Programmare in C su Amiga

di Dario de Judicibus

undicesima parte

Sviluppare un programma è qualcosa di più che saper scrivere il codice sorgente. Esiste tutta una serie di tecniche successive alla fase di preparazione del codice che sono fondamentali per la realizzazione di un qualunque programma. Da questa puntata inizieremo a presentare, in aggiunta agli ormai classici discorsi sulle funzioni che l'Amiga ci mette a disposizione, anche quelle tecniche e, perché no, quei trucchi, che ci permetteranno di semplificare notevolmente lo sviluppo dei programmi e di renderli sempre più potenti diminuendo al tempo stesso le dimensioni dell'eseguibile

Introduzione

Questa puntata è divisa in due parti. Nella prima prenderemo spunto dall'esercizio proposto nella scorsa puntata per incominciare a parlare di tecniche di compilazione. Abbiamo visto finora, infatti, come si possono utilizzare alcune delle funzioni che Amiga ci mette a disposizione per scrivere programmi più o meno potenti. Tuttavia non ci siamo ancora posti il problema di *come* vanno compilati questi programmi. Qualcuno di voi forse penserà che basta scrivere il programma, mandarlo in pasto al compilatore ed al cosiddetto *linkage editor* per vedersi sfornare bello e pronto il nostro modulo eseguibile. D'accordo. In prima approssimazione possiamo anche cavarcela in questo modo, ma a cosa serve scrivere un codice chiaro, potenzialmente veloce e potente, se poi non sfruttiamo appieno le possibilità che i compilatori ci mettono a disposizione? Certo ci sono anche quelli che si comprano una macchina fotografica con dieci programmi, sistema autofocus, flash con lettura TTL e tre o quattro sistemi di lettura dell'esposimetro, per poi farci sì e no venti scatti all'anno in occasione del compleanno della bambina o dell'anniversario di matrimonio, ma chi si indirizza verso la difficile arte della programmazione è per definizione su un altro piano culturale. Non si tratta di una questione di intelligenza od impegno, ma di cultura appunto. La logica, la praticità, lo spirito di curiosità, il desiderio di comprendere e d'imparare tipici di chi programma per diletto, portano a voler controllare sempre di più gli strumenti che si hanno a disposizione e ad acquisirne di sempre più potenti.

Per questo motivo, a partire da questa puntata, incominceremo a portare avanti un discorso nuovo, parallelo a quello della programmazione pura e semplice. Incominceremo cioè anche a porci problemi quali:

- come si gestisce e si mantiene un ambiente di sviluppo;
- come ridurre le dimensioni dei moduli prodotti;
- come si commenta e si documenta il codice;
- come si aumenta la velocità di esecuzione di un programma;
- trucchi e tecniche di programmazione e compilazione;
- come si analizzano i problemi in fase di esecuzione *[debug]*;

e molti altri ancora. Ovviamente il tutto sarà sempre relativo al nostro Amiga ed al linguaggio C, ma molti discorsi, opportunamente adattati, possono essere utilizzati in altri ambienti e per altri linguaggi. A tal proposito, invito tutti i lettori che hanno scoperto trucchi, tecniche od anche semplicemente sequenze di operazioni particolarmente utili, a scriverci in redazione. Le migliori saranno pubblicate sulla rivista. Mi raccomando però: primo, non mandate cinquanta pagine di stampa o programmi molto lunghi, cercate di evidenziare solo la tecnica od il trucco in questione; secondo, specificate sempre il vostro ambiente di lavoro (modello dell'Amiga, versione del compilatore,...) e cercate di commentare il più chiaramente possibile il materiale spedito. So che in Italia ci sono molti programmatori in gamba. Purtroppo alcuni fra i migliori si sprecano piratando SW americano e tedesco o creando virus ed altri programmini analoghi. Penso che, se queste persone si decidessero a mettere le loro capacità al servizio dell'utenza Amiga, come si fa da anni negli Stati Uniti, riceverebbero certamente molte più gratificazioni e riconoscimenti di adesso. C'è ancora molto da fare nel campo del software di pubblico dominio. Cerchiamo di farci conoscere all'estero in modo più positivo di quanto, a ragione od a torto, siamo conosciuti. I buoni programmi sono sempre apprezzati. Un esempio per tutti **Guardian** di Leonardo Fei (Milano), incluso da Fred Fish nella sua ormai ultranota e prestigiosa *compilation* (Disco #154). Spero in breve di

vedere altri nomi italiani in questa ed altre raccolte di PD e Shareware per Amiga.

La seconda parte inizia a presentare le funzioni grafiche della *graphics.library* ed il loro utilizzo. Grazie a queste funzioni ed alle tecniche presentate nelle scorse puntate, sarete in breve in grado di scrivere tutta una serie di programmi grafici anche complessi, che vanno sotto il nome di *grafica non interattiva*. Programmi di questo tipo sono ad esempio molti programmi sui frattali od i programmi grafici del tipo di quelli presentati su *Le Scienze* da A.K. Dewdney. Si tratta di programmi che una volta lanciati con determinati parametri iniziali, generano immagini di vario tipo senza che l'utente possa intervenire direttamente. In genere, quest'ultimo può solo terminare il programma. Una volta che avremo appreso come comunicare con Intuiton tramite la porta IDCMP, impareremo anche come scrivere programmi grafici *interattivi*, in cui cioè l'utente, tramite mouse o tastiera, può intervenire sull'esecuzione del programma e sulle immagini che esso forma sullo schermo.

