Read the following problem descriptions (8.31, 8.32, 8.33) from the textbook "Visual C# How to Program, 5th Edition" by Paul Deitel and Harvey Deitel.

1. 8.31 (Machine-Language Programming) Let's create a computer called the Simpletron. As its name implies, it's a simple machine, but powerful. The Simpletron runs programs written in the only language it directly understands: Simpletron Machine Language, or SML for short.

   The Simpletron contains an accumulator—a special register into which information is put before the Simpletron uses it in calculations or examines it in various ways. All the information in the Simpletron is handled in terms of words. A word is a signed four-digit decimal number, such as `+3364`, `-1293`, `+0007` and `-0001`. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers `00`, `01`, …, `99`.

   Before running an SML program, we must load, or place, the code into memory. The first instruction (or statement) of every SML program is always placed in location `00`. The simulator will start executing at this location.

   Each instruction written in SML occupies one word of the Simpletron's memory (hence, instructions are signed four-digit decimal numbers). We shall assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron's memory may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the operation code specifying the operation to be performed. SML operation codes are summarized in Fig. 8.29.

   The last two digits of an SML instruction are the operand—the address of the memory location containing the word to which the operation applies. Let's consider several simple SML programs. The first SML program (Fig. 8.30) reads two numbers from the keyboard, then computes and displays their sum.

Operation code	Meaning
Input/output operations:	
`const int READ = 10;`	Read a word from the keyboard into a specific location in memory.
`const int WRITE = 11;`	Write a word from a specific location in memory to the screen.
Load/store operations:	
`const int LOAD = 20;`	Load a word from a specific location in memory into the accumulator.
`const int STORE = 21;`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations:	
`const int ADD = 30;`	Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).
`const int SUBTRACT = 31;`	Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator).
`const int DIVIDE = 32;`	Divide a word from a specific location in memory into the word in the accumulator (leave result in the accumulator).
`const int MULTIPLY = 33;`	Multiply a word from a specific location in memory by the word in the accumulator (leave the result in the accumulator).
Transfer of control operations:	
`const int BRANCH = 40;`	Branch to a specific location in memory.
`const int BRANCHNEG = 41;`	Branch to a specific location in memory if the accumulator is negative.
`const int BRANCHZERO = 42;`	Branch to a specific location in memory if the accumulator is zero.
`const int HALT = 43;`	Halt. The program has completed its task.

Fig. 8.29 | Simpletron Machine Language (SML) operation codes.

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Fig. 8.30 | SML program that reads two integers and computes their sum.

The instruction `+1007` reads the first number from the keyboard and places it into location `07` (which has been initialized to 0). Then instruction `+1008` reads the next number into location `08`. The load instruction, `+2007`, puts the first number into the accumulator, and the add instruction, `+3008`, adds the second number to the number in the accumulator. All SML arithmetic instructions leave their results in the accumulator. The store instruction, `+2109`, places the result in memory location `09`, from which the write instruction, `+1109`, takes the number and displays it (as a signed four-digit decimal number). The halt instruction, `+4300`, terminates execution.

The second SML program (Fig. 8.31) reads two numbers from the keyboard and determines and displays the larger value. Note the use of the instruction `+4107` as a conditional transfer of control, much the same as C#'s `if` statement.

Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 8.31 | SML program that reads two integers and determines the larger.

Now write SML programs to accomplish each of the following tasks:

a. Use a sentinel-controlled loop to read positive numbers and compute and display their sum. Terminate input when a negative number is entered.
b. Use a counter-controlled loop to read seven numbers, some positive and some negative, then compute and display their average.
c. Read a series of numbers, then determine and display the largest number. The first number read indicates how many numbers should be processed.

2. 8.32 (Computer Simulator) In this problem, you're going to build your own computer. No, you'll not be soldering components together. Rather, you'll use the powerful technique of software-based simulation to create an object-oriented software model of the Simpletron of Exercise 8.31. Your Simpletron simulator will turn the computer you're using into a Simpletron, and you'll actually be able to run, test and debug the SML programs you wrote in Exercise 8.31.

   When you run your Simpletron simulator, it should begin by displaying:

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** ( or data word ) at a time into the input ***
*** text field. I will display the location   ***
*** number and a question mark (?). You then  ***
*** type the word for that location. Enter    ***
*** -99999 to stop entering your program.     ***
```

Your app should simulate the memory of the Simpletron with a one-dimensional array `memory` of 100 elements. Now assume that the simulator is running, and let's examine the dialog as we enter the program of Fig. 8.31 (Exercise 8.31):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
```

Your program should display the memory location followed by a question mark. Each of the values to the right of a question mark is input by the user. When the sentinel value `-99999` is input, the program should display the following:

```
*** Program loading completed ***
*** Program execution begins  ***
```

The SML program has now been placed (or loaded) in array `memory`. Now the Simpletron executes the SML program. Execution begins with the instruction in location `00` and, as in C#, continues sequentially, unless directed to some other part of the program by a transfer of control.

