Programmare in C su Amiga

di Dario de Judicibus

Incominciamo con questa puntata a costruire uno scheletro di programma da utilizzare ogniqualvolta dobbiamo scrivere un codice che interfacci Intuition. Vedremo come esso può essere variato in funzione delle nostre necessità e quali vantaggi e limitazioni comporti il suo utilizzo.

Dato che nelle scorse puntate le due sottorubriche *Casella Postale* e *La Scheda Tecnica* si sono prese un po' dello spazio solitamente dedicato al tema vero e proprio di questi articoli, e cioè la programmazione in C su Amiga, ho pensato di dare questa volta più spazio ad Intuition, anche per mantenere una certa continuità nel discorso che altrimenti ne risulterebbe troppo spezzettato. In questa puntata imposteremo un primo abbozzo di scheletro di quello che diventerà il nostro programma di lavoro nelle prossime puntate. Esso ha due scopi: il primo è quello di mostrare come si utilizza Intuition per costruire una interfaccia a menu e, come vedremo in seguito, come si gestiscono altri oggetti quali quadri *[requester]*, aggeggi *[gadget]*, e via dicendo; il secondo è quello di fornire una base flessibile per tutti i programmi di questo tipo, capace di adattarsi a varie esigenze e comunque utile per evitare di riscrivere da capo l'intero programma ogniqualvolta vogliamo usare Intuition nel nostro codice. Alcune delle tecniche che presenterò sono elaborazioni di tecniche sviluppate da vari programmatori e rese disponibili dagli stessi a tutti gli utenti Amiga, altre sono tecniche personali che mi hanno permesso di standardizzare i miei programmi con indubbio vantaggio in termini di tempo e di manutenzione. La seconda parte dell'articolo mostrerà come usare la **grep.lib** per sfruttare la potenza di *GREP* dall'interno di un programma scritto in C.

Figura 1.
La struttura ***Menu***.

```
/* ========================================================================== */
/* === Menu ================================================================= */
/* ========================================================================== */
struct Menu
{
  struct Menu     *NextMenu;  /* il menù successivo nella lista oppure NULL */
  SHORT            LeftEdge,  /* la posizione dell'area di selezione:       */
                   TopEdge;   /* angolo in alto a sinistra (vedi articolo)  */
  SHORT            Width,     /* le dimensioni dell'area di selezione:      */
                   Height;    /* larghezza ed altezza (vedi articolo)       */
  USHORT           Flags;     /* stato del menù (vedi lista sotto)          */
  BYTE            *MenuName;  /* titolo del menù                            */
  struct MenuItem *FirstItem; /* puntatore alla catena delle voci del menù  */

/* Le seguenti variabili son riservate al solo utilizzo interno (Intuition) */
  SHORT JazzX, JazzY, BeatX, BeatY; /* *** RISERVATE *** NON USARE ******** */
};

/*
**  Segnalatori impostati da Intuition  Programma  Significato              */
*/
#define MENUENABLED 0x0001 /*    SI        SI      Menù [T] attivo - [F] no */
#define MIDRAWN     0x0100 /*    SI        NO      Voci visualizzate        */
```

Figura 2.
La struttura ***MenuItem***.

```
/* ========================================================================== */
/* === MenuItem ============================================================= */
/* ========================================================================== */
struct MenuItem
{
  struct MenuItem *NextItem;  /* la voce successiva nella lista oppure NULL */
  SHORT            LeftEdge,  /* la posizione dell'area di selezione:       */
                   TopEdge;   /* angolo in alto a sinistra (vedi articolo)  */
  SHORT            Width,     /* le dimensioni dell'area di selezione:      */
                   Height;    /* larghezza ed altezza (vedi articolo)       */
  USHORT           Flags;     /* segnalatori vari (vedi lista sotto)        */
  LONG             MutualExclude; /* maschera di esclusione (vedi articolo) */
  APTR             ItemFill;  /* immagine o testo principale: punta ad una  */
                              /* struttura Image, IntuiText od a NULL       */
  APTR SelectFill;            /* immagine o testo alternativo: punta ad una */
                              /* struttura Image, IntuiText od a NULL       */
  BYTE Command;               /* scorciatoia: carattere con cui selezionare */
                              /* la voce in combinazione con il tasto Amiga */
  struct MenuItem *SubItem;   /* lista delle sottovoci, se prevista         */
  USHORT NextSelect;          /* voce successiva selezionata nel caso di    */
                              /* selezioni multiple                         */
};

/*
**  Segnalatori impostati da Intuition  Programma  Significato              */
*/
#define CHECKIT     0x0001 /*    NO        SI      Attributo selezionabile  */
#define ITEMTEXT    0x0002 /*    NO        SI      [T] testo - [F] immagine */
#define COMMSEQ     0x0004 /*    NO        SI      Scorciatoia via comando  */
#define MENUTOGGLE  0x0008 /*    NO        SI      Selezione on/off         */
#define ITEMENABLED 0x0010 /*    NO        SI      Voce [T] attiva - [F] no */

#define HIGHFLAGS   0x00C0 /*    NO        SI      Modi di evidenziazione   */
#define HIGHIMAGE   0x0000 /*    NO        SI      Immagine o testo utente  */
#define HIGHCOMP    0x0040 /*    NO        SI      Colori complementari     */
#define HIGHBOX     0x0080 /*    NO        SI      Bordo rettangolare       */
#define HIGHNONE    0x00C0 /*    NO        SI      Nessuno                  */

#define CHECKED     0x0100  /*   SI        SI      Voce selezionata         */

#define ISDRAWN     0x1000  /*   SI        NO      Sottovoci visualizzate   */
#define HIGHITEM    0x2000  /*   SI        NO      Voce evidenziata         */
#define MENUTOGGLED 0x4000  /*   SI        NO      Selez. on/off già attiva */
```

