L. 7.000 - S.FR. 10,50

Amiga Transactor

PER

EDIZIONE ITALIANA

LA RIVISTA PER I PROGRAMMATORI DI AMIGA NOVEMBRE - DICEMBRE 1988: NUMERO 1

I messaggi di Amiga • Anteprima dell'AmigaDOS V1.3

I comandi dell'AmigaDOS V1.3 • Noi e i linguaggi

$13 frequenti errori dei programmatori di Amiga

Riflessioni sul workbench • ...E di cosa sono fatti

i piccoli Font? • Concetti di programmazione su Amiga

Compilatore lattice C versione 4.0

L'interfaccia a pacchetti dell'AmigaDOS, in C • Breakpoint

Virus e antivirus

GRUPPO EDITORIALE JACKSON
AREA CONSUMER

TAXE PERCUE (TASSA RISCOSSA) MILANO FERR. CORR. — GRUPPO IV/70

DIRETTORE RESPONSABILE
Giampietro Zanga

DIRETTORE EDITORIALE
Daniele Comboni

REDAZIONE
Leonardo Fei
Marco Ottolini

COORDINAMENTO REDAZIONALE
Angelo Cattaneo

GRAFICA
Wilma Germani

IMPAGINAZIONE ELETTRONICA
Piera Loddo

REVISIONE GRAFICA E IMPAGINAZIONE
Gianni De Tomasi

STAMPA
Grafika '78 - Pioltello - Milano

AUTORIZZAZIONE ALLA PUBBLICAZIONE
Numero in attesa di autorizzazione

AREA CONSUMER

PUBLISHER
Filippo Canavese

CONCESSIONARIO ESCLUSIVO
SODIP - via Zuretti, 25 - 20125 Milano

DIREZIONE, REDAZIONE, PUBBLICITÀ E AMMINISTRAZIONE
via Rosellini, 12 - 20124 MILANO
tel. 02/66800000-66800161-66800272-66800238
Telex: 333436 GEJITI
Telefax: 02/6948238

ABBONAMENTI E MAGAZZINO
via Gasparotto, 15 - 20092 Cinisello B. (MI)
tel.: 02/6122527 - 6187376

SEDE LEGALE
via Pietro Mascagni, 14
20122 MILANO

PREZZO DELLA RIVISTA: L. 7000
NUMERI ARRETRATI: L. 14000
ABBONAMENTO ANNUO (6 numeri): L. 25500
ESTERO: L. 51000

Transactor Canada

PUBLISHER
Richard Evers

EDITORS
Nick Sullivan, Chris Zamara

ASSISTANT EDITOR
Malcolm O'Brien

EDITORIAL ASSISTANT
Moya Drummond

CUSTOMER SERVICE
Renanne Turner

CONTRIBUTING WRITERS
S. Ahlstrom, S. Ballantine, C.B. Blish, J. Butterfield, Betty Clay, D. Curtis, M. Dillon, A. Finkel, C. Innes, C. Gray, P. Kivolowitz R. Mariani, B. Nesbitt, R. Peck, L. Phillips, B. Rakosky, J. Toebes, V.A. Wagner, D. Wood

Transactor UK Ltd
David H. Beatty
(UK Publisher)

per AMIGA
Transactor
La rivista dei programmatori di Amiga

Sommario

Autorevole

Da molti anni a questa parte si è assistito alla nascita di diverse testate dedicate a home computer che a più riprese si sono definite autorevoli. Cosa significa essere autorevoli? Insegnare all'utente i primi rudimenti dell'informatica? Imbottire la rivista di resoconti di fiere e interviste con personaggi vari? Presentare le proprie impressioni sugli ultimi giochi d'oltre oceano?

L'essere autorevoli non può essere il cappello sotto il quale tutti trovano posto per mascherare la propria povertà di contenuti e idee. Per noi essere autorevoli significa essenzialmente proporre un prodotto che pone nelle mani del lettore uno strumento utile e sicuro per l'apprendimento delle tematiche cui ci si riferisce. Quindi, fornire una rivista che non viene abbandonata in uno scaffale dopo una lettura sommaria, ma che viene continuamente consultata ogni volta che si presenta qualche problema o sorge qualche dubbio.

La strada da percorrere per creare una testata con queste caratteristiche non può, al momento, non prevedere un contributo sostanziale di materiale estero. Infatti in Italia non si è ancora affermata una conoscenza di Amiga sufficientemente estesa da permettere l'uso di soli contributi italiani.

L'unica rivista che proponesse materiale rispondente ai nostri criteri di qualità e autorevolezza è "Transactor for the Amiga", versione dedicata al solo Amiga della famosa rivista per il C64 "Transactor", di cui ora proponiamo la traduzione selezionata e arricchita da nostri interventi. Nonostante la ridotta quantità di materiale italiano non disperiamo ed anzi crediamo, entro breve tempo e servendoci di contributi dei lettori, di aumentare il numero di articoli made in Italy.

Amiga Transactor ha un prezzo decisamente basso se lo si considera in rapporto alla qualità e alla quantità di articoli pubblicati. La spesa mensile poi diviene ancora inferiore rispetto ad altre pubblicazioni se si pensa che la rivista è bimestrale e che non è necessario acquistare un dischetto supplementare per disporre dei listati. Sì, i listati sono pubblicati di seguito agli articoli, in quanto riteniamo che siano uno strumento indispensabile per apprendere i concetti enunciati nell'articolo. I rifermenti sono spesso così numerosi che il privare il lettore del listato avrebbe significato obbligarlo ad acquistare l'ulteriore dischetto per poter leggere gli articoli per i quali aveva già pagato il prezzo di copertina della rivista.

Chi poi volesse utilizzare i listati in un proprio programma, potrebbe risparmiare la fatica di digitarli ordinando il dischetto che li contiene unitamente a eventuali utility citate all'interno della rivista.

Amiga Transactor ha messo a punto un'altra serie di servizi, di cui si parla più estesamente nell'interno, per agevolare l'utente di Amiga nel reperimento di informazioni e programmi altrimenti di difficile reperibilità.

Prima di augurare buona lettura mi preme sottolineare che la quasi totalità degli esempi, in questo numero tutti, sono in C o in Assembly, poiché sono gli unici linguaggi tramite i quali si possono sfruttare tutte le caratteristiche di Amiga. Siamo anche consapevoli del fatto che la maggior parte dei lettori non conosce nè l'uno nè l'altro, ma nonostante ciò non prevediamo di iniziare dei corsi su di essi. Perché? Perché esistono già numerosi libri che possono assolvere brillantemente il compito e perché un corso avrebbe occupato dello spazio sulla rivista che invece preferiamo riempire con notizie e informazioni altrimenti non reperibili.

Marco Ottolini

LETTERE...

Device handler cercasi: Per prima cosa vorrei congratularmi con voi per la nuova rivista dedicata ad Amiga; l'annuncio della sua pubblicazione costituisce la migliore notizia che abbia ricevuto in questi ultimi tempi.

Sto scrivendo un programma C per gestire un BBS sul mio Amiga 500 e ho bisogno di creare un mio device che mandi la propria uscita contemporaneamente allo schermo del BBS e al modem. Dovrebbe inoltre combinare i dati provenienti dal modem e dalla tastiera in un flusso unico. Supponendo di chiamare questo device BBS:, se l'utente volesse scrivere un messaggio, il programma lancerebbe il programma Edit nel modo seguente:

Edit >BBS: <BBS: filename

Posso scrivere un programma che compia queste operazioni, ma non so come fare per farlo diventare un device regolare di Amiga, in modo da potergli mandare un file o utilizzare la redirezione da CLI come mostrato prima. Molte grazie.

Howard C. Cochran, Lillburn, Georgia

Quello di cui hai bisogno è un device handler personalizzato, cosa non molto facile da implementare. Sarebbe troppo lungo mostrare come scriverne uno in questa sede, ma esistono già alcuni esempi sui dischi di pubblico dominio. Uno di questi, particolarmente completo e scritto da Matt Dillon, si chiama "DosDev" e mostra come creare un RAM-disk, costituendo così una base eccellente per scrivere un DOS device personalizzato. Il codice sorgente e la documentazione di "DosDev" si trovano sul Fish disk #113. L'handler risultante viene messo nella directory DEVS: (normalmente la directory "devs" presente sul disco con il quale è stato effettuato il boot) e il device viene aggiunto a quelli di sistema creando una "MountList" e utilizzando il comando "Mount". Non abbiamo ancora affrontato un progetto come questo, ma non dovrebbe essere difficile implementarlo avendo a disposizione come base il codice del device handler.

