Programmare in C su Amiga

di Dario de Judicibus

Fare un programma non vuol dire solo scrivere del codice, ma anche eseguire tutta una serie di processi che vanno dalla progettazione alla verifica del risultato finale. Fra questi, c'è la definizione di alcune procedure che permettono di automatizzare tutte le altre, semplificando la generazione e la manutenzione del programma stesso

Lo scorso mese abbiamo incominciato a dividere queste puntate in due parti: una relativa alla programmazione vera e propria, l'altra, nuova, relativa alla gestione dell'ambiente di sviluppo nel suo complesso. Per ragioni di spazio e per evitare di spezzare un discorso che trova nella continuità la sua chiave di lettura, questa volta ci occuperemo solo del secondo argomento, introducendo uno dei programmi di supporto alla programmazione più importanti dopo il compilatore ed il *linker*. Ci rifaremo la prossima puntata dedicandola completamente alle funzioni grafiche dell'Amiga.

Introduzione

In questa puntata incominceremo a trattare un argomento estremamente importante per chi, lasciatosi alle spalle il programmino messo su in poche ore e poi dimenticato, intende affrontare qualche progetto un po' più impegnativo col desiderio di portarlo avanti nel tempo raffinandolo ed aggiungedovi di volta in volta sempre più funzioni e possibilità. In pratica questo vuol dire partire dal presupposto che non ci si fermerà al primo rilascio *[release]* od addirittura alla prima versione *[version]* del programma (vedi nota 1).

Lo sviluppo di un programma

Come abbiamo detto nella scorsa puntata, la programmazione vera e propria è solo una delle tante cose che un programmatore esperto deve gestire. Per quanto possa sembrare strano non è forse neanche la più importante. In realtà, per sviluppare un programma è necessario passare attraverso una serie di fasi ognuna delle quali ha regole ben precise, che comunque variano a seconda dell'ambiente in cui si programma e del tipo di programma che si sta sviluppando. Dato che questi articoli non intendono essere un trattato di programmazione, ma vogliono soltanto fornire una serie di utili indicazioni per chi, avendo a disposizione un Amiga ed avendo già una certa infarinatura di C, desidera imparare come sfruttare al meglio sia la macchina che il linguaggio in questione, non affronteremo l'argomento in modo sistematico e formale, ma cercheremo di introdurre il lettore, volta per volta e partendo sempre da esempi pratici legati in qualche modo all'Amiga, ad una serie di tecniche che le/gli permetteranno in breve tempo di costruirsi un ambiente di sviluppo facile da usare ma potente ed efficace.

L'ambiente di sviluppo

Un ambiente di sviluppo è definito dall'insieme delle tecniche, dei programmi di supporto e della struttura a file che si utilizzano quando si sviluppa un programma. Ad esempio, se avete un disco rigido e avete costruito una serie di

Figura 1 - MAKEFILE.

```
#
# file: E10.lmk
#
# Soluzione dell'esercizio proposto nella decima puntata - MAKEFILE
#
# 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 -b0 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 2
Sintassi di un MAKEFILE.

```
# ------------------------------------------------------------
#
# Questo è un commento. Viene ignorato da LMK
#

