Programmare in C su Amiga

di Dario de Judicibus

Con questa terza puntata termina la nostra breve carrellata sui misteri dell'AmigaDos, dal punto di vista del programmatore 'C'. In particolare analizzeremo la struttura gerarchica dei file e mostreremo come effettuare direttamente da programma molte operazioni che vengono di solito effettuate interattivamente da CLI

Vediamo in breve una possibile soluzione all'esercizio proposto nella puntata precedente prima di passare all'argomento di questo mese.

Come ricorderete, l'esercizio consisteva nello scrivere un programmino in grado di aprire una piccola finestra CLI nella quale visualizzare dieci numeri casuali [*random*], tenerla così per qualche secondo e poi chiuderla automaticamente restituendo il controllo alla finestra CLI da cui era stato chiamato.

La soluzione proposta, come d'altro canto la maggior parte dei programmi che presenterò in questa e nelle prossime puntate, è stata scritta in Lattice C 4.0, ma si può comunque facilmente adattare a qualsiasi compilatore C sul mercato.

La figura 1 è autoesplicativa, ma è bene comunque cogliere l'occasione per sottolineare un punto importante e che spesso provoca un po' di confusione a chi si avvicina a questo potente, ma decisamente complesso linguaggio che è il C.

Come si può vedere in figura, sono stati usati due tipi di aree di memoria [*buffer*] nel programma. Una del tipo **char *Buffer** ed un'altra del tipo **char *Buffer [SIZE]**.

Anche se in molti casi queste aree possono essere usate in modo equivalente, esse vengono trattate in modo diverso dal compilatore e soprattutto da molte funzioni interne [*built-in functions*] quali appunto le funzioni di conversione da intero a stringa.

Nel primo caso (**char *Buffer**) definiamo il puntatore ad una stringa di caratteri, senza specificarne la lunghezza. Quando il programma viene compilato, il compilatore riserva un'area ad un puntatore. In fase di esecuzione, nel momento in cui viene assegnata una stringa al puntatore in questione (ad esempio **Buffer = «Oggi è Sabato»;**), il programma carica in quell'area il puntatore alla stringa che era stata memorizzata in compilazione da un'altra parte. In teoria, l'unico limite che si ha alla lunghezza della stringa è la memoria disponibile.

Nel secondo caso, invece, viene prenotata un'area di dimensioni predefinite (**SIZE**) fin dalla compilazione, ma non si può più assegnare a **Buffer** una stringa, dato che questo non è altro che il puntatore al primo elemento dell'area fissata, cioè **&Buffer[0]** e quindi non può essere cambiato, non è cioè quello che in C si chiama un lvalue. In questo caso si dovrà usare la funzione interna **strcpy.** Inoltre dovremo stare bene attenti a non superare i limiti prefissati, per evitare di andare a modificare zone di memoria che contengono altri dati o addirittura istruzioni in linguaggio macchina.

In genere si usano puntatori quando non si sa a priori quanto spazio è necessario, mentre si preferiscono vettori di caratteri [*array*] quando è importante prenotare uno spazio di una certa lunghezza.

Nel nostro caso, mentre **buf** viene usata nella chiamata alla **Write** (vedi nota 1), perché non si può stabilire quanto potrà essere lunga la stringa da stampare, **use** deve essere lunga almeno 13 byte come richiesto dal manuale del Lattice C per la funzione **stcl_d**.

Introduzione

Veniamo ora alla seconda parte di questa serie dedicata all'AmigaDOS.

Questa puntata si propone di:

1. far vedere come è possibile chiamare da un programma, non solo le funzioni dell'AmigaDOS già viste, ma di fatto qualsiasi programma a condizione di... beh, questo lo vedremo tra poco;
2. introdurre il lettore alla struttura gerarchica nella quale sono organizzati i file e spiegare come acquisire informazioni sui singoli elementi di tale struttura (directory e file);
3. mostrare come:

- creare una directory,
- cambiare nome ad un file,
- cancellare un file,
- ottenere informazioni relative ad un dischetto.