Dal momento che questo approccio al problema di scrivere un BBS sembra interessante, ti auguriamo di riuscire a portarlo a termine con successo.

...E BIT

Una regola facile per i Minterms

Alcune delle funzioni della libreria grafica che permettono di sfruttare il blitter utilizzano un argomento chiamato "Minterm" per specificare l'operazione logica che deve essere eseguita fra il bitmap sorgente e destinazione, prima di copiare il risultato in quello di destinazione. Le funzioni sono:

BltBitMap()	blit da un BitMap a un altro
BltBitMapRastPort()	blit da un BitMap a una RastPort
BltMaskBitMapRastPort()	blit da un BitMap a una RastPort attraverso una maschera
ClipBlit()	blit da una RastPort a un'altra

Con queste funzioni, i bit dal 4 al 7 di Minterm descrivono come settare ogni bit destinazione in base allo stato di entrambi i corrispondenti bit sorgente e destinazione. L'operazione verrà ripetuta per ogni bitplane presente, o per quei bitplane specificati dal parametro "Mask" della funzione o dalla variabile Mask della RastPort. I BitMap sorgente e destinazione possono anche essere uno solo e coincidere.

Probabilmente il Minterm più comune è quello per copiare senza modifiche dal sorgente al destinazione, rappresentato dal valore esadecimale C0 (espresso come 0x0C0 nelle chiamate di funzione in C).

I Minterm per le altre operazioni possibili sono molto facili da calcolare usando questa semplice tavola della verità:

		Destinazione 1	Destinazione 0
Sorgente	1	m3	m2
	0	m1	m0

Le caselle da "m0" a "m3" rappresentano i bit del Minterm e determinano, per ogni bit sottoposto al blit, che cosa verrà scritto nella zona di destinazione quando i bit sorgente e destinazione sono come indicato dalla colonna a sinistra a dalla riga in alto.

Per esempio, possiamo calcolare il Minterm per la sola copia (vanilla-copy) seguendo queste regole: quando il bit sorgente contiene zero, quello di destinazione dovrà essere messo a zero; quando il bit sorgente contiene uno, quello di destinazione dovrà essere messo a uno. Per soddisfare a queste condizioni le caselle da m3 a m0 dovranno contenere:

m3	m2	m1	m0
1	1	0	0

Questo è il valore da mettere nei bit dal 4 al 7 del Minterm, in questo caso 0x0C0. Vediamo un altro esempio, una comune operazione di OR esclusivo utilizzata per modificare un bitmap in modo che una successiva applicazione della stessa operazione lo riporti alle condizioni originali. L'operazione di OR-esclusivo prevede che venga settato a uno il bit del risultato qualora o il sorgente o il destinazione contenga un uno, eccetto nel caso in cui entrambi siano a uno. Riempendo la tavola della verità otterremo:

```
m3  m2  m1  m0
 0   1   1   0
```

Questo dà come risultato il valore 0x060 per il valore da usare per un OR esclusivo. Possiamo estrapolare altri Minterm utili, come 0x000 per azzerare i bit nel rettangolo di destinazione e come 0x0E0 per effettuare un OR fra sorgente e destinazione.
Tutte e sedici le combinazioni che si possono ottenere dalla tavola della verità sono elencate di seguito, segnalate con un asterisco nel caso risultino spesso utili.
Minterm Operazione fra sorgente e destinazione

0000 * azzera tutti i bit del destinazione
0001 OR invertito
0010 setta a uno i bit dove sorgente=0 e destinazione=1
0011 * copia il sorgente invertito nel destinazione
0100 setta a uno i bit dove sorgente=1 e destinazione=0
0101 * inverte il destinazione
0110 * OR esclusivo
0111 setta a uno tutti i bit eccetto dove sorg. e dest. si sovrappongono
1000 setta a uno tutti i bit solo dove sorg. e dest. si sovrappongono
1001 inverso dell'OR esclusivo
1010 come il destinazione; non ha alcun effetto
1011 setta a uno tutti i bit eccetto sorgente=1 e destinazione=0
1100 * copia semplice da sorgente a destinazione
1101 setta a uno tutti i bit eccetto sorgente=0 e destinazione=1
1110 * OR semplice
1111 * setta a uno tutti i bit nel destinazione

Dalla lista precedente possiamo ricavare delle definizioni da mettere in un header file chiamandolo "minterms.h":

```
#define MINTERM_CLEAR       0x000
#define MINTERM_INVERT_SRC  0x030
#define MINTERM_INVERT_DST  0x050
#define MINTERM_XOR         0x060
#define MINTERM_COPY        0x0C0
#define MINTERM_OR          0x0E0
#define MINTERM_FILL        0x0F0
```

Per provare i risultati delle operazioni del blitter con tutti i Minterm, potremo utilizzare il programma "Minterm Tutor" che si trova sul disco di Transactor for the Amiga relativo a questo numero. Si può selezionare uno dei sedici Minterm gadget e vedere il risultato del blit fra un sorgente a singolo bitplane e le immagini di destinazione. Le immagini di destinazione sono cerchi di diverse dimensioni e posizioni; il risultato forma un diagramma di Venn mostrando l'effetto grafico di ogni Minterm. Ad ogni blit viene anche visualizzata la tavola della verità opportunamente riempita e la descrizione dell'effetto del Minterm.

Piccolo Type

Chi ha detto che bisogna scrivere molto codice per fare una qualunque cosa su Amiga? Questo programma copia dal suo input standard al suo output standard e può essere utilizzato come il comando TYPE, semplicemente utilizzando l'operatore di redirezione "<" prima del nome del file. Si può anche redirigere l'output in un file, anziche' averlo sullo schermo. Tutto il lavoro viene svolto da una sola linea di istruzione C:

```
/* Tiny-Type  Se compilate con il Manx usate gli interi a 32 bit */

#define BUFSIZE 512
char buf[BUFSIZE];
main()
{
   while (Write(Output(),buf,Read(Input(),buf,BUFSIZE)) ==
BUFSIZE);
}
```

Un bug di Intuition
di Rico Mariani

Ecco un modo piuttosto eccitante di mandare in crash il vostro sistema: mentre trascinate una finestra sullo schermo del Workbench, premete Amiga-sinistro-N o M (funzionerà anche se non c'è un secondo schermo).
Continuate a trascinare la finestra verso destra, fino a dove potete andare... Caspita! Si fa trascinare via, non è vero? Se lasciate andare la finestra mentre una parte di essa si trova oltre i confini dell'universo, succederanno cose molto brutte; fatelo solo se non vi spaventa un bel GURU in un futuro abbastanza prossimo. Se riportate, invece, la finestra nei confini sicuri dello schermo prima di lasciarla andare, tutto procederà normalmente e non succederà niente.

Salvataggio di un disco
di Bryce Nesbitt

Se trovate un disco che ha un "Read/Write Error":

non lasciatevi prendere dal panico!...

c'è un facile trucchetto che permette di recuperare giornate di lavoro in un sorprendente numero di casi.

Raggiungete il bordo del disco che c'è nel drive e afferratelo. Muovetelo un pochettino in sù o in giù e provate a riprendere (RETRY) l'operazione precedentemente interrotta. Potrebbe essere richiesti alcuni tentativi in una direzione o nell'altra, ma c'è una buona probabilità che si possano recuperare i dati che ormai si credevano perduti!

Questo fenomeno è dovuto a problemi di allineamento. Molti errori sui dischi sono causati da posizioni leggermente differenti della testina di lettura/scrittura nei vari drive. Piegando il disco in questa maniera, viene leggermente spostata la parte del supporto magnetico attualmente sotto la testina, compensando il problema. Assicuratevi di eseguire una copia di sicurezza del disco in questione al più presto possibile.

I messaggi di Amiga

di Don Curtis

Emulazioni, tastiera dell'A2000, nuovi Amiga

Don Curtis è un ufficiale di polizia a Denver, Colorado, assegnato durante gli ultimi due anni a progettare e sviluppare programmi. Ha curato inoltre la progettazione e la manutenzione di sistema di 10 computer AT&T Unix 3B2. Nel suo tempo libero Don è un assistente SYSOP nell'AmigaForum su CompuServe.

Notizie per i possessori di SideCar...

