coordinamento di Andrea de Prisco

Programmare in C su Amiga (24)

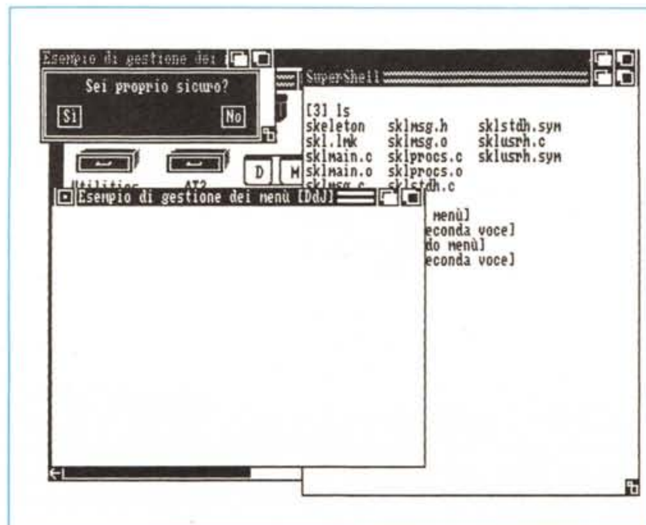
di Dario de Judicibus
(MC2120)

L'emissione di messaggi di errore e l'interazione diretta dell'utente nelle scelte logiche di un programma è un argomento base nel discorso delle tecniche di programmazione. In questa puntata vedremo come implementare tali tecniche nel nostro programma scheletro

Introduzione

Un programma scritto bene deve prevedere la possibilità che qualcosa vada storto. Ad esempio che non si possa aprire una libreria, o non si trovi un file a cui deve accedere, o più semplicemente che non ci sia abbastanza memoria libera per portare a termine una qualche operazione. Deve cioè prevedere tutta una serie di controlli a fronte di quelle operazioni che, se non effettuate o svolte non correttamente, possono avere un influsso negativo sulla continuazione del programma stesso od addirittura sul corretto funzionamento del sistema. Ovviamente, nel caso che uno di questi controlli metta in evidenza una situazione anomala, il programma deve passare il controllo ad una procedura che sia in grado di gestire il problema. In alcuni casi basterà modificare il flusso delle operazioni successive, ignorando di fatto l'evento verificatosi, in altri casi la decisione potrà essere più drastica, fino eventualmente al completamento dell'esecuzione con una indicazione di impossibilità a proseguire *[abend return code]*. Non solo. In alcuni casi il programma potrà prendere le sue decisioni da solo, in altri dovrà chiedere aiuto all'utente, vuoi per effettuare delle operazioni di compensazione, vuoi per determinare direttamente il flusso logico delle operazioni successive. Facciamo un paio di esempi. Un editore di testi viene chiamato passandogli il nome di un file. Il programma si accorge che il file non esiste. Nessun problema: se ne crea uno nuovo con quel nome. Il programma si è limitato ad operare in modo diverso saltando così il problema. A questo punto l'editore cerca il file di configurazione. Lo trova ma alcuni campi nel file non sono corretti. Il programma usa quelli di *default* ma avverte l'utente che c'è qualcosa di sbagliato nel file di configurazione. L'utente modifica il file, gli cambia nome e cerca di salvarlo. Il programma si accorge che esiste già un file con quel nome e non sa che fare. Non può certo decidere lui se è il caso o no di rimpiazzarlo. E se l'utente si fosse sbagliato? L'editore allora emette un messaggio chiedendo all'utente di confermare il salvataggio del nuovo file sul precedente. A questo punto l'utente chiede al programma di aprire una nuova finestra. Il programma ci prova, ma non c'è abbastanza memoria. Messaggio di errore: «*Non posso aprire la finestra. Non c'è abbastanza memoria per farlo*». L'utente allora chiude altri lavori concomitanti, ma nel far questo chiude per errore anche un processo collegato all'editore stesso. Chiede di nuovo, quindi, di aprire la seconda finestra, ma a questo punto il programma principale si accorge che non può comunicare con il processo di servizio che utilizzava per aprire e chiudere i documenti, emette un messaggio di errore di elevata severità e, non potendo neppure chiudere il documento principale, libera tutte le risorse allocate e ritorna il controllo al sistema operativo.

In questo scenario abbiamo visto un po' tutte le varie possibiità che si possono presentare. In pratica un buon programma deve poter comunicare con l'utente e ricevere da questi delle direttive, anche quando il modo normale di



Esempio di quadro automatico (autorequester).

operare non prevede esplicitamente una interazione diretta. I messaggi che un programma può emettere nei confronti dell'utente sono in sostanza di tre tipi:
Informativi
con i quali il programma avverte che un certo evento, in genere previsto, si è puntualmente verificato. Ad esempio, a seguito della richiesta di salvare un documento, il programma effettua l'operazione e quindi emette a video il messaggio *«Documento salvato»*.
Di Avvertimento
con i quali il programma denuncia una situazione anomala ma non tale da impedire di effettuare l'operazione. A seconda della situazione il programma può richiedere o meno l'intervento dell'utente. Un esempio è quello del salvataggio di un documento su uno già esistente, o l'identificazione di campi errati nel file di configurazione.
Di Errore
con i quali il programma denuncia il verificarsi di una situazione che rende impossibile il corretto svolgimento di una determinata operazione. Anche in questo caso l'utente può essere chiamato ad intervenire, infilando, ad esempio, il dischetto che contiene il documento da caricare nell'unità disco. Se l'errore tuttavia è grave, il programma può anche dover sospendere l'esecuzione se non addirittura terminarla del tutto. Ad esempio, se si è cercato di stampare ma non risulta esserci alcuna stampante collegata al computer, il sistema avvertirà l'utente della cosa e riprenderà quindi il normale corso delle operazioni. Se viceversa un programma si accorge che non può aprire, diciamo, in **intuition.library**, il flusso logico salterà direttamente alla procedura di chiusura del programma stesso.