Le strutture per i menu

Le due principali strutture che si utilizzano per definire i menu da associare ad una finestra sono la struttura **Menu** (vedi figura 1) e la struttura **MenuItem** (vedi figura 2). La prima serve a definire i menu, ed a costruire quindi la barra dei menu *[menu strip]*, la seconda viene utilizzata sia per le voci che per le sottovoci. Analizziamole in dettaglio.

La struttura Menu

Una barra di menu può contenere uno o più menu. Se questi sono più di uno, essa è descritta da una lista di strutture **Menu**, legate l'una all'altra tramite il primo campo della struttura che altro non è se non il puntatore alla struttura successiva, secondo la tecnica a liste già descritta in una delle prime puntate. Ovviamente l'ultimo menu avrà questo campo impostato a NULL.

I successivi quattro campi servono a posizionare e dimensionare l'area di selezione *[select box]* del menu stesso, cioè quell'area entro la quale si deve trovare il cursore affinché, premendo il tasto destro del mouse, il menu venga aperto. Le aree di selezione relative a due menu differenti non dovrebbero mai sovrapporsi. Se questo succede ed il cursore si trova nell'area comune all'atto dell'apertura del menu, viene aperto quello che nella catena di strutture **Menu** viene prima partendo dall'inizio della lista. Nella attuale versione del sistema operativo [1.3], Intuition ignora completamente i campi **TopEdge**, usando al suo posto il valore **TopBorder** relativo allo schermo su cui è aperta la finestra, ed **Height**, per il quale viene utilizzata l'altezza della barra del titolo dello schermo. È possibile che in futuro, specialmente se i menu potranno essere resi attraverso immagini grafiche invece del solo titolo, queste variabili vengano riconosciute da Intuition. Una sola considerazione su **LeftEdge**: questa è misurata a partire dal punto più a sinistra dello schermo, *bordo sinistro incluso*.

Il campo successivo viene utilizzato per i seguenti segnalatori:

MENUENABLED
indica se il menu è attivo o meno. Va impostato prima di chiamare la funzione **SetMenuStrip()**, a meno che non si voglia che il menu sia disattivato fin dall'inizio, e quindi non accessibile dall'utente. Ovviamente si possono sempre utilizzare le funzioni **OnMenu()** ed **OffMenu()** per modificare lo stato del menu.

MIDRAWN
indica se la lista delle voci relative al menu è visualizzata o meno. Sta infatti per **Menu's Items DRAWN** (voci del menu visualizzate).

Il campo **MenuName** è il puntatore alla stringa che apparirà sulla barra dei menu all'interno dell'area di selezione. Al momento infatti, come già accennato, i menu possono essere rappresentanti solo da *testi*, al contrario delle voci, come vedremo tra poco.

L'ultimo campo a disposizione del programmatore è il puntatore alla catena di strutture **MenuItem** che descrive la lista delle voci associate al menu in questione.

Ci sono poi altri quattro campi riservati al sistema e dai nomi alquanto curiosi, che di conseguenza ci limiteremo ad ignorare.

La struttura MenuItem

Questa struttura può essere utilizzata sia per le voci che per le sottovoci. Come nel caso dei menu, le voci e le sottovoci possono formare una catena grazie al primo campo della struttura che punta alla struttura successiva nella lista od a NULL se si tratta dell'ultimo elemento della lista.

Anche qui i successivi quattro campi della struttura servono a posizionare e dimensionare l'area selezionabile relativa alla voce. A differenza di quanto visto per i menu, tuttavia, tutti e quattro i campi sono utilizzati da Intuition. Vedremo, nella prossima puntata, in che modo, anche in funzione del tipo di elemento con cui si intende rappresentare la voce, se cioè un testo oppure un'immagine.

Il campo successivo viene utilizzato per i segnalatori relativi alla voce. Questi sono molti di più che nel caso dei menu, e verranno analizzati in dettaglio nella prossima puntata.

