时钟每次滴答一下时（即循环的每次迭代），打印一个有70个位置的行，在乌龟的位置上显示字母T，在兔子的位置上显示字母H。有时候，两个选手在同一格。这时候，乌龟会咬兔子，你的程序应该从该位置开始打印"OUCH!!!"。除了T、H和"OUCH!!!"（不分胜负的情况下）以外，其他打印位置都应该是空的。打印每一行之后，检测是否有动物已经到达或越过第70格。如果有，打印出胜利者并终止程序。如果乌龟赢，打印 "TORTOISE WINS!!! YAY!!!"；如果兔子赢，打印 "Hare wins. Yuch."。如果两个动物同时胜出，你也许偏向乌龟（"弱者"）赢，或者可以打印 "It's a tie."。如果两个都没赢，则再次循环，模拟下一个时钟滴答。准备运行程序时，让一组爱好者观看比赛。你会发现观众非常投入！

动物	移动类型	时间百分比	实际动作
乌龟	快走	50%	向右移动3个方格
	滑倒	20%	向左移动6个方格
	慢走	30%	向右移动1个方格
兔子	睡觉	20%	不动
	大步跳跃	20%	向右移动9个方格
	大步滑倒	10%	向左移动12个方格
	小步跳跃	30%	向右移动1个方格
	小步滑倒	20%	向右移动2个方格

图 8.38　移动乌龟和兔子的规则

特殊小节：构建自己的计算机

在后面的几个问题中，我们暂时离开高级语言编程的话题。我们"剥开"一个计算机，看看它的内部结构。下面将介绍机器语言编程，并编写几个机器语言程序。为了得到有价值的体验，我们接着建立一个计算机（通过基于软件的模拟手段），可以在上面执行自己的机器语言程序。

8.18 （**机器语言程序**）我们建立一个称为Simpletron的计算机。顾名思义，它是一个简单的机器，但是很快将会看到，它也具有强大的功能。Simpletron只能运行用它可以理解的唯一语言，即Simpletron机器语言（简称为SML）编写的程序。

Simpletron包含一个累加器（一个"特殊寄存器"），存放Simpletron用于计算和各种处理的信息。Simpletron中的所有信息都是按照"字"来处理的。字是带符号的4位十进制数，例如+3364、-1293、+0007、-0001，等等。Simpletron带有100个字的内存，并且这些字通过它们的位置编号00, 01,..., 99被引用。在运行一个SML程序之前，我们必须把程序载入或者放置到内存。每个SML程序的第一条指令（或语句）总是放在位置00处。模拟器从这个位置开始执行。

使用SML编写的每条指令占用Simpletron内存中的一个字；因此，指令是带符号的4位十进制数。假设SML指令的符号总是正号，但是数据字的符号可正可负。Simpletron内存中的每个位置可以包含一条指令、程序使用的一个数据值或未用到的（即未定义的）内存区。每个SML指令的前两位数字是操作码，指定要进行的操作。SML操作码如图8.39所示。

操作码	含义
输入/输出操作：	
const int READ = 10;	从键盘将一个字读入到内存中的特定位置
const int WRITE = 11;	将内存中特定位置的一个字写到屏幕
载入和存储操作：	
const int LOAD = 20;	将内存中特定位置的一个字载入累加器
const int STORE = 21;	将累加器中的一个字存储到内存中的特定位置
算术运算：	
const int ADD = 30;	内存中特定位置的一个字加累加器中的字（结果保留在累加器中）
const int SUBTRACT = 31;	累加器中的字减去内存中特定位置的一个字（结果保留在累加器中）
const int DIVIDE = 32;	累加器中的字除以内存中特定位置的一个字（结果保留在累加器中）
const int MULTIPLY = 33;	内存中特定位置的一个字乘以累加器中的字（结果保留在累加器中）

图 8.39　Simpletron 机器语言（SML）操作码

操作码	含义
控制转移操作：	
const int BRANCH = 40;	转移到内存的特定位置
const int BRANCHNEG = 41;	如果累加器为负值，转移到内存的特定位置
const int BRANCHZERO = 42;	如果累加器为零，转移到内存的特定位置
const int HALT = 43;	停止——程序已完成任务