VARIABILE = valore_della_variabile

ascendente: lista dei suoi discendenti

  processo_di_generazione_dell'ascendente_dai_discendenti
```

directory e sotto-directory per contenere il compilatore, il Linkage Editor, le librerie, gli *header* e così via, potreste essere tentati di caricarvi anche la directory che dovrà contenere i vostri programmi in fase di sviluppo. In effetti, sia gli uni che l'altra fanno parte del vostro ambiente di sviluppo. Ebbene, questo dovrebbe essere evitato. Quando compilate un programma, state compiendo un certo numero di operazioni di lettura e scrittura che servono a caricare il codice sorgente, salvare i file intermedi, leggere le librerie, scrivere il risultante file oggetto e via dicendo. Ora, più operazioni di I/O effettuate, specialmente in scrittura, maggiore è il rischio di rovinare un settore od una traccia del vostro disco *[stratch]*. Si tratta di una cosa normalissima anche se, per fortuna, non frequente. In molti casi basta utilizzare un programmino come **DiskDoctor** per risolvere il problema, perdendo al massimo uno o due file, altre volte la cosa è più seria. In formati disco quali quello del PC IBM, ad esempio, se viene rovinata la FAT, può risultare complicato recuperare i file a cui questa puntava. Nel caso dell'Amiga, che ha una gestione dei file differente, può capitare che una directory punti se stessa in un *loop* infinito. Anche questi casi sono risolvibili, ma le tecniche per correggere tali situazioni non sono alla portata di tutti. A volte, l'utente meno esperto non ha altra alternativa che riformattare il disco rigido e ricostruirlo utilizzando l'ultima copia di sicurezza *[backup]* effettuata.

Vediamo allora come impostare l'ambiente di sviluppo. Ci sono due soluzioni abbastanza sicure. La prima consiste nel mantenere sul disco rigido il compilatore e le altre utilità di sviluppo *[utilities]*, le librerie, i file di inclusione e comunque tutto ciò che viene solamente letto durante la preparazione dell'eseguibile. Inoltre la directory riservata ai file intermedi *[quad]* o temporanei va assegnata alla RAM: o comunque ad una directory in RAM: che dovrà essere stata creata precedentemente. Viceversa le directory che devono contenere i programmi in fase di sviluppo, saranno create su uno o più dischetti rimovibili. Questi, oltre a garantire una maggiore durata del disco rigido, sono anche più facilmente recuperabili in caso di danneggiamento. In genere si usano due serie di dischetti: la prima serie contiene una directory per ogni programma che si sta sviluppando e sono detti «dischetti di lavoro» *[work diskette]*; la seconda serie contiene una directory per ogni programma consolidato, che, cioè, è stato terminato e provato. La separazione in directory per quello che riguarda i dischetti di lavoro serve a fornire un ulteriore elemento di sicurezza contro eventuali problemi di scrittura/lettura da disco; quella dei dischetti dei programmi finiti, serve a garantire un maggior ordine per la classificazione e la manutenzione dei programmi, come vedremo in seguito.

La seconda soluzione è analoga alla precedente, ed è generalmente utilizzata dai programmatori professionisti che desiderano sfruttare al massimo la maggiore velocità in I/O fornita da un disco rigido. In pratica, mentre i programmi terminati vengono comunque archiviati su dischetti, i dischetti di lavoro sono sostituiti da una seconda partizione sul disco rigido od addirittura da un secondo disco rigido. Tale soluzione è comunque un po' più costosa, anche perché spesso i dischi rigidi utilizzati sono a grande capacità (centinaia di MByte) e tempo di accesso particolarmente rapido. Se la partizione (od il disco) di lavoro dovesse venire danneggiata da una qualche operazione di scrittura, le operazioni di recupero saranno effettuate solo su quest'ultima, senza interessare la partizione (od il disco) che contiene il sistema operativo ed il compilatore C.

Per quello che mi riguarda io utilizzo la prima soluzione. Se però avete un disco rigido da 30M o 40M, provate a dividerlo in due partizioni, una per i comandi del sistema operativo, il compilatore, le librerie e così via, l'altro per i sorgenti, i dati, ed in generale tutti quei file che rappresentano il prodotto del vostro lavoro (IFF, musica, testi). Una volta che avrete finito di elaborarli potrete salvarli su un dischetto archivio (vedi nota 2).

Le fasi dello sviluppo

Vediamo ora in prima approssimazione quali sono le fasi tipiche dello sviluppo di un programma su di un Personal Computer. Ci limiteremo a considerare una metodologia semplificata opportunamente per chi programma per diletto piuttosto che per professione. Possiamo dividere il tutto in tre fasi, ognuna formata da due o più sottofasi:

1. Disegno

Progettazione

È la definizione delle caratteristiche funzionali del programma e della struttura dello stesso. Ad esempio, se si vuole scrivere un programma in grado di leggere un file IFF di tipo ILBM e visualizzare l'immagine risultante sullo schermo, la definizione delle caratteristiche funzionali determina quali tipi di file ILBM il programma dovrà essere in grado di leggere (tutti, bassa risoluzione solamente, HAM) oppure se deve essere prevista anche la possibilità di stampare l'immagine risultante o meno. Viceversa definire la struttura del programma significa decidere se esso dovrà essere sviluppato in più moduli od in un modulo singolo, se utilizzerà delle librerie in linea od in fase di compilazione, e così via.

Pseudocodifica