Lo stesso dicasi per i tre campi successivi. Il primo ha lo scopo di definire se e quali voci possono essere selezionate contemporaneamente a quella in oggetto, gli altri due definiscono il tipo di oggetto (testo od immagine) da utilizzare per rappresentare la voce sullo

NOTE

1. Si dice *interfaccia* tra un codice chiamante ed una procedura o funzione, l'insieme delle regole che definiscono lo scambio di informazioni tra i due all'atto della chiamata, durante il periodo in cui viene eseguita la procedura chiamata, ed al momento che il controllo ritorna al codice chiamante. Una interfaccia definisce quindi:
- i parametri che il chiamante passa al chiamato, definendone i tipi e le modalità di passaggio (e.g. per valore o per variabile);
- l'eventuale valore di ritorno passato dal chiamato al chiamante (se funzione) o comunque quali parametri di ingresso possano essere stati modificati e quindi da considerare anche in uscita;
- le eventuali aree di lavoro comuni, i blocchi di controllo, i file, e le variabili globali utilizzate da entrambi;
- i prerequisiti alla chiamata (inizializzazione di aree, ordine di precedenza rispetto altre funzioni, e via dicendo).

2. Si chiamano variabili (o costanti) *globali* quelle variabili che sono viste da tutto il codice interno al programma. Si tratta del massimo livello di visibilità possibile, in quanto il suo campo di validità *[scope]* copre tutto il programma. Viceversa si dicono variabili *locali*, quelle variabili conosciute solo da una certa procedura, ed invisibili esternamente. Una variabile in effetti può poi essere locale ad una certa procedura, ma essere visibile ad altre procedure da questa chiamate. Il campo di visibilità può allora variare da una singola funzione atomica a tutto il programma.

Per evitare effetti collaterali nel passaggio del controllo da un livello funzionale ad un altro, si raccomanda di evitare di utilizzare variabili globali come meccanismo di comunicazione *(interfaccia)* tra una procedura ed un'altra, dato che questo non è immediatamente visibile all'atto della rilettura del codice, ed è facile quindi introdurre dei bachi facendo operare più funzioni sulle stesse aree dati. In generale il programma principale più le routine di inizializzazione e terminazione sono autorizzate a modificare le variabili globali, le altre possono solo utilizzarle. Naturalmente bisogna poi valutare caso per caso...

3. Dato che esistono al momento solo tre livelli nella gerarchia dei menu, e cioè i *menu*, le *voci* e le *sottovoci*, se la struttura **MenuItem** viene usata per definire una sottovoce il campo **SubItem** viene ignorato da Intuition. Tuttavia, al fine di garantirsi la compatibilità con le versioni future del sistema, è consigliabile assegnare il valore **NULL** a tale campo nel caso stiamo definendo una sottovoce.

schermo, sia quando questa è semplicemente mostrata, sia quando è evidenziata (cursore posizionato sopra e tasto destro del mouse premuto). In quest'ultimo caso il campo **Flags** deve contenere il valore **HIGHIMAGE**, anche se stiamo utilizzando solo testi e non immagini.

Il campo **Command** può contenere un singolo carattere alfanumerico. Se questo campo non è nullo e **Flags** contiene il valore **COMMSEQ**, allora la voce può essere selezionata direttamente da tastiera per mezzo della combinazione di tasti *[tasto Amiga destro]* + *[carattere alfanumerico specificato]*, senza che sia necessario aprire il menu tramite mouse.

A questo punto abbiamo il puntatore alla lista delle eventuali sottovoci associate alla voce in questione, oppure NULL se non sono previste sottovoci o questa struttura fa riferimento essa stessa ad una sottovoce (vedi nota 3).

L'ultimo campo, **NextSelect**, viene impostato da Intuition quando l'utente seleziona la voce o la sottovoce. Nel caso di selezioni multiple, infatti, è importante essere in grado di rilevare *tutte* le voci (o sottovoci) selezionate dall'utente, non solo la prima. Questo campo ci permette di gestire questa situazione. Esso infatti punta la successiva voce (o sottovoce) selezionata dall'utente, oppure assume il valore **MENUNULL** nel caso siamo arrivati all'ultima voce selezionata o nel caso che sia stata effettuata una sola selezione.

Il programma scheletro

Il programma riportato in figura 3 è uno scheletro su cui continueremo a lavorare nelle prossime puntate. Il suo scopo è quello di costituire una base su cui sviluppare la maggior parte dei programmi che interfacciano Intuition, ed in particolar modo quelli che devono gestire menu, quadri e gadget. Vediamo i criteri su cui il programma in questione è stato realizzato.

