Programmare in C su Amiga

di Dario de Judicibus

seconda puntata

Prima di introdurre l'argomento di questa seconda puntata, vediamo insieme una possibile soluzione al problema posto la volta scorsa.

Si trattava, come certamente ricorderete, di trovare una soluzione semplice, pratica ed elastica al problema di chiudere le librerie già aperte in caso di uscita forzata dal programma. Di fatto questa tecnica può essere adattata a qualunque altra operazione di chiusura che non sia quella relativa a librerie, come ad esempio:

- chiusura di file,
- deallocazione di memoria,
- cancellazione di aree temporanee su disco,
- e via dicendo...

Lo scopo di tale tecnica è fondamentalmente quello di garantire un disegno strutturato del programma, di facile manutenzione e lettura. Ricordatevi sempre che se un programma non è chiaro, sia per la forma in cui è scritto [*layout*], sia per la mancanza di commenti e prologhi (vedi riquadro), sarà poi difficile per lo stesso autore leggerlo e modificarlo, magari qualche mese dopo averlo scritto.

Facendo ora riferimento alla figura 1, vediamo come può essere risolto il problema. Ovviamente, al posto di

lib001.library, lib002.library... e Lib001Base, Lib002Base...

andranno poi sostituiti i nomi reali vuoi delle librerie Amiga, vuoi dei rispettivi puntatori base.

Innanzi tutto definiamo una maschera [*mask*] di quattro byte, per memorizzare le librerie già aperte man mano che andiamo avanti con il programma.

Quindi definiamo enne segnalatori [*flag*], uno per libreria. Ogni flag è formato da una parola con tutti i bit posti a zero tranne uno. Tutti i flag sono diversi, ovviamente.

Ogni volta che apriamo una libreria, poniamo ad uno il bit relativo a quella libreria nella maschera appositamente creata:

```
/* LIB_XXX è il flag relativo alla libreria aperta */
 LMask |= LIB_XXX
```

A questo punto, se qualcosa va male, prima di uscire chiamiamo una procedura (CleanClose) che «legge» la maschera e chiude quelle librerie corrispondenti ai bit posti ad uno. Ovviamente ognuno può chiamare tale procedura con un nome diverso se vuole, dato che non è una procedura di sistema. Nel momento in cui sorga la necessità di aggiungere un'altra libreria, basterà definire un altro flag, «aggiornare» la maschera subito dopo aver effettuato l'apertura ed aggiungere alla **CleanClose** una verifica sul nuovo flag. Semplice, non è vero?

Attenti però:

1) la maschera va sovrapposta al flag dopo che la libreria è stata aperta con successo;

2) non confondete l'espressione **LMask & LIB_001** con quella **LMask && LIB_001**, quest'ultima essendo sempre vera anche se è stata aperta una sola libreria, qualunque essa sia;

3) ricordatevi di chiamare la **CleanClose** anche alla fine del programma.

Si chiama *prologo* quel blocco di commenti che viene posto all'inizio di un programma che fornisce informazioni quali:

- l'autore o gli autori del programma,
- lo scopo del programma,
- la sintassi di chiamata (interfaccia con il CLI),
- i dati in ingresso e le aree utilizzate (su disco ed in memoria),
- i dati in uscita (su disco),
- i codici di ritorno,
- i prerequisiti minimali.

Questi ultimi riguardano le caratteristiche minime che deve avere l'ambiente di esecuzione [*Execution-Time Environment*] per poter girare il programma [*Program Run*], quali la versione minima necessaria del Sistema Operativo e delle librerie, l'esistenza o meno di altri programmi o comandi nel sistema, l'esistenza o meno di file iniziali [*profile*].

Introduzione

Ed eccoci finalmente all'argomento del giorno: AmigaDOS.

Chiunque di voi abbia lavorato con il CLI, conosce bene comandi come **dir, cd, info** ed **assign.** Questi comandi altro non sono che programmi che utilizzano i servizi offerti dall'AmigaDOS per permettere all'utente di lavorare con i propri file e di muoversi a piacere nella struttura a directory dell'Amiga [*directory tree*].