In tutti i casi l'utente dovrebbe essere avvertito di cosa è successo o sta succedendo, anche se non è in grado in quel momento di intervenire. Se ad esempio manca una libreria di sistema, il programma, prima di terminare, dovrebbe avvertire l'utente di *quale* libreria si tratta, dandogli così la possibilità di installarla prima di lanciare di nuovo il programma.

Nei sistemi mono-tasking bastava emettere un messaggio a console. In quelli multi-tasking basati su una interfaccia a finestre, è spesso necessario aprire una finestrella contenente il messaggio emesso e, possibilmente, un sistema di controllo interattivo che permetta all'utente di dire al programma: *«OK. Ho letto. Adesso fai così...»*.

In questa puntata vedremo come si può implementare un meccanismo del genere nel nostro programma scheletro.

Prima di passare alle modifiche effettuate, è necessario introdurre due nuove tecniche, una sempre relativa ad Intuition, l'altra propria di tutti i compilatori C conformi allo standard ANSI.

Il quadro automatico

Certamente vi sarà capitato più di una volta di veder apparire nell'angolo in alto a sinistra dello schermo del WorkBench una finestrella rettangolare con la richiesta di reinserire un determinato disco o di attivare la stampante. A volte è bastato effettuare l'operazione richiesta per farla scomparire e continuare a lavorare senza ulteriori interruzioni, a volte avete dovuto selezionare il rettangolino con la scritta *retry* o quello con la scritta *cancel* per poter andare avanti. La finestra in questione si chiama *quadro di sistema [system requester]*, ed è uno dei più semplici quadri che l'Amiga vi permette di costruire.

Un quadro è in sostanza una finestra pensata come supporto a tutta una serie di elementi il cui scopo è quello di ricevere e fornire informazioni al sistema o ad un programma che in esso gira. Ad

```
SINTASSI:

long AutoRequest
(
   struct Window *      /* la finestra a cui il quadro è associato o NULL */
 , struct IntuiText *   /* il testo informativo sopra i pulsanti          */
 , struct IntuiText *   /* il testo del pulsante positivo (a sinistra)    */
 , struct IntuiText *   /* il testo del pulsante negativo (a destra)      */
 , long                 /* il segnalatore relativo agli eventi "positivi" */
 , long                 /* il segnalatore relativo agli eventi "negativi" */
 , long                 /* la larghezza del quadro                        */
 , long                 /* l'altezza del quadro                           */
);

ESEMPIO:

long fatto = 0L;

struct IntuiText *testoBase =
{
  0,1,JAM1,50,5,NULL,"Rimetti il dischetto!",NULL
};
struct IntuiText *testoPositivo =
{
  0,1,JAM1,7,3,NULL,"Va bene",NULL
};
struct IntuiText *testoNegativo =
{
  0,1,JAM1,7,3,NULL,"Cancella",NULL
};

long eventiPositivi = DISKINSERTED;
long eventiNegativi = NULL;

long larghezza = 250;
long altezza   = 120;

fatto = AutoRequest(w,testoBase,testoPositivo,testoNegativo,
                    eventiPositivi,eventiNegativi,larghezza,altezza);
```

Figura 1
AutoRequest().

Figura 2 - stdarg.h.

```
#ifndef _STDARGH
#define _STDARGH 1

#ifndef _VA_LIST
#define _VA_LIST 1
typedef char *va_list;
#endif

#define va_start(a,b) a=(char *)(&b+1)
#define va_arg(a,b) *((b *)( (a+=sizeof(b))-sizeof(b) ))
#define va_end(a) /* end of varg a */

/**
*
* Define NULL if it's not already defined
*
*/
#ifndef NULL
#define NULL    (void *)0
#endif

#endif
```

```
void curva (tempo)
  struct tm *tempo;
{
  va_list vl; /* definisci una lista di parametri a numero variabile "vl" */

  anytype anyarg; /* ESEMPIO: questo è un argomento di un qualche tipo    */
  othtype otharg; /*          questo è un altro argomento di tipo diverso */

  va_start(vl,tempo); /* inizializza la lista - "vl" punta ora alla lista */
                      /* che segue il primo parametro, cioè "tempo"       */

  /* Qui inizia l'estrazione dei vari argomenti. E' responsabilità del    */
  /* programmatore:                                                       */
  /* - associare a ciascun argomento il tipo corretto                     */
  /* - sapere quando la lista è finita                                    */

  va_arg(anyarg,anytype); /* carichiamo ora in "anyarg" il contenuto del  */
                          /* primo parametro della lista (tipo "anytype") */
  va_arg(otharg,othtype); /* carichiamo ora in "otharg" il contenuto del  */
                          /* secondo parametro in lista (tipo "othtype")  */

  va_end(vl); /* termina la lista - non sempre è una NOOP - dipende dal C */

  /* CORPO DELLA FUNZIONE */

}
```

Figura 3
Utilizzo
delle macro
va_xxx.

```
printf("INTERO %ld\n",1234L);   /* stampa INTERO 1234            */
printf("ESADEC %8X\n",1234L);   /* stampa ESADEC 000004d2        */
```

▲
Figura 4 - printf().

esempio, il pannello di controllo del vostro videoregistratore è un «quadro». Su di esso sono infatti disponibili una serie di pulsanti, cursori, indicatori di vario tipo, che vi permettono di operare sul VCR e di verificarne il corretto funzionamento.

