Assignment 3

Due Wednesday by 11:55pm **Points** 60

For the following projects (and all future projects in this course):

- **Don't use the conditional operator (discussed on pages 207-209 of the textbook).** It may seem "efficient" because it's compact, but it's compact to the point of making code less readable (there are those who disagree with me, but for this class don't use it).

Project 3.a

Write a program that asks the user how many integers they would like to enter. You can assume they will enter a number >= 1. The program will then prompt the user to enter that many integers. After all the numbers have been entered, the program should display the largest and smallest of those numbers (no, you cannot use arrays, or any other material we haven't covered). When you run your program it should match the following format:

```
How many integers would you like to enter?
4
Please enter 4 integers.
-4
105
2
-7
min: -7
max: 105
```

The file must be named: **minmax.cpp**.

Project 3.b

Write a program that prints "Please enter your filename.", reads in the name of the file, and then tries to open it. If the input file is there and can be opened, the program should read the list of integers in the file, which will have one integer per line as in the following example:

```
14
9
12
-6
-30
8
109
```

Note: This example is just to demonstrate the format of the input file. Your program would not print these values out to the console or to the output file.

The program will then add together all the integers in the file, create an output file called sum.txt, write the sum to that file (just that number - no additional text), and then print (to the console) "result written to sum.txt". Remember to close both the input and output files. If the input file is not there (or is there but couldn't be opened for some reason), the program should just print out "could not access file".

Using a string variable as the parameter of the open function is a C++11 feature, so to compile, you'll need the "-std=c++0x" flag as discussed in the section "Note on different C++ standards".

The file must be named: **fileAdder.cpp**

Project 3.c

Write a program that prompts the user for an integer that the player (maybe the user, maybe someone else) will try to guess. If the player's guess is higher than the target number, the program should display "too high" If the user's guess is lower than the target number, the program should display "too low" The program should use a loop that repeats until the user correctly guesses the number. Then the program should print how many guesses it took. When you run your program it should match the following format:

```
Enter the number for the player to guess.
-12
Enter your guess.
100
Too high - try again.
50
Too high - try again.
-2000
Too low - try again.
-12
You guessed it in 4 tries.
```

The file must be named: **numGuess.cpp**

Assignment 3a project plan

Submit Assignment

Due Sunday by 11:55pm **Points** 20 **Submitting** a file upload **File Types** pdf

This part must be submitted as a pdf here in Canvas.

The page called "Problem Solving" discusses five steps for an approach to solving a problem and learning from the process. For project **3a** you will create a project plan based on steps two and three of that process. You will then write code based on your design (the fourth step), which will be submitted on Mimir as usual (together with the code for projects 3b and 3c). The specifications for project 3a (and 3b and 3c) are in Assignment 3. When doing your coding, you're allowed to deviate from the plan you made. You will also write your reflections on the problem-solving process (the fifth step).

Your design document should contain the two following sections:

- **Testing Plan:** Design some tests you can perform to verify that your program meets the given specifications. They should be representative enough that if your program passes them, you would have a high level of confidence that your program is ready to be submitted. Having just a couple of tests will not meet that criterion. Create a table with two columns: one for the description of each test and one for the expected results of each test. These are not meant to be general descriptions, but rather specific concrete examples of inputs and outputs. Try to come up with tests that will check different aspects of the program's functioning, and be creative in thinking of situations that might "break" the program. You do not need to test whether the inputs are legal - just whether the program behaves correctly when given legal inputs.
- **Design:** Describe or draw out your design for how the program should behave using pseudocode or flowcharts, following the Pseudocode & Flowchart Guidelines. You do not need to do both pseudocode and a flowchart - just one or the other.

Below is an example for a program that is supposed to determine whether a positive integer entered by the user is prime. If so, it should output “prime”, otherwise it should output “not prime”.

Example Testing Plan

User enters 1	Output “not prime”
User enters 2	Output “prime”
User enters 3	Ouptut “prime”

User enters 9	Output “not prime”
User enters 187	Output “not prime”
User enters 9001	Output “prime”
User enters 1000001	Output “not prime”
User enters 1000003	Output “prime”

Example Design

```
prompt user to enter a positive integer
initialize input to the user-entered value
initialize result to true
for all positive integers from 2 to input-1
            if the remainder of input divided by the current integer is zero
                        set result to false
output result
```