Innanzi tutto esso è fortemente strutturato, in modo da poter funzionare ancor prima di essere terminato, grazie all'ausilio di speciali procedure vuote dette «tronconi» *[stub routines]*. Queste procedure altro non sono che piccole routine la cui interfaccia con il codice

```
/*****************************************************************************
**  Programmare in C su Amiga  (c) 1989  Dario de Judicibus - Roma (I)     **
** ------------------------------------------------------------------------- **
**  Scheletro di un programma di gestione dei menù.                          **
**                                                                           **
**    Questo programma crea una struttura a menù da associare ad una         **
**    finestra che poi apre sullo schermo del WorkBench. Il programma        **
**    è altamente strutturato in modo da presentare uno scheletro            **
**    flessibile da dettagliare in più fasi successive.                      **
**                                                                           **
**  Riconoscimento: questo scheletro è basato in parte su di una tecnica     **
**  sviluppata da John T. Draper - Sausalito (USA).                          **
**                                                                           **
*****************************************************************************/

 #include "exec/types.h"
 #include "intuition/intuition.h"
 #include "graphics/gfxmacros.h"
 #include "proto/exec.h"
 #include "proto/intuition.h"
 #include "proto/graphics.h"
 #include "stdio.h"
 #include "stdlib.h"
 #include "string.h"

/*
**  Tipi
*/
 typedef struct IntuiMessage IMSG;

/*
**  Prototipi delle funzioni interne al programma
*/
 void StartAll        ( void );
 void CloseAll        ( void );
 void BuildMenus      ( void );
 void LetsGo          ( void );
 int  HandleEvent     ( IMSG * );
/* ------------------------------ STUBs --- */
 int  H_MenuVerify    ( IMSG * );
 int  H_MenuPick      ( IMSG * );
 int  H_MouseButtons  ( IMSG * );
/* ----------------------------------------- */
 int  H_CloseWindow   ( IMSG * );
/*
**  Costanti
*/
 #define IREV 0
 #define GREV 0
 #define INAME "intuition.library"
 #define GNAME "graphics.library"
 #define DJ_COLS 400
 #define DJ_ROWS 150
 #define DJ_TITL "Esempio di gestione dei menù [0dJ]"
 #define GOAHEAD 1
 #define CLOSEME 0
```

```
/*
**  Caratteristiche della finestra: gadget di CHIUSURA, di 'PROFONDITA',
**  di SPOSTAMENTO, restauro automatico intelligente, tipo GZZ, attiva.
*/
 #define DJ_BASE  WINDOWCLOSE|WINDOWDEPTH|WINDOWDRAG
 #define DJ_SPEC  GIMMEZEROZERO|SMART_REFRESH|ACTIVATE

/*
**  Puntatori alle principali strutture
*/
 struct IntuitionBase *IntuitionBase;
 struct GfxBase *GfxBase;
 struct Window *w;
 struct RastPort *rp;
 struct MsgPort *up;
 IMSG *imsg;

/*
**  Strutture di definizione della finestra e dei menù
*/
 struct NewWindow nw =
 {
   20, 20, DJ_COLS, DJ_ROWS,  /* posizione e dimensioni della finestra  */
   0, 1,                      /* colore delle penne di fondo e di segno */
   CLOSEWINDOW,               /* Segnalatori IDCMP: gadget di chiusura  */
   DJ_BASE|DJ_SPEC,           /* caratteristiche della finestra         */
   NULL, NULL, DJ_TITL,       /* gadget, checkmark, titolo              */
   NULL, NULL, 0, 0, 0, 0,    /* schermo, superbitmap, dimensioni min.  */
   WBENCHSCREEN               /* da aprire sullo schermo del WorkBench  */
 };

/*
**  Maschere di controllo
*/
 #define IMASK 0x0001
 #define GMASK 0x0002
 #define WMASK 0x0004
 UWORD  mask = 0x0000;


/* ========================================================================= **
**                 STRUTTURE DI DEFINIZIONE PER I MENU'                      **
** ========================================================================= */
/* * * * * * * * * * * * * * * DA RIEMPIRE! * * * * * * * * * * * * * * * */

/*****************************************************************************
** main: programma principale                                               **
*****************************************************************************/
 void main()
 {
   StartAll   ();    /* Effettuiamo le chiamete di partenza.              */
   BuildMenus ();    /* Costruiamo i menù da associare alla finestra.     */
   LetsGo     ();    /* Va bene. E' tutto pronto. Andiamo!                */
   CloseAll   ();    /* Finito. Chiudiamo tutto.                          */
 }
```

chiamante è già stata definita (vedi nota 1), ma che di fatto non fanno niente se non ripassare il controllo indietro in modo che il programma possa continuare come se nulla fosse. Il vantaggio consiste nel poter definire fin dall'inizio la struttura e la logica del programma, senza peraltro doverlo codificare tutto prima di poterne verificare il funzionamento. Grazie a questa tecnica è possibile costruire un codice base, la cui logica è in linea di massima già quella finale, controllarlo con una serie di prove *[test]*, per poi passare a codificare i singoli blocchi uno alla volta. Questo permette inoltre di procedere a piccoli passi: si codifica un pezzo, si compila tutto, lo si prova, e se tutto va bene si passa a codificare quello successivo. In caso di errore, risulta molto più semplice identificare il codice responsabile del problema, dato che nell'ottanta per cento dei casi questo è localizzato nell'ultimo blocco aggiunto, specialmente se si è sviluppato il programma seguendo una tecnica detta *a scatole cinesi*. Questa tecnica si base sui seguenti criteri:

