

# FILE BINARI

---

**Un file binario è una pura sequenza di byte,**  
senza alcuna strutturazione particolare

- È un'astrazione di memorizzazione **assolutamente generale**, usabile per memorizzare su file *informazioni di qualsiasi natura*
  - snapshot della memoria
  - rappresentazioni interne binarie di numeri
  - immagini, audio, musica, ...
  - ... volendo, *anche caratteri*
- I file di testo non sono indispensabili: sono semplicemente *comodi*

# FILE BINARI

---

- Un file binario è una sequenza di byte
- Può essere usato per archiviare su memoria di massa **qualunque tipo di informazione**
- Input e output avvengono sotto forma di una **sequenza di byte**
- **La fine del file è SEMPRE rilevata in base all'esito delle operazioni di lettura**
  - **non ci può essere EOF**, perché un file binario non è una sequenza di caratteri
  - **qualsiasi byte si scegliesse come marcatore**, potrebbe sempre capitare nella sequenza

## FILE BINARI

---

Poiché un file binario è una sequenza di byte,  
sono fornite due funzioni per *leggere e  
scrivere sequenze di byte*

- **fread()** legge una sequenza di byte
- **fwrite()** scrive una sequenza di byte

# OUTPUT BINARIO: `fwrite()`

---

Sintassi:

```
int fwrite(addr, int dim, int n, FILE *f);
```

- **scrive sul file n elementi, ognuno grande dim byte**  
(complessivamente, scrive quindi  $n \cdot dim$  byte)
- gli elementi da scrivere vengono **prelevati dalla memoria a partire dall'indirizzo addr**
- **restituisce il numero di elementi (non di byte) effettivamente scritti**, che possono essere meno di n

# INPUT BINARIO: fread()

---

Sintassi:

```
int fread(addr, int dim, int n, FILE *f);
```

- legge dal file **n** elementi, ognuno grande **dim** byte  
(complessivamente, tenta di leggere quindi  $n \cdot dim$  byte)
- gli elementi da leggere vengono **scritti in memoria a partire dall'indirizzo** **addr**
- restituisce il numero di elementi (non di byte) effettivamente letti, che possono essere meno di n se il file finisce prima. **Controllare il valore restituito è il SOLO MODO per sapere che cosa è stato letto**, e in particolare **per scoprire se il file è terminato**

# ESEMPIO 1

Salvare su un file binario **numeri.dat** il contenuto di un array di dieci interi

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
FILE *fp;
int vet[10] = {1,2,3,4,5,6,7,8,9,10};
if ((fp = fopen("numeri.dat", "wb"))==NULL)
    exit(1); /* Errore di apertura */
fwrite(vet, sizeof(int), 10, fp);
fclose(fp);
}
```

In alternativa:

```
fwrite(vet, 10*sizeof(int), 1, fp)
```

**sizeof()** è essenziale per la portabilità del sorgente: la dimensione di **int** non è fissa

## ESEMPIO 2

Leggere da un file binario `numeri.dat` una sequenza di interi, scrivendoli in un array

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    FILE *fp;
    int vet[40], i, n;
    if ((fp = fopen("numeri.dat", "rb")) == NULL)
        exit(1); /* Errore di apertura */
    n = fread(vet, sizeof(int), 40, fp);
    for (i=0; i<n; i++) printf("%d ", vet[i]);
    fclose(fp);
}
```