La soluzione

Nella scorsa puntata abbiamo proposto un esercizio relativamente semplice, in teoria, ma particolarmente interessante per due motivi: innanzi tutto perché veniva chiesto di risolverlo in un modo *elegante*, cioè in modo abbastanza generalizzato da renderne facile l'ampliamento; secondo perché ci darà lo spunto per accennare ad alcune tecniche di compilazione abbastanza utili.

Riassumiamo brevemente quali caratteristiche erano state richieste nella scorsa puntata per il programma in questione: innanzitutto esso deve essere in grado di aprire un certo numero di finestre sullo schermo del WorkBench, ad ognuna delle quali va quindi associato un puntatore diverso. Inoltre le finestre devono avere solo la barra di spostamento con il titolo, mentre la loro chiusura deve poter venir effettuata tramite un gadget di chiusura posto in un'altra finestra, alta come la barra del titolo. Una tecnica del genere viene usata da alcuni *hack*. Questi sono dei programmini, PD solitamente di piccole dimensioni ma alquanto spettacolari. Una ca-

```
/*
** file: E10_1.c
**
** Soluzione dell'esercizio proposto nella decima puntata - prima parte
**
** 1989 (c) Dario de Judicibus - Creato il 27 Gennaio 1989
*/

 #include "exec/types.h"
 #include "exec/lists.h"
 #include "intuition/intuition.h"
 #include "proto/exec.h"
 #include "proto/intuition.h"

 UBYTE OpenWindows(void);
 void CloseWindows (void);

 #define NW 4
 #define FILL 0L
 #define DONE 0x01
 #define FAIL 0x00
 #define WF WINDOWDEPTH|WINDOWDRAG|SMART_REFRESH|NOCAREREFRESH

/*
** Scheletro valido per tutte le finestre
*/
 struct NewWindow skeleton =
   {0,0,0,0,-1,-1,NULL,FILL,NULL,NULL,FILL,NULL,NULL,0,0,0,0,WBENCHSCREEN};

/*
** Caratteristiche delle singole finestre
*/
 WORD x[NW] = {400,75,250,450};
 WORD y[NW] = {10,25,150,75};
 WORD w[NW] = {200,160,160,160};
 WORD h[NW] = {10,60,60,60};
 UBYTE *t[NW] = {"Testa","Sinistra","Al centro","Destra"};
 struct Window *wl[NW];

/*
** Puntatori alle strutture immagine per i puntatori - Vedi seconda parte
*/
 extern UWORD *mp[NW];

/*
** Dimensioni e centraggio dei vari puntatori
*/
 LONG mpdata[NW][4] =
 {
   {16, 16, -7, -7},
   { 9,  9, -4, -4},
   { 9, 16, -7, -7},
   { 9, 11,  0,  0}
 };
```

```
/*
** Programma principale
*/
 void main()
 {
   if (IntuitionBase = OpenLibrary("intuition.library",0L))
   {
     if ((OpenWindows() == DONE))
     {
       WaitPort(wl[0]->UserPort);
     }
     CloseWindows();
   }
 }

/*
** Preparazione ed apertura delle finestre
*/
 UBYTE OpenWindows()
 {
   UBYTE i, done = DONE;

   for (i=0; (i < NW) && (done != FAIL); i++)
   {
     skeleton.LeftEdge   = x[i];
     skeleton.TopEdge    = y[i];
     skeleton.Width      = w[i];
     skeleton.Height     = h[i];
     skeleton.Title      = t[i];
     skeleton.IDCMPFlags = (i == 0)?(ACTIVATE|CLOSEWINDOW):NULL;
     skeleton.Flags      = (i == 0)?(WF|WINDOWCLOSE):WF;
     if ((wl[i] = OpenWindow(&skeleton)) == NULL) done = FAIL;
     else SetPointer(wl[i], mp[i], mpdata[i][0],   /* * * * * * * * */
                                   mpdata[i][1],   /* Qui associamo */
                                   mpdata[i][2],   /*  i puntatori  */
                                   mpdata[i][3]);  /* alle finestre */
                                                   /* * * * * * * * */
   }
   return(done);
 }

/*
** Chiusura delle finestre
*/
 void CloseWindows()
 {
   UBYTE i;
   for (i=0; i < NW; i++) if (wl[i]) CloseWindow(wl[i]);
 }
```

Figura 1 - Esercizio: codice della prima parte.

tegoria di hack ha la caratteristica di duplicare lo schermo del WorkBench su di uno schermo utente, posizionarlo di fronte a quello vero, e «distruggerlo» un po' alla volta seguendo un algoritmo predeterminato. Roba da infarto! L'utente inesperto vede il suo schermo sciogliersi come neve al sole oppure polverizzarsi in pochi secondi. Niente paura: non è un virus. Se tirate giù lo schermo frontale con il mouse, vi accorgerete che il vostro WorkBench è ancora là, intatto come sempre. L'unica differenza sarà appunto la presenza di una nuova finestra, generalmente dell'altezza di una barra non interlacciata (10 pixel) dotata di titolo e gadget di chiusura. Basterà fare click su quest'ultimo per terminare il programma «assassino».