- le variabili globali sono in sola lettura da parte di tutte le procedure interne del programma, ad esclusione delle funzioni di inizializzazione e di terminazione, le uniche autorizzate a modificare una variabile globale (vedi nota 2);
- le interfacce tra le varie procedure sono ben definite ed avvengono esclusivamente per mezzo del passaggio di parametri; l'uso di aree di lavoro comuni è permesso a condizione che sia ben chiaro chi faccia cosa e che si cerchi di non utilizzarle per lo scambio di informazioni che possano alterare il flusso logico del programma stesso;
- ogni procedura ha una serie di responsabilità ben definite (possibilmente in numero limitato), ed il suo funzionamento non può alterare quello di altre procedure in dipendenza dell'ordine in cui queste vengono chiamate; fanno eccezione le sole procedure di inizializzazione, che vanno chiamate prima di tutte le altre, e quelle di terminazione, da chiamare prima di lasciare il programma;
- ogni procedura deve essere una scatola nera per il codice chiamante, questo non deve cioè fare assunzione alcuna sul modo di operare della routine

```
/*****************************************************************************
** StartAll: chiamate di partenza                                           **
*****************************************************************************/
void StartAll()
{
 /*
 **  Apre le librerie (Intuition & Graphics) e la finestra
 */
  IntuitionBase = (struct IntuitionBase *)OpenLibrary(INAME,IREV);
  if (IntuitionBase == NULL) CloseAll();
  mask |= IMASK;
  GfxBase = (struct GfxBase *)OpenLibrary(GNAME,GREV);
  if (GfxBase == NULL) CloseAll();
  mask |= GMASK;
  w = (struct Window *)OpenWindow(&nw);
  if (w == NULL) CloseAll();
  mask |= WMASK;
  rp = w->RPort;       /* RastPort per la grafica */
  up = w->UserPort;    /* Porta utente per IDCMP  */
}

/*****************************************************************************
** CloseAll: chiamate di chiusura                                           **
*****************************************************************************/
void CloseAll()  /* ordine inverso rispetto StartAll()!!! */
{
  if (mask & WMASK) CloseWindow(w);
  if (mask & GMASK) CloseLibrary(GfxBase);
  if (mask & IMASK) CloseLibrary(IntuitionBase);
  Exit(0);
}

/*****************************************************************************
** LetsGo: OK. Blocco principale di controllo                               **
*****************************************************************************/
void LetsGo(void)
{
 /*
 **  Svuotiamo la coda messaggi o mettiamoci in attesa del successivo
 */
  FOREVER  /* Ciclo infinito: si interrompe con "break" */
  {
    if ((imsg = (IMSG *)GetMsg(up)) == NULL)  WaitPort(up);
    else if (HandleEvent(imsg) == CLOSEME)    break;
  }
}

/*****************************************************************************
** BuildMenus: Costruisce i menù                                            **
*****************************************************************************/
void BuildMenus()
{
   /* * * * * * * * * * * * * * DA RIEMPIRE! * * * * * * * * * * * * * * */
}
```

```
/*****************************************************************************
** HandleEvent: gestione dei messaggi da Intuition                          **
*****************************************************************************/
int HandleEvent(msg)
  IMSG *msg;
{
  IMSG localmsg;     /* Questa è una fotocopia del messaggio ricevuto */
  int  result;       /* Questo è il valore da restituire al chiamante */
                     /* Fotocopiamo il messaggio originale            */
  CopyMem((char *)msg,(char *)&localmsg,sizeof(IMSG));

 /* ------------------------------------------------------------------ *
  * ATTENZIONE: i controlli di tipo VERIFY vanno messi PRIMA di        *
  *             rispondere al messaggio, altrimenti non servono a      *
  *             niente! Potevamo anche usare msg->Class, ovviamente.   *
  * ------------------------------------------------------------------ */
  switch (localmsg.Class)
  {
    case MENUVERIFY  : result = H_MenuVerify   (&localmsg);   break;
  }

  ReplyMsg((struct Message *)msg); /* OK. Adesso possiamo rispondere. */

 /* ------------------------------------------------------------------ *
  * ATTENZIONE: da questo punto in poi il messaggio puntato da "msg"   *
  *    non è più disponibile a questo task.  Useremo la copia locale   *
  *    salvata in "localmsg".  MENUVERIFY è ripetuto per evitare che   *
  *    l'assegnazione di "result" in "default:" si sovrapponga a       *
  *    quella precedente. Questo è uno dei tanto modi per evitarlo.    *
  * ------------------------------------------------------------------ */

  switch (localmsg.Class)
  {
    case CLOSEWINDOW : result = H_CloseWindow  (&localmsg);   break;
    case MENUPICK    : result = H_MenuPick     (&localmsg);   break;
    case MOUSEBUTTONS: result = H_MouseButtons (&localmsg);   break;
    case MENUVERIFY  : /* # già trattato in precedenza # */   break;
    default          : result = GOAHEAD                ;     break;
  };
  return (result);
}

/*****************************************************************************
** STUB ROUTINES: per il momento non fanno niente                           **
*****************************************************************************/
int H_MenuVerify    (msg)  IMSG *msg; { return(GOAHEAD); }
int H_MenuPick      (msg)  IMSG *msg; { return(GOAHEAD); }
int H_MouseButtons  (msg)  IMSG *msg; { return(GOAHEAD); }

/*****************************************************************************
** H_CloseWindow: gestisce l'evento CLOSEWINDOW                             **
*****************************************************************************/
int H_CloseWindow   (msg)  IMSG *msg; { return(CLOSEME); }
```

