```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

**Fig. 8.22** | Two-dimensional array representation of a maze.

Write recursive function `mazeTraverse` to walk through the maze. The function should receive arguments that include a 12-by-12 character array representing the maze and the starting location of the maze. As `mazeTraverse` attempts to locate the exit from the maze, it should place the character `X` in each square in the path. The function should display the maze after each move, so the user can watch as the maze is solved.

**8.17** (*Generating Mazes Randomly*) Write a function `mazeGenerator` that randomly produces a maze. The function should take as arguments a two-dimensional 12-by-12 character array and pointers to the `int` variables that represent the row and column of the maze's entry point. Try your function `mazeTraverse` from Exercise 8.16, using several randomly generated mazes.

## Special Section: Building Your Own Computer

In the next several problems, we take a temporary diversion away from the world of high-level-language programming. We "peel open" a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (using software-based *simulation*) on which you can execute your machine-language programs!

**8.18** (*Machine-Language Programming*) Let's create a computer we'll call the Simpletron. As its name implies, it's a simple machine, but, as we'll soon see, it's a powerful one as well. The Simpletron runs programs written in the only language it directly understands, that is, Simpletron Machine Language, or SML for short.

The Simpletron contains an *accumulator*—a "special register" in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of *words*. A word is a signed four-digit decimal number, such as `+3364`, `-1293`, `+0007`, `-0001`, etc. 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 program 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; thus, instructions are signed four-digit decimal numbers. 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* that specifies the operation to be performed. SML operation codes are shown in Fig. 8.23.

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.

| 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 and 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 result in accumulator). |
| `const int SUBTRACT = 31;` | Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator). |
| `const int DIVIDE = 32;` | Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator). |
| `const int MULTIPLY = 33;` | Multiply a word from a specific location in memory by the word in the accumulator (leave result in 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. 8.23** | Simpletron Machine Language (SML) operation codes.

Now let's consider two simple SML programs. The first (Fig. 8.24) reads two numbers from the keyboard and computes and prints their sum. The instruction `+1007` reads the first number from the keyboard and places it into location `07` (which has been initialized to zero). Instruction `+1008` reads the next number into location `08`. The *load* instruction, `+2007`, places (copies) 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 (copies) the result back into memory location `09`. Then the *write* instruction, `+1109`, takes the number and prints it (as a signed four-digit decimal number). The *halt* instruction, `+4300`, terminates execution.

The SML program in Fig. 8.25 reads two numbers from the keyboard, then determines and prints 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 | +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. 8.24** | SML Example 1.

| 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. 8.25** | SML Example 2.

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

a) Use a sentinel-controlled loop to read positive numbers and compute and print 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, and compute and print their average.

c) Read a series of numbers, and determine and print the largest number. The first number read indicates how many numbers should be processed.

**8.19** (*Computer Simulator*) It may at first seem outrageous, but in this problem you are going to build your own computer. No, you won't be soldering components together. Rather, you'll use the powerful technique of *software-based simulation* to create a *software model* of the Simpletron. Your Simpletron simulator will turn the computer you are using into a Simpletron, and you actually will be able to run, test and debug the SML programs you wrote in Exercise 8.18.

When you run your Simpletron simulator, it should begin by printing

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

Your program should simulate the Simpletron's memory with a single-subscripted, 100-element array `memory`. Now assume that the simulator is running, and let's examine the dialog as we enter the program of Example 2 of Exercise 8.18:

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

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

The numbers to the right of each `?` in the preceding dialog represent the SML program instructions input by the user.

The SML program has now been placed (or loaded) into array `memory`. Now the Simpletron executes your SML program. Execution begins with the instruction in location `00` and, like 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 Simpletron begins execution, the special registers are all initialized to zero.

Now let's "walk through" the execution of the first SML instruction, `+1009` in memory location `00`. This 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 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*, etc.). 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. 8.26 (we leave the others to you).

The *halt* instruction also causes the Simpletron to print the name and contents of each register, as well as the complete contents of memory. Such a printout is often called a *register and memory dump*. To help you program your dump function, a sample dump format is shown in Fig. 8.26. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. To format numbers with their sign as shown in the dump, use stream manipulator **showpos**. To disable the display of the sign, use stream manipulator **noshowpos**. For numbers that have fewer than four digits, you can format numbers with leading zeros between the sign and the value by using the following statement before outputting the value:

```
cout << setfill( '0' ) << internal;
```

```
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. 8.26** | A sample register and memory dump.

Parameterized stream manipulator **setfill** (from header `<iomanip>`) specifies the fill character that will appear between the sign and the value when a number is displayed with a field width of five characters but does not have four digits. (One position in the field width is reserved for the sign.) Stream manipulator **internal** indicates that the fill characters should appear between the sign and the numeric value .

Let's proceed with the execution of our program's first instruction—`+1009` in location `00`. As we've indicated, the `switch` statement simulates this by performing the C++ statement

```
cin >> memory[ operand ];
```

A question mark (`?`) should be displayed on the screen before the `cin` statement executes to prompt the user for input. The Simpletron waits for the user to type a value and press the *Enter* key. The value is then read into location `09`.

At this point, simulation of the first instruction is complete. All that remains is to prepare the Simpletron to execute the next instruction. The instruction just performed was not a transfer of control, so we need merely increment the instruction counter register as follows:

```
++instructionCounter;
```

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

Now let's consider how to simulate the branching instructions (i.e., the transfers of control). All we need to do is adjust the value in the `instructionCounter` appropriately. Therefore, the unconditional branch instruction (`40`) is simulated in 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 8.18. The variables that represent the Simpletron simulator's memory and registers should be defined in `main` and passed to other functions by value or by reference as appropriate.

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 use a `while` loop to test that each number entered is in this range and, if not, keep prompting the user to reenter 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, accumulator overflows (i.e., arithmetic operations resulting in values larger than `+9999` or smaller than `-9999`) and the like. Such serious errors are called **fatal errors**. When a fatal error is detected, your simulator should print an error message such as

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

and should print a full register and memory dump in the format we've discussed previously. This will help the user locate the error in the program.

**8.20** (*Project: Modifications to the Simpletron Simulator*) In Exercise 8.19, 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. In Exercises 20.31–20.35, we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to SML. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler. [*Note:* Some modifications may conflict with others and therefore must be done separately.]

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 modulus calculations. This requires an additional Simpletron Machine Language instruction.
c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.
d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions.
e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language 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 decimal equivalent of a character. Add a machine-language instruction that inputs 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.]