Se avete la versione tedesca del SideCar attaccata al vostro A1000, ci sono buone notizie! Potrete usare il nuovo software sviluppato per l'A2088 (BridgeBoard) con il vostro SideCar e trarne sostanziali vantaggi. Alcuni dei miglioramenti che si possono ottenere installando il nuovo software sono: nessun messaggio di debugging viene mandato sulla porta seriale al momento dell'attivazione di PCWindow (mono o colore) e questi programmi si caricano e girano più velocemente. Il SideCar lavora con le espansioni di memoria e ora funziona correttamente anche quando l'Amiga lavora intensamente (prima, un intenso lavoro da parte dell'Amiga poteva portare a un arresto della parte PC). Il trasferimento dei file dal PC all'Amiga e viceversa avviene senza problemi. Non ci sono più limitazioni per quanto riguarda il numero di partizioni Amiga sull'HardDisk del PC (sembra che in precedenza, quando il PC disk superava il 60 per cento di riempimento, finiva sulla partizione Amiga).

Per poter usufruire di queste migliorie, dovete procurarvi una copia del disco di installazione di A2088 dal vostro rivenditore. Non è ancora chiaro se il rivenditore può farvi avere il disco come upgrade del vecchio software, oppure se dovete acquistarlo nuovamente.

Quando sarete in possesso del disco dell'A2088, leggete lo script file chiamato SideCarInstall. Fate una copia del vostro Workbench 1.2 poi eseguite questo script file. Questa operazione crea un nuovo disco di boot per il SideCar con tutto il nuovo software installato. Non usate lo script file BridgeInstall, in quanto copierebbe il file LPT1, mentre il SideCar necessita del programma LPT1S. Se avevate una partizione sull'HardDisk del PC dovrete riformattarla. Fate una copia di tutti i dati contenuti nella partizione, quindi riformattate e ricopiate i dati nella nuova partizione. Ora potete rimettervi al lavoro con una nuova macchina... o, almeno, tale vi sembrerà!

Parliamo della BridgeBoard

Molti di coloro che hanno acquistato una BridgeBoard hanno dei problemi a collegare l'altoparlante alla loro scheda. L'uscita sul jack (J2) non è sufficente a produrre un volume ragionevole pilotando un altoparlante normale. Ho esaminato questo segnale con un oscilloscopio, scoprendo che sulla mia macchina l'uscita è di circa 2v picco picco. Questa è più che sufficiente per pilotare un amplificatore, ma in tale evenienza sareste costretti a portare la linea fuori dal cabinet dell'A2000. Esistono tuttavia altre alternative. Potreste portare un filo conduttore dal pin destro di J2 (visto dal lato componenti) al pin 6 del connettore seriale interno. Il connettore seriale interno si trova sulla MotherBoard appena dietro la porta seriale esterna. Questa modifica renderà disponibile l'audio del PC sui normali connettori audio di Amiga.

Un altro metodo consiste nell'utilizzare un cicalino piezoelettrico. Normalmente potete reperirne diversi modelli nei negozi di materiale elettronico. Dopo averne provati parecchi, devo dire che il suono prodotto è decente, senza essere niente di eccezionale. Il metodo che alla fine io stesso ho adottato è quello di usare il microfono di un vecchio registratore. Certo, un microfono. Molti di questi, infatti, funzionano altrettanto bene come altoparlanti. Questa soluzione si è rivelata la migliore tra quelle esaminate, in quanto è di facile attuazione, potendo installare il microfono all'interno dello chassis di Amiga. Se utilizzate il microfono/altoparlante, assicuratevi che il posto dove lo piazzerete non lo faccia interferire nel normale funzionamento del computer. Inizialmente lo avevo piazzato sulla parte frontale della motherboard, vicino al drive da 5.25". A causa delle interferenze magnetiche che si creavano, il drive segnalava moltissimi errori. Allora ho spostato il piccolo altoparlante alla sinistra della BridgeBoard e ora funziona tutto bene.

Le voci di corridoio, riguardo la BridgeBoard, continuano. Sembra che la CBM abbia smesso di produrre la versione 8088 (A2088) in favore della versione 80286 (stile AT... A2286?). Nel momento in cui scrivo non posso confermare queste voci, ma è pur vero che la Commodore sta lavorando ad una BridgeBoard di tipo AT. Si suppone che il prezzo per la scheda AT si aggiri intorno a un milione e mezzo. Se questo fosse vero, per me non avrebbe alcun senso... ma non mi sorprenderebbe. La CBM non è famosa per avere fatto sempre cose sensate. Sebbene ci saranno sicuramente degli utenti interessati esplicitamente a una BridgeBoard di tipo AT, mi chiedo quanti di coloro a cui basta la semplice compatibilita' PC saranno disposti a pagare il mezzo milione supplementare per avere quella AT. Non mi piacerebbe vedere la Commodore perdere potenziali vendite della A2088 durante il periodo di attesa che precederebbe la diponibilità della nuova BridgeBoard. Proprio in questo momento la domanda di schede A2088 è assai superiore alla disponibilità delle stesse. Penso che ci sia posto per entrambi i modello sul mercato, così come c'è posto sia per l'A500 che per l'A2000. Tutto dipende da quello che vogliono gli utenti finali e la Commodore dovrebbe renderle entrambe disponibili.

Emulatori e cloni

La discussione sulla BridgeBoard rievoca l'argomento degli

emulatori e dei cloni. Sono una buona idea? Non c'è una risposta definitiva e assoluta. Da una parte, la compatibilità è una buona cosa. Che vi piaccia o no, i PC MS- DOS sono qui per rimanere e costituiscono una forza di primaria importanza nel mercato. Rappresentano, de facto, lo standard nel campo del generico impiego per affari. D'altronde per i prossimi anni, se vi porterete a casa del lavoro dall'ufficio, più che probabilmente avrete bisogno di un PC per compiere quel lavoro. L'Amiga e il Mac si stanno lentamente introducendo nel mondo del business, ma a tutt'oggi il PC non ha veri concorrenti. Sulla base di queste osservazioni, aggiungere la compatibilità PC all'Amiga è una buona cosa. Significa che le persone che vogliono un Amiga, ma hanno bisogno di far girare applicazioni MS-DOS, possono avere entrambi in una sola macchina. La BridgeBoard, essendo un PC su una scheda per Amiga, costituisce una valida e meno costosa alternativa a un PC separato.

Che cosa dire, però, degli altri emulatori? Se Magic Sac per Amiga farà mai la sua comparsa, sarà la compatibilità con il Mac una buona cosa? Che cosa dire degli emulatori per il C64? Sarebbe un emulatore Atari (ST o 8 bit) o un emulatore per la serie Apple II un'idea degna di considerazione?

Penso di no.

Guardate il Transformer, il primo emulatore PC. Era lento, solo parzialmente compatibile e, dopo tutto, non molto utile. Se aveste mostrato a qualcuno il vostro Amiga mentre faceva girare del software PC sotto Transformer, quel qualcuno vi avrebbe fatto una risatina e vi avrebbe detto "Cavoli..., mica male, ma non posso aspettare 20 minuti perchè lo spreadsheet faccia quel calcolo che al lavoro mi richiede solo 5 minuti!". Gli emulatori per il C64 sono nella stessa barca. Funzionano, più o meno, ma sono lenti. Non sono totalmente compatibili e probabilmente non lo saranno mai. Non abbiamo emulatori per Atari 8 bit o ST e neanche emulatori per Apple II o Macintosh, ma se li avessimo e se l'emulazione fosse via software, sarebbero altrettanto o più lenti degli emulatori che già abbiamo.

Chiunque veda girare uno di questi lenti emulatori sull'Amiga potrebbe concludere che l'Amiga non è neanche all'altezza della macchina che intendeva emulare e noi tutti sappiamo checiò non è vero. Potrebbe addirittura formarsi la convinzione che non c'è neanche del buon software per l'Amiga stesso, tanto che la gente ha bisogno di ricorrere agli emulatori per avere il software desiderato.

Che dire, allora, degli emulatori hardware? Questi dovrebbero darvi la stessa velocità e compatibilità che la BridgeBoard ci fornisce per l'emulazione PC. Per calcolatori a 8 bit, come l'Atari 800, il C64 e perfino l'Apple II, il costo di un emulatore hardware sarebbe più che probabilmente prossimo o superiore all'originale. Non sarebbe dunque conveniente dal punto di vista economico.