Nell'Amiga i vari elementi di controllo sono detti *gadget*. D'ora in poi useremo indifferentemente sia il termine inglese che il termine italiano *controllo* (non è la traduzione letterale, ma è il termine che di fatto si usa per le varie apparecchiature che usiamo ogni giorno). Abbiamo già visto quando abbiamo parlato di finestre alcuni controlli di sistema associabili ad una finestra: i controlli di profondità, quello per la chiusura della finestra, la barra di spostamento, il controllo delle dimensioni. Intuition ci mette e disposizione vari tipi di controlli e svariati modi di utilizzo e di interazione con essi. Alcuni sono del tipo *a pulsante*, cioè vanno «spinti» con il puntatore del mouse per essere attivati, altri sono *a cursore*, cioè vanno agganciati e fatti scorrere lungo un binario od all'interno di un rettangolo definito, altri ancora sono *a campo d'ingresso*, cioè permettono l'immissione di informazioni di vario tipo sotto forma di stringa di caratteri. Vedremo tutto ciò in dettaglio dalla prossima puntata, quando incominceremo a parlare dei vari tipi di controlli. Per il momento vediamo un modo estremamente semplice per generare un quadro tipo quello di sistema senza dover definire le varie strutture associate al quadro stesso ed ai controlli che utilizza.

Il quadro in questione si chiama *quadro automatico [autorequester]*. È un quadro estremamente semplice da definire ma, proprio per questo, poco flessibile e di limitato utilizzo. Il quadro automatico (vedi schermo pubblicato nella prima pagina dell'articolo) è un rettangolo di altezza e larghezza definibili che viene visualizzato *sempre* nell'angolo in alto a sinistra dello schermo, sebbene una volta emesso, possa essere spostato dall'utente essendo dotato di barra di spostamento. Il quadro contiene tre elementi, un testo e due controlli a pulsante, ognuno contenente a sua volta un testo. Il testo posizionato sopra i pulsanti serve a fornire informazioni e, in genere, a richiedere all'utente di effettuare una determinata operazione. Questa può essere da sola sufficiente a «soddisfare» il quadro o può richiedere la selezione esplicita di uno dei due pulsanti. È anche possibile che, data la peculiarità di una determinata richiesta, sia disponibile un solo pulsante, invece di due.

Facendo riferimento alla figura 1, vediamo ora come si invoca un quadro automatico e quali sono i parametri da passare alla funzione apposita.

La funzione si chiama **AutoRequest()**. Il suo compito è quello di creare automaticamente il quadro, mettersi in attesa di uno o più eventi che soddisfino la richiesta effettuata, e quindi di cancellare il quadro deallocando la memoria utilizzata ritornando al programma il tipo di risposta ottenuta. Questa può essere solo di due tipi: positiva o negativa. È positiva se l'utente ha selezionato il pulsante positivo, posto a sinistra, oppure se il sistema ha segnalato uno degli eventi definiti dal programmatore come «positivi». È negativa se viceversa l'utente ha selezionato il pulsante negativo, posto a destra, oppure se il sistema ha segnalato uno degli eventi definiti dal programmatore come «negativi». Nel primo caso la funzione ritorna il valore **TRUE**, nel secondo caso il valore **FALSE**. In figura è anche mostrato un esempio. Nel nostro caso è stato definito l'evento relativo all'inserzione di un dischetto in una unità disco come evento positivo, atto cioè a soddisfare il quadro. Grazie a questa possibilità l'utente deve solo inserire il dischetto richiesto nel computer, senza necessariamente dover selezionare anche il pulsante di sinistra, cosa che rende molto più flessibile l'utilizzo dei quadri di sistema.

Il quadro può essere associato ad una finestra oppure, se il primo parametro passato è nullo, la funzione creerà una finestra *ad hoc* a cui associare il quadro. Inoltre un quadro automatico deve contenere sempre almeno il pulsante negativo. Quello positivo può essere omesso passando **NULL** al posto della struttura **IntuiText** relativa al pulsante sinistro.

Quando parleremo più in dettaglio dei quadri, analizzeremo anche come funziona internamente la funzione **AutoRequest()**. Per il momento l'importante è averne capito l'utilizzo.

Un'ultima cosa prima di proseguire. Affinché la funzione **AutoRequest()** ritorni il controllo al programma chiamante, è necessario che Intuition sia libero di girare ed accettare la risposta dall'utente. Se l'utente preme il bottone di destra del mouse ed il segnalatore **MENUVERIFY** era attivato, Intuition si pone in attesa che il programma risponda all'indicazione di verifica dell'apertura di un menu. D'altra parte il vostro codice di controllo situato nella procedura **HandleEvent()** non può girare finché **AutoRequest()** non torna il controllo al vostro programma. Risultato: un circolo vizioso da cui non ne uscite più *[deadlock]*.

Per evitare tutto ciò bisogna temporaneamente cancellare il segnalatore **MENUVERIFY** dalla lista degli eventi legati alla finestra interessata, se era stato attivato in precedenza, e quindi riattivar-

```
extern int vprintf(char *, va_list);
extern int vfprintf(FILE *, char *, va_list);
extern int vsprintf(char *, char *, va_list);
```