Figura 3 - Il programma scheletro.

```
PATTERN re_gen( char * );

Converte una espressione regolare fornita come stringa di caratteri, in
una rappresentazione interna utilizzabile da re_match() ed are_match().
In caso di errore ritorna NULL.

int re_match( char *, PATTERN );

Controlla se la stringa di caratteri fornita in ingresso soddisfa o meno
l'espressione regolare specificata. Quest'ultima deve essere già stata
convertita con re_gen() nell'opportuna rappresentazione interna.
In caso di errore ritorna -1, altrimenti restituisce la posizione del-
l'ultimo carattere della prima occorrenza che soddisfa l'espressione
regolare fornita.

int are_match( char *, int, PATTERN );

Opera come re_match(), salvo che la ricerca non avviene a partire dal
primo carattere della stringa, ma da una posizione specificata.

int re_smatch( char *, char * );

Opera come re_match(), solo che la stringa in cui avviene la ricerca è
la seconda, mentre la prima rappresenta l'espressione regolare nella sua
forma esplicita. Essa infatti chiama automaticamente la re_gen().

int are_smatch( int, char *, char * );

Opera come re_smatch(), salvo che la ricerca non avviene a partire dal
primo carattere della stringa, ma da una posizione specificata.
Anch'essa infatti chiama automaticamente la re_gen().
```

◀ *Figura 4. Le funzioni della* ***grep.lib***.

```
/*
**  Cerca la data del Natale di un qualunque anno del XX secolo in archivio.
*/
 #include "pat.h"
 #define DA_TROVARE "25 [Dd]icembre 19[0-9][0-9]"
 PATTERN espressione;
 .
 .
 .
 if ((espressione = re_gen(DA_TROVARE)) == NULL) CloseAll();
 numero_blocchi = 1;
 do
 {
   blocchi_letti = fread(blocco,BYTE_IN_BLOCCO,numero_blocchi,archivio);
   if (re_match(blocco,espressione) >= 0) Trovato();
 }
 while (blocchi_letti == numero_blocchi);
 .
 .
 .
```

Figura 5. Due esempi della ***grep.lib***.

chiamata, ma deve agire solo in base all'interfaccia che la caratterizza.

Ovviamente ci sono le dovute eccezioni, ma vanno sempre ben documentate, per evitare di perderle di vista allorché si metta di nuovo le mani sul codice in questione.

La seconda caratteristica dello scheletro riportato in figura, è che le varie operazioni sono ben identificate e raggruppate in procedure, rendendo più scorrevole la lettura del programma, senza necessariamente dover entrare nel dettaglio per comprenderne il funzionamento. Analogamente le varie definizioni ed i dati utilizzati dalle varie funzioni sono ben definite in testa al programma, rendendo particolarmente semplice la modifica delle stesse, senza che sia necessario ritoccare il codice.

Questo è costruito in modo da poter aggiungere facilmente nuove funzioni senza dover modificare le precedenti. Lo stesso dicasi nel momento in cui sorgesse la necessità di eliminare una funzione. Basta sostituirla con una procedura troncone senza preoccuparsi eccessivamente di possibili effetti collaterali. Anche qui, naturalmente, si sta facendo un discorso generale, tuttavia il livello di mantenibilità ed espandibilità di questo genere di programmi è decisamente elevato.

Entriamo ora nel dettaglio, facendo riferimento al codice in figura 3.