Questo ci limita agli emulatori hardware per il Mac e l'ST. Ci guadagneremmo qualcosa? C'è un effettivo bisogno di emulare una delle due macchine? Ancora una volta, penso di no. Magic Sac per l'ST viene venduto a quelle persone che vogliono un Mac economico o vogliono usare software che non può essere trovato per l'ST stesso. Vogliamo veramente che le persone comprino un Amiga per trasformarlo in un clone economico del Mac? Se ci limitiamo a considerare l'ST, dobbiamo osservare che l'Amiga ha già un prezzo inferiore rispetto all'ST e che il software disponibile per l'ST è disponibile anche per l'Amiga. Non ci si guadagnerebbe nulla.

Infine, che cosa si verifica se voi già possedete una di quelle macchine e volete passare a un Amiga, pur mantenendo il vostro vecchio software? Questo è probabilmente un argomento valido. La mia esperienza dimostra che le persone che passano a una macchina nuova si stancano presto del loro vecchio software e comprano comunque nuovo software. A lungo andare, si accorgeranno probabilmente che sarebbe stato più conveniente tenersi il vecchio calcolatore. Il denaro che si ottiene "dando indietro" il calcolatore o vendendo tutto in blocco non è molto differente dal costo dell'emulatore.
Quando poi si stancheranno del vecchio calcolatore, avranno sia il computer sia il software per il medesimo, il che rappresenta un notevole vantaggio quando si cerca di vendere un calcolatore usato. Vendere il solo computer prima e il software in un secondo tempo non dà buoni risultati. Come esempio, ho venduto il solo ST che ho usato per lo sviluppo di software. La prima cosa che la gente mi chiedeva era quanto software io avessi da vendere insieme al computer. Non appena si accorgevano che io avevo solo software per lo sviluppo di programmi (compilatori, text editor e così via), perdevano rapidamente interesse o chiedevano un prezzo più basso.

Comprendo che queste stesse argomentazioni possono benissimo essere applicate alla BridgeBoard. E' solo perché il PC è così radicato nel mondo degli affari (e nelle menti di molte persone) come "il" calcolatore, che la BridgeBoard, tra tutti i possibili emulatori, è l'unico ad avere senso. Ci sono assolutamente troppi PC e troppo software per il PC in circolazione perchè li si possa ignorare.

E parlando di cloni...

Argomento interessante. La Discovery International vende il Marauder II, un programma copiatore che clona (copia) quasi tutti i programmi sul mercato. La Discovery include perfino delle routine che sproteggono certi programmi in modo che li possiate installare su un hard disk. Utilities come questa hanno differenti impieghi e, se vengono impiegate legalmente, non vedo perchè non debbano essere usate. Il manuale del Marauder inizia specificando che le Leggi Federali sul Copyright permettono al possessore di un programma di farne una copia per scopi di archiviazione. Prosegue affermando che, con alcuni programmi, ottenete solo la licenza di usare il programma e non vi è quindi permesso farne una copia. Fino a questo punto tutto bene.Ora arriva il problema: provate a indovinare chi ha recentemente pubblicato un programma che non può essere copiato con il Marauder II? Già, la Discovery Software ha di recente pubblicato Arkanoid e

il Marauder II non lo copia. Non c'è nessuna clausola che precisi che voi non possedete Arkanoid e neppure che affermi che è proibito fare copie per l'archiviazione.
Non sto criticando ilsoftware; tutti e due i programmi sono buoni. Sono solo curioso di capire perchè una società produca un programma in grado di copiare quasi ogni programma protetto altrui e non i propri.

Dirò questo in difesa della Discovery: col programma viene dato un coupon che vi permette di avere una copia di backup per tre dollari. E' un prezzo e una politica molto ragionevole che consente di avere una seconda copia di Arkanoid nel caso in cui l'originale non funzioni.

Messa a punto della tastiera dell'A2000

Alcuni dei nuovi Amiga A2000 presentano un piccolo ma fastidioso problema con la tastiera. Sembra che il primo (e solo il primo) tasto premuto in una nuova finestra CLI venga perso. Fortunatamente, la CBM ha deciso di provvedere alla riparazione ufficialmente.
In primo luogo, tutto ciò riguarda solo la tastiera della Hitek. La Hitek può essere identificata in parecchie maniere: i tasti di funzione sono più grandi dei tasti nel resto della tastiera e, se guardate attentamente negli spazi tra i tasti, lo schermo sottostante è nero. Ovviamente, il modo definitivo di determinare quale tastiera abbiate è quello di controllare se perdete il primo tasto premuto in una nuova finestra CLI.

Ricordatevi che fare questa modifica da soli invaliderà la garanzia. Se il vostro A2000 è ancora in garanzia, riportatelo dal vostro negoziante per la riparazione. I negozianti sono stati autorizzati a fare la modifica come riparazione in garanzia.

Anche se il vostro A2000 è fuori garanzia, dovreste contattare il vostro rivenditore per fare la modifica. A lungo termine risulta più sicuro che farlo da soli. In questa maniera, se qualcosa va storto, il rivenditore sarà responsabile per qualsiasi ulteriore riparazione che possa essere richiesta.
La modifica richiede la rimozione della intelaiatura di supporto per i drive e l'eliminazione di due condensatori (c910 e c911) dalla Motherboard vicino al connettore della tastiera. Questo lavoro non è indicato per chi non è pratico di riparazioni di apparecchi elettronici.

Revisioni della motherboard per l'A2000

Ho parlato con un po' di persone preoccupate di avere una motherboard obsoleta o difettosa nel loro A2000 (la versione West Chester o B2000 è stata rivista un pò di volte da quando ha fatto la sua comparsa nei negozi).Tutte le revisioni sono consistite in piccole modifiche e non hanno alcun effetto sul funzionamento della macchina. Secondo Dave Haynie della Commodore-Amiga (il progettista della B2000), non è uscita alcuna macchina con la motherboard rev 4.0. Alcune macchine, principalmente unità dimostrative per rivenditori, sono uscite con la motherboard rev 4.1. Queste hanno una uscita video di scarsa qualità a causa di alcune perline di ferrite sovradimensionate nella circuiteria di uscita del segnale video.Pochissime di queste macchine sono andate ai clienti. Se vi è capitato uno di questi A2000 dalla cattiva uscita video, il rimedio è molto semplice: eliminate le perline di ferrite nella circuiteria video.
La revisione 4.2 ha risolto il problema. Nella motherboard rev 4.3 sono state aggiunte due resistenze di pull-up per il chip GARY prodotto dalla Commodore (MOS). Questa modifica fu apportata quando venivano ancora usati i GARY prodotti dalla Toshiba in modo da evitare sorprese nel momento in cui si fosse passati ad impiegare i componenti della MOS. Che cos'è un chip GARY? GARY sta per GAte aRraY, insieme di porte, e viene usato in combinazione con Agnus.

Come potete vedere, l'unica versione in cui si possa apprezzare una differenza è la rev 4.1. Con una macchina di questo tipo, notereste un video confuso. Guardando sul retro della motherboard, dalle parti del connettore RGB, dovreste vedere 3 perline di ferrite su cui sono avvolti dei fili.
Le motherboard rev 4.2 e 4.3 hanno i fili dell'uscita RGB che semplicemente passano attraverso le perline e non sono avvolti su di esse. Se avete una di queste macchine, portatela dal vostro rivenditore e questi dovrebbe eliminare le perline. Dovreste pagare poco o nulla per questa operazione.

Che cosa c'è di nuovo alla CBM...

Ci sono tantissime nuove voci intorno a un nuovo modello di Amiga. Vorrei sottolineare che, al momento in cui scrivo, sono solo voci e non annunci ufficiali.
La macchina in questione (A3000?) dovrebbe essere basata sul 68020 o sul 68030 a seconda del costo della CPU quando inizierà la produzione.
Se si tratterà di una macchina col 68020, questa avrà una MMU 68851; sul 68030, la MMU è contenuta nel microprocessore stesso. Si vocifera che, in entrambi i casi, il coprocessore matematico sarà il nuovo chip 68882. La motherboard ospiterà 2 o forse 3 megabyte di RAM come configurazione base e ci sarà un maggior numero di slot per le espansioni.

Si dice anche che la macchina supporterà un display non interlacciato di 1024x920 e una gamma di 2 milioni di colori. Con una risoluzione di questo tipo, ogni bitplane userà circa un megabyte di RAM, per cui questa macchina risulterà una divoratrice di memoria.