È una forma di codifica del programma «a grandi linee», utilizzando un linguaggio formale più vicino al linguaggio naturale che a quello del compilatore. Esistono molte tecniche alternative (diagrammi di flusso, BNF, ecc.), ma la pseudocodifica è forse la più semplice per chi parte da zero in questo campo.

2. Codifica

Codice Sorgente

È la codifica vera e propria del codice che dovrà poi essere compilato. Questa fase comprende anche la scrittura dei file di inclusione specifici per quel programma, esclusi cioè quelli forniti con il compilatore e quelli già scritti per programmi precedenti e validi anche per quello in questione.

Dati e file ausiliari

Sono tutti quei file che, pur essendo necessari al funzionamento del programma, non contribuiscono alla preparazione dell'eseguibile vero e proprio. Sono, ad esempio, tabelle, sequenze di record, profili (vedi nota 3) e simili. Possono venir creati utilizzando semplicemente un editore di testi *[text editor]* od essere il prodotto di un altro programma (ad es.: il reindirizzamento dell'*output* di un comando di sistema operativo). A questi si aggiungono quei file, detti ausiliari, che servono a creare e/o mantenere il programma (i cosiddetti **make** file), eventuali procedure per la compilazione (**script** file), e documentazione varia di sviluppo. Sono esclusi i file di documentazione per l'utente ed i vari **ReadMe** file. Quest'ultimi sono generalmente scritti alla fine, una volta che il programma è stato verificato.

3. Generazione

Compilazione

Il processo di compilazione serve a produrre uno o più oggetti non eseguibili che andranno poi legati fra di loro e con le librerie di sviluppo.

Linkage Edition

Questa attività è svolta appunto dal programma di legame *[linkage editor]*, che risolve tutte le referenze tra le varie chiamate interne ed esterne, come già detto nella scorsa puntata.

Verifica

È il passo più difficile e, per definizione, mai esaustivo. La verifica *[test]* serve ad eliminare il maggior numero di errori possibili, in tempi accettabili.

LMK

Nella scorsa puntata abbiamo mostrato un piccolo file, riportato per comodità in figura 1, che abbiamo detto essere servito ad ottenere il modulo eseguibile relativo all'esercizio proposto nella decima puntata, a partire dal codice sorgente. Stiamo quindi parlando delle prime due sottofasi del processo di *Generazione* del modulo eseguibile (vedi nota 4).

Tale file viene utilizzato da un programma chiamato *LMK* e fornito con il Lattice C 5.0 (vedi nota 5).

LMK trova il suo analogo in ambiente UNIX nel programma di utilità **make**. Chi già conosce quest'ultimo, non avrà certo difficoltà ad utilizzare *LMK*. Per gli altri, cercheremo, in questa e nelle prossime puntate, di fornirvi una buona base per fruttare al meglio le caratteristiche di tale programma.

Innanzitutto bisogna dire che *LMK* non è un programma specifico per lo sviluppo di altri programmi. In realtà, si tratta di un *gestore di progetti*, di un prodotto cioè che, controllando il processo che permette di sviluppare un progetto a partire da un certo numero di file, evita allo sviluppatore di eseguire operazioni superflue e di ricordare procedure complesse ed articolate.

Per semplicità faremo riferimento al caso specifico dell'esercizio risolto la volta scorsa, ma andrebbe bene anche il caso della produzione di un rapporto basato su un certo numero di file di una base di dati, oppure la stampa di una lettera e della relativa busta a partire da uno scheletro generalizzato ed un archivio contenente una serie di indirizzi.

Analizziamo il processo che porta alla generazione di **E18**. Vi consigliamo di procedere tenendo sotto mano l'articolo presentato nel numero di aprile di MC microcomputer. Per prima cosa dobbiamo compilare i due file sorgente, in modo da ottenere i rispettivi file oggetto. Inoltre le opzioni di compilazione per i due file non sono le stesse, in quanto i dati relativi a quattro sprite vanno caricati nella memoria CHIP (da cui l'opzione **-ad**) mentre il codice contenuto nel sorgente principale deve essere in grado di accedere a tali dati dalla memoria FAST (da cui l'opzione **-b0**). Possiamo allora dire che **E10_1.o** si ottiene da **E10_1.c** tramite il processo **1c -b0 E10_1.c** mentre **E10_2.o** si ottiene da **E10_2.c** tramite il processo **1c -ad E10_2.c.** I due file sorgente si dicono in tal caso «dipendenti» dei rispettivi file oggetto *[dependent]* (vedi nota 6). Attenzione a non confondervi, però: per «dipendente» non si intende tanto «che dipende da», quanto il rapporto «padre/figlio» di una struttura gerarchica. *LMK* infatti, si basa su una visione gerarchica dei legami tra i file *prodotto* e quelli *dipendenti* (vedi figura 4). Quindi, da un punto di vista funzionale, sono i file oggetto a dipendere dai sorgenti, da quello gerarchico è vero il viceversa. Analogamente l'eseguibile ha come dipendenti proprio i file oggetto, da cui peraltro dipende in quanto ottenuto da questi in seguito al processo di *link*. Per evitare confusione, useremo il termine italiano *discedente* al posto della traduzione letterale dell'inglese *dependent*, e *ascendente* per indicare il livello superiore.

Chiameremo inoltre *radice* il risultato finale dell'intero processo e *discendenti terminali* le «foglie» dell'albero che rappresenta i file in gioco e le relazioni fra di loro.

Tornando al nostro esempio, i file oggetto sono a loro volta discendenti dell'eseguibile, che si ottiene appunto da questi tramite il processo di Linkage Edition.

La sintassi del nostro **makefile** è quindi abbastanza semplice (vedi figura 2): un ascendente è descritto dal suo nome seguito subito dopo dal segno di interpunzione *due punti* (:) e, separata