La prima parte è la solita: gli *include* necessari, un po' di definizioni sia di tipi, che di costanti, con in testa i prototipi delle procedure interne. Quindi i soliti puntatori alle librerie, alla struttura **Window**, a quella della relativa **RastPort** e, per comodità, anche alla porta utente per la comunicazione con Intuition. In più c'è il puntatore ad una struttura **IntuiMessage**. Di seguito definiamo la finestra, che per comodità prendiamo di tipo GZZ con **SMART_REFRESH**, in modo da rendere più semplici le future operazioni grafiche e da ridurre le interazioni con Intuition. Ovviamente questo non è necessario, specialmente se si vuole evitare uno spreco di memoria ed un possibile abbassamento delle performance (sempre che si scriva un codice veloce ed ottimizzato). Tuttavia, per evitare di mettere per ora un ulteriore codice di gestione del restauro della finestra, di una eventuale ridimensionamento della stessa da parte dell'utente, e delle operazioni grafiche ai bordi della stessa, ho deciso di impostare i due codici or ora menzionati in modo anche da evidenziare meglio la parte che ci interessa.

Per comodità lasciamo inoltre gestire ad Intuition il riposizionamento della finestra (**WINDOWDRAG**) e gli spostamenti in profondità (**WINDOWDEPTH**). Dato che per ora lo scheletro non include ancora il codice relativo ad i menu, sebbene sia già predisposto per la sua introduzione, l'unico evento di cui saremo notificati è quello relativo alla chiusura della finestra, per cui è stato aggiunto un gadget di chiusura ed arrivato il segnalatore IDCMP **CLOSEWINDOW**. Per ora apriremo la finestra direttamente sullo schermo del WorkBench.

Per finire le classiche maschere di apertura e chiusura, già adottate in precedenza.

Il blocco successivo è pronto a ricevere le definizioni della struttura a menu da associare alla finestra, e che costruiremo seguendo una tecnica sviluppata da *John T. Draper* e da me migliorata in modo da renderla ancora più potente.

A questo punto siamo al programma vero e proprio. Cosa fa? Basta leggere:

- apre l'ambiente di lavoro ed inizializza le strutture appropriate;
- costruisce i menu (per ora un troncone);
- parte con il ciclo principale;
- richiude l'ambiente di lavoro.

Semplice, vero? Già così, senza entrare nel dettaglio, abbiamo una chiara idea di ciò che fa il nostro programma.

Analizziamo adesso le singole procedure.

StartAll()

Questa procedura apre le varie librerie e la finestra, impostando opportunamente le principali variabili globali. Come di consueto si tiene traccia delle varie operazioni con la già acquisita tecnica delle mascherine.

CloseAll()

Questa procedura chiude quanto aperto da StartAll() o da altre procedure, utilizzando le maschere come lista di chiusura. In questo modo essa può essere richiamata da un qualunque punto del programma riuscendo a chiudere tutto e solo quello che è stato aperto

fino a quel momento, prima di terminare il programma stesso.

LetsGo()

È il blocco principale di controllo, che si preoccupa per il momento di svuotare la coda dei messaggi arrivati alla porta utente da Intuition, o di mettersi in attesa del messaggio successivo se questa è vuota. In caso l'utente abbia selezionato il gadget di chiusura della finestra, il ciclo termina ed il controllo ritorna al programma principale. Vengono utilizzati i due segnalatori **GOAHEAD** e **CLOSEME** che abbiamo introdotto nella puntata apparsa sul numero di dicembre di MCmicrocomputer.

BuildMenus()

Questa procedura per il momento è vuota. Il suo compito è quello di utilizzare le strutture che descrivono la struttura a menu da associare alla finestra, per costruire il blocco di strutture da passare ad Intuition affinché possa poi rendere graficamente i menu veri e propri. Vedremo nelle prossime puntate come riempirla. Di fatto, se compilate il programma, questo già funziona, pur avendo queste aree vuote.

HandleEvent()

Di questa procedura abbiamo già parlato due puntate fa. Rispetto a quella precedente, tuttavia, abbiamo fatto una serie di modifiche.

- Innanzi tutto essa è già pronta a gestire gli eventi relativi ad i menu, sebbene le procedure chiamate sono per ora realizzate come tronconi.
- In secondo luogo, sono stati introdotti dei controlli *prima* di rispondere al messaggio spedito da Intuition. Essi sono relativi agli eventi di tipo **VERIFY**, di cui parleremo in seguito. Basti dire per il momento che essi devono essere effettuati prima di restituire il controllo del messaggio ad Intuition, dato che il loro scopo è quello di intervenire tra una richiesta di operazione da parte dell'utente ad Intuition, ed il momento in cui questi soddisferà tale richiesta.
- Sono stati aggiunti degli operatori monadici per riallineare alcuni tipi ad i prototipi interni del Lattice C ed evitare così alcuni fastidiosi messaggi di avvertimento *[warning]*.