La Commodore ha inoltredeciso di supportare fino a 4 megabyte di RAM sulla sua scheda a coprocessore A2620 per l'A2000. La A2620 è la scheda 68020/ 68851/68881 di prossima uscita che andrà nello slot del coprocessore dell'A2000.
Verrà fornita con 2 megabyte di RAM, ma sarà espandibile sulla medesima scheda a 4 Mega.

L'A2620 è stata provata su un A2000 con il BridgeBoard installato ed entrambi hanno funzionato senza problemi.

Anteprima dell'AmigaDOS V1.3

di Chad Innes

Nuove aggiunte e miglioramenti sostanziali

Questo articolo é una breve anteprima delle modifiche e delle aggiunte all'AmigaDOS V1.3 rispetto alla versione 1.2. E' basato sulla "versione gamma 4" e quindi bisogna tenere presente che la versione 1.3 rilasciata potrà avere qualche piccola differenza rispetto alla presente. Inoltre non si tratta di un esame dettagliato dell'AmigaDOS V1.3 ma è solamente una breve occhiata.

Ecco una panoramica dei maggiori cambiamenti della versione 1.3:

- Nuovo KickStart che permette di effettuare il boot da device diversi da DF0:.

- Nuovo file system veloce (FFS) per gli hard disk (incrementa la velocità da quattro a cinque volte).

- L'aggiunta del device PIPE:.

- Nuovo CLI con una shell. Permette il richiamo di comandi precedenti, l'editing e altro.

- Esiste un RAM disk recuperabile (sopravvive al reset).

- Nuovo driver veloce per le stampanti e un maggior numero di stampanti supportate.

- Nuove librerie matematiche IEEE veloci con supporto del coprocessore matematico 68881.

- Nuovi comandi CLI e modifiche ad alcuni comandi CLI esistenti.

Entriamo nei dettagli. Una delle prime cose che si notano nel Workbench 1.3 è che, nel cassetto System, "SlowMemLast" (usa la memoria lenta per ultima) è stato cambiato in "FastMemFirst" (usa la memoria veloce per prima).
I programmi sono identici e hanno la stessa funzione, solo il nome è stato cambiato: forse per confondere i principianti? La seconda è che l'importantissimo cassetto "Demos" (qualcuno l'ha mai usato?) è stato spostato dal disco Workbench all'Extras.Il Workbench è così pieno che alcune keymap e alcuni driver per le stampanti sono stati anch'essi spostati nel disco Extras.

- Nuovo KickStart con auto-boot

Il nuovo KickStart permette di effettuare il boot da device diversi da DF0:. Quali? Si è saputo da una fonte molto attendibile che la Commodore sta lavorando a un controller per hard disk in grado di effettuare l'auto-boot. La seconda possibilità è effettuare l'auto-boot dal RAM disk recuperabile (RAD, da Recoverable Auto-bootable Device)! Sì, è proprio vero, si può fare un re-boot da RAD. Questo nuovo RAD è conosciuto nell'ambiente di West Chester (dove ha sede il quartier generale Amiga) come RAMBO. Più avanti nell'articolo si trovano maggiori informazioni sul RAD.

Al momento non si hanno notizie di altri device, disponibili o in via di preparazione, che sfruttino l'auto-boot, ma con questa nuova possibilità si può essere certi che presto ne appariranno di nuovi.

- Nuovo file system veloce per hard disk

Il file system veloce (FFS, da Fast File System) è progettato per gli hard disk e per gli analoghi device non rimuovibili. Per usare il nuovo FFS occorre modificare la mountlist per il proprio disco fisso e riformattarlo prima di usarlo.

Vediamo alcuni punti interessanti.

Per prima cosa bisogna mantenere ancora una piccola partizione formattata AmigaDOS in aggiunta a quella FFS. La partizione DOS normale può consistere anche in un solo cilindro, ma deve esistere e deve essere la prima partizione del drive.

Secondo punto: non è permesso l'auto-boot da un device FFS. Quindi, se si pensa di aggiungere al proprio sistema un controller per hard disk che supporti l'auto-boot, bisognerà comunque mantenere una partizione DOS normale sul proprio hard disk. Tale partizione deve essere dimensionata a sufficienza per poter contenere tutte le informazioni necessarie (leggi comandi) ad attivare il proprio Amiga e per trasferire il controllo alla parte FFS del proprio disco fisso. Durante alcuni test si è notato un sorprendente aumento di velocità, che fornisce una nuova sensazione di potenza e velocità all'Amiga.

Il tipico "addbuffers 20" aggiunge un ulteriore turbo al FFS. Se avete già usato "Facc II", un programma che aumenta le prestazioni dei floppy, avete un'idea di che tipo di prestazioni possa raggiungere il disco fisso con FFS. Bisogna levarsi tanto di cappello di fronte alla CBM: FFS è uno dei migliori potenziamenti dell'Amiga da lungo tempo a questa parte.

- Il device PIPE:

La Commodore ha finalmente aggiunto un device PIPE:. Questo

é un device standard di Amiga e non una pipe nel senso Unix del termine. Per usare il nuovo PIPE: bisogna aggiungere il device alla propria mountlist e poi montarlo (comando MOUNT) prima dell'uso. Ecco un esempio di come il nuovo device può essere usato:

```
1> run copy df0:s/startup-sequence pipe:sam
[CLI 2]
1> type pipe:sam
```

Naturalmente questo é un esempio stupido, ma mostra, per chi già conosce i principi del pipelining, come lavora il nuovo device.

- Nuova shell CLI

Che dire di una shell per il CLI? E' possibile usare "Resident" per mantenere dei comandi in memoria e richiamarli istantaneamente senza caricarli da disco. E' stata aggiunta la possibilità di usare alias, cioé stringhe definite dall'utente che si sostituiscono al nome dei comandi. Per esempio:

```
1> alias cp copy
1> cp DF0:pippo DF1:pluto
```

Evidenzia come cp possa diventare un sinonimo del comando copy. Un'altra caratteristica interessante é il modo con cui la shell gestisce il comando prompt. Con la versione 1.2 "prompt %n" stampava come prompt il numero del CLI (al posto di %n). Questo é ancora valido con la nuova shell, ma ora se si usa "prompt %n %s" si vedrà il numero del CLI e il "%s" sarà sostituito dal path della directory corrente.

Si possono anche richiamare le linee di comando inserite in precedenza, tramite i tasti cursore su e giù, e si possono modificare usando i tasti cursore destra e sinistra. Ciò significa che se accidentalmente si scrive quanto segue:

```
1>SIR df0:devs/printers
```

e si preme return, non è più necessario riscrivere l'intera linea. Basta premere il tasto cursore su e la lina ricompare. Poi non bisogna più premere in continuazione il tasto backspace per cambiare la "S" in "D", ma ci si può portare sulla "S" con i tasti cursore per poi premere DELETE, "D" e infine return.

- Un RAM disk recuperabile (RAD)

Il nuovo device RAD differisce dal vecchio disco RAM: in alcuni punti sostanziali. Per prima cosa deve essere aggiunto alla propria mountlist, montato e formattato prima dell'uso.
Ciò significa che si deve impostarne la dimensione nella mountlist e che, a differenza di RAM:, non cresce e si riduce a seconda di ciò che contiene. Inoltre si può scegliere se usare l'FFS e/o il file system normale ma, se si desidera effettuare il re-boot da RAD occorre usare il file system normale.

A conti fatti, il RAD Commodore non differisce di molto nell'uso normale dal RAD commerciale (VDK:) o da quello di pubblico dominio (VD0:).

- Driver veloci per stampanti

I miglioramenti ai driver per stampanti riguardano: auto centratura, supporto per tre tipi di dithering, correzione del colore per l'RGB, incremento di velocità da 3 a 20 volte rispetto alla versione 1.2, supporto per la stampa di figure HAM e una nuova versione di Preferences con opzioni aggiuntive.

Le aggiunte a Preferences includono due nuovi gadget (che corrispondono a due schermi), "Graphic 1" e "Graphic 2". Graphic 1 (secondo schermo per la stampante) ha una nuova selezione per le sfumature di stampa (scala di grigi). Graphic 2 (terzo schermo per la stampante) dispone di numerose nuove opzioni: antialiasing, offset da sinistra, centratura, densità da 1 a 7, correzione del colore per l'RGB, dithering ordinato/mezzotono/F-S, supporto per la scalatura intera o frazionaria e limiti in altezza e larghezza.