Use variable `accumulator` to represent the accumulator register. Use variable `instructionCounter` to keep track of the location in memory that contains the instruction being performed. Use variable `operationCode` to indicate the operation currently being performed (i.e., the left two digits of the instruction word). Use variable `operand` to indicate the memory location on which the current instruction operates. Thus, `operand` is the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then "pick off" the left two digits and place them in `operationCode`, and "pick off" the right two digits and place them in `operand`. When the Simpletron begins execution, the special registers are all initialized to zero.

Now, let's "walk through" execution of the first SML instruction, `+1009` in memory location 00. This procedure is called an instruction execution cycle.

The `instructionCounter` tells us the location of the next instruction to be performed. We fetch the contents of that location from `memory` by using the C# statement

```
instructionRegister = memory[ instructionCounter ];
```

The operation code and the operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a read (versus a write, a load, or whatever). A `switch` differentiates among the 12 operations of SML. In the `switch` statement, the behavior of various SML instructions is simulated as shown in Fig. 8.32. We discuss branch instructions shortly and leave the others to you.

Instruction	Description
read:	Display the prompt `"Enter an integer"`, then input the integer and store it in location `memory[ operand ]`.
load:	`accumulator = memory[ operand ];`
add:	`accumulator += memory[ operand ];`
halt:	This instruction displays the message `*** Simpletron execution terminated ***`

Fig. 8.32 | Behavior of several SML instructions in the Simpletron.

When the SML program completes execution, the name and contents of each register, as well as the complete contents of memory, should be displayed. This is often called a memory dump. To help you program your dump method, a sample dump format is shown in Fig. 8.33. A dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated.

```
REGISTERS:
accumulator          +0000
instructionCounter     00
instructionRegister  +0000
operationCode          00
operand                00

MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 8.33 | A sample memory dump.

Let's proceed with the execution of our program's first instruction—namely, the `+1009` in location `00`. As we've indicated, the `switch` statement simulates this task by prompting the user to enter a value, reading the value and storing it in memory location `memory[ operand ]`. The value is then read into location `09`.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Since the instruction just performed was not a transfer of control, we need merely increment the `instructionCounter`.

This action completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins a new with the fetch of the next instruction to execute.

Now let's consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated within the `switch` as

```
instructionCounter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if ( accumulator == 0 )
   instructionCounter = operand;
```

At this point, you should implement your Simpletron simulator and run each of the SML programs you wrote in Exercise 8.31. If you desire, you may embellish SML with additional features and provide for these features in your simulator.

Your simulator should check for various types of errors. During the program-loading phase, for example, each number the user types into the Simpletron's `memory` must be in the range `-9999` to `+9999`. Your simulator should test that each number entered is in this range and, if not, keep prompting the user to re-enter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes and accumulator overflows (i.e., arithmetic operations resulting in values larger than `+9999` or smaller than `-9999`). Such serious errors are called fatal errors. When a fatal error is detected, your simulator should display an error message, such as

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

and should display a full computer dump in the format we discussed previously. This treatment will help the user locate the error in the program.

3. 8.33 (Project: Simpletron Simulator Modifications) In Exercise 8.32, you wrote a software simulation of a computer that executes programs written in Simpletron Machine Language (SML). In this exercise, we propose several modifications and enhancements to the Simpletron Simulator.
    a. Extend the Simpletron Simulator's memory to contain 1000 memory locations to enable the Simpletron to handle larger programs.
    b. Allow the simulator to perform remainder calculations. This modification requires an additional SML instruction.
    c. Allow the simulator to perform exponentiation calculations. This modification requires an additional SML instruction.
    d. Modify the simulator to use hexadecimal values rather than integer values to represent SML instructions.
    e. Modify the simulator to allow output of a newline. This modification requires an additional SML instruction.
    f. Modify the simulator to process floating-point values in addition to integer values.
    g. Modify the simulator to handle string input. [Hint: Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII (see Appendix C) decimal equivalent of an uppercase character. Add a machine-language instruction that will input a string and store the string beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII

character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a half-word.]

h. Modify the simulator to handle output of uppercase strings stored in the format of Part g. [Hint: Add a machine-language instruction that will display a string beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and displays the string by translating each two-digit number into its equivalent character.]

In this project, you will be required to release three versions of software which implement the Simpletron simulator. Each version will add new functionality as well as correct errors from the previous version. All code must be kept in a repository under GitHub. You will be required to create branches for each version, and after successfully testing the functionality, you should merge your changes back to the main branch and create a label for the newly released version.

Here is a list of items that should be incorporated into each version:

1.0:

- All operations from Fig. 8.29.

2.0:

- Ability to read in a program from a text file rather than input it via the program.
- Memory dump after the completion of a program.

3.0:

- Operations 8.33b, 8.33c, and 8.33e.