Figura 5 - Funzioni della famiglia vprintf().

```
/****************************************************************************
** ShowMsgReq: visualizza un messaggio in un quadro                        **
****************************************************************************/
 BOOL ShowMsgReq(wptr,msg,type)
  struct Window *wptr;
  char          *msg;
  UWORD         type;
 {
  BOOL answer;                  /* Risposta dell'utente                  */
  SHORT width;                  /* Larghezza del quadro                  */
  ULONG oldflags;               /* Salva temporaneamente lo stato IDCMP  */
  ITXT mtxt[3] =
  {
     {0,1,JAM1,50,5,NULL,NULL,NULL}                     /* Messaggio       */
    ,{0,1,JAM1, 7,3,NULL,NULL,NULL}                     /* Testo positivo  */
    ,{0,1,JAM1, 7,3,NULL,NULL,NULL}                     /* Testo negativo  */
  };

 /*
 **  Che tipo di bottoni?
 */
  /*P*/if (type & PBT_YES)  mtxt[1].IText = (UBYTE *)"Si";
  else                      mtxt[1].IText = (UBYTE *)"Continua";
  /*N*/if (type & NBT_NO)   mtxt[2].IText = (UBYTE *)"No";
  else                      mtxt[2].IText = (UBYTE *)"Cancella";

 /*
 **  Calcola la larghezza del quadro in modo da farci entrare tutto.
 **  Posiziona il messaggio al centro del quadro.
 */
  mtxt[0].IText = (UBYTE *)msg;
  width = ITXTL(&mtxt[2]) + 7;
  if (type & TWOGADG) width += ITXTL(&mtxt[1]) + 7;
  width = max(ITXTL(&mtxt[0]),width) + 100;
  mtxt[0].LeftEdge = ((width - ITXTL(&mtxt[0])) >> 1) - 5;

 /*
 **  Prima di chiamare AutoRequest(), disabilita lo stato MENUVERIFY,
 **  per evitare che Intuition si ponga in attesa della risposta e
 **  non possa a sua volta gestire il quadro ("deadlock").
 */
  if (wptr != NULL)
  {
    oldflags = wptr->IDCMPFlags;
    ModifyIDCMP(wptr,oldflags & ~MENUVERIFY);
  }

  answer = (BOOL)AutoRequest(wptr,&mtxt[0],(type & TWOGADG)?&mtxt[1]:NULL,
                             &mtxt[2],NULL,NULL,width,50);

  if (wptr != NULL) ModifyIDCMP(wptr,oldflags);
  return (answer);
 }
```

◀ *Figura 6*
Show MsgReq().

```
/*
**  Definizioni da precompilare per generare la tabella SKLUSRH.SYM
*/

/*
**  Costanti
*/
 #define ONEGADG 0x0000
 #define TWOGADG 0x0001
 #define PBT_YES 0x0100
 #define NBT_NO  0x0200

/*
**  Tipi
*/
 typedef struct Node           NODE;
 typedef struct Message        EMSG;
 typedef struct IntuiMessage IMSG;
 typedef struct IntuiText      ITXT;
 typedef struct Menu           MENU;
 typedef struct MenuItem       ITEM;

 typedef struct iDesc
 {
   UBYTE  cmd;
   UBYTE *txt;
   UBYTE *alt;
 }
 IDESC;

/*
**  Prototipi delle funzioni di servizio
*/
 int  TotTextNumber( IDESC*, int );
 BOOL ShowMsgReq   ( struct Window *, char *, UWORD );
 void SetupMenu    ( MENU *, MENU *, BYTE *, ITEM * );
 void SetupItemList( ITEM *, ITEM *, int, USHORT, ITXT *, IDESC *, ITEM ** );
 void SetExclude   ( ITEM *, int, int );
 void ClearExclude ( ITEM *, int, int );
 void CloseSafelyWindow ( struct Window *, struct TextFont * );
```

▲
Figura 7 - sklusrh.c.

lo una volta che **AutoRequest()** vi ha restituito il controllo.

Funzioni a parametri variabili

Una delle funzioni più utilizzate in C è senza dubbio la **printf()**. Lo scopo di questa funzione è quello di convertire gli argomenti che le sono stati passati secondo un certo formato e di includerli in una stringa di caratteri ASCII da emettere a video. La caratteristica più peculiare, tuttavia, di tale funzione — e di quelle appartenenti alla stessa famiglia, come **scanf()** — rispetto alle altre funzioni interne del C, è quella di non avere *a priori* un numero di argomenti predefinito. In pratica è possibile passare a queste funzioni da uno ad enne argomenti, senza dover dichiararne il numero prima. Tali funzioni sono dette *a numero di parametri variabile*. Si tratta di una proprietà estremamente utile, che può far comodo in molte situazioni. Ad esempio, immaginiamo di dover scrivere una funzione **curva()** che calcoli e tracci sullo schermo la curva che meglio approssima una serie di dati sperimentali, per esempio l'umidità relativa ad un certo giorno dell'anno, con la condizione che i dati possano anche non essere presi ad intervalli regolari nel tempo. Avremo allora in ingresso due vettori, uno relativo all'ora alla quale sono stati effettuati i vari rilevamenti, e l'altro relativo alle corrispondenti misurazioni effettuate. Il prototipo sarà allora

```
void curva( struct tm *tempo, float *umido);
```

dove **tempo** punta al vettore dei tempi ed **umido** a quello dei dati rilevati. Ad un certo punto ci accorgiamo che per le caratteristiche proprie della strumentazione utilizzata, le misurazioni effettuate durante l'arco della giornata non hanno la stessa precisione, per cui è necessario introdurre un vettore di *pesi* che tengano conto dei differenti errori di misurazione determinati nel tempo. Il prototipo va quindi modificato come segue:

```
void curva( struct tm *tempo, float *umido,
float *peso);
```

