g) Visualizzate gli elementi del vettore **values** utilizzando gli indici con un puntatore che faccia riferimento al vettore stesso.
h) Puntate all'elemento 5 del vettore **values** utilizzando la notazione con gli indici di vettore, quella con puntatore e offset usando il nome del vettore come puntatore, quella con gli indici di puntatore e quella con puntatore e offset.
i) Quale indirizzo sarebbe puntato da **vPtr + 3**? Quale valore sarebbe immagazzinato in quella locazione?
j) Supponendo che **vPtr** faccia riferimento a **values[4]** quale indirizzo sarebbe puntato da **vPtr -= 4**? Quale valore sarebbe immagazzinato in quella locazione?

7.10 Per ognuna delle seguenti attività scrivete una singola istruzione che esegua il compito indicato. Supponete che siano già state dichiarate le variabili intere **long value1** e **value2** e che **value1** sia stata inizializzata a **200000**.

a) Dichiarate la variabile **lPtr** come puntatore a un oggetto di tipo **long**.
b) Assegnate l'indirizzo di **value1** alla variabile di tipo puntatore **lPtr**.
c) Visualizzate il valore dell'oggetto puntato da **lPtr**.
d) Assegnate il valore dell'oggetto puntato da **lPtr** alla variabile **value2**.
e) Visualizzate il valore di **value2**.
f) Visualizzate l'indirizzo di **value1**.
g) Visualizzate l'indirizzo immagazzinato in **lPtr**. L'indirizzo visualizzato è identico a quello di **value1**?

7.11 Eseguite ognuna delle seguenti attività.

a) Scrivete l'intestazione della funzione **zero** che avrà come parametro un vettore **bigIntegers** di interi **long** e non restituirà alcun valore.
b) Scrivete il prototipo per la funzione della parte (a).
c) Scrivete l'intestazione della funzione **add1AndSum** che avrà come parametro un vettore di interi **oneTooSmall** e restituirà un intero.
d) Scrivete il prototipo per la funzione descritta nella parte (c).

Nota: gli Esercizi dal 7.12 al 7.15 sono abbastanza impegnativi. Una volta che avrete risolto questi problemi, dovreste essere in grado di implementare facilmente i giochi di carte più comuni.

7.12 Modificate il programma della Figura 7.24, in modo che la funzione di distribuzione dispensi le cinque carte di una mano di poker. In seguito scrivete le seguenti funzioni aggiuntive:

a) Determinate se la mano contenga una coppia.
b) Determinate se la mano contenga una doppia coppia.
c) Determinate se la mano contenga un tris (per esempio, tre fanti).
d) Determinate se la mano contenga un poker (per esempio, quattro assi).
e) Determinate se la mano contenga un colore (ovverosia, cinque carte dello stesso seme).
f) Determinate se la mano contenga una scala (ovverosia, cinque carte con valori consecutivi).

7.13 Utilizzate le funzioni sviluppate nell'Esercizio 7.12 per scrivere un programma che distribuisca le cinque carte di due mani di poker, le valuti e determini qual è la mano migliore.

7.14 Modificate il programma della Figura 7.13 in modo da simulare un mazziere. Le cinque carte della mano del mazziere saranno distribuite "a faccia in giù" così che il giocatore non le possa vedere. Il programma dovrà quindi valutare la mano del mazziere e, basandosi sulla qualità di quella, dovrà estrarre altre carte per sostituire quelle scartate dalla mano originaria. Il programma dovrà quindi rivalutare la mano del mazziere. (*Attenzione*: questo è un problema difficile!).

7.15 Modificate il programma sviluppato nell'Esercizio 7.14 così che possa gestire automaticamente la mano del mazziere, mentre consenta al giocatore di decidere quali carte della sua mano sostituire. Il programma dovrà quindi valutare entrambe le mani e determinare chi avrà vinto. Utilizzate ora questo nuovo programma per giocare 20 partite contro il computer. Chi ne vince di più, voi o il computer? Fate giocare uno dei vostri amici in 20 partite contro il computer. Chi ne vince più? Basandovi sui risultati di queste partite, apportate le opportune modifiche per raffinare il vostro programma di gioco (anche questo è un problema difficile). Giocate altre 20 partite. Gioca meglio il vostro nuovo programma?

7.16 Nel programma per il mescolamento e la distribuzione delle carte della Figura 7.24, abbiamo intenzionalmente utilizzato un algoritmo di mescolamento inefficiente che ha introdotto la possibilità di differimenti indefiniti. In questo esercizio creerete un algoritmo di mescolamento ad alta efficienza che eviterà il differimento indefinito.