```
# -------------------------------------------------------------------
# Supponiamo di aver modificato solo E10_2.c e di lanciare di nuovo LMK
# -------------------------------------------------------------------


#
#  Questo passo va effettuato in quanto E10_2.o è cambiato
#
E10: E10_1.o E10_2.o
  LC:blink FROM LIBo+E10_1.o+E10_2.o TO E10 LIB $(LIBS) $(LOPT)

#
#  Questo passo  non va effettuato in quanto E10_1.c non è cambiato
#
E10_1.o: E10_1.c
  LC:lc -b0 E10_1.c

#
#  Questo passo va effettuato in quanto E10_2.c è cambiato
#
E10_2.o: E10_2.c
  LC:lc -ad E10_2.c

# -------------------------------------------------------------------
```

Figura 3 - Esempio di riutilizzo di un MAKEFILE.

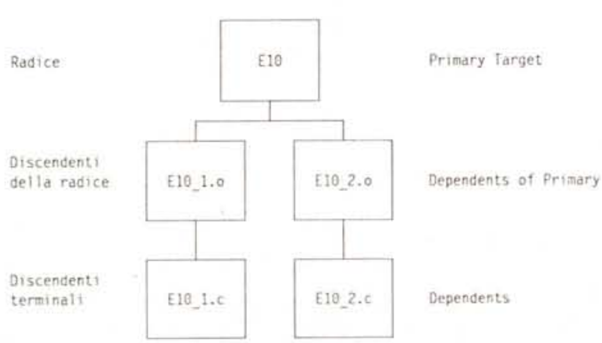


Figura 4 - La struttura gerarchica per E10.

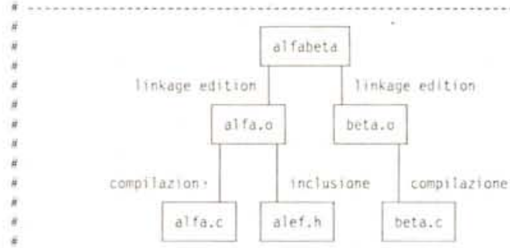
```
# -------------------------------------------------------------------

#
#  I due oggetti alfa.o e beta.o sono due discendenti di alfabeta.
#  Viceversa c.o è un discendente implicito, in quanto lo è potenzialmente,
#  ma, non essendo in lista, viene ignorato da LMK.
#
alfabeta: alfa.o beta.o
  LC:blink FROM LIBo+alfa.o+beta.o TO E10 LIB $(LIBS) $(LOPT)

#
#  Il sorgente alfa.c è un discendente di alfa.o.
#  Il file di inclusione alef.h, invece, pur non essendo presente tra i
#  parametri del comando che definisce il processo di compilazione, è
#  stato esplicitato in modo da far rieseguire questo passo nel caso esso
#  venga modificato.
#
alfa.o: alfa.c  alef.h
  LC:lc -$(AOPTS) alfa.c

#
#  L'oggetto beta.o ha come suo unico discendente il sorgente beta.c
#
beta.o: beta.c
  LC:lc -$(BOPTS) beta.c


# -------------------------------------------------------------------
#
#   Albero che rappresenta il processo appena descritto
#
# -------------------------------------------------------------------
```