- Librerie matematiche IEEE

Le nuove librerie matematiche permettono il supporto del coprocessore matematico 68881 sia tramite una scheda con 68020 che, con driver speciali, come un add-on. In aggiunta al supporto del 68881, le librerie matematiche sono state riscritte per essere più veloci di quelle della versione 1.2 anche se sono usate senza coprocessore matematico.

- Nuovi comandi CLI e modifiche ai comandi CLI esistenti

Per le aggiunte e le modifiche ai comandi CLI rimandiamo all'articolo intitolato "Comandi dell'AmigaDOS V1.3" che segue.
Abbiamo dato un breve sguardo all'AmigaDOS V1.3.
In chiusura, mi preme ricordare nuovamente che si è trattato solamente di un'anteprima della versione 1.3 e non di una analisi dettagliata.

Comunque, buona fortuna con la V1.3. Se volete discutere dell'articolo o semplicemente parlare dell'Amiga, potete lasciarmi una nota al Berks Amiga BBS (215) 678-7691 (24 ore, 300/1200/2400 baud).

Per gli interessati, il numero di telefono, ovviamente non appartiene alla rete italiana me é di quella americana (NdT).

Comandi dell'AmigaDOS V1.3

I nuovi comandi DOS e quelli che sono cambiati

Questo articolo deve essere considerato come una guida da usare in aggiunta al manuale originale sull'AmigaDOS della Commodore. Le informazioni contenute in quest'articolo sono ricavate dalla versione gamma 4 dell'AmigaDOS 1.3 e di conseguenza possono essere soggette a piccole modifiche nella versione definitiva.
La sintassi seguente sarà usata per tutte le descrizioni dei comandi:

[1.3] (se è un comando nuovo) NOME COMANDO

FORMATO COMANDO

Descrizione completa del nuovo comando oppure elenco delle modifiche rispetto alla versione 1.2.

ADDBUFFERS

ADDBUFFERS <drive>: <nn>

Quando è usato con il FFS, ADDBUFFERS velocizza sempre gli accessi al disco, a differenza del file system normale. Con l'FFS vengono mantenute in memoria centrale solo le informazioni sulle directory e non i blocchi di dati. Il tipo di memoria usata da ADDBUFFERS può essere specificata nella mountlist per mezzo della voce "BUFMEMTYPE".

COPY

COPY [FROM]<nome> [TO]<nome> [ALL] [QUIET] [BUF=BUFFER] [CLONE] [DATE] [PRO] [COM]

BUF => numero di buffer da 512 byte da usare durante la copia.
CLONE => combinazione di DATE, PRO e COM.
DATE => copia anche la data di creazione.
PRO => copia anche i bit di protezione.
COM => copia anche il commento.

DATE

DATE <data> <ora> [TO|VER <file>]

Non richiede più la presenza di zeri non significativi prima di cifre singole.

DELETE

DELETE <nome> [ALL] [QUIET]

Continua a cancellare i file rimanenti, se fallisce la cancellazione di uno di essi durante una operazione multipla.

DIR

DIR [<directory>] [OPT A=ALL|I=INTER|D=DIRS]

Ora accetta "A", "I" e "D" al posto di "ALL", "INTER" (interattivo) e DIRS rispettivamente. C'è un'opzione "COMMAND=" in modo interattivo che permette l'esecuzione della maggior parte dei comandi. L'opzione DIRS visualizza solamente le directory.

DISKDOCTOR

DISKDOCTOR <drive>:

DISKDOCTOR ora controlla che sia disponibile memoria sufficiente prima di iniziare ad operare.

ECHO

ECHO <stringa> [NOLINE]

Quando si usa l'opzione NOLINE, non viene stampato il linefeed e quindi non si va a capo.

EXECUTE

EXECUTE <file di comandi> [<argomenti>]

EXECUTE controlla l'esistenza del device logico T: per usarlo se è presente; altrimenti usa la directory :T.

[1.3] FF

FF -0|-n

FF (FastFonts, font veloci) velocizza la visualizzazione di testo sullo schermo dell'Amiga. L'opzione "-0" abilita FF, mentre "-n" lo disabilita.

INFO

INFO [<device>]

INFO ora accetta anche nomi di device più lunghi. La possibilità di specificare un device permette di ottenere informazioni solo su di esso.

INSTALL

INSTALL [DRIVE] <df0|df1|df2|df3>: [NOBOOT] [CHECK]

La memoria usata per creare il blocco di boot viene cancellata prima dell'operazione di installazione di un disco. Con NOBOOT si disattiva la possibilità di effettuare un boot con quel disco, che rimane comunque ancora un disco AmigaDOS. Con CHECK viene controllato che il boot block sia quello standard Commodore (utile per verificare la presenza di virus). Il codice di ritorno del comando è posto a 0 se non viene trovato il boot-block oppure se é quello standard, altrimenti viene posto a 5 (WARN).

LIST

LIST [[DIR] <dir>] [P=PAT <pattern>] [KEYS] [[NO]DATES] [TO

<nome>] [S <stringa>] [SINCE <data>] [UPTO <data>] [QUICK] [BLOCK]

Vengono visualizzati i nuovi bit di protezione: H - nascosto (hidden), S - file di comandi (script), P - puro, A - archivio. L'opzione QUICK non visualizza le colonne aggiuntive dopo il nome del file. L'opzione BLOCK visualizza la dimensione in blocchi anziché in byte. Il bit H rende il file invisibile durante un comando DIR o LIST, ma ciò è valido solo se si usa la nuova shell Commodore. Il bit S permette l'esecuzione di un file di comandi come se fosse un vero programma (cioé senza usare il comando EXECUTE). Per una descrizione del bit P, riferirsi al comando RESIDENT. Il bit A è usato come indicatore del fatto che il file sia stato modificato. Se il bit è a 1, qualunque comando causi una scrittura nel file (come COPY o ED) provocherà l'azzeramento del bit, indicando che è stata effettuata una modifica.

[1.3] LOCK

LOCK <drive>: ON|OFF [<password>]

LOCK permette di proteggere in scrittura un hard disk o una partizione FFS. La protezione rimane attiva finché non viene disattivata con il comando LOCK o il sistema non effettua un reboot. Quando si protegge un disco, la password permette di specificare una stringa di quattro caratteri che deve essere fornita tale e quale per riabilitare la scrittura.

NEWCLI

NEWCLI [<specifiche della finestra>] [FROM <file>]

Il nuovo CLI diventa automaticamente una shell se SHELL-SEG é residente. Se non è specificato nessun file FROM, viene usato, se esiste, il file di default "S:CLI-StartUp".

PROTECT

PROTECT [FILE] <file> [FLAGS [+=ADD][-=SUB] <stato>]

PROTECT può essere usato per aggiungere (ADD|+) o togliere (SUB|-) bit di protezione a un file. Adesso sono disponibili i nuovi bit di protezione: H, S, P ed A. Riferitevi al comando LIST per una spiegazione delle loro funzioni.

[1.3] RESIDENT

RESIDENT <nome residente> <file> [DELETE] [ADD] [REPLACE] [PURE]
RESIDENT rende residente in memoria il comando specificato. Una volta che un comando è divenuto residente, non c'è più bisogno di effettuare accessi al disco per caricarlo. RESIDENT usato senza parametri, visualizza una lista dei comandi attualmente residenti. Non tutti i comandi possono diventare residenti. Solo quelli che sono ri-entranti e ri-eseguibili. I programmi che soddisfanno questi requisiti avranno il bit di protezione P messo a 1. RESIDENT funziona solamente usando la Shell Commodore.

RUN

RUN <comando> [+<comando>]

RUN è ora in grado di eseguire comandi in background senza che ciò pregiudichi la possibilità di chiudere il CLI da cui era stato eseguito. Ciò si ottiene ridirezionando l'output del comando RUN. Quando RUN è usato per iniziare un nuovo task CLI, viene usato il file "S:CLI-StartUp", se è presente, per inizializzarlo.

SEARCH

SEARCH [FROM] <file>|<pattern> [SEARCH]<stringa> [ALL] [NONUM] [QUICK]

SEARCH ora restituisce come valore di ritorno uno 0 se l'oggetto è stato trovato e un 5 (WARN) altrimenti; inoltre ferma la ricerca se si preme CTRL-C. Se si usa l'opzione NONUM, SEARCH non stampa il numero (o i numeri) di linea di dove è stata trovata la stringa. L'opzione QUICK usa una forma più compatta di visualizzazione.