Modificate il programma della Figura 7.24 nel modo seguente. Inizializzate la matrice **deck** come mostrato nella Figura 7.29. Modificate la funzione **shuffle** in modo che iteri su ogni riga e colonna della matrice, toccando ognuno degli elementi una sola volta. Ogni elemento dovrà essere scambiato con un altro selezionato a caso dalla matrice.

Visualizzate la matrice risultante così che possiate determinare se il mazzo di carte sia stato mescolato in modo soddisfacente (come nella Figura 7.30, per esempio). Per assicurarvi un mescolamento soddisfacente potreste anche fare in modo che il programma richiami più volte la funzione **shuffle**.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Figura 7.29 La matrice **deck** non mescolata.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Figura 7.30 Esempio di matrice **deck** mescolata.

Osservate che sebbene l'approccio utilizzato in questo problema migliori l'algoritmo di mescolamento, quello di distribuzione richiederà ancora delle ricerche nella matrice **deck** per la carta 1, poi per la 2, poi per la 3 e così via. Come se non bastasse, l'algoritmo di distribuzione continuerà a cercare una carta nel mazzo anche dopo averla già ritrovata e distribuita. Modificate il programma della Figura 7.24 in modo che, una volta che abbia distribuito una carta, non faccia ulteriori tentativi per ritrovarla e che proceda immediatamente con la distribuzione di quella successiva. Nel Capitolo 10 svilupperemo un algoritmo che richiederà una sola operazione per carta.

7.17 (*Simulazione: la tartaruga e la lepre*) In questo esercizio ricreerete uno dei momenti storici più importanti, vale a dire la classica corsa tra la tartaruga e la lepre. Utilizzerete la generazione dei numeri casuali per sviluppare una simulazione di questo memorabile evento.

I nostri contendenti cominceranno la loro corsa nella "casella 1" delle 70 che compongono il percorso della gara. Ogni casella rappresenta una delle posizioni possibili lungo il percorso della gara. La linea di arrivo è nella casella numero 70. Il primo contendente che avrà raggiunto o superato la casella 70 sarà premiato con un secchio di carote e lattughe fresche. Il percorso si inerpica serpeggiando sul fianco di una montagna sdrucciolevole, perciò i contendenti potranno perdere terreno di quando in quando.

C'è un orologio che scandisce ogni secondo. Ad ogni tic dell'orologio il vostro programma dovrà aggiustare la posizione degli animali secondo le seguenti regole:

Animale	Tipo di mossa	Percentuale di tempo	Mossa effettiva
Tartaruga	Arrancata rapida	50%	3 caselle a destra
	Scivolone	20%	6 caselle a sinistra
	Arrancata lenta	30%	1 casella a destra
Lepre	Dormita	20%	Nessuna mossa
	Salto lungo	20%	9 caselle a destra
	Scivolone lungo	10%	12 caselle a sinistra
	Salto corto	30%	1 casella a destra
	Scivolone corto	20%	2 caselle a sinistra

Utilizzate le variabili per mantenere traccia delle posizioni degli animali (ovverosia dei numeri da 1 a 70). Fate partire ogni animale dalla posizione 1 (vale a dire dalle "gabbie di partenza"). Riportate nella casella 1 l'animale che dovesse eventualmente scivolare indietro in una posizione precedente a quella.

Generate le percentuali mostrate nella tabella precedente producendo un intero casuale i compreso nell'intervallo 1 = i = 10. Nel caso della tartaruga eseguirete una "arrancata rapida" quando 1 = i = 5, una "scivolata" quando 6 = i = 7 o una "arrancata lenta" quando 8 = i = 10. Utilizzate una tecnica simile per far correre la lepre.

Cominciate la gara visualizzando:

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

In seguito, per ogni tic dell'orologio (ovverosia per ciascuna ripetizione del ciclo), visualizzerete una linea di 70 posizioni che mostri una lettera `T` in quella della tartaruga e una `L` in quella della lepre. Di quando in quando, i contendenti si ritroveranno nella stessa casella. In tal caso la tartaruga morderà la lepre e il vostro programma dovrà visualizzare `OUCH!!!` in quella posizione. Tutte le posizioni di visualizzazione diverse da `T`, `L` od `OUCH!!!` dovranno restare vuote.

Dopo che ogni linea sarà stata visualizzata, controllerete se uno dei due animali abbia raggiunto o superato la casella 70. In caso affermativo visualizzerete il nome del vincitore e terminerete la simulazione. Nel caso che abbia vinto la tartaruga visualizzerete "`TORTOISE WINS!!! YAY!!!`". Nel caso che abbia vinto la lepre visualizzerete "`Hare wins. Yuch.`" Nel caso che entrambi gli animali tagliassero il traguardo nello stesso tic dell'orologio, potreste favorire la tartaruga (poverina), oppure potreste visualizzare "`It's a tie.`" Nel caso che nessuno dei due animali abbia ancora vinto, eseguite un'altra volta il ciclo per simulare il successivo tic dell'orologio. Radunate un gruppo di tifosi che assistano alla gara, quando sarete pronti a mandare in esecuzione il vostro programma. Il coinvolgimento del vostro pubblico vi sorprenderà.