```
# -------------------------------------------------------------------
```

Figura 5 - Discendenti impliciti ed espliciti.

da almeno uno spazio da quest'ultimo, dalla lista dei suoi discendenti, il tutto sulla stessa linea. Nella linea seguente, indentato di almeno uno spazio bianco, c'è il comando che corrisponde al processo che genererà l'ascendente a partire dai suoi discendenti. A loro volta questi, come abbiamo visto, possono essere descritti da un blocco di linee analogo. Ogni blocco deve avere il nome dell'ascendente allineato alla prima colonna. Nel caso che certe stringhe di caratteri vengano usate spesso, o per semplificare le definizioni nel caso fossero troppo lunghe, è possibile definire delle variabili tramite l'assegnazione:

VARIABILE = valore

da usare in seguito ponendo il nome tra parentesi e facendole precedere dal simbolo del dollaro, nel modo seguente:

... $(VARIABILE)...

Queste variabili vengono anche chiamate *Macro* e possono essere paragonate a quelle definite in C tramite la direttiva #**define**. Come quest'ultime, anche le macro di *LMK* devono precedere le linee nelle quali vengono usate, altrimenti saranno espanse in una *stringa nulla*.

In ogni caso, se una linea è più lunga di 80 caratteri, si può andare a capo a condizione di usare la barra diagonale inversa (\) come carattere di continuazione.

Quindi, riassumendo, per automatizzare l'esecuzione di un processo tramite *LMK*, è necessario:
1. definire gli oggetti da cui si parte, quello di arrivo, e tutti quelli intermedi;
2. definire i processi che permettono di andare da un livello all'altro, cioè i vari passi *[step]*;
3. scrivere un file che descrive il tutto usando la sintassi appena descritta;
4. invocare *LMK*.

Vediamo ora che succede quando *LMK* viene lanciato.

Innanzi tutto esso cerca nella directory corrente un file chiamato **lmkfile, lmkfile.lmk** o **lmkfile.lmk,** a meno che lo sviluppatore non abbia usato l'opzione **-f** nel modo seguente:

1> lmk -f filename.lmk
2> lmk -f filename

Nel primo caso *LMK* utilizza come file di descrizione del processo **filename.lmk**, mentre nel secondo caso esso cerca prima un file chiamato **filename**, e solo nel caso questo non esista, **filename.lmk**. Attenzione quindi a non usare il secondo formato se nella directory corrente esiste un altro file con quel nome, magari proprio la radice! È un errore molto comune. Notate inoltre lo spazio bianco che separa l'opzione dal nome del file. Differentemente dalla sintassi usata nel compilatore per certe opzioni, quello spazio è obbligatorio.

Trovato il file, questo viene letto e vengono identificati tutti gli ascendenti, quei file, cioè, che sono il risultato di un qualche processo. A questo punto i vari processi vengono effettuati a partire dai discendenti terminali, passando per tutti i passi intermedi, fino alla radice.

Tuttavia, se fosse tutto qui, l'utilità di *LMK* non sarebbe maggiore di quella di un file *script* (vedi nota 7) opportunamente preparato. Una delle caratteristiche più importanti di *LMK* è quella di controllare la data e l'ora associata ad ogni discendente, e di confrontarla con i rispettivi ascendenti. Se l'ascendente è più recente di tutti i suoi discendenti, allora il processo associato viene saltato, in quanto non necessario.

Facciamo un esempio. Supponiamo di aver già creato l'eseguibile **E10** corrispondente all'esercizio della decima puntata. Ci siamo accorti tuttavia che uno degli sprite non è venuto come avremmo voluto. Fate riferimento alla figura 3.

Innanzi tutto modifichiamo il file sorgente **E10_2.c** che contiene i dati relativi allo sprite in questione. A questo punto lanciamo di nuovo *LMK* specificando il file **E10.lmk** (compresa l'estensione).

Dato che il file **E10_1.c**, unico discendente di **E10_1.o** non è cambiato, non c'è ragione di ricompilarlo, e difatti, avendo ancora il sorgente una data/ora più vecchia di quella del file oggetto ottenuto compilandolo, *LMK* salta questo passo, con ovvio risparmio di tempo. Viceversa, l'altro file sorgente avrà una data più recente del rispettivo oggetto, essendo stato appena modificato. *LMK* effettuerà quindi il processo di compilazione per ottenere un nuovo oggetto. A questo punto rimane il passo finale. Dato che **E10_2.o** è stato appena ottenuto compilando il sorgente modificato, esso ha quindi una data/ora più recente dell'eseguibile, e questo è sufficiente a far sì che *LMK* decida di ripetere anche il passo di Linkage Edition, anche se l'altro oggetto è più vecchio della radice.

Risultato finale: un nuovo eseguibile ottenuto con solo due passi invece di tre, il tutto automaticamente.

Naturalmente vi sarete subito resi conto di una cosa: il tutto non può funzionare correttamente se la data e l'ora del sistema non vengono sempre regolarmente aggiornate. Per chi possiede un Amiga 2000 (od un Amiga 1000 con scheda aggiuntiva con orologio tampone) non c'è alcun problema, a condizione di impostare la data e l'ora corrette almeno la prima volta. Chi invece ha un Amiga 1000 base dovrà ricordarsi di farlo almeno ogni qual volta deciderà di lavorare nel suo ambiente di sviluppo, anche solo per editare i sorgenti!