Ovviamente nei due casi la formula utilizzata sarà differente, tuttavia sarebbe ridondante scrivere due differenti funzioni, una nel caso in cui i dati sperimentali hanno tutti lo stesso peso, ed una in cui i pesi siano differenti. Potremmo allora essere tentati a scrivere solo la seconda, ma questo ci costringerebbe a passare comunque il terzo vettore, riempito da una serie di valori identici, ad esempio da *uno*. Si tratta di una soluzione sporca che, oltre a costringere il programmatore ad uno spreco inutile di memoria, riduce comunque le *performance* di **curva()** dato che utilizza comunque la formula completa anche nel caso semplificato. Come fare allora, visto che comunque un prototipo va definito? Semplice. Basta definire una funzione a parametri variabili.

```
/*************************************************************************
** H_MenuPick: gestisce l'evento MENUPICK                              **
*************************************************************************/
/*
**  Alcune definizioni per la stampa
*/
 #define PRT_MENU(m)     printf("Menù      [%s]\n",(m))
 #define PRT_ITEM(v,w)   printf("Voce      [%s]\n",itemname[v][w].txt)
 #define PRT_SUBI(s,t,u) printf("Sottovoce [%s]\n",subiname[s][t][u].txt)

 int H_MenuPick(msg)
   IMSG *msg;
 {

  :


    /*
    **  BLOCCO PER LA GESTIONE DEI CODICI
    */
     switch (menunum)
     {
       case MENU_100:
            PRT_MENU(MENU_1TX);
            (void)ShowMsgReq(whichwin,MENU_1TX,ONEGADG); /* Solo CANCELLA */
            switch (itemnum)
            {
              case ITEM_110: PRT_ITEM(MENU_100,ITEM_110); break;
              case ITEM_120: PRT_ITEM(MENU_100,ITEM_120); break;
              case ITEM_130: PRT_ITEM(MENU_100,ITEM_130); break;
              case ITEM_140: PRT_ITEM(MENU_100,ITEM_140);
                   switch (subnum)
                   {
                     case SUBI_141:
                          PRT_SUBI(MENU_100,ITEM_140,SUBI_141); break;
                     case SUBI_142:
                          PRT_SUBI(MENU_100,ITEM_140,SUBI_142); break;
                   }
                   break;
              case ITEM_150: PRT_ITEM(MENU_100,ITEM_150); break;
            }
            break;
       case MENU_200:
            PRT_MENU(MENU_2TX);
            (void)ShowMsgReq(whichwin,MENU_2TX,TWOGADG); /* CONT. & CANC. */
            switch (itemnum)
            {
              case ITEM_210: PRT_ITEM(MENU_200,ITEM_210); break;
              case ITEM_220: PRT_ITEM(MENU_200,ITEM_220); break;
              case ITEM_230: PRT_ITEM(MENU_200,ITEM_230);
                   switch (subnum)
                   {
                     case SUBI_231:
                          PRT_SUBI(MENU_200,ITEM_230,SUBI_231); break;
                     case SUBI_232:
                          PRT_SUBI(MENU_200,ITEM_230,SUBI_232); break;
                     case SUBI_233:
                          PRT_SUBI(MENU_200,ITEM_230,SUBI_233); break;
                     case SUBI_234:
                          PRT_SUBI(MENU_200,ITEM_230,SUBI_234); break;
                   }
                   break;
            }
            break;

  :


   return(GOAHEAD);
 }
```

Figura 8
H_MenuPick().

Lo standard ANSI prevede la possibilità di definire un prototipo anche per questo tipo di funzioni, utilizzando un particolare simbolo: *i tre puntini*. Nel nostro caso il prototipo verrebbe ad essere scritto così:

```
void curva( struct tm *tempo, ...);
```

Abbastanza intuitivo, no? Ma come si fa a gestire un numero variabile di argomenti? E come fa il C a controllare a questo punto la coerenza dei tipi passati con quelli definiti nel prototipo, visto che in effetti nel prototipo non sono stati affatto definiti! A tale proposito ci vengono in aiuto tre macro standard ANSI, definite nel file **stdarg.h** riportato in figura 2. Vediamo il loro utilizzo, lasciando al lettore il compito di comprendere il meccanismo che sta sotto al loro funzionamento.

Innanzi tutto il prototipo della nostra funzione deve *sempre* contenere almeno un argomento esplicitamente. Non è cioè possibile un prototipo del tipo

```
void funzione (...);
```

Quindi la funzione va codificata seguendo la falsariga riportata in figura 3. Come si può vedere, l'unico parametro riportato nella testata della funzione è quello dichiarato esplicitamente nel prototipo. Va quindi dichiarata una variabile di tipo **va_list**. Questa variabile rappresenta in sostanza il puntatore alla lista degli argomenti passati dopo il parametro esplicito. Quindi essa va inizializzata utilizzando la **va_start()**, alla quale va anche passato il parametro dichiarato nella testata. A questo punto **vl** (nel nostro esempio) punta al primo argomento della lista variabile. Per caricarsi questo valore in una variabile locale (o globale) di un certo tipo, è necessario usare la **va_arg()** la quale vuole in ingresso sia la variabile in cui deve essere caricato il parametro passato, sia il tipo di tale variabile. Essa inoltre sposta in avanti il puntatore alla lista, che viene così a puntare all'argomento successivo. Si procede così per tutti i parametri della lista. A questo punto va chiamata la **va_end()** che si occupa di chiudere la lista. Nel caso del *Lattice C* è una macro nulla, ma non è così in tutte le implementazioni. La definizione di queste macro dipende infatti dal sistema, anche se l'utilizzo è poi lo stesso in tutti i sistemi.

Ma come si fa a sapere quanti argomenti sono stati passati, e chi verifica che il tipo associato a ciascun argomento è quello corretto? La risposta è: *è un problema del programmatore, non del compilatore.* Ad esempio, nella **printf()** la funzione è in grado di sapere il numero di argomenti passati contando il numero di sequenze di sostituzione nella stringa di controllo, quelle cioè che iniziano col segno di *percento*. In questo caso le stesse sequenze dicono al programma come interpretare i vari argomenti per quello che riguarda il tipo. In questo modo, a parità di argomenti, è possibile effettuare associazioni differenti, come mostrato in figura 4. Un'altra tecnica potrebbe essere quella di passare come primo parametro proprio il numero di argomenti passati, o di utilizzare un argomento *tappo* che dice al programma che è arrivato alla fine della lista. In quanto al tipo degli argomenti, se è vero da un lato che il controllo è lasciato tutto al programmatore, non avendo alcun modo il compilatore di verificare la coerenza tra la dichiarazione iniziale e l'utilizzo della funzione, è anche vero che ciò ci mette a disposizione una tecnica potente, che va molto al di là del passaggio di una serie fissa di elementi dello stesso (vettori) o di differente (strutture) tipo, permettendo di utilizzare la stessa funzione per passare oggetti anche molto differenti. Ad esempio, si può pensare di creare una funzione **somma()** che è in grado di gestire sia la somma di enne numeri

```
/**************************************************************************
** CloseAll: chiamate di chiusura                                        **
**************************************************************************/
 void CloseAll(msgid)  /* ordine inverso rispetto StartAll()!!! */
   int msgid;
 {
   BOOL quit = TRUE;
   char outmsg[MSGLENGTH];
   va_list vl;

   if (msgid != EMSG_NONE)
   {
     va_start(vl,msgid);  /* Ora "vl" punta alla lista delle variabili    */
     (void)vsprintf(outmsg,ErrorMessages[msgid],vl);
    /*
    **  Funziona anche se w == NULL
    */
     if (msgid == EMSG_SAFEQUIT)
       quit = ShowMsgReq(w,outmsg,TWOGADG|PBT_YES|NBT_NO);
     else
       quit = !ShowMsgReq(w,outmsg,ONEGADG);
     va_end(vl);
   }

   if (!quit) return;                          /* Allora si continua!!! */

   if (mask & MSK_MST)
   {
     ClearMenuStrip(w);
     ModifyIDCMP(w,SaveFlags);
   }
   if (mask & MSK_MEM) FreeRemember(&rememory,TRUE);
   if (mask & MSK_WIN) CloseWindow(w);
   if (mask & MSK_GFX) CloseLibrary(GfxBase);
   if (mask & MSK_INT) CloseLibrary(IntuitionBase);
   Exit(0);
 }
```

