we test your program, we will use our own versions of these files that we have specially instrumented for automated testing.

By Wednesday, October 30, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in, because we will not accept excuses like "My network connection at home was down, and I didn't have a way to copy my files and bring them to a SEASnet machine." There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later.