SETDATE

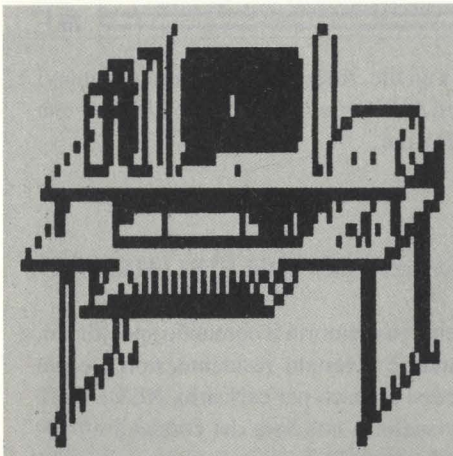
SETDATE <file> <data> [<ora>]

Il modo di ricevere l'input di SETDATE è stato modificato, in modo che ora possa ricevere l'output di DATE. Non è più richiesto l'uso di zeri non significativi prima delle singole cifre. Se non si specifica nessuna data, viene presa per default quella corrente, agendo quindi come il comando Unix "touch".

STATUS

STATUS [PROCESS] <processo> [FULL] [TCB] [SEGS] [CLI=ALL]

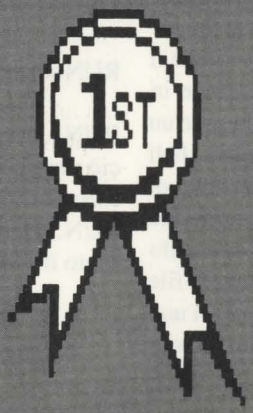
(segue a pag. 30)



AMIGA™

COMMODORE

COMPUTER CENTER
030/223230



AMIGALINE BBS 24 h
030/2420452

Il Centro TUTTO COMMODORE

più qualificato e completo!

..impossibile chiedere di più..

..assurdo accontentarsi di meno!!

COMPUTER CENTER POINT

Via Cipro int. 62 - 25124 BRESCIA

Noi e i linguaggi

di Larry Phillips

Larry Phillips è un hacker hardware e software di Vancouver, nel British Coloumbia, e un Sysop nell'Amigaforum su

Vi siete mai fermati un attimo a pensare cosa avviene quando scrivete un programma? Non intendo dire come far capire al computer come si deve eseguire una particolare operazione. Mi riferisco alle differenze che ci sono tra ciò che pensate e il risultato finale, il programma eseguibile. Il compilatore, assemblatore o interprete che utilizzate per tradurre le vostre idee in sequenze di istruzioni, riveste un ruolo molto importante nel processo di programmazione.

Quindi, quale linguaggio usare? Quale fra i tanti è il migliore? Come tante cose nella vita, la risposta non è nè semplice nè veloce, senza considerare quali sono gli usi specifici di ogni singolo linguaggio. Ritenere che un linguaggio può essere più conveniente di altri per particolari operazioni non è sbagliato, ma non sempre è così.

Si tende in genere a dividere i linguaggi in categorie: il FORTH è utile per il controllo di processi, il C per scrivere sistemi operativi, il Lisp per le elaborazioni a liste, il Prolog per l'intelligenza artificiale e sistemi esperti e così via. Il problema di queste settorialità sta nel fatto che ogni categoria di linguaggi tende sempre più a autospecializzarsi: un programmatore ad esempio rifiuterà un linguaggio per un dato progetto se lo riterrà semplicemente di una categoria non adatta rispetto al problema. D'altra parte, egli proverà a utilizzare il suo linguaggio preferito per realizzare il programma.

Prendiamo ad esempio il LOGO. Anni fa cercai di approfondire la mia conoscenza di LOGO, e chiesi ad alcune persone dove potevo ottenere maggiori informazioni, oppure se erano previsti dei corsi. La reazione fu praticamente unanime: "Non c'è bisogno di un corso, semplicemente devi giocarci, la tartarughina è semplice da usare". Oppure: "Ah, il linguaggio con la tartaruga?". Quando finalmente trovai un posto in cui si tenevano corsi di LOGO, fui amareggiato del fatto che ciò che realmente veniva spiegato nel corso era l'uso della turtle graphics, e non si faceva neppure accenno alle potenzialità del linguaggio. Seymour Papert, ideatore del LOGO, intendeva i comandi della tartaruga come la parte più semplice del linguaggio anche se, per coloro che usano il computer, si sarebbe potuto limitare solo alla loro implementazione.

Questo è uno dei più macroscopici esempi di preconcetti che limitano l'uso di un linguaggio. Il LOGO, comunque, non è il solo linguaggio a soffrire di questi pregiudizi. Fra i linguaggi più popolari, ciascuno di noi tende a presentare il linguaggio favorito con ampi meriti, quasi con un entusiasmo religioso, dimenticando i benefici di altri. Queste prese di posizione sono spesso basate su percezioni e preconcetti, piuttosto che su fatti ed analisi approfondite. In qualsiasi discussione che abbia come argomento i linguaggi di programmazione, capita spesso di sentire cose come:
"Quello non ti permette di scendere a fondo, a livello di byte",
"E' troppo prolisso, troppi comandi", "E' troppo rigido", "Non è strutturato",
"Non permette un buon uso dell'I/O e dei numeri in virgola mobile", "Non ha gestione delle stringhe",
"E' troppo (rigido o libero) nel controllo sui tipi di dati".

In questa sede ho intenzione di avvalorare le mie ipotesi, invitando chiunque a criticarle. Sono d'accordo sul fatto che molti delle frasi accennate sopra siano vere, ma devono essere applicate al linguaggio corretto. Certo, è verissimo che il Pascal non permette di lavorare a livello di byte, che il C è criptico, il Modula 2 troppo prolisso, e anche che il FORTRAN non permette del facile I/O e il PILOT non ha il floating point. Appurato ciò occorre però precisare che le affermazioni descritte non sono dipendenti solo dal linguaggio, ma anche da come il programma è stato scritto, le implementazioni disponibili per il linguaggio e le risorse native della macchina che ospita il programma. Ciò può sembrare abbastanza vago ma contiene elementi sufficienti ad avvalorare le mie ipotesi e merita un'occhiata più approfondita.

Analizziamo alcuni problemi di programmazione. Prendiamo ad esempio dei problemi abbastanza semplici, che ogni programmatore medio è in grado di risolvere:

1. Generare una tabella ASCII
2. Controllare uno strumento, come ad esempio un tornio
3. Calcolare il risultato di una espressione

Se ora si riuscissero a raccogliere le proposte sui linguaggi da usare, e ognuno scrivesse i programmi indicati con il proprio linguaggio preferito, i risultati potrebbero essere estremamente interessanti. Nel caso della tabella ASCII, quale programmatore finirebbe per primo? Quale programma sarebbe il più corto, quale il più veloce? E' veramente importante che un linguaggio permetta una o più delle prestazioni qui sopra indicate? La risposta ovviamente dipende dall'uso che deve essere fatto del programma. Se il programma deve essere usato una volta sola, la velocità di stesura del codice dovrebbe essere più importante dello spazio che occuperebbe in memoria. Le dimensioni e la velocità di esecuzione sono invece elementi da prendere in considerazione se il programma deve essere usato spesso, come potrebbe essere nel caso in cui si intenda mantenere la tabella ASCII disponibile "on-line".

Il secondo esempio potrebbe essermi fatale in un confronto con coloro che programmano in FORTH e assembler. Essi potrebbero affermare che non vi sono altri linguaggi con i quali è possibile scendere a un così diretto controllo dell'hardware.

Nel caso dell'ultimo esempio si tratta effettivamente di program-

mi che devono essere utilizzati più volte, per cui velocità e dimensioni sono importanti. In generale può sembrare un problema semplice, ed alcuni linguaggi lo potrebbero gestire semplicemente. Altri linguaggi richiederebbero invece un parser completo, routine di input, controllo degli errori e così via. Questo problema è stato inserito apposta, come un trabocchetto.

In ognuno di questi casi, ci sono linguaggi che sono chiaramente individuabili come la scelta sbagliata, anche perchè alcuni non possono proprio eseguire certe operazioni, oppure occorrono troppi artifici per realizzarle.