Torniamo ora al nostro esercizio. La parte più difficile consiste nello scrivere qualcosa di abbastanza flessibile da non richiedere un grosso lavoro di riscrittura nel momento che si decida di diminuire od aumentare il numero di finestre da aprire. Per far questo si è adottata la seguente tecnica (vedi figura 1).

Per prima cosa definiamo una struttura di tipo **NewWindow** contenente quei parametri i cui valori saranno gli stessi per tutte le finestre. Ad esempio, lo schermo è sempre lo stesso, e cioè quello del WorkBench; quindi **NewWindow.Type** è WBENCHSCREEN per tutte le finestre.

Quindi definiamo un certo numero di vettori per ogni parametro che assumerà valori differenti per ogni finestra, come ad esempio la posizione della finestra sullo schermo od il titolo da inserire nel bordo superiore.

Analogamente definiamo il vettore che contiene i puntatori agli sprite da associare ad ogni finestra, e la matrice che ne definisce le dimensioni e il centraggio (vedi MCmicrocomputer n. 83). Come si può vedere in figura, il vettore dei puntatori è dichiarato *esterno*. Vedremo tra poco perché.

Il programma principale è vergognosamente semplice! In pratica si limita ad aprire la libreria di Intuition ed a chiamare la funzione che apre le finestre. Quindi si mette in attesa sulla porta IDCMP della prima finestra, quella cioè che ha il gadget di chiusura. Appena arriva un segnale su quest'ultima, chiude tutto. In realtà l'unico segnale

```
/*
** file: E10_2.c
**
** Soluzione dell'esercizio proposto nella decima puntata - seconda parte
**
** 1989 (c) Dario de Judicibus - Creato il 27 Gennaio 1989
*/

 #include <exec/types.h>

/*
** Immagini per i puntatori
*/

 UWORD sphere[] =
 {
   0,0,
   0x07E0, 0x0320, 0x1FF8, 0x0CC8, 0x3FFC, 0x18E4, 0x7FFE, 0x4E18,
   0x7FFE, 0x5E1C, 0xFFFF, 0x9E0E, 0xFFFF, 0xE1F1, 0xFFFF, 0xC1F1,
   0xFFFF, 0xC1F1, 0xFFFF, 0xE1F1, 0xFFFF, 0x9E0E, 0x7FFE, 0x5E1C,
   0x7FFE, 0x4E18, 0x3FFC, 0x18E4, 0x1FF8, 0x0CC8, 0x07E0, 0x0320,
   0,0
 };

 UWORD cross[] =
 {
   0,0,
   0xC180, 0x4100, 0x6380, 0xA280, 0x3700, 0x5500,
   0x1600, 0x2200, 0x0000, 0x0000, 0x1600, 0x2200,
   0x2300, 0x5500, 0x4180, 0xA280, 0x8080, 0x4100,
   0,0
 };

 UWORD sexcl[] =
 {
   0,0,
   0x0FC3, 0x0000, 0x3FF3, 0x0000, 0x30C3, 0x0000,
   0x0000, 0x3C03, 0x0000, 0x3FC3, 0x0000, 0x03C3,
   0xC033, 0xC033, 0xFFC0, 0xFFC0, 0x3F03, 0x3F03,
   0,0
 };

 UWORD tower[] =
 {
   0,0,
   0xF000, 0x0000, 0x7800, 0x0000, 0x0000, 0x0000,
   0x1E00, 0x0000, 0x0F00, 0x0000, 0x0000, 0x0000,
   0x0780, 0x0000, 0x03C0, 0x0000, 0x0000, 0x0000,
   0,0
 };

 UWORD *mp[4] = {&sphere[0], &cross[0], &sexcl[0], &tower[0]};
```

Figura 2 - Esercizio: codice della seconda parte.

```
#
# file: E10.lmk
#
# Soluzione dell'esercizio proposto nella decima puntata - terza parte
#
# 1989 (c) Dario de Judicibus - Creato il 27 Gennaio 1989
#

#
# Variabili
#
LIBS = LIB:lc.lib+LIB:amiga.lib
LOPT = NODEBUG SC SD

#
# Come ottenere E10 da E10_1.o ed E10_2.o
#
E10: E10_1.o E10_2.o
  LC:blink FROM LIBo+E10_1.o+E10_2.o TO E10 LIB $(LIBS) $(LOPT)

#
# Come ottenere E10_1.o da E10_1.c
#
E10_1.o: E10_1.c
  LC:lc -bO E10_1.c

#
# Come ottenere E10_2.o da E10_2.c
#
E10_2.o: E10_2.c
  LC:lc -ad E10_2.c
```

Figura 3 - Esercizio: istruzioni per la creazione dell'eseguibile.

```
/*
** Prototipi delle funzioni grafiche: Versione 1.3
*/

/*
** *** TESTI ***
*/
 void  ClearEOL    (struct RastPort *);
 void  ClearScreen (struct RastPort *);
 long  TextLength  (struct RastPort *, char *, long);
 long  Text        (struct RastPort *, char *, long);
/*
** *** FONTS ***
*/
 void               AddFont      (struct TextFont *);
 void               AskFont      (struct RastPort *, struct TextAttr *);
 long               AskSoftStyle (struct RastPort *);
 void               CloseFont    (struct TextFont *);
 struct TextFont *  OpenFont     (struct TextAttr *);
 long               RemFont      (struct TextFont *);
 long               SetFont      (struct RastPort *, struct TextFont *);
 long               SetSoftStyle (struct RastPort *, long, long);
```

Figura 4 - Le funzioni grafiche dell'Amiga: Testi. ▶

che può arrivare è quello del gadget di chiusura, dato che abbiamo specificato

SMART_REFRESH|NOCAREREFRESH

nella struttura di scheletro iniziale.