L'AmigaDOS è infatti un Sistema Operativo multi-processo, un sistema operativo cioè, che permette a più lavori [*job*] di essere presenti contemporaneamente nel sistema. Come abbiamo già accennato, un processo è formato da un

task più altre informazioni che vedremo in seguito, rappresentate da particolari strutture dati definite in **libraries/dosextens.h.**

Questa puntata cercheremo di portare il lettore ad una coscenza più approfondita del Sistema Operativo, visto dal punto di vista del programmatore C, lo introdurremo al Sistema di Archiviazione [*Filing System*] dell'Amiga e lo prepareremo ad affrontare un componente fondamentale del primo livello (vedi Prima Puntata): EXEC.

Ci occuperemo in particolare di:

- operazioni di I/O da e su CLI
- apertura e chiusura di file
- operazioni I/O su file
- comandi di gestione dell'I/O

L'AmigaDOS

I programmi che vedremo nei paragrafi successivi vengono tipicamente eseguiti da CLI, dato che spesso utilizzano direttamente od indirettamente informazioni relative alla finestra CLI dalla quale sono stati lanciati.

Si consiglia pertanto di evitare, almeno per il momento, di utilizzare gli esempi riportati in programmi eseguibili da WorkBench. Una volta acquisita una certa dimestichezza con i concetti principali dell'AmigaDOS, potrete usare una parte delle funzioni qui presentate anche in programmi più complessi, lanciabili anche da WorkBench.

I/O da e su CLI

Vediamo innanzi tutto come si passano i parametri ad un programma che va eseguito da CLI, come scrivere sulla finestra CLI e come ricevere un testo dalla tastiera. Vedremo in seguito altri modi di leggere e scrivere da CLI, tuttavia sarà necessario prima introdurre qualche nuovo concetto. Per adesso limitiamoci al metodo più semplice.

Tra parentesi, tale tecnica di I/O è quella più portabile tra sistemi differenti, dato che fa uso di funzioni standard del C.

Come passare i parametri al programma

Dopo aver compilato un programma, sia esso formato da più moduli, sia esso un unico file, è necessario legare [*link*] i moduli oggetto (quelli che terminano con **.o**) fra di loro e

```
/*************************************************************************\
 *  ATTENZIONE: quanto segue è situato PRIMA del programma principale    *
\*************************************************************************/

/*
 *  Definisci i flag di controllo per la chiusura delle librerie
 */

/* LONG = 4 bytes > 1 2 3 4 */ LONG LMask = 0L;

 #define LIB_001 0x00000001 /* Libreria Numero 1 - Byte #4 -> 0000 0001 */
 #define LIB_002 0x00000002 /* Libreria Numero 2 - Byte #4 -> 0000 0010 */
 #define LIB_003 0x00000004 /* Libreria Numero 3 - Byte #4 -> 0000 0100 */
 #define LIB_004 0x00000008 /* Libreria Numero 4 - Byte #4 -> 0000 1000 */
 #define LIB_005 0x00000010 /* Libreria Numero 5 - Byte #4 -> 0001 0000 */
 #define LIB_006 0x00000020 /* Libreria Numero 6 - Byte #4 -> 0010 0000 */
 #define LIB_007 0x00000040 /* Libreria Numero 7 - Byte #4 -> 0100 0000 */

 extern struct Library *OpenLibrary();

 struct Lib001Base *Lib001Base;
 struct Lib002Base *Lib002Base;
 struct Lib003Base *Lib003Base;
 struct Lib004Base *Lib004Base;
 struct Lib005Base *Lib005Base;
 struct Lib006Base *Lib006Base;
 struct Lib007Base *Lib007Base;
```

1a

```
/*************************************************************************\
 *  *** * Procedura per la chiusura delle librerie: CleanClose() * ***  *
\*************************************************************************/

 VOID CleanClose()
 {
   if (LMask & LIB_001) CloseLibrary(Lib001Base);
   if (LMask & LIB_002) CloseLibrary(Lib002Base);
   if (LMask & LIB_003) CloseLibrary(Lib003Base);
   if (LMask & LIB_004) CloseLibrary(Lib004Base);
   if (LMask & LIB_005) CloseLibrary(Lib005Base);
   if (LMask & LIB_006) CloseLibrary(Lib006Base);
   if (LMask & LIB_007) CloseLibrary(Lib007Base);
 }
```