Understanding, Testing Plan and Design

Criteria	Ratings			Pts
coverage of proposed tests	5.0 pts proposed tests are representative of the specifications	3.0 pts proposed tests do not adequately cover the specifications	0.0 pts proposed tests are absent or inadequate	5.0 pts
correctness of expected results	5.0 pts expected results are correct for proposed tests	3.0 pts there are flaws in the expected results	0.0 pts expected results are absent or inadequate	5.0 pts
design adheres to pseudocode & flowchart guidelines	5.0 pts pseudocode is specific without looking like C++; flowchart notation is used correctly	3.0 pts pseudocode is vague or looks similar to C++; incorrect use of flowchart notation	0.0 pts pseudocode is quite vague or is essentially C++ code; flowchart notation is largely incorrect	5.0 pts
correctness of design logic	5.0 pts design logic seems sound	3.0 pts design logic is somewhat adequate, but has gaps or flaws	0.0 pts design logic has many gaps or flaws	5.0 pts
				Total Points: 20.0

Problem Solving

How do you get from a set of requirements to a working program? There is no cut-and-dried recipe to follow. If there were, someone would write a program to do it, and computer programmers would be obsolete. However, there are general principles you can follow that will help to guide you toward a solution. (Based on *How to Solve it* by George Pólya.)

1. Make sure you have a clear understanding of all the requirements. In the real world, this is sometimes the most challenging part.
2. Plan tests you can use to check potential solutions for correctness.
3. Design your approach. One of the most commonly used methods for this is *stepwise refinement*. This is a process of breaking up the original problem into some number of smaller sub-problems. Next, you take each sub-problem and break it up into even smaller sub-problems. This is repeated until you get sub-problems that are small enough to be translated into code.
4. Translate your design into computer code (checking it with the tests from #2).
5. Reflect on how you arrived at your solution. What worked and what didn't? How can this experience help you solve future problems?

As an example of stepwise refinement, consider the problem of wanting to paint your house, which is a big job. It might seem overwhelming at first, but you can break that top-level problem into the following sub-problems:

- buy paint
- paint house
- clean up

Those sub-problems can in turn be broken down further:

- buy paint
  - choose colors
  - choose brand of paint
- paint house
  - tape over parts that shouldn't get painted
  - purchase/borrow supplies (ladders, rollers, etc.)
  - bribe friends with pizza
  - schedule a day/time
  - actually paint the house
- clean up
  - return/put away supplies
  - un-tape the taped bits
  - finish any fine detail work

This process continues until you've defined everything to a level of detail where you feel comfortable that you know everything that needs to happen. For programming, this process often involves pseudocode or flowcharts, which embody the steps required and which (once you've refined them to a sufficient level of detail) can then be translated into a computer language. See the Pseudocode and Flowchart Guidelines in this module.

Pseudocode & Flowchart Guidelines

Adapted from John Dalbey, Bob Roggio and Farshad Barahimi.

Pseudocode and flowcharts allow the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudocode or flowchart needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a matter of translating from the pseudocode or flowchart into source code.

Pseudocode

In general the vocabulary used in the pseudocode should be the vocabulary of the problem domain, not of the implementation domain. The pseudocode is a narrative for someone who knows the requirements (problem domain) and is trying to learn how the solution can be organized. That is, someone who is a non-programmer should be able to read and understand your pseudocode. For example:

Extract the next word from the line (good)
Set word to get next token (poor)

Append the file extension to the name (good)
Name = name + extension (poor)

For all the characters in the name (good)
For character = first to last (poor)

Note that the logic must be decomposed to the level of a single loop or decision. Thus "Search the list and find the customer with highest balance" is too vague because it takes a loop **and** a nested decision to implement it.

Each textbook and each individual designer may have their own personal style of pseudocode. Pseudocode is not a rigorous notation, since it is read by other people, not by the computer, but **this is the style you should follow for this class.**

Examples:

```
Get student's grade from user

If student's grade >= 60
```

```
    Print "passed"
else
    Print "failed"
```