La maggior parte del codice è situato nella funzione **OpenWindows()** (notare la «s» finale!). Questa non è, ovviamente, una funzione di Intuition, ma è una routine interna che utilizza lo scheletro di base ed i vettori precedentemente definiti, per preparare ed aprire le singole finestre. In pratica essa riutilizza sempre la stessa struttura **NewWindow** per tutte le finestre, variandone però prima i parametri definiti nei vari vettori. Inoltre, mentre per tutte le finestre *figlie* **IDCMPFlags** è nullo e **Flags** assume un valore comune definito come **WF**, la finestra *madre* viene targata come «attiva all'apertura» e le viene aggiunto il gadget di chiusura. A questo punto la finestra è aperta e, se tutto è andato bene, le viene associato un puntatore (del mouse). Quello invece della relativa struttura **Window** viene memorizzato in un apposito vettore, in modo da poter essere riutilizzato nella fase di chiusura. Notate che anche una sola operazione di apertura fallita fa terminare il programma. Se il numero delle finestre è grande e non è poi così importante che siano tutte aperte, si può viceversa propendere per una logica *minimale* in cui cioè, il programma apre tutte le finestre che riesce ad aprire ed ignora le altre. Dipende ovviamente da quel che si deve fare in seguito.

Noi, in seguito, non facciamo niente, dato che si tratta solo di un esercizio. In realtà ci metteremo in attesa che l'utente, dopo essersi divertito un po' a spostare le finestre e ad osservare i puntatori cambiare, si stufi e chiuda il tutto con il gadget della finestra madre.

Il codice di chiusura è, se possibile, ancora più semplice, ed è anche abbastanza generico da funzionare nel caso si sia scelto di utilizzare la logica minimale di cui sopra. In pratica non fa altro che scorrere il vettore che contiene i puntatori alle varie strutture **Window** e chiudere quelle finestre a cui corrisponde un puntatore non nullo.

«E gli sprite?» direte voi... Semplice: sono in un altro file (vedi figura 2).

Ricordate che nella scorsa puntata dicemmo che i dati che descrivono l'immagine di uno sprite vanno nella memoria di tipo CHIP? Bene, un modo per far questo è quello di separare il codice ed i dati che vanno in FAST dai dati che vanno in CHIP (vedi nota 1). I primi vanno quindi compilati usando l'opzione **-b0** che si assicura che tutti i dati definiti come *statici, esterni* e le stringhe di caratteri siano indirizzate per mezzo di un campo di spostamento *[offset]* rispetto al registro base A4 da 32 bit piuttosto che 16 bit come è in genere. Questo perché un *offset* di 16 bit permette al massimo di indirizzare campi distanti non più di 64K dalla base, mentre, se il programma si trova nella memoria FAST ed alcuni dati si trovano in CHIP, quasi certamente la distanza tra le due aree di memoria è superiore a 64K. Per garantire viceversa che i dati specificati nel secondo file vadano realmente in CHIP, è necessario utilizzare l'opzione di compilazione **-ad**. Quest'ultima implica *comunque* **-b0** per il file in questione. Non è quindi necessario che sia specificata esplicitamente anche l'opzione di indirizzamento a 32 bit, in questo caso.

Se avete qualche problema a decifrare le strutture in figura 2, andate a rileggere nell'ultima puntata come si costruisce una struttura dati per uno sprite a partire da un'immagine disegnata su di un foglio quadrettato.

Un altro vantaggio nel dividere il nostro programma in due, è il seguente.

Figura 6
Le funzioni grafiche dell'Amiga: Disegno.

```
/*
** Prototipi delle funzioni grafiche: Versione 1.3
*/

/*
** *** PUNTI ***
*/
 void  Move       (struct RastPort *, long, long);
 long  ReadPixel  (struct RastPort *, long, long);
 void  WritePixel (struct RastPort *, long, long);
/*
** *** LINEE ***
*/
 void  Draw       (struct RastPort *, long, long);
 void  PolyDraw   (struct RastPort *, long, short *);
/*
** *** AREE ***
*/
 long  AreaDraw   (struct RastPort *, long, long);
 void  AreaEnd    (struct RastPort *);
 long  AreaMove   (struct RastPort *, long, long);
 void  InitArea   (struct AreaInfo *, short *, long);
 void  RectFill   (struct RastPort *, long, long, long, long);
/*
** *** REGIONI ***
*/
 void            AndRectRegion   (struct Region *, struct Rectangle *);
 long            AndRegionRegion (struct Region *, struct Region *);
 long            ClearRectRegion (struct Region *, struct Rectangle *);
 void            ClearRegion     (struct Region *);
 void            DisposeRegion   (struct Region *);
 struct Region * NewRegion       (void);
 void            OrRectRegion    (struct Region *, struct Rectangle *);
 long            OrRegionRegion  (struct Region *, struct Region *);
 void            XorRectRegion   (struct Region *, struct Rectangle *);
 long            XorRegionRegion (struct Region *, struct Region *);
/*
** *** COLORI ***
*/
 void              FreeColorMap (struct ColorMap *);
 long              GetRGB4      (struct ColorMap *, long);
 struct ColorMap * GetColorMap  (long);
 void              LoadRGB4     (struct ViewPort *, short *, long);
 void              SetAPen      (struct RastPort *, long);
 void              SetBPen      (struct RastPort *, long);
 void              SetDrMd      (struct RastPort *, long);
 void              SetRGB4      (struct ViewPort *, long, long, long, long);
 void              SetRGB4CM    (struct ColorMap *, long, long, long, long);
```