In ogni caso, i vantaggi di *LMK* non si fermano qui, ma di questo ce ne occuperemo tra un paio di puntate al massimo.

LMK è particolarmente utile quando una, od entrambe le seguenti condizioni vengono a verificarsi:
1. il progetto in questione è basato su molti file e/o su di un buon numero di processi,
2. esso è portato avanti da più di uno sviluppatore allo stesso tempo.

In quest'ultimo caso *LMK* è uno strumento prezioso per garantire la consistenza e l'integrità degli elementi che compongono il progetto.

Discendenti impliciti ed espliciti

Certamente avrete notato che per ottenere l'eseguibile del nostro esempio, è necessario agganciare fra di loro non solo i due file oggetto **E10_n.o,** ma anche il codice di partenza *[startup]* **c.o** — che serve ad assegnare eventuali parametri passati al momento di eseguire il programma, al vettore **char *argv[]**, ad impostare i vari File Handle di I/O (**stdin, stdout, stderr,...**), e molte altre cose — e le librerie di compilazione, quelle cioè che in genere si trovano nella directory **LIB:**.

Perché allora questi oggetti non sono presenti nella lista dei discendenti?

Non solo. Supponiamo che i nostri sorgenti prevedano l'inclusione di alcuni file tramite la direttiva #**include**. Anche se tali file non compaiono esplicitamente nella definizione del processo che genera l'oggetto, se io li modifico è necessario ricompilare comunque il sorgente che li include. Come faccio a dirlo ad *LMK*?

Rispondiamo prima a quest'ultima domanda: «Basta semplicemente aggiungere anche tale file alla lista dei discendenti». Difatti, non è necessario che un discendente appaia nella definizione di un processo perché esso sia tale. È sufficiente che esso sia in qualche modo legato alla generazione di un ascendente, e che quindi una sua eventuale modifica possa influenzare il processo di generazione, perché esso debba essere aggiunto tra i discendenti del file in questione (vedi figura 5). Tale operazione si dice *rendere esplicito un discendente*.

Viceversa, file come **c.o** o le librerie di compilazione, che vengono fornite con il compilatore, in linea di massima cambiano solo se si cambia il compilatore, e quindi non è necessario esplicitarle. Tenete presente però che, in questo

caso, l'installazione di un nuovo compilatore non sarebbe sufficiente a far rieseguire l'ultimo passo, quello cioè di Linkage Edition. A vostra scelta, quindi! Quei file che sono potenzialmente dei discendenti, ma non sono stati esplicitati, si dicono *discendenti impliciti.*

File multipli

Per terminare rispondiamo ad una domanda che sicuramente qualche lettore a questo punto si sarà posto (no, non è «Ma chi me lo ha fatto fare di imparare il C?»). Il quesito che sorge spontaneo è piuttosto: «Perché devo dividere il mio sorgente in più file?».

Benché il motivo sia chiaro per quello che riguarda l'esempio utilizzato in precedenza, si potrebbe obbiettare che non capita poi così spesso di dover utilizzare diverse opzioni di compilazione su diversi file appartenenti allo stesso programma.

Vi risponderemo con una serie di domande:

- Se dovete scrivere un libro utilizzando un processore di testi *[word processor]*, lo mettete tutto in un solo enorme file, o lo spezzate nei singoli capitoli?
- Se dovete fare una presentazione, caricate musica, testi ed immagini in un solo file IFF (si può fare, si può fare...) oppure in diversi file a seconda del tipo di oggetto?
- Se dovete costruire una tabella molto grande e complessa, la caricate tutta nel vostro foglio elettronico, o cercate di spezzarla in più tabelle legate fra di loro attraverso referenze incrociate *[cross-reference]*?

Se a tutte queste domande la vostra risposta è la prima, aspettatevi di passare lunghe ore di fronte al vostro Amiga a caricare file ed aspettare che i programmi corrispondenti abbiano finito di elaborarli, il tutto col rischio di rovinare tutto con una sola operazione sbagliata.

Se invece avete fornito la seconda risposta, allora avete capito uno dei trucchi più importanti dell'informatica (e della vita): *dividi et impera.*

Bando agli scherzi, la suddivisione di un progetto in più file porta a certi indiscutibili vantaggi.

- Caricare il file che serve in un certo momento è certamente più rapido che caricare tutto ogni volta; questo vale anche per i sorgenti da editare.
- Il programma stesso sarà più veloce ad elaborare un file piccolo, piuttosto che uno grande, anche perché spesso le informazioni contenute negli altri file in cui si è suddiviso il tutto, non servono realmente ad elaborare quello specifico su cui si sta operando.
- Se per errore si distrugge anche parzialmente il file su cui si sta lavorando, gli altri sono comunque salvi. Ricordate le due partizioni del disco fisso di cui avevamo parlato in precedenza? La filosofia è la stessa.
- Spesso strutturare un progetto in più parti permette di avere una visione più ordinata dello stesso.