In realtà ci sarebbero molte altre cose da dire sull'AmigaDOS.

```
/*                                                          *\
 *  PRTRAND   -   Soluzione all'esercizio della seconda puntata   *
 *                                                                 *
 *                Apre una nuova finestra CLI e vi stampa dentro   *
 *                dieci numeri random. Attende 4 secondi e poi     *
 *                chiude la finestra.                              *
 *                                                                 *
 *                Scritto e compilato con il Lattice C 4.0         *
\*                                                                 */

#include "exec/types.h"
#include "proto/dos.h"
#include "libraries/dosextens.h"
#include "stdio.h"
#include "string.h"

VOID main()
{
  struct FileHandle *fh;
  WORD i,l;
  char use[13], *line, *buf="";

  /*
   *  Prova ad aprire una nuova finestra CLI. Se ci sono problemi esci.
   */
  fh = Open("CON:300/50/300/150/Random",MODE_NEWFILE);
  if (fh == NULL)
  {
    printf("Error during Open(): %d\n",IoErr());
    Exit(RETURN_ERROR);
  }

  /*
   *  Stampa il titolo e fai partire il loop.
   */
  line = strcpy(buf,"Random Numbers generation:\n\n");

  for (i=1;i<12;++i)
  {
    l = Write(fh,buf,strlen(buf));
    line =  stpcpy(buf,"Number #");
    l = stcl_d(use,i);
    line = stpcpy(stpcpy(line,use),": ");
    l = stcl_d(use,(LONG)rand());
    line = stpcpy(stpcpy(line,use),"\n");
  }

  /*
   *  Attendi 4 secondi e poi chiudi tutto.
   */
  Delay(200);
  Close(fh);
}
```

Figura 1
Soluzione all'esercizio della seconda puntata.

```
/*                                                          *\
 *  DATA      -   Aggiorna la data di sistema prendendola dal  *
 *                file "Oggi".                                 *
\*                                                          */

#include "exec/types.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

VOID main()
{
  struct FileHandle *infh;
  BOOL Success;

  infh = Open("S:Oggi",MODE_OLDFILE);
  if (infh == NULL) Exit(RETURN_FAIL);
  Success = Execute("Date ?",infh,0);
  if (!Success) Exit(RETURN_FAIL);
  Close(infh);
}
```

Figura 2
Esempio di utilizzo di Execute() con il File Handle di ingresso non nullo.

Note

1. Attenzione a non confondere le funzioni interne del Lattice C **read e write** da quelle di AmigaDOS **Write** e **Read** . In C il fatto che una lettera sia minuscola piuttosto che maiuscola è importante!
2. Non è certo il caso di **Date** ovviamente.
3. Per evitare problemi è bene identificare un elemento sempre attraverso il formato completo del nome, specificando l'intero cammino dal dispositivo fisico, fino all'elemento stesso (ad esempio **df1:codice/C/programma**).
4. La funzione **CleanExit()** è stata introdotta solo per chiarezza. Si tratta chiaramente di una funzione utente tipo quella già vista nella seconda puntata e che serviva a chiudere le librerie.

Innanzitutto ci sono ancora alcune funzioni secondarie come ad esempio **SetComment()** che permette di associare un commento ad un file. Una lista completa può essere trovata in *The Amiga DOS Manual* che consigliamo a chiunque voglia approfondire l'argomento. Inoltre molte funzioni fornite da AmigaDOS possono essere ottenute solo tramite una tecnica speciale, detta della comunicazione a pacchetti. Questi pacchetti [*packet*] sono strutture costruite sulla base del meccanismo a messaggi [*message*] fornito da EXEC, di cui parleremo nella prossima puntata. Dato che si tratta di una tecnica avanzata, che richiede una buona conoscenza di EXEC, rimandiamo il discorso ad un altro momento. Allo stesso modo tralasceremo, almeno per il momento, quelle strutture di AmigaDOS che vengono utilizzate da alcuni comandi molto usati, come **assign**, e per i quali non c'è una funzione equivalente tra i servizi del sistema operativo. Vedremo che si può comunque risolvere il problema in un altro modo, grazie alla funzione **Execute**.

Execute

Supponiamo di aver bisogno di assegnare un nome logico ad una certa directory. Se ci trovassimo in una finestra CLI scriveremmo semplicemente

1>assign PIPPO: df1:fumetto/pippo

Come si può fare la stessa cosa dall'interno di un programma scritto in C?