Figura 5
Le funzioni grafiche dell'Amiga: Animazione.

```
/*
** Prototipi delle funzioni grafiche: Versione 1.3
*/

/*
** *** GELS ***
*/
 void  AddAnimOb    (struct AnimOb *, long, struct RastPort *);
 void  AddBob       (struct Bob *, struct RastPort *);
 void  AddVSprite   (struct VSprint *, struct RastPort *);
 void  Animate      (long, struct RastPort *);
 long  AreaEllipse  (struct RastPort *, long, long, long, long);
 void  DoCollision  (struct RastPort *);
 void  DrawEllipse  (struct RastPort *, long, long, long, long);
 void  DrawGList    (struct RastPort *, struct ViewPort *);
 void  GetGBuffers  (struct AnimOb *, struct RastPort *, long);
 void  InitGels     (struct VSprint *, struct VSprite *, struct GelsInfo *);
 void  InitGMasks   (struct AnimOb *);
 void  InitMasks    (struct VSprite *);
 void  RemIBob      (struct Bob *, struct RastPort *, struct ViewPort *);
 void  RemVSprite   (struct VSprite *);
 void  SetCollision (long, __fgptr, struct GelsInfo*);
 void  SortGList    (struct RastPort *);
/*
** *** SPRITE ***
*/
 void  ChangeSprite (struct ViewPort *, struct SimpleSprite *, short *);
 void  FreeSprite   (long);
 void  FreeGBuffers (struct AminOb *, struct RastPort *, long);
 long  GetSprite    (struct SimpleSprite *, long);
 void  MoveSprite   (struct ViewPort *, struct SimpleSprite *, long, long);
```

Supponiamo che, fatto girare il programma, vi accorgiate che uno degli sprite non è venuto bene o, comunque, volete modificarlo. Per far questo dovete andare a modificare la struttura dati corrispondente all'immagine del puntatore in questione. Se questa fosse stata definita nello stesso file del programma vero e proprio, avreste dovuto ricompilare *tutto* il programma, anche se aveste cambiato un solo bit! Vediamo invece come possiamo fare con due file. Innanzi tutto dovete aver conservato i file *oggetto non eseguibili*, quelli cioè che, nel caso del C terminano con **.o**, oltre naturalmente al codice sorgente. Non cancellate mai questi file, specialmente se il vostro programma è formato da più *sorgenti* (vedi nota 2). Nell'esempio riportato nelle figure già viste, il programma principale si trova nel file E10_1.c mentre le strutture immagine dei puntatori del mouse si trovano in E10_2.c. Inoltre, dopo la compilazione avremo anche E10_1.o ed E10_2.o. Se adesso modifichiamo solo il contenuto (non le dimensioni però) delle strutture immagine, basterà ricompilare il solo E10_2.c e lanciare di nuovo la fase di legame *[link]*. Un notevole risparmio di tempo, no? Se poi avessimo voluto anche modificare le dimensioni od il centraggio dei nostri puntatori, avremmo dovuto spostare in E10_2.c la matrice **mpdata[NW][4]** presente in E10_1.c. Naturalmente, in questo caso, avremmo dovuto ricompilare anche E10_1.c, ma *solamente* la prima volta. Naturalmente questo ha lo svantaggio di forzare il caricamento di questa struttura nella memoria CHIP, il che non sarebbe necessario, ma in fondo si tratta solo di un centinaio di byte più o meno. Questo può comunque essere evitato se si fa uso della nuova dichiarativa *chip* del Lattice C 5.0 per le sole strutture immagine.

Per esercizio, provate a riscrivere E10_1.c ed E10_2.c in modo da renderli abbastanza flessibili da poter ricompilare solo il secondo anche nel caso si voglia aumentare il numero delle finestre. Attenzione però, c'è questa volta in gioco una costante predefinita... **NW**.