- Infine, proprio grazie a programmi come *LMK*, la suddivisione in più file, permette spesso di diminuire il numero di passi da effettuare per riottenere la radice, nel caso si siano modificate solo alcune delle parti di cui è composto il progetto in questione.

Ovviamente, come sempre, per ogni regola ci sono delle eccezioni, e sicuramente vi capiterà il caso in cui un file è meglio di molti. Proporremo un buon metodo anche per questo caso, non vi preoccupate. Per il momento provate a scrivere dei **makefile** per i programmi che avete già sviluppato, anche se il sorgente è formato da un solo file. Se non avete *LMK*, potete provare qualcuno dei PD **make** disponibili su molti BBS. In particolare, sui dischetti di *Fred Fish* ci sono i seguenti programmi:

Disco # 2 **Make** - Autore: Landon Dyer
Disco #2 **Make2** - Autore: Marc Mengel
Disco #45 **Make** - Autore sconosciuto
Disco #69 **Make** - Autore: caret@fairlight.OZ (indirizzo rete)
Disco #74 **Makemake** - Autore: Tim McGrath

Quest'ultimo serve a costruire automaticamente un **makefile** a partire dal sorgente. Se qualcuno ha altre informazioni a riguardo, sarò ben lieto di pubblicarle, purché sia sempre indicata la fonte e dove è reperibile il programma.

Conclusione

Bene. Siamo arrivati alla conclusione anche stavolta. Parleremo ancora di *LMK* in futuro, tuttavia, per accontentare i patiti della grafica, la prossima puntata sarà interamente dedicata alle funzioni grafiche. Buon lavoro!

MC

Note

1. In genere un programma è identificato da due numeri, spesso separati da un punto decimale (ad es.: 1.5). Il primo numero è detto *versione*, il secondo *rilascio*. Ogni volta che un prodotto viene modificato, il numero corrispondente alla versione viene incrementato di uno se il programma è stato in gran parte riscritto e le caratteristiche base del prodotto sono state pesantemente influenzate dalle modifiche apportate. Se viceversa tali modifiche si riferiscono solo a qualche nuova opzione (o comando, nel caso di un sistema operativo) o comunque il cuore *[kernel]* del programma è praticamente rimasto lo stesso, è il numero corrispondente al rilascio ad aumentare. In pratica, mentre un possibile AmigaDOS 1.4 avrebbe comunque la stessa base funzionale dell'1.3, un incremento della versione (AmigaDOS 2.0) sarebbe indubbiamente indizio di profondi cambiamenti nel sistema operativo.

Tale convenzione è, ad esempio, seguita da buona parte dei prodotti commerciali e dai PD più importanti.

Molti programmi poi hanno anche un altro carattere per indicare piccole modifiche o la «riparazione» di alcuni errori di programmazione (o bachi) *[fix]*. A volte è un numero attaccato direttamente a quello del rilascio (ad es.: 4.01), altre volte una lettera in minuscolo (ad es.: 2.3a).

2. Tali dischetti non vanno confusi con quelli eventualmente utilizzati per le copie di *backup* del disco rigido. Un dischetto archivio è un dischetto che contiene il risultato di un vostro lavoro ad un certo livello di consolidamento: la prima versione di un programma, la terza versione di un documento, una immagine fatta con un programma grafico, un progetto disegnato con un CAD. In genere per sicurezza, si tengono le due ultime versioni di ognuno di questi.

3. Si chiamano profili *[profile]* quei file che contengono una serie di valori iniziali per variabili e *keyword* che possono essere utilizzate dal programma. Ad esempio, per un programma di comunicazione, un profilo può servire a definire la velocità della linea in *baud*, il numero di bit di parità, il tipo di protocollo da usare per il trasferimento dei file, e così via.

4. Dato che la trattazione della fase di *Disegno* e quella della sotto fase di *Verifica* vanno molto al di là degli scopi di questa serie di articoli, mentre la fase di *Codifica* viene già trattata nella parte dell'articolo che si occupa specificatamente delle funzioni Amiga, ci concentreremo in seguito soprattutto sulle prime due sottofasi della *Generazione* del modulo eseguibile.

5. Nelle versioni precedenti del Lattice C, **lmk** era disponibile solo nella confezione avanzata (più costosa), oppure come prodotto a parte. Esistono comunque in giro varie versioni PD per Amiga del programma Unix a cui **lmk** si rifà, e cioè **make**.

6. *Dependent* vuol dire letteralmente *a carico di* od anche *domestico, servitore*, dal latino *de-* e *pendere*, cioè «pendere in giù», da cui anche il nostro *dipendente*.

7. Uno script file è una sequenza di comandi dell'AmigaDOS che può essere eseguito per mezzo dei comandi **execute** e **run** [1.2] oppure attivando il segnalatore **s** associato al file per mezzo del comando **protect** [1.3]. In seguito useremo questo termine in modo più generale ampliandolo anche, ad esempio, ai sorgenti **ARexx** che possono essere usati in modo analogo per eseguire condizionatamente una serie di comandi del sistema operativo.

Quotha 32

software & hardware

SOFTWARE

Originale, sigillato, con garanzia ufficiale e possibilità di aggiornamento

SPREADSHEET INTEGRATI

Microsoft Excel 2.0 in italiano	750.000
Microsoft Works in italiano	295.000
Lotus 1-2-3 2.01 in italiano con impress	695.000
Lotus Symphony 2.0 in italiano	895.000
Ashton Tate Framework in italiano	990.000