`fread` tenta di leggere 40 interi, ne legge meno se il file finisce prima

`n` contiene il numero di interi effettivamente letti

## ESEMPIO 3

---

Scrivere su un file di caratteri **testo.txt** una sequenza di caratteri

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *fp; int n;
    char msg[] = "Ah, l'esame\nsi avvicina!";
    if ((fp = fopen("testo.txt", "wb")) == NULL)
        exit(1); /* Errore di apertura */
    fwrite(msg, strlen(msg)+1, 1, fp);
    fclose(fp);
}
```

Dopo averlo creato, provare ad aprire questo file con un editor qualunque

Un carattere in C ha sempre **size=1**  
Scelta: salvare anche terminatore stringa

## ESEMPIO 4: OUTPUT DI NUMERI

---

L'uso di file binari consente di rendere evidente la differenza fra la *rappresentazione interna* di un numero e la sua *rappresentazione esterna* come *stringa di caratteri in una certa base*

- Supponiamo che sia `int x = 31466;`
- Che differenza c'è fra:

`fprintf(file,"%d", x);`

`fwrite(&x, sizeof(int), 1, file);`

## ESEMPIO 4: OUTPUT DI NUMERI

---

Se **x** è un intero che vale 31466, internamente la sua rappresentazione è (su 16 bit):      01111010 11101010

- **fwrite()** emette direttamente tale sequenza, scrivendo quindi i **due byte** sopra indicati
- **fprintf()** invece emette la sequenza di caratteri **ASCII** corrispondenti alla rappresentazione esterna del numero 31466, ossia i **cinque byte**

00110011 00110001 00110100 00110110 00110110

Se per sbaglio si emettessero **su un file di testo** (o su video) direttamente i due byte: 01111010 11101010 si otterrebbero *i caratteri corrispondenti al codice ASCII di quei byte*: **éz**

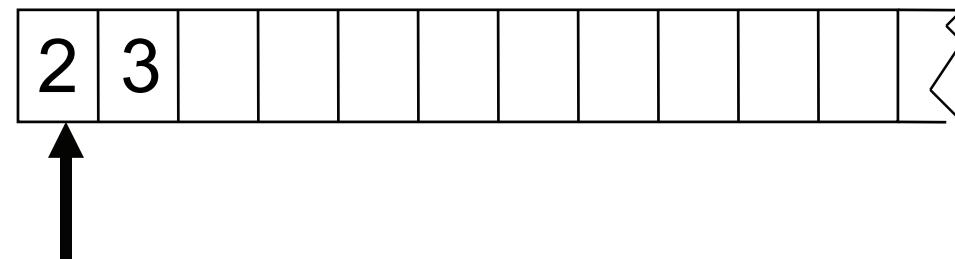
## ESEMPIO 5: INPUT DI NUMERI

---

Analogamente, che differenza c'è fra

`fscanf(file, "%d", &x);` e  
 `fread(&x, sizeof(int), 1, file);`

nell'ipotesi che il file (di testo) contenga la sequenza di caratteri “23”?



## ESEMPIO 5: INPUT DI NUMERI

---

`fscanf()` preleva la **stringa di caratteri ASCII**

carattere '2' → 00110010 00110011 ← carattere '3'

che costituisce la rappresentazione esterna del numero, e la **converte** nella corrispondente rappresentazione interna, ottenendo i due byte:

00000000 00010111

che rappresentano in binario il valore **ventitre**

## ESEMPIO 5: INPUT DI NUMERI

---

`fread()` invece preleverebbe i due byte

carattere '2' 00110010 00110011 carattere '3'

credendoli già la rappresentazione interna di un numero, senza fare alcuna conversione

In questo modo sarebbe inserita nella variabile x esattamente la sequenza di byte sopra indicata, che verrebbe quindi interpretata come il numero **tredicimilacentosei**

## ESEMPIO COMPLETO FILE BINARIO

---

È dato un file binario **people.dat** i cui record rappresentano ciascuno i dati di una persona, secondo il seguente formato:

- **cognome** (al più 30 caratteri)
- **nome** (al più 30 caratteri)
- **sesso** (un singolo carattere, '**M**' o '**F**' )
- **anno di nascita**

Si noti che la **creazione del file binario deve essere sempre fatta da programma**, mentre per i file di testo può essere fatta con un text editor (che produce sempre e solo file di testo)

# CREAZIONE FILE BINARIO

---

È necessario scrivere un programma che lo crei strutturandolo in modo che ogni record contenga una

```
struct persona{  
    char cognome[31], nome[31], sesso[2];  
    int anno;  
};
```

I dati di ogni persona da inserire nel file vengono richiesti all'utente tramite la funzione `leggiel()` che non ha parametri e restituisce come valore di ritorno la `struct persona` letta. Quindi il prototipo è:

```
struct persona leggiel();
```

# CREAZIONE FILE BINARIO

---

```
struct persona leggiel(){
    struct persona e;

    printf("Cognome ? ");
    scanf("%s", e.cognome);
    printf("\n Nome ? ");
    scanf("%s", e.nome);
    printf("\nSesso ? ");
    scanf("%s", e.sesso);
    printf("\nAnno nascita ? ");
    scanf("%d", &e.anno);
    return e;
}
```

# CREAZIONE FILE BINARIO

---