1c

```
/*************************************************************************\
 *              *** * Programma principale:  main() * ***                *
\*************************************************************************/

 VOID main()
 {

  /*
   *  Apri le librerie. Se qualcosa va male chiudi in modo pulito ed esci.
   */
   if ((Lib001Base = (struct Lib001Base*)
         OpenLibrary("lib001.library",0L)) == NULL)
   {
      printf("Non posso aprire la libreria numero 001\n");
      CleanClose();
      Exit(0L);
   }
   LMask |= LIB_001;
   .
   .
   .
   if ((Lib007Base = (struct Lib007Base*)
         OpenLibrary("lib007.library",0L)) == NULL)
   {
      printf("Non posso aprire la libreria numero 007\n");
      CleanClose();
      Exit(0L);
   }
   LMask |= LIB_007;

  /*
   *  Corpo del programma.
   */
   .
   .
   .

  /*
   *  Chiudi le librerie prima di uscire definitivamente
   */
   CleanClose();
 }
```

1b

Figura 1 - Soluzione all'esercizio della prima puntata.

comunque con l'opportuno codice di inizializzazione [*startup*].

Questo è necessario perché AmigaDOS non fornisce al programma il nome del programma stesso e gli eventuali parametri specificati secondo quanto stabilito dallo «Standard ISO» del C.

Nel caso del Lattice C, (versione «classica» Amiga 'C') ad esempio, sono forniti due file di inizializzazione:

AStartup.obj
che va usato solo se non vengono utilizzate le funzioni di I/O del Lattice C, e quindi le librerie di compilazione vanno agganciate [*link*] nel seguente ordine: AMIGA.LIB + LC.LIB;

LStartup.obj
che va usato se vengono utilizzate anche le funzioni di I/O del Lattice C, e quindi le librerie di compilazione vanno agganciate [*link*] nel seguente ordine: LC.LIB + AMIGA.LIB;

A questo punto, se il programma non prevede parametri in ingresso, il codice del vostro programma sarà più o meno il seguente:

```
/* Scheletro nel caso non siano passati parametri */

main()
{
 /*
  *   Corpo del programma principale
  */
}
```

altrimenti somiglierà al seguente:

```
/* Scheletro nel caso siano passati dei parametri */

main(argc,argv)
int argc;
char *argv[];
{
 /*
  *   Corpo del programma principale
  */
}
```

dove:

argc è zero se il programma è stato chiamato da WorkBench, uno se non sono stati passati parametri al programma, oppure corrisponde al numero di parametri passati più uno;

argv è un vettore [*array*] di puntatori a stringhe di caratteri, corrispondenti ad i vari parametri passati. Se **argc** è maggiore di zero, allora **argv [0]** punta al nome del programma stesso.

È buona regola quindi, se il programma è stato chiamato da CLI e si vuole fare riferimento al nome del programma, usare quello fornito da **argv [0]** e non scriverlo direttamente nel codice, in modo da poter sempre rinominare il programma senza cambiare il codice. Perciò evitate assolutamente la tecnica (2) mostrata in figura 2, ma utilizzate invece la (1), sempre nella stessa figura.

Differentemente da quanto detto sopra, programmi scritti per essere compilati dal Lattice C 4.0 si possono avvalere delle nuove routine di inizializzazione che forniscono, in caso **argc** sia nulla, un puntatore ad una struttura di tipo **WBStartup** purché venga incluso nel file la seguente riga:

```
#include "workbench/startup.h"
```

Per ulteriori informazioni, fare riferimento al *Lattice (R) AmigaDOS C Compiler - Programmer's Reference Manual* che viene fornito insieme al compilatore.

Come si scrive e si legge da CLI

Per mandare un messaggio sullo stesso CLI dal quale il programma è stato lanciato non è necessario fare niente di particolare. Praticamente il codice è lo stesso di quello che si utilizzerebbe in un sistema mono-tasking quale un compatibile IBM:

```
/* Il buon vecchio "Hello world" */
#include "stdio.h"

main()
{
  printf("Hello world\n");
}
```

Naturalmente lo stesso discorso vale per **write (stdout,...), putchar()** e via dicendo. Si assume che il lettore abbia familiarità con tali funzioni tipiche del C.