STUBs

Infine ci sono le procedure per la gestione dei singoli eventi, per la maggior parte realizzate come tronconi, a parte quella relativa alla chiusura della finestra, già attiva, sebbene altrettanto semplice tanto da non essere poi così diversa da un troncone.

E questo è tutto per il momento. Lo scheletro è già in grado di essere compilato con un semplice

1c - L menu.c

sebbene per ora il tutto dia luogo semplicemente ad un programma che apre una finestra ed aspetta che venga chiusa per terminare. Ma crescerà... crescerà!

GREP

Nelle ultime due puntate abbiamo parlato di *GREP* come di un programma di utilità particolarmente potente in grado di ricercare all'interno di un certo numero di file tutte le linee che soddisfino quella che viene chiamata *espressione regolare*, cioè una espressione che descrive una ben determinata stringa di carattere od addirittura un'intera classe di stringhe.

Per fare un esempio pratico *GREP* permette di risolvere problemi di tipo: *Cerca in tutti i file che iniziano con «ddj» ed hanno estensione «memo» tutte le righe che contengono la data del Natale di un qualunque anno del XX secolo ed in cui la prima lettera del mese può essere maiuscola o minuscola. Quindi visualizzale sullo schermo.*

Il tutto si ottiene col comando:

```
[1] grep "25 [Dd]icembre 19[0-9][0-9]" ddj#?.memo
```

Vediamo ora come si può fare la stessa cosa da un programma scritto in C.

Questo è possibile grazie alla libreria di compilazione **grep.lib**. Una lista delle funzioni di questa libreria è riportata in figura 4. Leggetela attentamente prima di continuare.

Innanzi tutto è necessario aggiungere al codice l'istruzione:

#include "pat.h"

a meno che non si preveda di utilizzare solo la forma esplicita delle espressioni regolari (e quindi solo le funzioni **re_smatch()** e **are_smatch()**). Quindi bisogna aggiungere alla lista delle librerie di compilazione in fase di legame *[link]* la libreria **grep.lib**. Ad esempio, supponendo che il programma si chiami **cerca**, il comando di **LINK** potrebbe essere:

```
blink FROM LIB:c.o+cerca.o TO cerca
      LIBRARY grep.lib+LIB:lc.lib+LIB:amiga.lib
```

Vediamo ora un esempio pratico (vedi figura 5). Cerchiamo, come sopra, tutti i record di un certo file che contengono la data di Natale di un qualunque anno del XX secolo. Supponiamo di aver aperto il file in questione e di aver ottenuto indietro in **archivio** il puntatore al file da usare con la **fread()**. Dato che dobbiamo chiamare più volte la funzione di ricerca, non ci conviene usare la **re_smatch()**. Questa infatti chiamerebbe ogni volta la **re_gen()** per generare la rappresentazione interna dell'espressione regolare da utilizzare per la ricerca. In questo caso è più veloce chiamare direttamente la **re_gen()** per generare l'espressione nella sua rappresentazione interna (e quindi è necessario includere **pat.h**), per poi utilizzare nel ciclo che scandaglia il file la funzione **re_match()**, più veloce.

Le funzioni di tipo **are_...** sono analoghe a quelle di tipo **re_...** salvo che la ricerca non avviene a partire dal primo carattere della linea da scandagliare, ma da una qualunque posizione specificata come parametro di ingresso.

Tutte le funzioni di ricerca restituiscono in uscita la posizione dell'ultimo carattere della prima sottostringa che soddisfa l'espressione regolare, in modo da permettere al programma di continuare la ricerca sulla stessa linea nel caso ci sia una seconda sottostringa che soddisfi la stessa espressione. Quindi, se cerchiamo 19[0-9] [0-9] nella stringa:

```
          1    1    2    2    3    3    4    4    5    5    6
1...5....0....5....0....5....0....5....0....5....0....5....0
Il crollo avvenne nella notte del 4 Gennaio 1976, alle tre.
```

la funzione di ricerca ritornerà l'intero 48.

Ah, dimenticavo. Il codice in figura 5 è corretto come codice, ma non è quello più adatto al tipo di ricerca che in genere si effettua su file di testo. Perché? La risposta il prossimo mese.

Conclusione

Nella prossima puntata vedremo come si costruisce un menu utilizzando le strutture che abbiamo descritto in questo articolo, e come si determina quale voce e/o sottovoce è stata selezionata dall'utente, completando così la prima stesura del nostro scheletro. Nel frattempo, studiate con cura lo scheletro proposto in figura 3 e provate a fare delle variazioni per renderlo più veloce e flessibile. Niente *sporchi* trucchi però. Potranno anche dare grossi vantaggi, all'inizio, ma alla lunga si pagano cari. Provate inoltre a modificare l'esempio riportato in figura 5 per tener conto del caso in cui in una stessa linea ci sia più di una sottostringa che soddisfi l'espressione regolare.

E come sempre, buon lavoro! MC