▲ *Figura 9 - CloseAll().*

```
/**************************************************************************
** StartAll: chiamate di partenza                                        **
**************************************************************************/
 void StartAll()
 {
  /*
  **  Apre le librerie (Intuition & Graphics) e la finestra
  */

  :
```

In questo primo esempio, **CloseAll()** è chiamata ad emettere un messaggio *variabile*, cioè un messaggio che contiene un parametro modificabile dal programma stesso durante l'esecuzione. Nel caso in esame si tratta di una stringa, cioè di un parametro di tipo %s.

```
   w = (struct Window *)OpenWindow(&nw);
   if (w == NULL) CloseAll(EMSG_CANTOPEN,"la finestra");
   mask |= MSK_WIN;
   rp = w->RPort;       /* RastPort per la grafica */
   up = w->UserPort;    /* Porta utente per IDCMP  */

  /*
  **  Alloca dinamicamente le strutture per i menù e le voci
  **  Usa AllocRemember() per una gestione ordinata e flessibile
  */
   rememory = NULL;
```

Nel secondo esempio, **CloseAll()** è chiamata ad emettere un messaggio *fisso*, cioè un messaggio che non contiene alcun parametro e quindi non modificabile dal programma stesso durante l'esecuzione.

```
  /*
  **  Menù
  */
   menulist = (MENU *)AllocRemember(&rememory,
                 MENU_NUM * sizeof(MENU), MEMF_CLEAR);
   if (menulist == NULL) CloseAll(EMSG_NOMEMORY);
   mask |= MSK_MEM;                       /* Nota, basta la prima allocazione */

  :

 }
```

Figura 10 - StartAll(). ▶

interi, sia quella di enne numeri complessi a coefficienti interi. Al lettore il compito di scoprire come...

È anche possibile non aver bisogno di estrarre i singoli argomenti dalla lista utilizzando la **va_arg()**, qualora la lista vada passata così com'è ad una o più funzioni che a loro volta sono a numero di parametri variabile. È di fatto il nostro caso, come vedremo tra poco, avendo utilizzato nel nostro scheletro un'altra funzione interna del C ANSI, la **vsprintf()**. Questa altri non è che una **sprintf()** che gestisce *direttamente* una lista variabile di argomenti. In figura 5 sono riportate tutte le funzioni C ANSI che hanno fra i parametri in ingresso una lista variabile di parametri.

Da notare che, benché il *Lattice C 5.0* sia completamente conforme allo standard ANSI, e quindi supporti completamente sia le macro che le funzioni fin qui descritte, esse mancano completamente nella documentazione, lasciando pensare, erroneamente, che non siano supportate. Il problema è comunque secondario, dato che chi fosse interessato, può trovare in un qualunque manuale di riferimento dello standard ANSI per il C tutte le informazioni a riguardo. L'importante è che il compilatore segua tale standard (come di fatto fa il *Lattice C 5.0*).

Il programma scheletro

Vediamo ora di applicare quanto appreso fin qui al fine di implementare un meccanismo di comunicazione semplificata tra l'utente ed il programma, principalmente mirato alla gestione dei messaggi d'errore.

Per far questo è necessario introdurre due nuovi file ed una nuova procedura, con qualche modifica qua e là al codice già scritto.

ShowMsgReq()

Questa procedura, riportata in figura 6, ha il compito di emettere un quadro automatico contenente un messaggio ed uno o due pulsanti, a seconda delle necessità. Essa accetta in ingresso tre argomenti. Il primo è il puntatore alla finestra alla quale il quadro va associato, il secondo è la stringa di caratteri che corrisponde al messaggio da emettere, il terzo è il tipo di quadro da emettere. per il momento ho previsto due possibili classi di quadri, combinabili fra loro a formare quattro tipi distinti:

- la prima classe è basata sulla presenza di uno o di due pulsanti;
- la seconda classe è definita sulla base del testo da associare al, od ai pulsanti del quadro.

In figura 7 è riportato il sorgente della tabella simbolica utente che contiene, in testa, le quattro costanti usate per identificare le varie istanze delle due classi. Il file contiene anche tutti i prototipi delle funzioni di servizio, come al solito.

Vediamo in dettaglio la procedura **ShowMsgReq()**.