Il metodo più semplice (ne esiste un altro, ma richiede una conoscenza un

po' più approfondita dell'AmigaDOS) consiste nell'utilizzare la funzione **Execute()**. Tale funzione, da non confondersi con il comando **execute** che si trova nella directory «C:», riceve in ingresso una stringa di caratteri e la esegue come se l'avessimo scritta da tastiera in una finestra CLI, senza tuttavia creare tale finestra.

Al contrario quindi del comando **execute**, il cui scopo è di eseguire una serie di istruzioni caricate all'interno di un file di AmigaDOS detto CLI script o macro CLI, la funzione **Execute()** permette di eseguire un solo comando in modo analogo a quanto fa il comando **run**.

Putroppo tale funzione ha due restrizioni:
1. il comando **run** deve essere presente nella directory «C:»;
2. il comando da eseguire deve essere o nella directory «C:», oppure in quella corrente al momento dell'esecuzione.

La sintassi di **Execute** è la seguente:

```
BOOL fatto;
char *comando;
struct FileHandle *ingresso, *uscita;
fatto = Execute (comando, ingresso, uscita);
```

dove:
comando è una stringa che contiene il comando da eseguire, gli argomenti da passare a tale comando ed eventuali operatori di reindirizzamento («<» ed «>»), scritta cioè né più né meno come la scriveremmo in CLI;
ingresso specifica come reindirizzare l'ingresso standard del comando se non è stato specificato nessun operatore di reindirizzamento dell'ingresso nella stringa fornita come primo parametro;
uscita specifica come reindirizzare l'uscita standard del comando se non è stato specificato nessun operatore di reindirizzamento dell'uscita nella stringa fornita come primo parametro;
fatto se falso (FALSE) vuol dire che non è stato possibile eseguire correttamente il comando.

Due parole sui FileHandle di ingresso e di uscita.

Supponiamo che il FileHandle di ingresso sia nullo. In questo caso la funzione cerca di eseguire il comando descritto nella stringa fornita come primo parametro per poi ripassare il controllo al programma chiamante. Se invece il FileHandle di ingresso **non** è nullo, **Execute()** esegue la stringa fornita (la quale in tal caso può anche essere nulla, come vedremo in seguito) e quindi legge i dati in ingresso dal FileHandle specificato fino alla fine del file relativo. Ad esempio, se il file **Oggi** contiene la data odierna, il programmino in figura 2 mostra come utilizzare il FileHandle di ingresso per far eseguire l'aggiornamento della data di sistema. Ovviamente la stessa cosa potrebbe essere fatta semplicemente scrivendo:

```
fatto = Execute("Date <S:Oggi ?",0,0);
```

Tuttavia questa seconda soluzione, pur essendo più rapida e semplice da ricordare, talvolta può creare dei problemi (vedi nota 2) dato che si richiede di eseguire il comando senza prima fare una verifica di esistenza sul file in ingresso.

Un possibile utilizzo del FileHandle di ingresso è il seguente:

▼

```
struct FileHandle *dosfh;
BOOL Success;

dosfh = Open("CON:10/10/600/250/Prova",MODE_NEWFILE);
if (dosfh == NULL) Exit(RETURN_FAIL);
Success = Execute("",dosfh,0);
if (!Success) Exit(RETURN_FAIL);

:
```

Questa tecnica permette di aprire una finestra CLI in modo analogo a quanto avviene utilizzando il comando **NewCLI**. Tale finestra si comporta come un normale finestra CLI, e quindi può essere utilizzata sia per immettere nuovi comandi in modo interattivo, sia per ricevere dati in uscita dagli stessi. Naturalmente, per poter chiudere questa finestra e far continuare il programma, è necessario usare il comando **EndCLI**.