```
#include <stdio.h>
#include <stdlib.h>
struct persona{
    char cognome[31], nome[31], sesso[2];
    int anno;
};
struct persona leggiel();
int main(void){
FILE *f; struct persona e; int fine=0;
f=fopen("people.dat", "wb");
while (!fine)
    { e=leggiel();
        fwrite(&e,sizeof(struct persona),1,f);
        printf("\nFine (SI=1, NO=0)?");
        scanf("%d", &fine);
    }
fclose(f); }
```

## CREAZIONE FILE BINARIO

---

L'esecuzione del programma precedente crea il file binario contenente i dati immessi dall'utente.  
Solo a questo punto il file può essere utilizzato

Il file **people.dat** non è visualizzabile tramite un text editor: questo sarebbe il risultato

```
rossi >L yy @| T —8| □ | â3 mario L
```

## ESEMPIO COMPLETO FILE BINARIO

---

Ora si vuole scrivere un programma che

- legga record per record i dati dal file
- ponga i dati in un array di persone
- ... (*poi svolgeremo elaborazioni su essi*)

# ESEMPIO COMPLETO FILE BINARIO

---

## 1) Definire una struttura di tipo **persona**

Occorre definire una **struct** adatta a ospitare i dati elencati:

- **cognome** → array di 30+1 caratteri
- **nome** → array di 30+1 caratteri
- **sesso** → array di 1+1 caratteri
- **anno di nascita** → un intero

```
struct persona{  
    char cognome[31], nome[31], sesso[2];  
    int anno;  
};
```

## ESEMPIO COMPLETO FILE BINARIO

---

- 2) definire un array di **struct persona**
  - 3) aprire il file in lettura
- 

```
int main(void) {  
    struct persona v[DIM];  
    FILE* f = fopen("people.dat", "rb");  
    if (f==NULL) {  
        printf("Il file non esiste");  
        exit(1); /* terminazione del programma */  
    }  
    ...  
}
```

## ESEMPIO COMPLETO FILE BINARIO

---

- 4) leggere i record dal file, e porre i dati di ogni persona in una cella dell'array
- 

### Come organizzare la lettura?

`int fread(addr, int dim, int n, FILE *f);`

- legge dal file **n** elementi, ognuno grande **dim** byte  
(complessivamente, legge quindi  $n \cdot dim$  byte)
- gli elementi da leggere vengono scritti in memoria a partire dall'indirizzo **addr**

*Uso fread()*

# ESEMPIO COMPLETO FILE BINARIO

---

```
#define DIM 30
#include <stdio.h>
#include <stdlib.h>

struct persona{
    char cognome[31], nome[31], sesso[2];
    int anno;
};

int main(void) {
    struct persona v[DIM]; int i=0; FILE* f;
    if ((f=fopen("people.dat", "rb"))==NULL) {
        printf("Il file non esiste!"); exit(1);
    }
    while(fread(&v[i], sizeof(struct persona), 1, f)>0)
        i++;
}
```

# ESEMPIO COMPLETO FILE BINARIO

---

Che cosa far leggere a **fread()** ?

*Se vogliamo, anche l'intero vettore di strutture:  
unica lettura per **DIM** record (solo se sappiamo  
a priori che i record da leggere sono **esattamente**  
**DIM**)*

```
fread(v,sizeof(struct persona) ,DIM,f)
```

# ESEMPIO COMPLETO FILE BINARIO

---

```
#define DIM 30
#include <stdio.h>
#include <stdlib.h>

struct persona{
    char cognome[31], nome[31], sesso[2];
    int anno;
};