Manutenzione ed aggiornamento di un programma

Supponiamo ora che abbiate scritto un programma tipo quello dell'esercizio (E10) e lo mettiate su di un *Bullettin Boards (BBS)*, come ad esempio *MC-Link*. Come si fa solitamente in questi casi, il pacchetto *[package]* contiene il programma eseguibile, tutti i file sorgente (*header* inclusi), un file di documentazione ed il classico file **Read.Me** che contiene il vostro nome, cognome, indirizzo elettronico e/o postale e qualche importante avvertimento da leggere *prima* di far girare il programma (vedi nota 3). Un utente che come voi è appassionato di C, decide di modificare il sorgente per sistemare, ad esempio, un baco nel programma od aggiungere una nuova funzione. Ovviamente, dato che dopo aver modificato il sorgente, bisogna ricostruire l'eseguibile, il tizio (o la tizia) in questione prova a ricompilare il programma, esegue tutti i passi necessari e, non avendo ricevuto alcun messaggio di errore, lancia il nuovo programma. Risultato: l'Amiga va in GURU. Dopo ore passate a scervellarsi sul perché e sul percome, decide di ricompilare il vecchio sorgente per riottenere il programma come era prima, avendo utilizzato per il nuovo programma lo stesso nome del vecchio ed avendo quindi perso l'eseguibile originale. Compilazione, *link*, esecuzione... **GURU!** E questo senza aver modificato una sola linea di codice! Sempre più sconsolato il tizio in questione prende il telefono, vi chiama, e si mette d'accordo con voi per vedervi. Vi porta il nuovo programma, voi vi sedete di fronte al vostro Amiga e provate a ricompilare il programma. Avete appena finito di compilare il primo file e siete passati al secondo che il tizio vi guarda e dice, un po' scocciato: «Poteva scriverlo da qualche parte che si deve usare -ad per compilare il secondo file, no?». Morale della favola: spesso non basta il solo codice per ottenere un programma funzionante e privo di bachi. Nella maggior parte dei casi è anche necessario sapere *come* è stato prodotto quel programma. In particolare bisogna conoscere le opzioni di compilazione, di legame, le librerie usate, e molte altre cose. Una soluzione potrebbe essere quella di associare ad ogni programma una dettagliata descrizione di tutto ciò che si deve fare per ricostruirlo a partire dal codice. Ma a parte la perdita di tempo, chi volesse ricompilare il vostro codice dovrebbe leggersi le vostre istruzioni. E se fosse americano e voi avete scritto il tutto in italiano? O viceversa, come capita spesso? E poi, non tutti sono molto chiari quando si tratta di spiegare qualcosa, indipendentemente dalle loro capacità od intelligenza. È quindi necessario trovare un metodo *standard* per descrivere il processo di produzione di un programma a partire dal sorgente. Ancora meglio se tale descrizione può essere usata per automatizzare il processo stesso, rendendo più semplice la manutenzione e le operazioni di aggiornamento del programma. Un file di que-

Note

1. Il codice non ha mai la necessità di essere caricato nella memoria di tipo CHIP, i dati sì.
2. In seguito useremo la seguente terminologia:
- Un file sorgente *[source file]* è quel file che contiene codice o dati compilabili. Da non confondere con i *file di inclusione* ed i *file dati*. In C termina sempre con **.c**.
- Un file di inclusione *[include* o *header file]* è quel file che contiene codice non direttamente compilato, ma che viene appunto incluso nel sorgente in fase di precompilazione *[preprocessor]*, prima cioè della compilazione vera e propria. In C termina sempre con **.h**.
- Un file dati *[data file]* è quel file che viene letto da programma durante l'esecuzione. Non esiste una convenzione particolare per questo tipo di file. A volte si usano le estensioni **.dat** o **.fil**.
- Un file oggetto non eseguibile *[non executable module* od *object]* è il risultato *finale* della fase di compilazione. Come dice il nome, esso non può essere eseguito a meno di non farlo passare per un'altra fase, detta di legame od aggancio *[link]*. Questa fase risolve tutti i legami interni ed esterni al file, serve cioè ad assicurarsi che tutte le chiamate a funzioni interne od esterne possano essere effettuate senza problema. Il file oggetto è quindi il prodotto dell'ultima fase della compilazione vera e propria, qualora questa venga effettuata in più passi *[step]*. L'estensione C è **.o**.
- Un file intermedio *[quad file]* è prodotto da quei compilatori che operano in due passi. Nel caso del Lattice C l'estensione è **.q**.
- Un file o modulo eseguibile *[executable* o *module]* è il risultato finale del processo di compilazione e legame, è cioè il vostro programma pronto per essere eseguito. Nel caso dell'Amiga, non è prevista alcuna estensione. Il nome è in genere quello del sorgente senza l'estensione **.c**. Nel caso tuttavia di file multi-sorgente, il nome può essere qualunque, a piacere.

Esistono poi altri file (librerie di aggancio e di esecuzione, file di preparazione) con convenzioni ed utilizzi che variano da sistema operativo a sistema operativo. Ne vedremo alcuni in seguito.