Il FileHandle di uscita, al contrario di quello di ingresso, viene utilizzato abbastanza spesso, dato che molti comandi danno come risultato una serie di dati in uscita (ad esempio **dir** oppure **info**). Se questi fosse nullo e, allo stesso tempo, nessun operatore di reindirizzamento in uscita fosse stato incluso nel comando da eseguire, **Execute()** utilizzerebbe come uscita quella standard, quella cioè della finestra CLI dalla quale è stato lanciato il programma. Niente di male, direte voi... ed in linea di massima cioè è vero. Tuttavia, se il programma fosse stato lanciato da WorkBench, **Execute()** non saprebbe dove reindirizzare il risultato.

Fate quindi molta attenzione quando mettete a zero il FileHandle di uscita.

Vediamo adesso quando in genere può essere utile usare **Execute()**:
1. quando si vuole eseguire un comando presente in «C:» il quale non abbia una corrispondente funzione AmigaDOS (ad esempio **assign**);
2. quando si vuole eseguire un comando per il quale esiste una funzione AmigaDOS corrispondente, ma si vogliono specificare nomi parziali per i file utilizzando i simboli «#» e «?» (ad esempio **copy df0:pippo.#? TO df1:fumetti**);
3. quando si vuole eseguire un proprio programma presente nella directory corrente;
4. altri usi particolari come quello già descritto per aprire una nuova finestra CLI interattiva.

Ovviamente resta il fatto che le performance sono inferiori rispetto a quelle che si possono ottenere utilizzando direttamente una funzione dell'AmigaDOS, che esistono le restrizioni sopracitate relative a **run** e che il comando da eseguire non può essere ovunque.

La struttura gerarchica dei file

Chiunque abbia lavorato con il CLI, sa che i file in AmigaDOS sono organizzati secondo una struttura gerarchica. La radice [*root directory*] di tale struttura è rappresenata dal dispositivo fisico per la memoria di massa [*physical device*] (dischetto da 3"1/2, floppy disk da 5"1/4, harddisk, CD-ROM, RAM, e via dicendo). Questa può contenere sia file che altre directory, chiamate subdirectory. Ogni directory ha un nome. Un file viene identificato in modo univoco per mezzo del cammino [*path*] che bisogna percorrere attraverso la struttura gerarchica esistente e del nome del file stesso. Tale cammino può essere specificato completamente nel seguente modo:

device:dir/subdir1/.../subdirN/file

oppure in forma abbreviata

subdir5/.../subdirN/file

In quest'ultimo caso si assume che la prima parte del cammino corrisponda a quella che si deve fare per arrivare alla

```
/*                                                          *\
 *  LOCKDEMO  -  Esempio di bloccaggio e sbloccaggio di un file.  *
\*                                                          */

#include "exec/types.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

extern struct FileLock *Lock();

VOID main()
{
  struct FileLock *lock;

  /*
   *  Prova a bloccare il file.
   */
  lock = Lock("df0:prove/lucchetto",SHARED_LOCK);
  if (lock == NULL)
  {
    printf("Non posso bloccare il file - codice: %ld\n",IoErr());
    Exit(RETURN_FAIL);
  }

:

  /*
   *  Sblocca il file.
   */
  UnLock(lock);
}
```

▲ *Figura 3 Bloccaggio e sbloccaggio di un file.*

Figura 4 ▶ *Sintassi di alcune funzioni ed esempi vari.*

```
/*
 *  Esempio A: Sintassi di CurrentDir() e ParentDir()
 */
struct FileLock *nuovo, *vecchio;
vecchio = CurrentDir(nuovo);

struct FileLock *attuale, *precedente;
precedente = ParentDir(attuale);

/*
 *  Esempio B: Due modi di spostarsi in una directory
 */
/* - 1 ------------------------------------------------------ */
BOOL fatto;                          /* Usando Execute()    */
fatto = Execute("cd df1:pippo",0,0);
/* - 2 ------------------------------------------------------ */
struct FileLock *nuova, *vecchia;    /* Usando CurrentDir() */
nuova = Lock("df1:pippo",ACCESS_READ);
vecchia = CurrentDir(nuova);

/*
 *  Esempio C: Sintassi di Examine() e ExNext()
 */
BOOL fatto;
struct FileLock *lucchetto;
struct FileInfoBlock *blocco;

fatto = Examine(lucchetto, blocco);
fatto = ExNext(lucchetto, blocco);

/*
 *  Esempio D: Come usare Examine()
 */
BOOL fatto;
char *elemento;
struct FileLock *lucchetto;
struct FileInfoBlock *blocco;
typedef struct FileInfoBlock FIBLK

blocco = (FIBLK *)AllocMem(sizeof(FIBLK), MEMF_CLEAR);
lucchetto = Lock(elemento, ACCESS_READ);
fatto = Examine(lucchetto, blocco);
if (fatto)
{

:
```

directory corrente [*current directory*] (nell'esempio **subdir4**).