图 8.39（续）　Simpletron 机器语言（SML）操作码

SML 指令的后两位数字是操作数，也就是要操作的字的内存位置。

现在让我们考虑两个简单的 SML 程序。第一个 SML 程序（如图 8.40 所示）从键盘读入两个数，然后计算并打印它们的和。指令 +1007 从键盘读入第一个数字，并把它存放到内存位置 07（初始化为 0）处。指令 +1008 读入下一个数到内存位置 08 处。载入指令 +2007，把第一个数放置（复制）到累加器，然后加法指令 +3008 把第二个数和累加器中的数相加。所有的 SML 算术运算指令都把它们的结果留在累加器中。存储指令 +2109 把结果放回（复制）到内存位置 09。然后，写入指令 +1109 取得这个数并打印它（作为一个带符号的 4 位十进制）。停止指令 +4300 终止执行。

位置	代码	指令
00	+1007	（读入 A）
01	+1008	（读入 B）
02	+2007	（加载 A）
03	+3008	（加 B）
04	+2109	（存储 C）
05	+1109	（写入 C）
06	+4300	（停止）
07	+0000	（变量 A）
08	+0000	（变量 B）
09	+0000	（结果 C）

图 8.40　SML 示例 1

图 8.41 中的 SML 程序从键盘读入两个数，然后确定并打印较大值。请注意，指令 +4107 是用来进行条件控制转移的，和 C++ 的 if 语句相类似。

现在编写 SML 程序完成下列任务：

a) 使用一个标记控制的循环读取正数，计算并打印它们的和。当遇到一个负数时，停止输入。

b) 使用一个计数器控制的循环读入 7 个数字，其中有正数和负数，然后计算并打印它们的平均值。

c) 读入一系列数，然后确定并打印最大数。第一个读入的数指出要处理多少个数字。

位置	代码	指令
00	+1009	（读入 A）
01	+1010	（读入 B）
02	+2009	（加载 A）
03	+3110	（减去 B）
04	+4107	（分支负数到 07）
05	+1109	（写入 A）
06	+4300	（停止）
07	+1110	（写入 B）
08	+4300	（停止）
09	+0000	（变量 A）
10	+0000	（变量 B）

图 8.41　SML 示例 2

8.19（**计算机模拟程序**）最初看起来似乎有点不可理解，但在这个问题中，你将要建立自己的计算机。这里不是要把计算机的硬件连接起来，而是用基于软件模拟器的强大技术建立一个 Simpletron 的软件模型。读者是不会失望的，Simpletron 模拟器可以将你使用的计算机变成 Simpletron，而且实际上可以运行、测试和调试在习题 8.18 中编写的 SML 程序。

当运行 Simpletron 模拟器时，它首先打印如下信息：

```
*** 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. ***
```

你的程序应该用一个具有 100 个元素的一维数组 memory 模拟 Simpletron 的内存。现在假设模拟器正在运行，让我们检查一下输入习题 8.18 示例 2 的程序时的对话：

```
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 ***
```

请注意，前面对话中的每个“?”右边的数值表示用户输入的 SML 程序指令。

SML 程序现在已经被放到（载入到）数组 memory 中。Simpletron 开始执行你的 SML 程序。执行从位置 00 处的指令开始，和 C++ 一样，按顺序继续执行，除非通过控制转移定向到程序的其他部分。

使用变量 accumulator 表示累加寄存器。使用变量 counter 跟踪内存中包含当前执行指令的内存位置。使用变量 operationCode 表示当前正在进行的操作（即指令字的左边两位）。使用变量 operand 表示当前指令所操作的内存位置。因此，operand 是当前执行指令的最右边两位。不要直接从内存执行指令，而是将下一个要执行的指令从内存中转移到变量 instructionRegister 中。然后“摘取出”左边两位并把它们放到 operationCode 中，再“摘取出”右边两位放到 operand 中。当 Simpletron 开始执行时，所有特殊寄存器都初始化为 0。

下面，让我们看一下第一条 SML 指令（即内存位置 00 处的指令 +1009）的执行过程。这个过程称为一条指令的执行周期。

counter 指出包含下一条要执行的指令的内存位置。我们用下面的 C++ 语句从 memory 中取得该位置的内容：

```
instructionRegister = memory[ counter ];
```