Per prima cosa definiamo tre strutture **IntuiText** standard. A seconda del tipo specificato dal programmatore selezioniamo uno dei due testi previsti per ognuno dei due pulsanti. Calcoliamo quindi la larghezza del quadro in modo da poterci far stare tutto il messaggio e carichiamo il messaggio nel testo principale. A questo punto disabilitiamo lo stato **MENUVERIFY**, come raccomandato in precedenza, dopo aver salvato il vecchio stato in una variabile temporanea (ovviamente solo se il puntatore alla finestra non è nullo). Chiamiamo quindi la funzione **AutoRequest()** e restituiamo il controllo al chiamante dopo aver eventualmente restaurato lo stato originale IDCMP avendo cura di riportare il responso fornito dal quadro automatico. Da notare che la funzione **AutoReque-**

st() mette in attesa il programma di una risposta da parte del sistema o dell'utente. Qualunque operazione effettuata sui menu mentre il quadro è in attesa viene quindi accodata per essere successivamente elaborata allorché il controllo ritorna al programma principale.

In quesstra procedura ci sono due punti che sarebbe opportuno modificare. Uno in quanto potrebbe portare ad un potenziale errore, l'altro per ragioni di mantenibilità. A voi trovarli...

H_MenuPick()

Vediamo ora come si può utilizzare direttamente questa funzione. Un esempio è riportato in figura 8 dove, durante la scansione degli eventi che avviene nel ciclo principale della procedura **H_MenuPick()**, oltre a visualizzare nella finestra CLI il nome delle varie voci selezionate, ho introdotto due chiamate alla **ShowMsgReq()**. Nella prima, corrispondente al primo menu, viene visualizzato un quadro con il solo pulsante negativo; nella seconda, relativa al secondo menu, il quadro riporta entrambi i pulsanti di scelta.

CloseAll()

Vediamo ora come usare in modo più generale la **ShowMsgReq()** per gestire i molti messaggi di errore che il programma è chiamato ad emettere in caso di errore nello svolgimento delle varie operazioni, soprattutto in fase di inizializzazione. Il tutto cercando di sconvolgere il meno possibile il codice già esistente, ovviamente.

Ogni volta che il nostro programma incontra una situazione di errore tale da impedirgli di andare avanti, invoca la **CloseAll()** per chiudere in modo pulito l'esecuzione. Sembra quindi logico considerare questa funzione come la naturale responsabile dell'emissione del messaggio di errore prima di terminare il programma. Per far ciò è necessario modificare la procedura in questione come riportato in figura 9. Per prima cosa introduciamo un parametro in ingresso corrispondente all'*identificativo* del messaggio che vogliamo emettere *[message identifier]*. Questi altro non è che l'indice di un vettore di stringhe che contiene i vari messaggi utilizzati dal programma. Tale vettore è contenuto nel file messaggi **sklmsg.c** riportato in figura 11. L'utilizzo di un file esterno per i messaggi di errore rende estremamente semplice la traduzione dei messaggi in altre lingue, senza dover per questo ricompilare tutto il programma. Basta solo ricompilare il file messaggi e riassemblarlo con gli altri moduli oggetto per generare un nuovo eseguibile. Ovviamente la cosa è più complessa se si vuole aggiungere o cancellare un messaggio, ma in questo caso è ovvio che bisogna quanto meno modificare la procedura che utilizza il nuovo messaggio. Si guadagna comunque in flessibilità ed in chiarezza. Per poter utilizzare il file messaggi, il programma deve includere il file degli identificativi riportato in figura 12. Questo file è stato separato dal file messaggi vero e proprio per evitare di dover ricompilare tutti i moduli che lo includono quando si traduce il file messaggi in un'altra lingua.

A questo punto sorge un problema. In genere un messaggio contiene delle informazioni legate all'errore verificatosi, non predefinibili nel file messaggi a priori. Ad esempio, se un editore non riesce ad aprire il file **pippo**, certo non può avere nel file messaggi il messaggio «*Non riesco a trovare "pippo"*». D'altro canto emettere un messaggio generico del tipo «*Non riesco ad aprire il file*» può essere di scarso aiuto se non addirittura confondere le idee all'utente, a volte. Come fare, allora? La tecnica da me sviluppata consiste nello scrivere i messaggi seguendo il formato dei modelli utilizzati da funzioni come **printf()**, in modo da utilizzare poi la **sprintf()** per generare il messaggio a partire dal modello base più i dati contingenti relativi allo specifico errore. Nell'esempio di cui sopra il modello sarà allora

modello = «Non riesco ad aprire il file %s /* diciamo %s un parametro */

mentre il messaggio finale sarà generato come segue

(void) sprintf (messaggio, modello, nomedelfile);

```
/*
** Questo è il file che contiene tutti i messaggi: gli avvertimenti,
** quelli di errore, quelli informativi, e via dicendo.
*/
char *ErrorMessages[] =
{
   "Sei proprio sicuro?"
  ,"Non riesco ad aprire %s"
  ,"Non ho abbastanza memoria"
};
```

Figura 11 - sklmsg.c.

```
#ifndef SKLMSG_H
#define SKLMSG_H

/*
** Questo è il file che contiene tutti gli identificativi per i messaggi:
** gli avvertimenti, quelli di errore, quelli informativi, e via dicendo.
*/
extern char *ErrorMessages[];

#define MSGLENGTH 256

#define EMSG_NONE       0xFFFF
#define EMSG_SAFEQUIT   0
#define EMSG_CANTOPEN   1
#define EMSG_NOMEMORY   2

#endif
```

Figura 12 - sklmsg.h.

```
/*************************************************************************
**************************************************************************
** main: programma principale                                          **
**************************************************************************
*************************************************************************/
void main()
{
  StartAll   ();        /* Effettuiamo le chiamete di partenza.         */
  BuildMenus ();        /* Costruiamo i menù da associare alla finestra. */
  for(;;)
  {
    LetsGo   ();        /* Va bene. E' tutto pronto. Andiamo!           */
    CloseAll (EMSG_SAFEQUIT);   /* Finito. Chiudiamo tutto.             */
    /* ------------------------------------------------------------------ */
    /*  Se si arriva qui, vuol dire che CloseAll() non ha terminato il    */
    /*  programma, dietro richiesta dell'utente. Il ciclo continua...     */
    /* ------------------------------------------------------------------ */
  }
}
```

Figura 13 - main().