Analogamente, per leggere da CLI possono essere usate funzioni «classiche» come **scanf()** oppure **read(stdin,...)**.

Due parole devono essere invece spese per quanto riguarda **getchar()**: AmigaDOS non supporta operazioni di lettura di singoli caratteri dalla *console device* standard, in quanto la lettura da CLI non viene considerata terminata dal sistema operativo fin quando non viene premuto **ENTER.** Questo vale tanto per la finestra classica del CLI (a cui viene associato all'inizio sia lo standard input [*stdin*], sia lo standard output [*stdout*]), sia per qualunque altra finestra aperta come CON:.

Per potere compiere tali operazioni, sarà dunque necessario aprire tali finestre come RAW:. Come si effettuano operazioni di I/O da finestre RAW:, esula tuttavia dagli scopi di questa puntata. Ritorneremo su questo più avanti.

```
/* (1) fate così:                          (2) non così:                */

main(ac,av)                               main(ac,av)
int ac;                                   int ac;
char **av;                                char **av;
{                                         {
  if (ac>1) pgmname = av[0];
  if (ac == 1 && av[1][0] == '?')           if (ac == 1 && av[1][0] == '?')
  {                                         {
    printf("%s:\n",pgmname);                  printf("PROVA:\n");
    printf("   Programma di prova \n");       printf("   Programma di prova \n");
    exit(0);                                  exit(0);
  }                                         }
  .                                         .
  .                                         .
  .                                         .
}                                         }
```

Figura 2 - Riferimento al nome del programma nel codice.

I/O su file

Vediamo ora come si compiono operazioni di lettura e scrittura da e su file. Vedremo che tale tecnica può essere utilizzata anche per scrivere e leggere da finestre CLI differenti da quella da cui è stato lanciato il programma. In questo modo si può, ad esempio, fornire informazioni relative allo stato di esecuzione di un programma ([*log*] o [*trace*]) senza occupare spazio sulla finestra iniziale.

È necessario a questo punto introdurre un nuovo concetto, quello di File Handle.

Un File Handle è una sorta di puntatore al file, un modo per agganciarlo in modo da potervi fare riferimento ogni qualvolta vi si voglia compiere sopra una operazione di I/O. Tale aggancio viene fornito da AmigaDOS all'apertura del file e non va assolutamente confuso con il tipo FILE definito dal Lattice C come puntatore ad un file ed utilizzato dalle funzioni di I/O di secondo livello di tale compilatore. Certamente l'utilizzo del tipo FILE dà ad un programma maggiori garanzie di portabilità. D'altra parte, pur essendo il codice basato sulle funzioni AmigaDOS che utilizzano i File Handle fortemente orientato alla serie Amiga, e quindi poco portabile, si hanno notevoli vantaggi in fase di esecuzione, essendo tale codice più efficiente del precedente.

Le funzioni di I/O dell'AmigaDOS che tratteremo e che usano i File Handle sono le seguenti:

Open() Apre un file
Close() Chiude un file
Read() Legge da un file
Write() Scrive su di un file
Seek() Si posiziona in un punto del file

A queste vanno aggiunte le seguenti funzioni, sempre dell'AmigaDOS, di vario utilizzo:

Input() Ritorna il valore iniziale assunto da **stdin**
Output() Ritorna il valore iniziale assunto da **stdout**
IsInteractive() Verifica se un file è interattivo
WaitForChar() Attende per un certo tempo un carattere da un file interattivo

Prima di entrare nel dettaglio, apriamo una breve parentesi relativa agli operatori di ridirezionamento dell'AmigaDOS.

Quando lanciate un programma da CLI, la sintassi più generale è la seguente:

Nome Del Programma [< Input] [> Output] [opzioni] [parametri]

dove le parentesi quadre indicano che l'elemento incluso è opzionale, cioè non è obbligatorio.