AmigaDOS mette a disposizione del programmatore diverse funzioni per poter muoversi su e giù lungo la struttura ad albero in cui sono organizzati i file.

Prima di descriverle, tuttavia, è necessario introdurre un nuovo concetto: quello di lucchetto [*lock*].

Un lucchetto, come dice il termine, è un meccanismo attraverso il quale si chiede ad AmigaDOS il controllo temporaneo di un certo cammino, vuoi per modificarne alcuni attributi, vuoi semplicemente per impedire a qualcun altro di farlo mentre si stanno esaminando. Tale meccanismo è fondamentale in un sistema multitasking. Provate a pensare cosa potrebbe succedere se un processo potesse cancellare una directory vuota proprio mentre il vostro programma sta creando un nuovo file nella stessa directory.

Dire che si possiede il controllo di un cammino, vuol dire in sostanza possedere il controllo su di un elemento della struttura gerarchica nella quale sono organizzati i file in AmigaDOS. Tali elementi sono di due tipi: directory e file. Da questo momento ci riferiremo ad essi come elementi della struttura gerarchica, o più semplicemente elementi.

Vedremo che esiste la possibilità di definire due tipi di lucchetto: quello esclusivo, quando ci si vuole assicurare il controllo esclusivo del cammino specificato, e quello condiviso [*shared*]. In quest'ultimo caso più processi possono operare sullo stesso elemento.

Bloccaggio e sbloccaggio

Un elemento può essere bloccato per mezzo della funzione **Lock()**. Questa funzione accetta in ingresso due parametri, come mostrato in figura 3: il primo è una stringa che contiene il nome dell'elemento da bloccare (vedi nota 3), il secondo specifica il modo di accesso all'elemento.

Se si desidera ottenere il controllo esclusivo dell'elemento, allora si può usare indifferentemente **EXCLUSIVE_LOCK** oppure **ACCESS_WRITE**, dato che rappresentano lo stesso modo di accesso. Analogamente, si può specificare un accesso condiviso, usando **SHARED_LOCK** oppure **ACCESS_READ**.

Se tutto va bene, **Lock()** ritorna un puntatore ad una struttura specifica (**struct FileLock**) che serve ad AmigaDOS a contenere informazioni relative all'elemento bloccato.

Chi programma può ignorare completamente, se vuole, il contenuto di tale struttura, limitandosi ad usare il puntatore menzionato (il lucchetto, appunto!) come identificatore per qualunque successiva operazione sull'elemento bloccato.

Dato che il bloccaggio di un elemento è un'operazione che può in un qualche modo limitare l'operatività di altri processi, specialmente nel caso di un lucchetto esclusivo, è importantissimo sbloccare qualunque elemento si sia bloccato prima di terminare il programma. In caso contrario si può inficiare il corretto funzionamento di AmigaDOS. Di fatto, il nostro suggerimento è quello si sbloccare un elemento non appena non se ne abbia più bisogno, a meno che la logica del vostro programma non richieda di mantenerlo sotto controllo per evitare un'inconsistenza nelle operazioni successive. Un esempio classico consiste nel bloccare una directory vuota mentre si stanno preparando dei dati da scaricare in un file da creare appunto in tale directory.

Lo sbloccaggio di un elemento si effettua per mezzo della funzione di AmigaDOS **UnLock()**. Unico parametro in ingresso il lucchetto associato all'elemento da sbloccare.