3. Quando caricate da un BBS un programma, abbiate sempre l'avvertenza di leggere alcuni piccoli file di testo che sono *sempre* presenti (**Read.Me**, **Readme.first**, **Me.first**) nel pacchetto. Spesso essi contengono informazioni importanti da sapere prima di far girare il programma. Non farlo può anche causarvi la visita del buon vecchio GURU! In una delle prossime puntate vedremo come si costruisce un pacchetto.

```
/*
** Prototipi delle funzioni grafiche: Versione 1.3
*/

/*
** *** BLITTER & BITMAP ***
*/
 long  BltBitMap              (struct BitMap *, long, long, struct BitMap *,
                               long, long, long, long, long, long, char *);
 void  BltBitMapRastPort      (struct BitMap *, long, long, struct RastPort *,
                               long, long, long, long, long);
 void  BltClear               (char *, long, long);
 void  BltMaskBitMapRastPort  (struct BitMap *, long, long, struct RastPort *,
                               long, long, long, long, long, APTR);
 void  BltPattern             (struct RastPort *, struct RastPort *,
                               long, long, long, long, long);
 void  BltTemplate            (char *, long, long, struct RastPort *,
                               long, long, long, long);
 void  ClipBlit               (struct RastPort *, long, long, struct RastPort *,
                               long, long, long, long, long);
 void  CopySBitMap            (struct Layer *);
 void  DisownBlitter          (void);
 void  InitBitMap             (struct BitMap *, long, long, long);
 void  OwnBlitter             (void);
 void  QBlit                  (struct BltNode *);
 void  QBSBlit                (struct BltNode *);
 void  SyncSBitMap            (struct Layer *);
 void  WaitBlit               (void);
/*
** *** COPPER ***
*/
 void  CBump             (struct UCopList *);
 void  CMove             (struct UCopList *, long, long);
 void  CWait             (struct UCopList *, long, long);
 void  FreeCopList       (struct CopList *);
 void  FreeCprList       (struct cprlist *);
 void  FreeVPortCopLists (struct ViewPort *);
 void  MrgCop            (struct View *);
 void  UCopperListInit   (struct UCopList *, long);
```

```
/*
** *** RAST & VIEW PORTS ***
*/
 PLANEPTR         AllocRaster  (long, long);
 void             Flood        (struct RastPort *, long, long, long);
 void             FreeRaster   (PLANEPTR, long, long);
 void             InitRastPort (struct RastPort *);
 struct TmpRas *  InitTmpRas   (struct TmpRas *, char *, long);
 void             InitView     (struct View *);
 void             InitVPort    (struct ViewPort *);
 void             LoadView     (struct View *);
 void             MakeVPort    (struct View *, struct ViewPort *);
 void             ScrollRaster (struct RastPort *, long, long, long,
                                long, long, long);
 void             ScrollVPort  (struct ViewPort *);
 void             SetRast      (struct RastPort *, long);
/*
** *** LAYERS & ALTRE ***
*/
 long  AttemptLockLayerRom (struct Layer *);
 void  LockLayerRom        (struct Layer *);
 void  UnlockLayerRom      (struct Layer *);
 long  VBeamPos            (void);
 void  WaitTOF             (void);
 void  WaitBOVP            (struct ViewPort *);
```

Figura 7
Le funzioni grafiche dell'Amiga: Avanzate.

sto tipo si chiama **makefile**. Vedremo nella prossima puntata come si usa e quali vantaggi comporta. In figura 3 è mostrato quello relativo al programma E10. Lo commenteremo la prossima volta.

La libreria grafica dell'Amiga

La libreria grafica dell'Amiga (**graphics.library**) è una delle più interessanti. Essa non solo ci permette di disegnare all'interno di una finestra o nel *raster* relativo ad uno schermo, ma ci mette a disposizione tutta una serie di funzioni per utilizzare gli speciali chip Amiga dedicati prevalentemente alla grafica.

Le funzioni della libreria grafica [1.3] sono listate, sotto forma di prototipi, da figura 4 a figura 7.

Come si può vedere, la maggior parte (soprattutto quelle specifiche per disegnare) fa riferimento ad una struttura **RastPort**. Tale struttura infatti, come già detto in una delle puntate precedenti, contiene tutte le informazioni relative all'area di memoria che corrisponde alla finestra od allo schermo in cui vogliamo operare. Nel caso di una finestra, in particolare, il puntatore a tale struttura si ottiene da quella della struttura **Window** ottenuto all'apertura della finestra, nel seguente modo:

```
struct Window *w;
struct RastPort *rp;
...
rp = w->RPort;
...
```

```
#define GNAME "graphics.library"
#define GVERS 0          /* o la versione che REALMENTE vi serve */
struct GfxBase *GfxBase; /* Non cambiate MAI il nome del puntatore BASE! */
...
GfxBase = (struct GfxBase *)OpenLibrary(GNAME,GVERS);
if (GfxBase == NULL) Error(NOGRAPHICSLIBRARY); /* o quello che volete voi */
...
```

Figura 8

Naturalmente, prima di usare una qualunque funzione grafica, è necessario aprire la libreria grafica nel modo indicato in figura 8:

La prossima volta incominceremo ad analizzare in dettaglio le singole funzioni.

Conclusione

Abbiamo iniziato con questa puntata due argomenti molto importanti. Uno è quello relativo alle tecniche di compilazione e manutenzione dei file sorgente, l'altro è quello relativo alla libreria grafica dell'Amiga. Questa struttura ad argomenti *paralleli* caratterizzerà anche le prossime puntate. Questo perché ora che siete già in grado di scrivere programmi anche di una certa complessità, è importante che incominciate ad acquisire tutta una serie di tecniche di sviluppo che sono fondamentali per la realizzazione di un programma. Per analogia, scrivere un programma è un po' come scattare una fotografia. Il fotografo esperto non si limita allo scatto. Egli sa bene che in realtà il processo di sviluppo e di stampa è altrettanto, se non più importante, di quello che porta al semplice scattare la foto (preparazione dell'ambiente, delle luci se in interno, inquadratura, scatto). Il processo di creazione dell'eseguibile a partire dal sorgente corrisponde appunto alle attività che si svolgono in una camera oscura. Ma di questo avremo ancora modo di parlare. Alla prossima puntata.