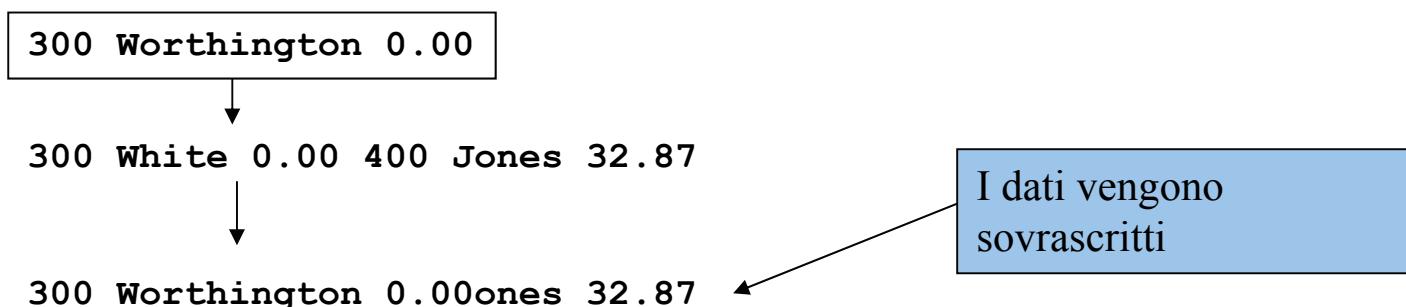
int main(void) {
    struct persona v[DIM]; int i=0; FILE* f;
    if ((f=fopen("people.dat", "rb"))==NULL) {
        printf("Il file non esiste!"); exit(1);
    }
    fread(v,sizeof(struct persona),DIM,f);
}
```

# Files ad accesso sequenziale:

- Se vengono modificati i dati possono essere distrutti.
- I campi possono variare in dimensione
  - (non sono la rappresentazione interna)
  - 1, 34, -890 sono tutti interi `int`, ma possono avere differenti rappresentazioni.

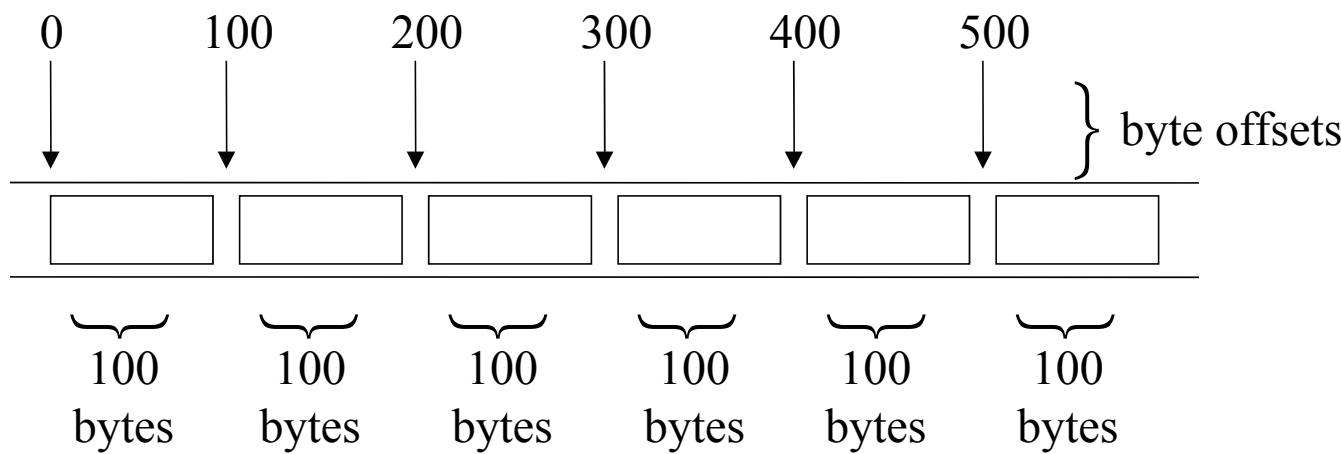
300 white 0.00 400 Jones 32.87 (vecchio valore)

Se vogliamo cambiarlo:



# Files ad accesso diretto

- Files ad accesso diretto:
  - Accedono i record individualmente e direttamente
  - I dati possono essere inseriti senza distruggere altri dati
  - I dati precedenti possono essere modificati o cancellati.
- Realizzati usando record di lunghezza fissa
  - (I files sequenziali non hanno lunghezza fissa)



# Creare un file ad accesso diretto (binario)

- Dati:
  - Non sono formattati (sono solo bytes)
    - Tutti i dati dello stesso tipo usano la stessa memoria (`int`, per esempio)
    - Tutti i record dello stesso tipo hanno la stessa lunghezza
    - I dati sono “illeggibili” per l’utente