Sezione speciale: costruite il vostro computer

Nei prossimi esercizi abbandoneremo temporaneamente il mondo della programmazione con linguaggi di alto livello. "Sbucceremo" un computer e daremo uno sguardo alla sua struttura interna. Introdurremo la programmazione in linguaggio macchina e con questo scriveremo diversi programmi. In seguito, per fare in modo che sia un'esperienza di particolare valore, costruiremo un computer (attraverso la tecnica della *simulazione* software) sul quale potrete far eseguire i vostri programmi scritti in linguaggio macchina!

7.18 (*Programmazione in linguaggio macchina*) Creeremo ora un computer che chiameremo Simpletron. Come il suo nome lascia intendere si tratta di una macchina semplice, ma come vedremo presto anche potente. Il Simpletron eseguirà dei programmi scritti nell'unico linguaggio che sia in grado di comprendere direttamente, vale a dire il Linguaggio Macchina del Simpletron, che abbrevieremo in LMS.

Il Simpletron contiene un *accumulatore*, un "registro speciale" in cui saranno inserite tutte le informazioni, prima che il Simpletron possa utilizzarle per i calcoli o per esaminarle in vari modi. Tutte le informazioni saranno manipolate all'interno del Simpletron come *parole*. Una parola è un numero decimale di quattro cifre con segno, come `+3364`, `-1293`, `+0007`, `-0001`, ecc. Il Simpletron è equipaggiato con una memoria di 100 parole alle quali faremo riferimento attraverso i loro numeri di locazione `00`, `01`, ... `99`.

Prima di poter eseguire un programma LMS dovremo *caricarlo* o immetterlo nella memoria. La prima istruzione (o comando) di ogni programma LMS sarà sempre caricata nella posizione `00`.

Ogni istruzione scritta in LMS occuperà una parola nella memoria del Simpletron e, di conseguenza, corrisponderà a un numero decimale di quattro cifre con segno. Ovviamente il segno di ogni istruzione LMS sarà sempre quello positivo, mentre quelli delle parole che contengano dei dati potranno essere positivi e negativi. Ogni locazione della memoria del Simpletron potrà contenere un'istruzione, il valore di un dato utilizzato dal programma, oppure un'area di memoria inutilizzata (e quindi indefinita). Le prime due cifre di ogni istruzione LMS corrisponderanno al *codice dell'operazione* che dovrà essere eseguita. I codici delle operazioni LMS sono riassunti nella Figura 7.31.

Codice dell'operazione	Significato
Operazioni di input/output:	
`#define READ 10`	Legge una parola dal terminale e la immagazzina in una specifica locazione di memoria.
`#define WRITE 11`	Scrive sul terminale la parola contenuta in una specifica locazione di memoria.
Operazioni di caricamento/immagazzinamento:	
`#define LOAD 20`	Carica nell'accumulatore la parola contenuta in una specifica locazione di memoria.
`#define STORE 21`	Archivia il contenuto dell'accumulatore in una specifica locazione di memoria.
Operazioni aritmetiche:	
`#define ADD 30`	Aggiunge la parola contenuta in una specifica locazione di memoria a quella contenuta nell'accumulatore (lasciando in questo il risultato).
`#define SUBTRACT 31`	Sottrae la parola contenuta in una specifica locazione di memoria da quella contenuta nell'accumulatore (lasciando in questo il risultato).
`#define DIVIDE 32`	Divide la parola contenuta in una specifica locazione di memoria per quella contenuta nell'accumulatore (lasciando in questo il risultato).
`#define MULTIPLY 33`	Moltiplica la parola contenuta in una specifica locazione di memoria per quella contenuta nell'accumulatore (lasciando in questo il risultato).
Operazioni di trasferimento del controllo:	
`#define BRANCH 40`	Salta a una specifica locazione di memoria.
`#define BRANCHNEG 41`	Salta a una specifica locazione di memoria, se l'accumulatore contiene un valore negativo.
`#define BRANCHZERO 42`	Salta a una specifica locazione di memoria, se l'accumulatore contiene un valore uguale a zero.
`#define HALT 43`	Ferma l'esecuzione del programma.

Figura 7.31 I codici di operazione del Linguaggio Macchina del Simpletron (LMS).