使用下列语句：

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

从指令寄存器提取出操作码和操作数。

Simpletron 必须确定该操作码实际上是读（不是写、载入等）操作。switch 区分 SML 的 12 种操作。在 switch 语句中，对各种 SML 指令的行为进行了模拟，如图 8.42 所示（其他的留给读者）。

halt 指令还使 Simpletron 打印每个寄存器的名字和内容，以及打印内存的全部内容。这种输出通常称为计算机转储（computer dump，计算机转储并不是指旧计算机被放置的地方）。为了帮助读者编写自己的转储函数，图 8.43 给出了一个说明转储格式的例子。注意，执行 Simpletron 程序之后，计算机转储将显示执行终止时的指令实际值和数据值。为了使数字和其符号以转储中的形式显示，使用流操纵符 showpos。要禁止符号的显示，使用流操纵符 noshowpos。对于位数少于 4 位的数，可以在输出值之前，使用以下语句在符号和数值之间加上前导的 0：

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

当一个不足 4 位的数字以 5 个字符的域宽显示时，参数化的流操纵符 setfill（来自头文件<iomanip>）指定符号和数值之间的填充字符（其中一个字符宽度是留给符号使用的）。流操纵符 internal 指示填充字符应该出现在符号和数字值之间。

读入：	cin >> memory[operand];
载入：	accumulator = memory[operand];
相加：	accumulator += memory[operand];
分支：	稍后我们将讨论分支指令
停止：	这条指令打印如下的消息：
	*** Simpletron execution terminated ***

图 8.42　SML 指令的行为

```
REGISTERS:
accumulator          +0000
counter                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
```

图 8.43　一个转储的例子

我们执行程序的第一条指令，即位置 00 中的 +1009。如前所述，switch 语句通过执行以下 C++ 语句模拟这个过程：

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

在执行 cin 语句之前，应该在屏幕上显示一个问号（?），以提示用户输入。Simpletron 等待用户输入一个值并按回车键。然后这个值被读入到内存位置 09 处。

此时，第一条指令的模拟已经完成了。剩下的事就是准备让 Simpletron 执行下一条指令。由于刚才执行的指令不是一个控制转移，因此我们只需要对指令计数寄存器加 1，如下所示：

```
++counter;
```

这样就完成了第一条指令的模拟执行。接下来整个过程（即指令的执行周期）重新开始，读取下一条要执行的指令。

现在，我们考虑一下如何模拟分支指令（即控制转移指令）。我们所需要做的就是要恰当地调整指令计数器的值。因此，无条件转移指令（40）在 switch 中模拟如下：

```
counter = operand;
```

“如果累加器为 0，则转移”的条件指令可以模拟如下：

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

此时，读者可以实现自己的 Simpletron 模拟程序，并运行在习题 8.18 中编写的每个 SML 程序。也许，你还可以增加其他特性来润色 SML，并在模拟程序中提供这些特性。

模拟程序应该检查各种类型的错误。例如，在装入程序阶段，用户输入到 memory 中的每个数都应当在范围 -9999 到 +9999 之间。模拟程序应该用一个 while 循环检测每个输入的数字在这个范围；否则，提示用户重新输入数字，直到用户输入一个正确的数为止。

在执行期间，模拟程序应该检查各种严重的错误情况。例如，除数为 0 的情况、执行不合法操作码的情况、累加器溢出（即算术运算的结果大于 +9999 或小于 -9999）的情况，等等。这些严重的错误称为致命的错误。当检测到一个致命的错误时，模拟程序应该打印以下错误信息：

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

并按照前面讨论的格式打印完整的计算机转储。这可以帮助用户找出程序中的错误。

更多的指针习题

8.20 修改图 8.25 至图 8.27 的洗牌和发牌程序，使洗牌和发牌操作由同一个函数（shuffleAndDeal）完成。该函数应包含类似于图 8.26 中 shuffle 函数的嵌套循环语句。

8.21 请问下面的程序做了什么？

```
1  // Ex. 8.21: ex08_21.cpp
2  // What does this program do?
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  void mystery1( char *, const char * ); // prototype
9
10 int main()
11 {
12    char string1[ 80 ];
13    char string2[ 80 ];
14
15    cout << "Enter two strings: ";
16    cin >> string1 >> string2;
17    mystery1( string1, string2 );
18    cout << string1 << endl;
19    return 0; // indicates successful termination
20 } // end main
21
22 // What does this function do?
23 void mystery1( char *s1, const char *s2 )
24 {
25    while ( *s1 != '\0' )
26       ++s1;
27
28    for ( ; *s1 = *s2; s1++, s2++ )
29       ; // empty statement
30 } // end function mystery1
```