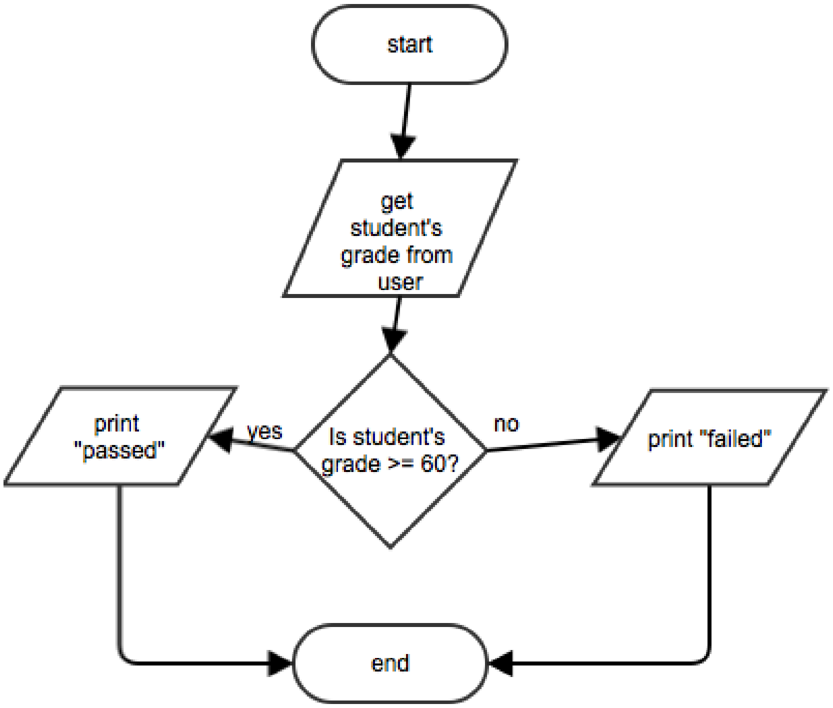
```
Set total to 0
Set grade counter to 1
While grade counter <= 10
    Input the next grade
    Add the grade to the total
    Add 1 to the grade counter
Set the class average to the total divided by 10
Print the class average
```

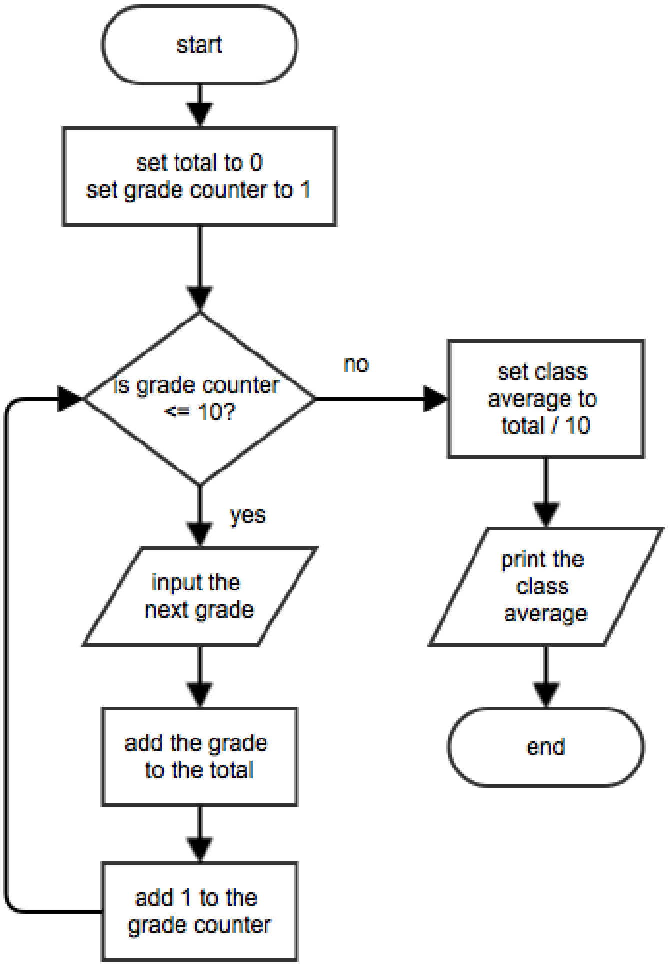
Flowcharts

A flowchart depicts an algorithm in a graphical format. Here are descriptions of the most common symbols:

- Oval: start and end
- Arrow: represents flow of control in a program.
- Rectangles: shows a computation or a specific process, for example "multiply X by 2".
- Parallelogram: Used for getting **input** from user or sending **output** to user.
- Diamond: Used for conditional flow control where a program has to decide which way to go, for example: if X is divisible by 2 do this thing; if not, do this other thing. Without conditional flow control, a program would have just one path from start to end, but with it there may be many different paths from start to end.

Examples:


start
get student's grade from user
Is student's grade >= 60?
yes
no
print "passed"
print "failed"
end


start
set total to 0
set grade counter to 1
is grade counter <= 10?
no
yes
set class average to total / 10
print the class average
end
input the next grade
add the grade to the total
add 1 to the grade counter

Assignment 3a reflection

Submit Assignment

Due Wednesday by 11:55pm **Points** 10 **Submitting** a file upload **File Types** pdf

This part must be submitted as a pdf here in Canvas.