Le ultime due cifre di un'istruzione LMS rappresenteranno invece l'*operando*, ovverosia l'indirizzo della locazione di memoria che conterrà la parola su cui l'operazione sarà applicata. Consideriamo ora alcuni semplici programmi in LMS.

Esempio 1

Locazione	Numero	Istruzione
00	+1007	(Legge A)
01	+1008	(Legge B)
02	+2007	(Carica A nell'accumulatore)
03	+3008	(Somma B all'accumulatore)
04	+2109	(Memorizza il valore dell'accumulatore in C)
05	+1109	(Stampa C)
06	+4300	(Halt)
07	+0000	(Variabile A)
08	+0000	(Variabile B)
09	+0000	(Risultato C)

Questo programma in LMS leggerà due numeri dalla tastiera e calcolerà e visualizzerà la loro somma. L'istruzione **+1007** leggerà il primo numero dalla tastiera e lo inserirà nella locazione di memoria **07** (che sarà già stata azzerata). In seguito **+1008** leggerà il secondo numero nella locazione **08**. L'istruzione *load*, **+2007**, sistemerà il primo numero nell'accumulatore, mentre l'istruzione *add*, **+3008**, aggiungerà il secondo numero a quello contenuto nell'accumulatore. *Tutte le operazioni aritmetiche del linguaggio LMS lasciano il loro risultato nell'accumulatore.* L'istruzione *store*, **+2109**, riporterà il risultato nella locazione di memoria **09** dalla quale l'istruzione *write*, **+1109**, lo preleverà e lo visualizzerà come un numero decimale di quattro cifre con segno. L'istruzione *halt*, **+4300**, terminerà l'esecuzione del programma.

Esempio 2

Locazione	Numero	Istruzione
00	+1009	(Legge A)
01	+1010	(Legge B)
02	+2009	(Carica A nell'accumulatore)
03	+3110	(Sottrae B dall'accumulatore)
04	+4107	(Salta a 07 se l'accumulatore è negativo)
05	+1109	(Stampa A)
06	+4300	(Halt)
07	+1110	(Stampa B)
08	+4300	(Halt)
09	+0000	(Variabile A)
10	+0000	(Variabile B)

Questo programma LMS leggerà due numeri dalla tastiera e determinerà e visualizzerà quello maggiore. Osservate l'uso della istruzione **+4107** come un trasferimento di controllo condizionale molto simile all'istruzione **if** del C. Scrivete ora dei programmi in LMS che eseguano ognuna delle seguenti attività.

a) Utilizzate un ciclo controllato da un valore sentinella per leggere 10 numeri positivi e calcolare e visualizzare la loro somma.

b) Utilizzate un ciclo controllato da un contatore per leggere sette numeri, positivi e negativi, e calcolate e visualizzate la loro media.

c) Leggete una serie di numeri e determinate e visualizzate quello maggiore. Il primo numero letto indicherà quanti valori dovranno essere elaborati.

7.19 (*Un simulatore di computer*) Potrà sembrarvi esagerato, ma con questo esercizio costruirete il vostro computer. No, non dovrete saldare dei componenti. Utilizzerete piuttosto la potente tecnica della *simulazione software* per creare un *modello software* del Simpletron. Non rimarrete delusi. Il vostro simulatore Simpletron trasformerà il computer che state utilizzando in un Simpletron e sarete effettivamente in grado di eseguire, provare e mettere a punto i programmi LMS che avete scritto nell'esercizio 7.18.

Nel momento in cui eseguirete il vostro simulatore Simpletron questo dovrà incominciare visualizzando:

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

Simulate la memoria del Simpletron con il vettore unidimensionale **memory** di 100 elementi. Supponete che il simulatore sia già in esecuzione ed esaminiamo il dialogo che si svilupperà con esso, man mano che immettiamo il programma mostrato nell'Esempio 2 dell'Esercizio 7.18:

```
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  ***
```

Ora che il programma LMS è stato immesso (o caricato) nel vettore **memory**, il Simpletron provvederà a eseguirlo. L'esecuzione comincerà con l'istruzione nella locazione **00** e, come in C, continuerà in modo sequenziale, sempre che non si dirami in qualche altra parte del programma a causa di un trasferimento di controllo.

Utilizzate la variabile **accumulator** per rappresentare il registro accumulatore. Usate la variabile **instructionCounter** per conservare l'indirizzo di memoria in cui sarà contenuta l'istruzione da eseguire. Utilizzate la variabile **operationCode** per indicare l'operazione da eseguire, ovverosia le due cifre a sinistra nella parola dell'istruzione. Usate la variabile **operand** per indicare la locazione di memoria su cui opererà l'istruzione corrente. In altri termini, **operand** corrisponderà alle due cifre più a destra dell'istruzione da eseguire. Non eseguite direttamente le istruzioni contenute nella memoria. Trasferite piuttosto quella da eseguire dalla memoria in una variabile chiamata **instructionRegister**. In seguito "staccherete" le due cifre di sinistra per sistemarle in **operationCode** e "separerete" le due cifre di destra per sistemarle in **operand**.