Nome Del Programma
è appunto il nome del programma da lanciare

Opzioni e parametri
vengono passati al programma nel vettore **argv** già visto. Di fatto AmigaDOS non fa distinzione tra i primi ed i secondi, tuttavia è opportuno pensare a tali elementi sempre come oggetti separati, con scopi ed utilizzo separati.

< Input
il simbolo < è l'operatore di ridirezione di Ingresso) definisce cioè da dove AmigaDOS deve prendere i dati in Ingresso.

> Output
il simbolo > è l'operatore di ridirezione di Uscita, definisce cioè dove AmigaDOS deve ridirigere i dati in Uscita.

Ad esempio, si può utilizzare il comando **echo** per creare un file contenente il testo che altrimenti avrebbe stampato a terminale nel modo seguente:

echo > pippo.out "Questo è il testo da reindirizzare"!

Quando vengono utilizzati tali operatori, AmigaDOS ridefinisce i puntatori **stdin** ed **stdout** come File Handle ai file specificati da CLI. Quindi, nell'esempio precedente, **stdin** fa ancora riferimento alla finestra CLI da dove è stato lanciato il comando **echo**, mentre **stdout** è in realtà il File Handle del file **pippo.out.**

Questa tecnica si rivela particolarmente utile per reindirizzare verso un file i messaggi di errore prodotti dal compilatore C, qualora questi siano parecchi (come capita spesso la prima volta che si compila un nuovo programma!). In questo modo li si può andare a leggere con calma con uno dei tanti programmi scritti a questo scopo [*browser*].

Torniamo ora alle nostre funzioni di I/O. Innazitutto qualunque programma voglia utilizzarle, deve contenere i seguenti #include:

```
#include "exec/types.h"
#include "libraries/dosextens.h"
#include "stdio.h"
```

Il primo, come già detto nella scorsa puntata, serve a definire alcuni tipi e costanti C referenziate dagli altri #include; il secondo definisce le strutture utilizzate da AmigaDOS quali appunto il File Handle; il terzo si fa carico della definizione dei tipi e delle costanti usate nelle operazioni standard di I/O.

In figura 3 vediamo un esempio di lettura da file.

Commentiamola:

1. Innanzi tutto verifichiamo che almeno un parametro sia stato passato al programma. Se così è, assumiamo che sia il nome del file da leggere.
2. Quindi cerchiamo di aprire il file. Se il File Handle è nullo, qualcosa deve essere andato storto, stampiamo a terminale un messaggio di errore ed usciamo.

Come si può vedere nella figura, nel messaggio di errore si richiama la funzione **IoErr()**. Tale funzione ritorna un codice di errore AmigaDOS nel caso una funzione di I/O sia andata male. Sarebbe opportuno chiamarla dopo ogni operazione di I/O risoltasi in un insuccesso. In figura 4 sono riportati alcuni dei codici di errore di AmigaDOS.

Inoltre, nell'uscire, si fa uso di una costante predefinita in

```
/*
 *  Esempio di I/O su un file
 */
#include "exec/types.h"
#include "libraries/dosextens.h"
#include "stdio.h"

extern struct FileHandle *Open(); /* Open deve ritornare il puntatore */
                                  /* ad un File Handle                */
main(argc,argv)
int argc;
char **argv;
{
  struct FileHandle *infh;        /* INput File Handle */
  char buffer[BUFSIZ];            /* buffer di ingresso */

  if (argc<2)  /* non è stato passato il nome del file */
  {
    printf("Devi supplire il nome del file da leggere!\n");
    Exit(RETURN_ERROR);
  }

  /* altrimenti assumiamo che argv[1] punti al nome del file */

  infh = Open(argv[1],MODE_OLDFILE);
  if (infh==0) /* Qualcosa è andato storto */
  {
    printf("Errore nell'aprire il file: %d\n",IoErr());
    Exit(RETURN_ERROR);
  }

  /* Tutto OK. Leggiamo fino alla fine (EOF) */

  while (Read(infh,buffer,BUFSIZ)!=EOF)
  {
    /* Fai quello che vuoi con i dati letti */
  }

  /* Bene, chiudi il file */
  Close(infh);
}
```