```

1 /* Fig. 11.11: fig11_11.c
2      Creating a randomly accessed file sequentially */
3 #include <stdio.h>
4
5 /* clientData structure definition */
6 struct clientData {
7     int acctNum;          /* account number */
8     char lastName[ 15 ];  /* account last name */
9     char firstName[ 10 ]; /* account first name */
10    double balance;      /* account balance */
11}; /* end structure clientData */
12
13 int main()
14 {
15     int i; /* counter */
16
17     /* create clientData with no information */
18     struct clientData blankClient = { 0, "", "", 0.0 };
19
20     FILE *cfPtr; /* credit.dat file pointer */
21
22     /* fopen opens the file; exits if file cannot be opened */
23     if ( ( cfPtr = fopen( "credit.dat", "wb" ) ) == NULL ) {
24         printf( "File could not be opened.\n" );
25     } /* end if */

```

```
26     else {
27
28         /* output 100 blank records to file */
29         for ( i = 1; i <= 100; i++ ) {
30             fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
31         } /* end for */
32
33         fclose ( cfPtr ); /* fclose closes the file */
34     } /* end else */
35
36     return 0; /* indicates successful termination */
37
38 } /* end main */
```

## Posizionamento del puntatore di posizione all'interno del file

- **fseek**
  - Setta il file position pointer ad una certa posizione
  - `fseek( pointer, offset, symbolic_constant );`
    - *pointer* – puntatore al file
    - *offset* – file position pointer (0 e` la prima locazione del file)
    - *symbolic\_constant* – specifica da dove partire
    - SEEK\_SET – dall'inizio del file
    - SEEK\_CUR – dalla corrente locazione
    - SEEK\_END – alla fine del file

# File position pointer

- File position pointer
  - Indica la posizione da cui leggere/scrivere I prossimi byte
  - Non un puntatore, ma un valore intero (specifica locazioni di bytes)
  - E` anche chiamato byte offset
- **rewind( cfPtr )**
  - Riposiziona il file position pointer all'inizio del file (byte 0).

```
1 /* Fig. 11.12: fig11_12.c
2      Writing to a random access file */
3 #include <stdio.h>
4
5 /* clientData structure definition */
6 struct clientData {
7     int acctNum;          /* account number */
8     char lastName[ 15 ];  /* account last name */
9     char firstName[ 10 ]; /* account first name */
10    double balance;      /* account balance */
11}; /* end structure clientData */
12
13 int main()
14 {
15     FILE *cfPtr; /* credit.dat file pointer */
16
17     /* create clientData with no information */
18     struct clientData client = { 0, "", "", 0.0 };
19
20     /* fopen opens the file; exits if file cannot be opened */
21     if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
22         printf( "File could not be opened.\n" );
23     } /* end if */
24     else {
25
```

```

26     /* require user to specify account number */
27     printf( "Enter account number"
28             " ( 1 to 100, 0 to end input )\n? " );
29     scanf( "%d", &client.acctNum );
30
31     /* user enters information, which is copied into file */
32     while ( client.acctNum != 0 ) {
33
34         /* user enters last name, first name and balance */
35         printf( "Enter lastname, firstname, balance\n? " );
36
37         /* set record lastName, firstName and balance value */
38         fscanf( stdin, "%s%s%lf", client.lastName,
39                 client.firstName, &client.balance );
40
41         /* seek position in file of user-specified record */
42         fseek( cfPtr, ( client.acctNum - 1 ) *
43                 sizeof( struct clientData ), SEEK_SET );
44
45         /* write user-specified information in file */
46         fwrite( &client, sizeof( struct clientData ), 1, cfPtr );
47
48         /* enable user to specify another account number */
49         printf( "Enter account number\n? " );
50         scanf( "%d", &client.acctNum );

```

```
51     } /* end while */  
52  
53     fclose( cfPtr ); /* fclose closes the file */  
54 } /* end else */  
55  
56 return 0; /* indicates successful termination */  
57  
58 } /* end main */
```