Nel momento in cui il Simpletron comincerà l'esecuzione i registri speciali saranno dunque inizializzati nel modo seguente:

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

Ora "seguiamo" l'esecuzione della prima istruzione LMS: il **+1009** sistemato nella locazione di memoria **00**. Quello che seguiremo è detto *ciclo di esecuzione dell'istruzione*.

La variabile **instructionCounter** ci indica la locazione della prossima istruzione da eseguire. "Preleveremo" dunque il contenuto di quella posizione dal vettore **memory** usando l'istruzione C:

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

Il codice dell'operazione e l'operando saranno estratti dal registro delle istruzioni con:

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

Ora il Simpletron è in grado di determinare che il codice dell'operazione corrisponde in realtà a una *read* (e non a una *write*, a una *load*, ecc.). Una struttura **switch** distinguerà le dodici operazioni del linguaggio LMS.

All'interno della struttura **switch** il comportamento delle varie istruzioni sarà simulato nel modo seguente (lasciamo a voi le altre istruzioni):

```
read:   scanf("%d", &memory[operand]);
load:   accumulator = memory[operand];
add:    accumulator += memory[cperand];
```

Varie istruzioni di salto: ne discuteremo tra breve.

```
halt:   Questa istruzione visualizza il messaggio
   *** Simpletron execution terminated ***
```

e in seguito visualizzerà il nome e il contenuto di tutti i registri, così come quello di tutta la memoria. Un output di questo genere è spesso chiamato *dump del computer* (ovverosia deposito del computer, che non è però il posto dove vanno a finire i vecchi computer). Per aiutarvi a implementare la vostra funzione di dump nella Figura 7.32 abbiamo riportato un esempio per il formato del dump. Osservate che un dump eseguito alla fine dell'esecuzione di un programma Simpletron dovrebbe mostrare i valori correnti delle istruzioni e dei dati, così come sono in quel momento.

Procediamo con l'esecuzione della prima istruzione del nostro programma, vale a dire la **+1009** della posizione **00**. Come abbiamo affermato prima, la struttura **switch** la simulerà eseguendo l'istruzione C

```
scanf("%d", &memory[operand]);
```

Prima che la funzione **scanf** sia eseguita, dovrà essere visualizzato un punto interrogativo (**?**) per richiedere l'input dell'utente. Il Simpletron attenderà che l'utente immetta un valore e prema il *tasto invio*. A quel punto il valore sarà sistemato nella locazione **09**.

La simulazione della prima istruzione è stata finalmente completata. Ci rimane solo la preparazione del Simpletron all'esecuzione della prossima istruzione. Dato che l'istruzione appena eseguita non era un trasferimento di controllo, avremo semplicemente bisogno di incrementare il registro contatore delle istruzioni, nel modo seguente:

```
++instructionCounter;
```

Questo passo completa davvero l'esecuzione simulata della prima istruzione. L'intero processo (ovverosia il ciclo di esecuzione dell'istruzione) ricomincerà nuovamente con il recupero della prossima istruzione da eseguire.

Ora esaminiamo in che modo saranno simulate le istruzioni di salto, ovverosia i trasferimenti di controllo. In effetti, sarà necessario solo aggiornare in modo appropriato il valore contenuto nel contatore di istruzioni. Di conseguenza, l'istruzione di salto incondizionato (**40**) sarà simulata all'interno della struttura **switch** con:


```
instructionCounter = operrand;
```

```
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
```

Figura 7.32 Un esempio di dump.

L'istruzione condizionale "salto se l'accumulatore è uguale a zero" sarà simulata con:

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

A questo punto, dovreste essere in grado di implementare il vostro simulatore di Simpletron e di eseguire ognuno dei programmi LMS che avete scritto nell'Esercizio 7.18. Potrete abbellire il linguaggio LMS con delle caratteristiche aggiuntive e implementarle nel vostro simulatore.

Il vostro simulatore dovrà effettuare dei controlli per vari tipi di errore. Per esempio, durante la fase di caricamento del programma, ogni numero che l'utente tenterà di immettere nel vettore **memory** del Simpletron dovrà essere compreso nell'intervallo da **-9999** a **+9999**. Il vostro simulatore dovrà quindi utilizzare un ciclo **while** per controllare che ogni numero immesso sia compreso in quell'intervallo e, in caso contrario, continuare a richiederlo all'utente finché non ne avrà immesso uno corretto.