Figura 3 - Apertura, lettura e chiusura di un file.

```
121   File is not an object module
202   Object in use
203   Object already exist
204   Directory not found
205   Object not found
218   Device not mounted
219   Seek error
221   Disk full
222   File is protected from deletion
223   File is protected from writing
224   File is protected from reading
225   Not a DOS disk
226   No disk in drive
```

Figura 4
Alcuni codici di errore ritornati da IoErr().

libraries/dos.h, incluso automaticamente da **libraries/dosextens.h** se il programmatore non l'ha già fatto esplicitamente. È buona pratica utilizzare tali costanti il più possibile nei propri programmi, sia per ragioni di mantenibilità che di portabilità fra differenti versioni dell'AmigaDOS:

```
RETURN_OK      Tutto a posto.
RETURN_WARN    Fatto, ma tieni presente che...
RETURN_ERROR   Spiacente, qualcosa è andato storto.
RETURN_FAIL    Aiuto! Sono nei guai!
```

Analogamente, nell'aprire il file, oltre al nome del file stesso è necessario specificare il modo di apertura, e per farlo abbiamo usato anche qui una costante predefinita, in accordo alla seguente tabella:

```
MODE_OLDFILE     Apri un file già esistente e posizionati
                 all'inizio
MODE_NEWFILE     Apri un file appena creato, eventualmente
                 dopo aver cancellato quello vecchio se già
                 esisteva
MODE_READWRITE   [DOS 1.2] Apri un file già esistente in modo
                 esclusivo
MODE_READONLY    [DOS 1.2] Sinonimo per MODE_OLDFILE
```

In genere sconsigliamo di usare **MODE_OLDFILE** per scrivere su di un file se avete la possibilità di usare **MODE_READWRITE** (solo AmigaDOS 1.2), ad esclusione dei file di tipo CON: e RAW:. Quest'ultimo modo infatti, impedisce ad altri programmi di accedere al file evitando sovrapposizioni che possono anche danneggiare il file. In ogni caso, vedremo nella prossima puntata come si blocca [*lock*] un file in modo da assicurarsene comunque l'accesso in esclusiva.

3. A questo punto iniziamo a leggere il file. Anche in questo caso facciamo uso di una costante predefinita in **stdio.h** chiamata End Of File (EOF). La funzione **Read** infatti, richiede in ingresso tre parametri:

a. il File Handle per agganciare il file
b. un'area di lavoro [*buffer*] per memorizzare i dati letti
c. la lunghezza del buffer

e ritorna in uscita il numero di caratteri effettivamente letti. Questo valore in genere corrisponde alla dimensione del buffer salvo in due casi:

a. si è raggiunta la fine del file (EOF)
b. si è verificato un errore di I/O.

In quest'ultimo caso sarebbe opportuno chiamare **IoErr()** per ricavarne il codice di errore relativo. In ogni caso è bene interrompere il programma, chiudere il file (non dimenticatevelo aperto!) ed uscire con un codice **RETURN_ERROR** o **RETURN_FAIL** a seconda dei casi.

4. Ed eccoci finalmente alla fine. Prima di uscire chiudiamo il file utilizzando la **Close()** e passandole il File Handle associato al file stesso.

La **Write()** si utilizza in modo analogo, essendo la sintassi praticamente la stessa:

```
int WrittenChars;
struct FileHandle *outfh;
char buffer[BUFSIZ];

WrittenChars = Write(outfh,buffer,BUFSIZ);
```

Questa volta tuttavia, se la funzione ritorna EOF (cioè -1), allora vuol dire che si è verificato un errore. Anche in questo caso ci può venire in soccorso il codice di errore fornito da **IoErr()**.

Ma quando apro un file, quale è il primo byte che leggo? Ovviamente il primo! E se volessi il ventesimo, oppure il centesimo? E se volessi aprire un file già esistente in scrittura, senza riscrivergli sopra, ma aggiungendo altri byte alla fine [*append*]?

Bene, in tal caso posso usare la **Seek()**. Questa funzione mi permette infatti di posizionarmi in un punto qualunque del file. Essa richiede in ingresso il solito File Handle e la posizione relativa dalla quale iniziare le operazioni di I/O. Tale posizione è misurata in byte ed è individuata da due parametri:

1. di quanti byte mi devo spostare da un certo punto per posizionarmi come richiesto, e
2. da quale punto incominciare a contare tali byte.