Da notare che il modello non termina con un ritorno carrello (\n) in quanto il messaggio non va stampato ma viene caricato come testo di Intuition.

C'è tuttavia un altro problema. Dato che la procedura che riceve l'identificativo del messaggio ed i dati ad esso relativi è sempre la stessa (nel nostro caso la **CloseAll()**), e dato che noi non solo non sappiamo a priori il valore dei vari parametri definiti nel messaggio, ma neanche il numero di parametri che certamente varia da messaggio a messaggio, come possiamo scrivere una funzione sufficientemente generalizzata da essere utilizzabile per tutti i messaggi? Dovremmo scriverne una per quei messaggi che non hanno parametri, una per quelli che ne hanno uno, e così via. È qui che ci viene in aiuto la possibilità di definire funzioni a numero variabile di parametri. Basta infatti definire il prototipo della **CloseAll()** come segue:

```
void CloseAll (int, ...);
```

e poi utilizzare la funzione **vsprintf()** al posto della **sprintf()** come riportato in figura. Ed il gioco è fatto!

Per finire vorrei far notare che due identificativi di messaggio hanno un significato particolare:

```
EMSG_NONE
```

che indica che *non c'è* alcun messaggio da emettere, e

```
EMSG_SAFEQUIT
```

che emette il messaggio finale, per verificare se l'utente vuole realmente uscire o meno. Solo nel secondo caso viene emesso un quadro a due valori, dato che esso viene emesso qualora l'utente abbia deciso di chiudere *volontariamente* il programma. In tutti gli altri casi in cui la chiusura del programma è forzata da una situazione di errore, il quadro ha un solo bottone che serve solamente a dare il tempo all'utente di leggere il messaggio visualizzato.

Due esempi di utilizzo della nuova **CloseAll()** sono riportati in figura 10, entrambi presi dalla **StartAll()**. Nel primo esempio il messaggio emesso è di tipo variabile, nel secondo è di tipo fisso.

Figura 14 - skl.lmk.

```
#
# makefile for SKL
#
 NAME = skeleton
 MAIN = sklmain
 PROCS = sklprocs
 MSGS = sklmsg
 STDH = sklstdh
 USRH = sklusrh
#
 LC = lc:lc
 LINK = lc:blink
 LCOPTS = -H$(USRH).sym -O
 STARTUP = lib:c.o
 LIBS = lib:lc.lib+lib:amiga.lib
 LKOPTS = SD SC ND VERBOSE

#
# SKL definitions
#
$(NAME): $(MAIN).o $(PROCS).o $(MSGS).o
  $(LINK) FROM $(STARTUP)+$(MAIN).o+$(PROCS).o+$(MSGS).o \
          TO $(NAME) LIB $(LIBS) $(LKOPTS)

$(MAIN).o: $(MAIN).c $(USRH).sym $(MSGS).h
  $(LC) $(LCOPTS) $(MAIN).c

$(PROCS).o: $(PROCS).c $(USRH).sym
  $(LC) $(LCOPTS) $(PROCS).c

$(MSGS).o: $(MSGS).c
  $(LC) $(MSGS).c

#
# pre-compiled tables
#
$(STDH).sym: $(STDH).c
  $(LC) -ph -o$(STDH).sym $(STDH).c

$(USRH).sym: $(USRH).c $(STDH).sym
  $(LC) -ph -H$(STDH).sym -o$(USRH).sym $(USRH).c
```

main()

Nel caso l'utente cambi idea e decida di continuare, la **CloseAll()** ritorna il controllo al chiamante. Si è reso quindi necessario modificare anche la procedura principale come riportato in figura 13.

Infine in figura 14 è riportato il nuovo file di generazione che tiene conto del file messaggi e del file degli identificativi aggiunti in questa puntata.

Conclusione

Nella prossima puntata incominceremo a parlare di controlli *[gadget]* in modo da poter entrare in dettaglio in seguito nella descrizione dei vari tipi di quadro. Riprenderemo il discorso dei menu più avanti, per mostrare alcune interessanti tecniche avanzate.

MC

Casella Postale

Torna dopo tanto tempo la rubrica *Casella Postale*. Questo mese risponderò ad una lettera arrivatami il 28 maggio ma spedita a gennaio di quest'anno. Si riferisce ad un'altra lettera apparsa nella 17ª puntata di questa serie di articoli.

Caro Dario,
ti scrivo in relazione alla lettera di Matteo Olivieri apparsa su MCmicrocomputer di dicembre.

Vorrei far notare che l'Amiga ROM Kernel Reference Manual: Includes & Autodocs, versione 1.3, specifica chiaramente alla voce graphics.library/Flood che «allo scopo di usare Flood, il RastPort di destinazione deve avere un valido TmpRas le cui dimensioni siano grandi almeno quanto quelle del RastPort».

Sperando di averti fatto una cosa gradita, ti saluto cordialmente.
Alfonso Caschili - S. Anna Arresi (CA)

Alfonso, ti ringrazio per l'informazione. Al riguardo aggiungerò che la scheda relativa alla **Flood()** si trova a pag. A-86 del suddetto manuale e che a pag. 369 del primo volume dei nuovi RKMs, cioè *Libraries & Devices*, è riportata la seguente affermazione:
Flood-fill requires a TmpRas at least as large as the RastPort in which the Flood-fill in being done. This is to ensure that even if the Flood-filling «leaks», it will not flow outside the TmpRas and corrupt another task's memory.

Il che va a maggior onore di Matteo, che spedì la lettera nel lontano luglio '89, cioè diversi mesi prima che uscissero i RKMs 1.3.

Nella vecchia edizione di questi manuali (RMKs 1.1), infatti, tale informazione non era riportata.