Durante la fase di esecuzione, il vostro simulatore dovrà effettuare dei controlli per vari errori gravi, come i tentativi di eseguire delle divisioni per zero o di eseguire dei codici di operazione non validi, il superamento della capacità dell'accumulatore (ovverosia le operazioni aritmetiche che producono valori maggiori di **+9999** o minori di **-9999**) e altre cose simili. Insomma, occorrerà gestire gli *errori fatali*. Nel momento in cui avrà intercettato un errore fatale, il vostro simulatore dovrà visualizzare un messaggio di errore come:

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

e visualizzare un dump completo del computer nel formato di cui abbiamo discusso in precedenza. Ciò aiuterà l'utente a individuare l'errore del programma.

7.20 Modificate il programma di mescolamento e distribuzione delle carte proposto nella Figura 7.24, in modo che le suddette operazioni siano eseguite da una funzione unica (**shuffleAndDeal**).

Questa dovrà contenere una struttura di iterazione nidificata simile a quella della funzione **shuffle** mostrata nella Figura 7.24.

7.21 Che cosa farà questo programma?

```
#include <stdio.h>

void mystery1(char *, const char *);

main()
{
   char string1[80], string2[80];

   printf("Enter two strings: ");
   scanf("%s%s", string1, string2);
   mystery1(string1, string2);
   printf("%s\n", string1);
   return 0;
}

void mystery1(char *s1, const char *s2)
{
   while (*s1 != '\0')
      ++s1;

   for ( ; *s1 = *s2; s1++, s2++)
      ;  /* istruzione vuota */
}
```

7.22 Che cosa farà questo programma?

```
#include <stdio.h>

int mystery2(const char *);

main()
{
   char string[80];

   printf("Enter a string: ");
   scanf("%s", string);
   printf("%d\n", mystery2(string));
   return 0;
}

int mystery2(const char *s)
{
   int x = 0;

   for ( ; *s != '\0'; s++)
      ++x;

   return x;
}
```


7.23 Trovate l'errore in ognuno dei seguenti segmenti di programma. Spiegate in che modo sarà possibile correggerlo, sempre che lo sia.

a)
```
int *number;
printf("%d\n", *number);
```

b)
```
float *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```

d)
```
char s[] = "this is a character array";
int count;
for ( ; *s != '\0'; s++)
   printf("%c ", *s);
```

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```

f)
```
float x = 19.34;
float xPtr = &x;
printf("%f\n", xPtr);
```

g)
```
char *s;
printf("%s\n", s);
```

7.24 (*Quicksort*) Negli esempi e negli esercizi del Capitolo 6 abbiamo discusso le tecniche di ordinamento a bolle (bubble sort), di bucket sort e per selezione (selection sort). Presenteremo ora una tecnica ricorsiva di ordinamento chiamata Quicksort (ordinamento veloce). L'algoritmo fondamentale per un vettore di valori unidimensionale è il seguente:

1) *Passo di ripartizione:* prendete il primo elemento di un vettore disordinato e determinate la sua posizione finale in quello ordinato. Ciò avverrà quando tutti i valori del sottovettore sinistro, rispetto all'elemento, gli saranno inferiori e tutti quelli del sottovettore destro gli saranno superiori. A questo punto avremo un elemento sistemato nella sua giusta posizione e due sottovettori disordinati.
2) *Passo ricorsivo:* eseguite il passo 1 su ogni sottovettore disordinato.

Ogni volta che il passo 1 sarà eseguito su un sottovettore, un altro elemento sarà sistemato nella sua posizione finale all'interno del vettore ordinato e saranno creati due sottovettori disordinati. Considereremo sicuramente ordinato un sottovettore che sia formato da un solo elemento e, di conseguenza, questo sarà già nella sua posizione finale.

L'algoritmo di base sembra abbastanza semplice, ma in che modo determineremo la posizione finale del primo elemento di ogni sottovettore? Come esempio, consideriamo il seguente gruppo di valori (l'elemento in grassetto è quello utilizzato per la ripartizione, ovverosia, quello che sarà sistemato nella sua posizione finale all'interno del vettore ordinato):

37 2 6 4 89 8 10 12 68 45

1) Partendo da quello più a destra nel vettore, confrontate ogni elemento con il **37** finché non ne troviate uno minore che scambierete di posto con il **37**. Il primo elemento minore di **37** è 12, perciò **37** e 12 saranno scambiati di posto. Il nuovo vettore sarà:

12 2 6 4 89 8 10 **37** 68 45

L'elemento 12 è in corsivo per ricordare che è stato appena scambiato di posto con il **37**.

2) Partendo dalla sinistra del vettore, ma cominciando da quello successivo al 12, confrontate ogni elemento con il **37** finché non ne troviate uno maggiore che scambierete di posto con il **37**. Il primo elemento maggiore di **37** è 89, perciò **37** e 89 saranno scambiati di posto. Il nuovo vettore sarà:

   12 2 6 4 **37** 8 10 *89* 68 45

3) Partendo dalla destra, ma cominciando da quello precedente a 89, confrontate ogni elemento con il **37** finché non ne troviate uno minore che scambierete di posto con il **37**. Il primo elemento minore di **37** è 10, perciò **37** e 10 saranno scambiati di posto. Il nuovo vettore sarà:

   12 2 6 4 *10* 8 **37** 89 68 45

4) Partendo dalla sinistra, ma cominciando da quello successivo al 10, confrontate ogni elemento con il **37** finché non ne troviate uno maggiore che scambierete di posto con il **37**. In questo caso non ci sono altri elementi maggiori di **37** perciò, confrontando il **37** con se stesso, sapremo che sarà già stato sistemato nella sua posizione finale all'interno del vettore ordinato.

Una volta che al suddetto vettore sarà stata applicata la ripartizione, ci saranno due sottovettori disordinati. Il sottovettore dei valori minori di 37 conterrà 12, 2, 6, 4, 10 e 8, mentre quello dei valori maggiori di 37 conterrà 89, 68 e 45. L'algoritmo di ordinamento dovrà dunque procedere con la ripartizione di entrambi i sottovettori nello stesso modo utilizzato per ripartire quello originale.

Basandovi sulla discussione precedente scrivete una funzione ricorsiva **quicksort** che ordini un vettore unidimensionale di valori interi. La funzione dovrà ricevere come argomenti un vettore di valori interi, un indice di partenza e uno di fine. La funzione **partition** dovrà essere richiamata da **quicksort** per eseguire il passo di ripartizione.

7.25 (*Attraversamento di un labirinto*) La griglia seguente è la rappresentazione di un labirinto all'interno di una matrice.