Ricordatevi inoltre di controllare sempre se l'operazione di bloccaggio ha

```
/*
 *  Crea una directory
 */
struct FileLock *lucchettonuovadir;
char *nomenuovadir;

lucchettonuovadir = CreateDir(nomenuovadir);

/*
 *  Cambia il nome ad un file
 */
BOOL fatto;
char *vecchionome, *nuovonome;

fatto = Rename(vecchionome, nuovonome);

/*
 *  Cancella un file
 */
BOOL fatto;
char *nomefile;

fatto = Delete(nomefile);

/*
 *  Ottieni informazioni relativamente al disco su cui si trova
 *  una directory od un file precedentemente bloccato.
 *  Vedi Figura 6 per la definizione di InfoData.
 */
BOOL fatto;
struct FileLock *lucchetto;
struct InfoData *informazioni;

fatto = Info(lucchetto, informazioni);
```

Figura 5 - Altre funzioni AmigaDOS.

```
/*
 *  Informazioni su file e directory
 */
struct FileInfoBlock
{
  LONG fib_DiskKey;          /*                                  */
  LONG fib_DirEntryType;     /* se file < 0 - se directory > 0 */
  char fib_FileName[108];    /* Nome dell'elemento - 30 max     */
  LONG fib_Protection;       /* Protezione: RWXD -> bit 3210    */
  LONG fib_EntryType;        /*                                  */
  LONG fib_Size;             /* Dimensione del file in byte     */
  LONG fib_NumBlocks;        /* Dimensione del file in blocchi */
  struct DateStamp fib_Date; /* Data dell'ultima modifica       */
  char fib_Comment[116];     /* Eventuale commento: FileNote    */
}

/*
 *  Informazioni sui volumi (dischetti, disco fisso...)
 */
struct InfoData
{
  LONG  id_NumSoftErrors; /* Numero di errori "soft" sul disco */
  LONG  id_UnitNumber;    /* Unita su cui il disco e montato   */
  LONG  id_DiskState;     /* Stato del disco - vedi sotto      */
  LONG  id_NumBlocks;     /* Numero di blocchi nel disco       */
  LONG  id_NumBlocksUsed; /* Numero di blocchi usati           */
  LONG  id_BytesPerBlock; /* Numero di byte in un blocco       */
  LONG  id_DiskType;      /* Tipo del disco - vedi sotto       */
  BPTR  id_VolumeNode;    /* puntatore (CPL al nodo del volume */
  LONG  id_InUse;         /* 0 se il disco non e in uso        */
};

/*  Stato del disco:                                           */
/*                                                             */
/*  ID_WRITE_PROTECTED   Disco protetto da scrittura           */
/*  ID_VALIDATING        Disco sotto processo di verifica      */
/*  ID_VALIDATED         Disco pronto per lettura/scrittura    */
/*                                                             */
/*  Tipo del disco:                                            */
/*                                                             */
/*  ID_NO_DISK_PRESENT   Il disco non e montato                */
/*  ID_UNREADABLE_DISK   Non puo essere letto                  */
/*  ID_DOS_DISK          Disco formattato da AmigaDOS          */
/*  ID_NOT_REALLY_DOS    Disco parzialmente DOS                */
/*  ID_KICKSTART_DISK    Disco per la partenza del sistema     */
```

▲ *Figura 6 Strutture contenenti informazioni sui volumi, le directory ed i file.*

```
/*
 *   "dir" e una stringa che contiene la directory
 */
#define FIB struct FileInfoBlock

info = (FIB *)AllocMem(sizeof(FIB),MEMF_CLEAR);

/*
 *   Blocca la directory e controlla se e veramente tale
 *   (vedi nota liref page=no refid=clean)
 */
lucchetto = Lock(dir,ACCESS_READ);
if (!lucchetto) CleanExit(NOLOCK); /* Esci in modo pulito          */

fatto = Examine(lucchetto, info);  /* Riempi info per ExNext       */
if (!fatto) CleanExit(NOINFO);     /* problemi.. Esci in modo pulito */
if (info->fib_DirEntryType < 0)    /* e un file! Esci in modo pulito */
   CleanExit(ISFILE);
printf("Contenuto di %s\n\n",dir);

/* Ora usa ExNext per stamparne il contenuto                       */
/* Nota che il loop termina se ExNext fallisce. In realta bisognerebbe */
/* sempre chiamare IoErr() per verificare se si tratta di un errore  */
/* oppure se abbiamo finito la lista di elementi nella directory     */
while (fatto)
{
  fatto = ExNext(lucchetto,info);
  if (fatto)
  {
    if (info->fib_DirEntryType < 0) printf("File: ")
    if (info->fib_DirEntryType > 0) printf("Dir : ")
    printf(&(info->fib_Filename[0])); printf("\n");
  }
}
if (lucchetto) UnLock(lucchetto);
FreeMem(info,sizeof(FIB));
```