MC

LE PERIFERICHE

DFI HANDY SCANNER

400 dpi - 32 mezzi toni - 105 mm - compatibile con Windows/ Gem/Halo/PCX in tutti i modi grafici IBM (disponibile anche software OCR)
a sole L. 450.000

STAMPANTI

Panasonic tutti i modelli inclusa la nuova KX-P1124 (200 cps/24 aghi)	telefonare

MONITOR

monocromatico dual 14" flat	L. 220.000
monocromatico VGA	L. 285.000
monocromatico multisync	L. 450.000
colori Philips 8802 (Amiga/ST)	L. 340.000
colori Philips 8833 (CGA)	L. 450.000
colori Philips 9043 (EGA)	L. 550.000
colori multisync CTX (nuovo)	L. 850.000
schermi antiriflesso	da L. 18.500

SUPPORTI DI MEMORIZZAZIONE

chip RAM	telefonare
dischi 3,5" Precision	L. 2.000
dischi 3,5" Precision HD	L. 5.500
dischi 5,25" Precision	L. 900
dischi 5,25" Precision HD	L. 2.300
drive 5,25" 1.2 Mb	L. 175.000
drive 3,5" 720 Kb	L. 180.000
drive 3,5" 1.44 Mb	L. 210.000
hard disk Seagate 20 Mb	L. 380.000
hard disk Seagate 32 Mb	L. 550.000
hard disk Seagate 40 Mb	L. 660.000
hard disk Seagate 80 Mb	telefonare
hardcard 20 Mb Tandon	L. 590.000
data pac 20/40 Mb	telefonare

ADD-ON

coprocessore Intel 8087-5	L. 210.000
coprocessore Intel 80287-8	L. 480.000
coprocessore Intel 80287-10	L. 550.000
FAX Murata manuale italiano	L. 1.400.000
modem Smartlink esterno	da L. 230.000
modem Smartlink interno	da L. 195.000
mouse Z-nix 250 dpi	L. 85.000
tastiera 102 tasti Cherry	L. 130.000
tavoletta grafica Genius 12"	L. 750.000

SCHEDE

scheda copy card 4.5	L. 150.000
scheda eprom burner 4 pos.	L. 240.000
schede espansione memoria	telefonare
schede multifunzione XT/AT	telefonare
scheda Super EGA 640x480	L. 290.000
scheda Super EGA 1024x480	L. 330.000
scheda VGA 800x600	L. 450.000

130 tipi diversi di schede, accessori & add-on disponibili: richiedere il catalogo o telefonare!

I PERSONAL

tutti i tipi di cabinet:
desktop standard
desktop baby
desktop minibaby
trasportabile LCD
tower drive vert.
tower drive oriz.
minitower

PC XT 8088-10 desktop
512 Kb RAM espandibile 1 Mb
drive 360 Kb + hard disk 20 Mb
Hercules - tastiera 102 tasti
monitor 14" dual flat screen
Lire 1.600.000

PC AT 80286-12 desktop
512 Kb RAM espandibile 4 Mb
drive 1.2 Mb + hard disk 20 Mb
Hercules - tastiera 102 tasti
monitor 14" dual flat screen
Lire 2.100.000

PC 80386-20 tower (foto)
1 Mb RAM espandibile 8/16 Mb
drive 1.2 Mb + hard disk 32 Mb
Hercules - tastiera 102 tasti
monitor 14" dual flat screen
Lire 4.400.000

MODELLI BASE
assembliamo configurazioni su richiesta

LE NOVITÀ FANTASOFT

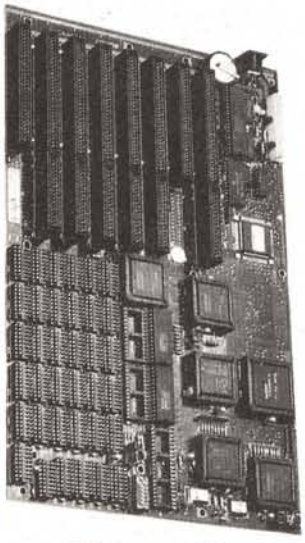
MOTHERBOARD 80386-20 MHz

- CPU 80386-20 ΣΣ
- zoccoli per 80287/80387
- memory interleaved
- shadow RAM
- espandibile fino a 16 Mb 32 bit RAM (41256/411000)
- Landmark 26.7 MHz

Lire 1.950.000

tutte le nostre motherboard e schede di espansione accettano anche i nuovi
CHIP RAM 1 Mbit-100
risparmio del 40%
sui vecchi chip 41256

SCHEDA ESPANSIONE 2 MB
EMS 4.0 a sole Lire 990.000



***SUNTAC* 80286**

- Landmark 16.1 MHz
- 6/8/12/16 MHz 0 wait
- 512/640/ 1024/4096 K onboard
- EMS 4.0
- Award bios con setup
- installabile in qualsiasi case

a sole Lire 450.000

FANTASOFT

COMPUTER HOUSE

Via O.Targioni Tozzetti 7/b - 57126 LIVORNO

TEL: 0586/805.200 - FAX: 0586/803.094

PREZZI IVA E TRASPORTO ESCLUSI - RICHIEDETE CATALOGO - SCONTI A RIVENDITORI