Borland Quattro in italiano	330.000

SPEDIZIONI GRATUITE
IN 24 ORE IN TUTTA ITALIA VIA CORRIERE

WORD PROCESSING

Microsoft Word 4.0 in italiano	750.000
Lotus Manuscript in italiano	650.000
Lotus Manuscript 2.0	750.000
MicroPro WordStar 4.0 in italiano	595.000
MicroPro WordStar 5.0	595.000
MicroPro WordStar 2000 Plus 3.0 it.	890.000
Borland Sprint	350.000
Ashton-Tate Multimate Advantage II it.	790.000

MULTI-LINGUAL SCHOLAR
Il W.P. che scrive e stampa in Russo, Arabo, Ebraico, Greco senza modifiche Hardware
890.000

DATABASE MANAGEMENT

Ashton-Tate dBASE III Plus in italiano	890.000
Ashton-Tate dBASE IV	1.090.000
Borland Paradox 2.0 in italiano	1.090.000

Borland Quattro in italiano
330.000

GRAFICI

Microsoft Chart 3.0	550.000
Lotus Freelance Plus 3.0	750.000
Adobe Illustrator per Windows	1.390.000
Harrard Grafics	690.000
Micrografx Designer	1.850.000

DESKTOP PUBLISHING

Aldus PageMaker 3.0 in italiano	1.390.000
Rank-Xerox Ventura Publisher 1.2 in italiano	1.390.000
Rank-Xerox Ventura Publisher 2.0	1.490.000
Rank-Xerox Ventura Publisher 2.0 Professional	690.000
Fonts aggiuntivi ed utilities per Ventura e PageMaker	Telefonare
Ventura 1.2. + Xerox Full Page Display	2.990.000
Microsoft Pageview in italiano	70.000
Ashton-Tate Byline	490.000

UTILITIES

Microsoft Windows 286 in italiano	195.000
Microsoft Windows 386 in italiano	295.000
Microsoft Windows 2.0 Toolkit	650.000
Norton Utilities 4.0	250.000
Borland Sidekick Plus	350.000

Quotha 32
PUNTO DI RIFERIMENTO
PER IL SOFTWARE PACCHETTIZZATO
MANTIENE A MAGAZZINO
LE PIU' RECENTI RELEASE

LINGUAGGI

Microsoft QuickBASIC 4.5	150.000
Microsoft QuickC 2.0	160.000
Microsoft BASIC Compiler 6.0	390.000
Microsoft C Compiler 5.1	595.000
Microsoft FORTRAN Compiler 4.1	595.000
Microsoft Macro Assembler 5.1	250.000
Microsoft COBOL Compiler 3.0	1.150.000
Microsoft Pascal Compiler 4.0	390.000

LINGUAGGI BORLAND SERIE TURBO

Borland Turbo Pascal 5.0 in italiano	**250.000**
Borland Turbo C 2.0 in italiano	**250.000**
Borland Turbo BASIC in italiano	**170.000**
Borland Turbo BASIC Telcom Toolbox	**150.000**
Borland Turbo Prolog 2.0	**195.000**
Borland Turbo Assembler/Debugger	**195.000**
Altri linguaggi e tools	**Telefonare**

Microsoft OS/2 Programmer's Toolkit 495.000

HARDWARE

Originale, imballato, con garanzia TOTALE di 1 anno

PERSONAL COMPUTER DESKTOP

Olivetti M240, M250, M290, M380, P500, P800	Telefonare
IBM Personal System 2	Telefonare

CONDIZIONI AGEVOLATE PER ENTI PUBBLICI, SCUOLE,UNIVERSITA', C.N.R.

PERSONAL COMPUTER LAPTOP

Olivetti M15	Telefonare
Zenith supersPORT/2	Telefonare

ZENITH SUPERSPORT / 20
640 KB RAM, 1 FDU 3,5", 1 HDU 20 MB, schermo LCD retroilluminato
3.750.000

Zenith supersPORT 286/20	Telefonare
Zenith supersPORT 286/40	Telefonare
Zenith turbosPORT 386	Telefonare

STAMPANTI

Panasonic KX-P1081	450.000
Altre Stampanti Panasonic	Telefonare

STAMPANTI NEC
P2200, P6 Plus, P7 Plus
Laser LC-866+, LC-890 PostScript
PREZZI FANTASTICI E CONSEGNA IMMEDIATA

MONITOR

NEC MultiSync GS Monoc.	499.000
NEC MultiSync II	1.190.000
NEC Monograph DTP Fulll Page	2.790.000

HARD DISK

Hard Card PLUS 20 MB + Kit 286	1.250.000
Hard Card PLUS 40 MB	1.550.000
Passport PLUS 20 MB	950.000
Passport PLUS 40 MB	1.290.000

DISKDOUBLER dd2000
raddoppia la capacità del Vostro Hard Disk indipendentemente dalle capacità attuali
495.000

COPROCESSORI MATEMATICI

Cop. Mat. Intel 80287-10	550.000
Altri coprocessori INTEL originali	Telefonare

SCHEDE SPECIALI, GRAFICHE, UPGRADE ED ESPANSIONE

Video Seven VEGA VGA	690.000
Orchid ProDesigner VGA	695.000

ORCHID PRODESIGNER VGA PLUS
512 KB RAM, risoluzione max. 1024 X 768 a 16 colori
1.090.000

Intel Inboard 386/PC 1MB RAM install.	1.590.000

DIGITHURST MICROEYE
Scheda acqisizione immagini da telecamera o videoregistratore anche in standard VGA completa di Editor di immagine compatibile con Windows, Gem e molti altri
1.595.000

Schede espansione RAM Telefonare

MOUSE E SCANNER

Microsoft Mouse Bus + Paintbrush	250.000
Microsoft Mouse Seriale-PS/2 + Paintbrush	250.000
Mouse Logitech C 7	195.000

SCANNER LOGITECH SCANMAN
completo di Editor di immagine compatibile con Windows, Gem e molti altri
450.000

TUTTI I PREZZI SONO AL NETTO DI I.V.A.

TERMINI E CONDIZIONI DI VENDITA: Tutti i prezzi sono al netto di I.V.A. - Pagamento in contrassegno con assegno circolare NT intestato a Quotha 32 s.r.l. o contante. - Sconto del 3% per pagamento anticipato. - Ci riserviamo di accettare ordini di importo inferiore a 500.000 lire. - La merce si intende salvo il venduto. - Ulteriori sconti per quantità. - La presente offerta è valida sino al 15 Giugno 1989 ed annulla e sostituisce ogni nostra precedente offerta'.

per ordini o informazioni telefonare allo

055 - 23.20.240

oppure spedire il tagliando compilato a:

Quotha 32 s.r.l.

Via Accursio, 2 - 50125 FIRENZE
Telefax 055 - 22.80.674

Ragione Sociale: _____

Nominativo: _____ Qualifica: _____

Indirizzo: _____

C.A.P.: _____ Città: _____ Prov: _____

Tel.: _____ Telefax: _____

☐ Desidero essere contattato da un vostro funzionario commerciale

☐ Desidero ricevere informazioni su: _____

☐ Inseritemi nella vostra mailing list

MC