◀ *Figura 7 Analisi di una directory.*

avuto successo o meno. Per fare questo basta verificare se il lucchetto è diverso od uguale a zero. In quest'ultimo caso l'operazione di bloccaggio è fallita e sarebbe opportuno utilizzate la funzione **IoErr()** per analizzare il motivo dell'insuccesso, come nell'esempio di figura 3.

Vedremo più avanti, tuttavia, che in alcuni casi il valore nullo per un lucchetto ha un particolare significato.

Infine, se si passa alla **Lock()** una stringa nulla, la funzione restituirà al programma il lucchetto relativo alla cosiddetta directory corrente [*current directory*], a quella directory cioè nella quale si stava operando al momento della chiamata.

Su e giù lungo l'albero...

AmigaDOS ci mette a disposizione due funzioni che ci permettono di muoverci avanti ed indietro lungo la struttura ad albero dei file:

- **CurrentDir()**
- **ParentDir()**

La prima ci permette di muoverci da una directory ad un'altra, mentre la seconda serve ad ottenere il lucchetto relativo alla directory di livello immediatamente superiore, se esiste. Tale directory si chiama directory madre [*parent directory*]. Vediamole più in dettaglio, eventualmente facendo riferimento agli esempi in figura 4.

CurrentDir()

La sintassi corretta per questa funzione è quella mostrata nell'esempio A. Come si può vedere, l'unico parametro in ingresso è il lucchetto relativo alla directory nella quale si vuole entrare. La funzione restituisce viceversa il lucchetto relativo alla directory nella quale il programma «si trovava» al momento della chiamata. In questo modo è sempre possibile tornare indietro chiamando di nuovo **CurrentDir()** e passandogli il lucchetto precedente.
> > > Attenzione < < <: Non utilizzate mai la **UnLock()** per sbloccare un lucchetto che non sia stato ottenuto precedentemente per mezzo della **Lock()**, quali appunto quelli restituiti dalla **CurrentDir().** Tali lucchetti, infatti, sono di proprietà esclusiva dell'AmigaDOS. Cancellandoli si rischia di impedire in seguito al sistema operativo l'accesso alle directory corrispondenti.

Nell'esempio B vediamo due possibili modi di entrare in una directory specifica. Il primo usa la **Execute()**, ed ha quindi lo svantaggio di richiedere il comando **run** nella directory C:, il secondo utilizza la **CurrentDir()**, appunto.

Ricordate inoltre che, al contrario di quanto già visto per la funzione di bloccaggio, un eventuale lucchetto nullo ritornato dalla **CurrentDir()** è perfettamente valido: vuol dire semplicemente che quando è stata effettuata la chiamata, il programma si trovava nella directory principale del disco dal quale è stato effettuato il boot del sistema.

ParentDir()

Questa funzione è analoga alla precedente (vedi sempre l'esempio A), salvo che questa volta il parametro in ingresso rappresenta una directory di cui abbiamo già in qualche modo ottenuto il lucchetto mentre in uscita abbiamo quello relativo alla directory madre. Anche in questo caso un lucchetto nullo significa che abbiamo raggiunto la cima dell'albero, che rappresenta la gerarchia AmigaDOS per i file. Questa volta però l'albero è solo quello relativo al volume (dischetto o disco fisso) nel quale si trova la directory corrente.

Informazioni su file *e* directory

Altri due importanti funzioni sono (vedi figura 4):

- **Examine()**
- **ExNext()**

Queste funzioni ci permettono di andare ad analizzare tutta una serie di informazioni relative ad un file o ad una directory ed hanno la stessa sintassi (esempio C).

Tali informazioni vengono memorizzate in una struttura apposita chiamata **FileInfoBlock** (vedi figura 6).