Quest'ultimo parametro può venire indicato tramite tre costanti predefinite in **libraries/dos.h**:

```
OFFSET_BEGINNING    a partire dall'inizio del file
OFFSET_CURRENT      a partire dalla posizione corrente
OFFSET_END          a partire dalla fine del file
```

In figura 5 sono riportati alcuni esempi di posizionamento.

```
/*
 *   Esempi di posizionamento: Seek() ritorna in MovedTo la posizione
 *   nel file dopo aver effettuato il posizionamento richiesto.
 *   Notare che il secondo parametro è:
 *     0 o positivo se OFFSET_BEGINNING è stato specificato
 *     0 o negativo se OFFSET_END è stato specificato
 *     0, negativo o positivo se OFFSET_CURRENT è stato specificato
 */
struct FileHandle *pfh;
int MovedTo;

/* Posizionati alla fine del file */
 MoveTo = Seek(pfh,0,OFFSET_END);

/* Posizionati a 100 bytes dall'inizio del file */
 MoveTo = Seek(pfh,100,OFFSET_BEGINNING);

/* Resta dove sei e ritorna la posizione corrente nel file */
 MoveTo = Seek(pfh,0,OFFSET_CURRENT);
```

Figura 5 - Esempi di utilizzo della Seek().

Come promesso, vediamo ora come aprire una nuova finestra CLI per operazioni di I/O. Semplice, invece di fornire alla **Open()** il nome di un file, gli si fornisce la definizione della finestra, con la stessa sintassi usata con il comando **NewCLI**:

```
newCLIfh = Open("CON:10/10/300/100/Nuovo_CLI",MODE_NEWFILE);
```

Analogamente possiamo sempre fare riferimento allo stesso CLI dal quale abbiamo lanciato il programma (e che quindi già esiste), così:

```
oldCLIfh = Open("*",MODE_OLDFILE);
```

Le restanti funzioni

Terminiamo questa seconda puntata con una veloce carrellata sulle restanti funzioni di I/O dell'AmigaDOS già menzionate. Useremo le seguenti definizioni:

```
struct FileHandle *fh;
BOOL status;
int timeout;
```

Le restanti quattro funzioni sono:

```
/*- 1 ---------------------------------------- INPUT -*/
/* Per ottenere l'aggancio iniziale in ingresso (stdin) */
fh = Input();

/*- 2 --------------------------------------- OUTPUT -*/
/* Per ottenere l'aggancio iniziale in uscita (stdout) */
fh = Output();

/*- 3 ------------------------------------ ISINTERACTIVE -*/
/* Per verificare se un file è collegato ad un terminale   */
/* virtuale. Un esempio di tali files è appunto:           */
/*         "CON:10/10/300/100/Terminale"                   */
status = IsInteractive(fh);

/*- 4 ------------------------------------------ WAITFORCHAR -*/
/* Per verificare se uno o più caratteri sono disponibili      */
/* entro un certo tempo da un file interattivo, cioè collegato */
/* ad un terminale virtuale                                    */
status = WaitForChar(fh,timeout);
```

A queste aggiungiamo un'altra utile funzione dell'AmigaDOS, spesso usata nelle operazioni di I/O su terminali virtuali (CLI, ad esempio):

```
/*- 5 ---------------------------------------- DELAY -*/
/* Attendi un certo numero di CINQUANTESIMI DI SECONDO */
Delay(timeout)

/* Il valore della frazione di secondi usata da questa ed altre */
/* funzioni di I/O è definito in libraries/dos.h come:          */
/* TICKS_PER_SECOND (attualmente appunto 50)                    */
```

L'esercizio

L'esercizio per la prossima volta è semplice. Scrivete un programma che scrive dieci numeri casuali [*random*] utilizzando la funzione **rand()** e li visualizza a terminale in una finestrella CLI in alto a destra rispetto a quella da cui viene chiamato.

Nella prossima puntata parleremo di bloccaggio e sbloccaggio di file e directory e vedremo come muoverci lungo la struttura ad albero [*directory tree*] dell'AmigaDOS.

Buon divertimento!

MK