```
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 0 1
0 0 1 0 1 0 1 1 1 1 0 1
1 1 1 0 1 0 0 0 0 1 0 1
1 0 0 0 0 1 1 1 0 1 0 0
1 1 1 1 0 1 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1 0 1
1 0 0 0 0 0 0 0 0 1 0 1
1 1 1 1 1 1 0 1 1 1 0 1
1 0 0 0 0 0 0 1 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

I numeri uno e zero rappresentano rispettivamente le pareti e i corridoi del labirinto.

Esiste un semplice algoritmo di attraversamento di un labirinto che garantisce il ritrovamento dell'uscita (sempre che ce ne sia una). Nel caso che l'uscita non ci fosse vi ritrovereste di nuovo al punto di partenza. Poggiate la mano destra sulla parete alla vostra destra e cominciate a camminare in avanti. Non rimuovete mai la vostra mano dalla parete. Se il labirinto svolta a destra, seguite il muro a destra. Alla fine arriverete sicuramente all'uscita del labirinto, se nel frattempo non avrete rimosso la vostra mano dalla parete. È probabile che esista un percorso più breve di quello che avete intrapreso, ma in questo modo avrete la certezza di uscire dal labirinto.

Scrivete la funzione ricorsiva **mazeTraverse** per attraversare il labirinto. La funzione dovrà ricevere come argomenti una matrice di caratteri 12 per 12, per rappresentare il labirinto, e la posizione di partenza all'interno dello stesso. Man mano che **mazeTraverse** tenterà di individuare l'uscita dal

labirinto, dovrà inserire il carattere **x** in ogni casella del percorso. La funzione dovrà visualizzare il labirinto dopo ogni mossa così che l'utente possa seguire la risoluzione del labirinto.

7.26 (*Generazione casuale di labirinti*) Scrivete una funzione **mazeGenerator** che riceva come argomento una matrice di caratteri 12 per 12 e produca un labirinto a caso. La funzione dovrà anche restituire i punti di ingresso e di uscita del labirinto. Provate la vostra funzione **mazeTraverse** dell'Esercizio 7.25 utilizzando vari labirinti generati a caso.

7.27 (*Labirinti di qualsiasi dimensione*) Generalizzate le funzioni **mazeTraverse** e **mazeGenerator** degli Esercizi 7.25 e 7.26, in modo da elaborare labirinti di qualsiasi larghezza e altezza.