Examine()

Questa funzione restituisce le informazioni relative ad uno specifico elemento.

Prima di poter chiamare **Examine()** è tuttavia necessario compiere due operazioni:
1. innanzitutto bisogna bloccare l'elemento di cui vogliamo ottenere le informazioni disponibili e memorizzare il lucchetto corrispondente;
2. poi è necessario allocare uno spazio in memoria, grande abbastanza da contenere la struttura menzionata, allineato ad una voce da quattro byte.

Quest'ultima operazione è necessaria perché AmigaDOS si aspetta di trovare questa particolare struttura allineata ad una voce.
Nel caso dell'Amiga, una voce corrisponde ad un blocco di 4 byte, cioè ad un **long int** o **LONG**.
Dato che la funzione **AllocMem()** (Exec) allinea la memoria richiesta ad una doppia voce, useremo proprio quella (vedi esempio D).

ExNext()

Questa funzione ci permette di analizzare, una dopo l'altra, le informazioni relative a tutti gli elementi contenuti in una stessa directory, siano essi file, siano esse altre directory.

La sintassi è la stessa della funzione precedente, solo che adesso il lucchetto è sempre quello della directory di cui si vuole analizzare il contenuto. Inoltre è necessario eseguire una **Examine()** sulla directory in questione in modo da riempire la prima volta il **FileInfoBlock** con dati relativi (vedi figura 7).

Tali informazioni saranno quindi passate alla **ExNext()** che le utilizzerà per cercare il primo elemento della directory.

A questo punto basterà chiamare più volte la **ExNext()** passandole sempre il lucchetto relativo alla directory madre e la struttura riempita dalla chiamata precedente.

Questo fintanto che non viene trovato l'elemento che interessa o la funzione ritorna un indicatore nullo.

È importante notare che questa funzione non va mai chiamata passandogli una struttura vuota, come invece spesso succede con **Examine()**.

Altre funzioni AmigaDOS

La figura 5 mostra altre quattro utili funzioni di AmigaDOS.

La prima serve a creare una directory e corrisponde al comando **makedir** nella directory C:. Accetta come parametro di ingresso una stringa contenente il nome della nuova directory da creare e restituisce il lucchetto alla stessa. Un valore nullo di questo sta ad indicare che qualcosa è andato storto.

La seconda serve a cambiare il nome ad un file. Corrisponde al comando **rename** e restituisce un indicatore che rappresenta l'eventuale successo o fallimento dell'operazione. Come si può vedere in figura, i parametri in ingresso corrispondono rispettivamente al vecchio ed al nuovo nome del file.

La terza serve a cancellare un file. Anche in questo caso viene restituito un indicatore [*flag*] per determinare se tutto è andato bene.

L'ultima funzione è, in un certo senso, analoga alle due funzioni precedentemente descritte **Examine()** ed **ExNext()**.

Stavolta, tuttavia, le informazioni restituite nella struttura **InfoData** non riguardano tanto l'elemento di cui si passa il lucchetto, quanto il volume (dischetto o disco fisso) che lo contiene.

Anche in questo caso è responsabilità del programmatore assicurarsi che la struttura sia allineata ad una voce (LONG) da quattro byte.

La definizione della struttura in questione è riportata in figura 6. Non preoccupatevi se per adesso c'è ancora qualche campo un po' misterioso. È necessario introdurre molti nuovi concetti prima di poter comprendere ogni aspetto di AmigaDOS. Ne vedremo qualcuno nella puntata dedicata ad EXEC.

L'esercizio

Per il momento avete abbastanza informazioni per affrontare l'esercizio di questa puntata. Si tratta di scrivere un programmino in grado di dirvi quanti byte avete ancora a disposizione su un qualunque dischetto. Facile, no?

Nella prossima puntata parleremo di EXEC e descriveremo il meccanismo a messaggi che utilizzeremo poi nelle puntate successive. Non ci soffermeremo comunque a lungo su questo complesso componente del sistema, dato che richiederebbe troppo tempo, ma incominceremo ad occuparci di grafica e di animazione, che rappresenta forse uno degli aspetti più spettacolari di Amiga.

Alla prossima puntata, dunque. MC