In this part of the assignment you'll write a reflection on your experience this week with creating a project plan for project **3a.** When writing your reflection, you don't have to do it in a question/answer format, but the following questions should be somehow addressed:

- **Understanding**: What did you learn about the problem as you went? Why or how did you learn it?
- **Testing plan**: What tests didn't work out the way you expected? What alterations did you have to make to your program due to failed tests? How could your planned tests have been more complete?
- **Design**: What was missing or needed to be altered from your initial design, and why?
- **Implementation**: What problems did you encounter during implementation? How were you able to solve those problems? What outside sources (sites, books, or other materials) did you find helpful?
- **Improvement**: How can you generalize any parts of your problem solving experience in a way that might help you on future assignments?

Reflection on project plan

About Exam 1

Exam 1 will consist of about thirty questions, many of them involving code snippets. You will need to understand how the code will behave just from reading it, so you will want to practice your hand-tracing skills. **This is very different from being able to figure out how code behaves when you have the benefit of a compiler, the textbook and the internet.** Don't assume that you know what the intent of the code is - read carefully and trace through each step. You will be allowed to use blank scratch paper (if proctored in person) or a whiteboard (if you use Proctor U). There is a time limit of 100 minutes (1 hour and 40 minutes) to complete the exam (I don't expect it to take most of you this long, but proctored testing tends to take a bit more time due to environment and the possibility of technical difficulties). Remember that this exam is worth 20% of your course grade.

The core topics you should focus on include:

- comments
- types and variables
- C++ strings (including length(), at(), and string concatenation)
- arithmetic operators and expressions
- input and output (you won't be tested on <iomanip> and formatted output)
- constants
- if-else statements
- nested if-else statements
- logical operators and expressions
- switch statements
- enums
- for loops
- while loops
- do-while loops
- nested loops
- blocks and scope
- file handling (opening files for input or output, reading from or writing to a file, closing a file, testing for EOF, testing whether a file was opened successfully)

A couple of example questions

1) What is the result of the example below, where a semicolon immediately follows the condition of the if statement?

```
if (14 % 2 == 1);
  cout << "14 is odd" << endl;
```

a) no output   b) it prints "14 is odd"   c) it prints "14 is even"   d) an error message

2) What would be the result of the following code?

```
string str = "apothecary";
cout << str.at(str.length()) << endl;
```

a) it would print "a" b) it would print "r" c) it would print "y" d) an error message

3) Assume that we initialize *num* prior to this section of code. Which value of num will cause it to print out "true"?

```
bool result = true;
for (int i=2; i<num; i++)
   if (num % i == 0)
      result = false;
if (result)
   cout << "true" << endl;
else
   cout << "false" << endl;
```

a) 6 b) 12 c) 17 d) 63

4) The following code is meant to write some text to a file, but right now it won't. Which line needs to be changed for that to happen?

```
ifstream myfile;                    //  line 1
myfile.open("greeting.txt");        //  line 2
myfile << "Hello world.\n";         //  line 3
myfile.close();                     //  line 4
```

a) line 1 b) line 2 c) line 3 d) line 4

5) Assume a variable named count has already been declared and initialized to 6. What would be the result of the following code?

```
int count = 1;
if (2 < count < 9)
 cout << "true" << endl;
else
 cout << "false" << endl;
```

a) It would print "true". b) It would print "false". c) There would be a compile error.

Criteria	Ratings			Pts
reflection on understanding	2.0 pts thoughtful response that includes support/reasoning/examples and is clearly expressed	1.0 pts response that is cursory, minimal, vague or disorganized	0.0 pts no response	2.0 pts
reflection on testing plan	2.0 pts thoughtful response that includes support/reasoning/examples and is clearly expressed	1.0 pts response that is cursory, minimal, vague or disorganized	0.0 pts no response	2.0 pts
reflection on design	2.0 pts thoughtful response that includes support/reasoning/examples and is clearly expressed	1.0 pts response that is cursory, minimal, vague or disorganized	0.0 pts no response	2.0 pts
reflection on implementation	2.0 pts thoughtful response that includes support/reasoning/examples and is clearly expressed	1.0 pts response that is cursory, minimal, vague or disorganized	0.0 pts no response	2.0 pts
reflection on improvement	2.0 pts thoughtful response that includes support/reasoning/examples and is clearly expressed	1.0 pts response that is cursory, minimal, vague or disorganized	0.0 pts no response	2.0 pts
				Total Points: 10.0