```
Enter account number ( 1 to 100, 0 to end input )  
? 37  
Enter lastname, firstname, balance  
? Barker Doug 0.00  
Enter account number  
? 29  
Enter lastname, firstname, balance  
? Brown Nancy -24.54  
Enter account number  
? 96  
Enter lastname, firstname, balance  
? Stone Sam 34.98  
Enter account number  
? 88  
Enter lastname, firstname, balance  
? Smith Dave 258.34  
Enter account number  
? 33  
Enter lastname, firstname, balance  
? Dunn Stacey 314.33  
Enter account number  
? 0
```

## Program Output

# Struttura in memoria

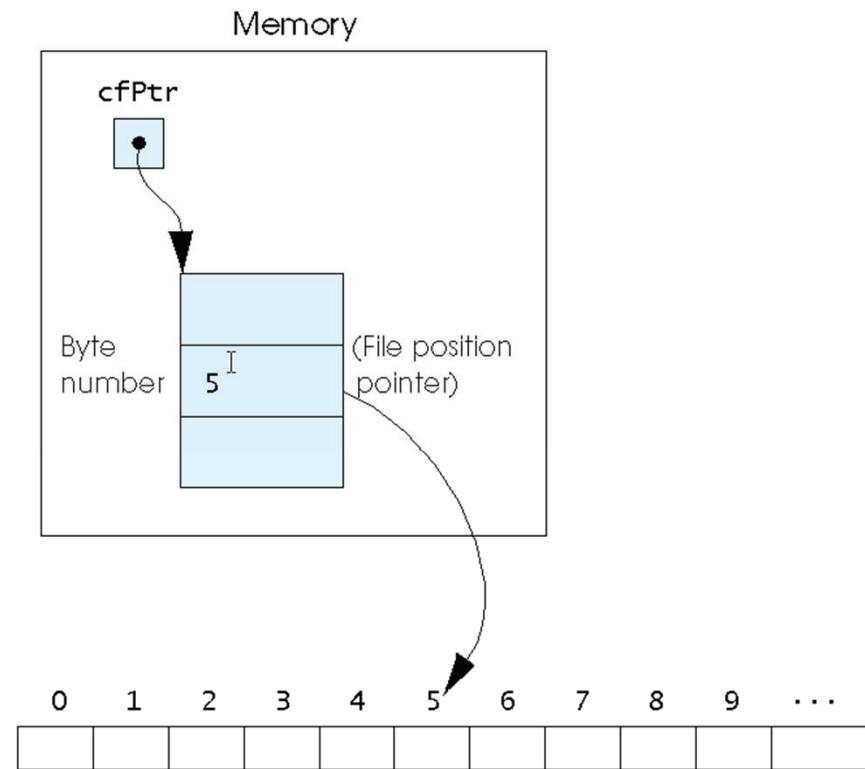


Fig. 11.14 The file position pointer indicating an offset of 5 bytes from the beginning of the file.

```
1 /* Fig. 11.15: fig11_15.c
2      Reading a random access file sequentially */
3 #include <stdio.h>
4
5 /* clientData structure definition */
6 struct clientData {
7     int acctNum;          /* account number */
8     char lastName[ 15 ];  /* account last name */
9     char firstName[ 10 ]; /* account first name */
10    double balance;      /* account balance */
11}; /* end structure clientData */
12
13 int main()
14 {
15     FILE *cfPtr; /* credit.dat file pointer */
16
17     /* create clientData with no information */
18     struct clientData client = { 0, "", "", 0.0 };
19
20     /* fopen opens the file; exits if file cannot be opened */
21     if ( ( cfPtr = fopen( "credit.dat", "rb" ) ) == NULL ) {
22         printf( "File could not be opened.\n" );
23     } /* end if */
```

```
24     else {
25         printf( "%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
26                 "First Name", "Balance" );
27
28         /* read all records from file (until eof) */
29         while ( !feof( cfPtr ) ) {
30             fread( &client, sizeof( struct clientData ), 1, cfPtr );
31
32             /* display record */
33             if ( client.acctNum != 0 ) {
34                 printf( "%-6d%-16s%-11s%10.2f\n",
35                         client.acctNum, client.lastName,
36                         client.firstName, client.balance );
37             } /* end if */
38
39         } /* end while */
40
41         fclose( cfPtr ); /* fclose closes the file */
42     } /* end else */
43
44     return 0; /* indicates successful termination */
45
46 } /* end main */
```

Acct	Last Name	First Name	Balance
29	Brown	Nancy	-24.54
33	Dunn	Stacey	314.33
37	Barker	Doug	0.00
88	Smith	Dave	258.34
96	Stone	Sam	34.98

# Program Output