7.28 (*Vettori di puntatori a funzioni*) Riscrivete il programma della Figura 6.22 in modo da utilizzare un'interfaccia guidata da menu. Il programma dovrà offrire all'utente le seguenti 4 opzioni:

```
Enter a choice:
  0 Print the array of grades
  1 Find the minimum grade
  2 Find the maximum grade
  3 Print the average on all tests for each student
  4 End program
```

Una restrizione all'utilizzo di vettori di puntatori a funzioni è che tutti i puntatori devono essere dello stesso tipo. Di conseguenza, devono essere dei puntatori a funzioni che restituiscano tutte lo stesso tipo di dato e che ricevano argomenti dello stesso tipo. Per questa ragione, le funzioni nella Figura 6.22 dovranno essere modificate in modo che restituiscano lo stesso tipo di dato e ricevano i medesimi parametri. Modificate le funzioni **minimum** e **maximum** in modo che visualizzino il valore minimo e quello massimo e non restituiscano niente. Per l'opzione 3, modificate la funzione **average** della Figura 6.22 in modo che invii in output la media di ogni studente (invece che quella di uno specifico). La funzione **average** non dovrà restituire nessun dato e riceverà gli stessi parametri di **printArray**, **minimum** e **maximum**. Immagazzinate i puntatori alle quattro funzioni nel vettore **processGrades** e utilizzate la scelta effettuata dall'utente come indice di vettore per richiamare le funzioni.

7.28 (*Modifiche al simulatore Simpletron*) Nell'Esercizio 7.19 avete scritto una simulazione software di un computer che esegue i programmi scritti nel Linguaggio Macchina Simpletron (LMS). In questo esercizio, proporremo diverse modifiche e miglioramenti al Simulatore Simpletron. Negli Esercizi 12.26 e 12.27, proporremo la costruzione di un compilatore che convertirà i programmi scritti in un linguaggio ad alto livello (una variante del BASIC) nel Linguaggio Macchina del Simpletron. Alcune delle seguenti modifiche e migliorie potranno essere necessarie per eseguire i programmi prodotti dal compilatore.

a) Estendete la memoria del Simulatore Simpletron, in modo che possa contenere 1000 locazioni e consentire al Simpletron di gestire programmi più corposi.
b) Fate in modo che il simulatore possa eseguire il calcolo del modulo. Sarà necessario aggiungere un'istruzione al Linguaggio Macchina Simpletron.
c) Fate in modo che il simulatore possa calcolare l'elevamento a potenza. Sarà necessario aggiungere un'istruzione al Linguaggio Macchina Simpletron.
d) Modificate il simulatore in modo che utilizzi dei valori esadecimali, invece di quelli interi, per rappresentare le istruzioni del Linguaggio Macchina Simpletron.
e) Modificate il simulatore per consentire la visualizzazione di un carattere newline. Sarà necessario aggiungere un'istruzione al Linguaggio Macchina Simpletron.
f) Modificate il simulatore in modo che possa elaborare anche dei valori in virgola mobile oltre a quelli interi.
g) Modificate il simulatore in modo che possa gestire l'input di stringhe. Suggerimento: ogni parola del Simpletron potrà essere suddivisa in due gruppi contenenti ognuno un intero

di due cifre. Ogni intero di due cifre rappresenterà il valore decimale ASCII equivalente a un carattere. Aggiungete un'istruzione in linguaggio macchina che prenda in input una stringa e la immagazzini in una specifica locazione della memoria del Simpletron. In quella locazione, la prima metà della parola conterrà il numero di caratteri inclusi nella stringa (ovverosia, la sua lunghezza). Ogni mezza parola successiva conterrà un carattere ASCII espresso con un valore decimale di due cifre. L'istruzione in linguaggio macchina convertirà ogni carattere nel suo equivalente ASCII e lo assegnerà alla mezza parola.

h) Modificate il simulatore in modo che gestisca l'output delle stringhe immagazzinate nel formato descritto nella parte (g). Suggerimento: aggiungete al linguaggio macchina un'istruzione che visualizzi una stringa cominciando da una certa locazione della memoria del Simpletron. In quella locazione, la prima metà della parola conterrà il numero dei caratteri inclusi nella stringa (ovverosia, la sua lunghezza). Ogni mezza parola successiva conterrà un carattere ASCII espresso con un valore decimale di due cifre. L'istruzione in linguaggio macchina controllerà la lunghezza e visualizzerà la stringa, traducendo ogni numero di due cifre nel carattere equivalente.

7.30 Che cosa farà il seguente programma?

```
#include <stdio.h>

int mystery3(const char *, const char *);

main()
{
  char string1[80], string2[80];

  printf("Enter two strings: ");
  scanf("%s%s", string1, string2);
  printf("The result is %d\n", mystery3(string1, string2));

  return 0;
}

int mystery3(const char *s1, const char *s2)
{
  for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++)

    if (*s1 != *s2)
      return 0;

  return 1;
}
```