Nel caso della tabella ASCII, se il programma deve essere usato una sola volta, un linguaggio ad alto livello può essere certamente più funzionale di un assemblatore, poichè il problema si presta a un approccio strutturato.
Fra i linguaggi ad alto livello, un interprete è certamente la scelta migliore, per la relativa velocità nel ciclo di codifica/verifica. La scelta sul linguaggio ad alto livello dovrebbe essere basata su altri parametri di valutazione.
Questi parametri sono il vero punto cruciale dell'articolo. Ci si dovrebbe orientare su un modo particolare di osservare il problema, piuttosto che sui linguaggi che si conoscono. Chiunque al giorno d'oggi possiede il Basic sulla propria macchina, e questa potrebbe essere una buona scelta. Il COMAL è un'altra buona scelta, così come l'Icon, il Lisp e l'APL. Il tutto dipende da come viene visto il problema. Se si ha la possibilità di valutare come può andare il programma in Basic, ma non si ha assolutamente idea di come potrà funzionare in APL, il Basic è ovviamente la scelta corretta, e vice versa.

Il controllo diretto dell'hardware è uno di quei campi in cui le controversie costituiscono la norma.
I programmatori FORTH sostengono che non esista linguaggio migliore del loro per realizzare programmi in quest'area; i programmatori assembler replicano la stesa cosa per i loro assemblatori. Questi però non sono i soli linguaggi disponibili: C e Modula 2, ad esempio, sarebbero validi sostituti.
Ambedue questi linguaggi sono in grado di scendere a livello hardware tramite l'indirizzamento assoluto, ed ambedue sono sufficientemente veloci per il controllo di applicazioni meccaniche, offrendo sicuramente una struttura più nitida di assembler e FORTH.
La scelta del linguaggio appropriato rispetto al programmatore che lo deve utilizzare è il vero strumento che permette di facilitare la risoluzione di un problema. Prima di cominciare a programmare con questo o quel linguaggio bisognerebbe considerare se si è soliti seguire un approccio top-down oppure bottom-up, se si ragiona più per termini astratti piuttosto che considerare il problema come un insieme di statement di un linguaggio, in modo da operare la scelta migliore.

Il terzo problema, il calcolo del risultato di una qualsiasi espressione, è stato scelto appositamente per le particolarità che presenta. Analizziamo ora il problema tenendo conto dei diversi linguaggi: occorre accettare i dati in input, verificare la validità dell'espressione, quindi scandire e valutare l'espressione, infine stampare il risultato. Ciò è ovviamente una operazione abbastanza complessa in assembler, FORTH, Basic ecc., mentre altri linguaggi possiedono un livello di routine più elevato che rendono il lavoro più semplice. L'input e il controllo sull'inserimento sono operazioni relativamente semplici per tutti i linguaggi di alto livello, così come è la stampa del risultato. La parte complessa del programma avviene nel momento in cui si deve valutare l'espressione: ciò richiede un particolare tipo di parser, e se avete già avuto esperienze su vari tipi di parser, scrivendo ad esempio un compilatore o leggendo i libri del caso, vi renderete conto che non è una operazione certamente banale.

In tutto ciò il trabocchetto si ritrova nel fatto che alcuni linguaggi dispongono della valutazione di stringhe tramite una singola istruzione. ARexx (vedi nota) è un linguaggio di comandi per Amiga simile, ma più potente, a quello supportato dal CLI: è del tutto simile al REXX disponibile sui mainframe IBM o Personal REXX, disponibile per IBM PC e compatibili. Il programma completo è:

```
arg x

interpret 'say' x
```

Confrontando ARexx con linguaggi del tipo assembler, C o Modula 2, ci si rende conto che è certamente il prodotto che meglio esegue il lavoro, a patto che già lo possediate e che lo abbiate utilizzato con un certo numero di programmi. Fra gli altri linguaggi ad alto livello se ne trovano pochi, l'eccezione è rappresentata da COMAL, LOGO e Lisp, che presentano caratteristiche simili.

Cosa si può trarre da tutto ciò? Il nocciolo della questione sta nel fatto che i linguaggi sono molto personali. Possono essere vicini al nostro modo di pensare, riflettendo il metodo adottato per risolvere i vari problemi di programmazione. Di conseguenza una critica al nostro linguaggio preferito viene spesso presa come una critica alle nostre scelte. Ricordatevi di questo la prossima volta che parlerete di linguaggi con un programmatore che preferisce un linguaggio diverso dal vostro. Ricordate che egli vede le cose in maniera diversa da voi e che ciò che non è adatto per voi può esserlo per qualcun'altro.

Se vi piace programmare, o state considerando di diventare programmatore, dovete certamente considerare che più linguaggi conoscete, meglio siete in grado di risolvere problemi e realizzare buoni lavori. Un carpentiere si scandalizza nel vedere un improvvisato hobbista usare un cacciavite come scalpello o svitare un bullone con una pinza, così il programmatore si scandalizza nel momento in cui non vengono usati gli strumenti corretti. Una delle cose piacevoli della programmazione consiste nel fatto che ci sono innumerevoli strumenti disponibili per compiere le stesse operazioni: si potrà quindi scegliere lo strumento più consono alla propria indole, permettendo di concentrare l'attenzione solo sul vero problema.

Tutto ciò che ho scritto non lascia troppo spazio alla moderazione. In altre parole, anch'io soffro degli stessi pregiudizi che qualsiasi programmatore possiede riguardo ai linguaggi che non gli sono familiari. In effetti, sono molto scettico riguardo ad alcuni lin-

guaggi ed alla loro semplicità di programmazione. Non mi voglio riferire a linguaggi per applicazioni speciali, utilizzati per un ristretto numero di applicazioni, ma tratterò un pò di quei linguaggi che sono considerati dalla maggior parte dei programmatori adatti per un uso generale.

Il C, sponsorizzato dai suoi seguaci come stringato, trasportabile, efficiente, estendibile, flessibile e, a detta dei più anziani, "il linguaggio che ti fà ricrescere i capelli e ti fa sentire più giovane", non rientra nei miei ideali di buon linguaggio.

Le mie critiche sul C riguardano:

* L'inconsistenza delle regole applicate.

* Non si può considerare un linguaggio né di alto né di basso livello.

* E' difficile da leggere, a causa dell'eccessiva punteggiatura per gli operatori e i delimitatori a blocchi di struttura.

* Troppe caratteristiche del linguaggio sono lasciate a chi scrive il compilatore

* I moduli separati non sono in realtà del tutto separati, e possono interagire fra loro in modo catastrofico nel momento in cui si apportano modifiche.

La scalata all'apprendimento del C è ripida e nessuno la può portare a termine senza per lo meno sbucciarsi le ginocchia. Per prima cosa ci si deve sempre scontrare con le trappole tese al programmatore che sono portate sia dall'implementazione del compilatore che da ciò che viene indicata come "definizione del linguaggio". Una definizione che consiste piuttosto in una incoerente, contradditoria e incompleta specifica chiamata "The C programming Language", il libro di Brian Kernighan e Dennis Ritchie (edito in Italia dal Gruppo Editoriale Jackson con il titolo "Linguaggio C").

A tutt'oggi, il comitato per gli standard ANSI sta lavorando per definire un C standard. Certamente è un nobile sforzo, che potrebbe realmente portare a uno standard, dando la possibilità ai realizzatori di compilatori di attenersi alle regole. Ci potrebbe comunque volere un po' di tempo, visto che la bibbia di Kernighan & Ritchie è stata interpretata in mille maniere differenti.

Il C è stato paragonato a un macro assembler molto sofisticato. A mio avviso un assembler è per lo meno un linguaggio onesto, che non pretende di essere considerato di alto livello. Il C vi nasconderà dei dettagli, fornendovi delle macro come l'equivalente per alcune costruzioni molto potenti, ma allo stesso tempo nascondendovi ciò che sarebbe meglio porre in evidenza, soprattutto se si considera che è un assembler molto sofisticato. Come linguaggio ad alto livello, non fa molta strada. Vi dirà quando avrete causato degli errori nell'assegnamento fra tipi, dimenticando di rilevare altri errori più avanti, ma ve lo dirà solo in alcuni casi, assumendo in altri casi che voi sappiate quello che state facendo.

Prendiamo come esempio il casting. Modula 2 è spesso criticato per essere troppo ristretto nel controllo sui tipi. Gli aficionados del C sostengono che non vogliono che il compilatore li "tenga per mano" e non accettano nemmeno il fatto che il casting effettuato automaticamente dal compilatore possa avere risultati disastrosi.

Questi risultati non vengono visti subito, ma quando il valore contenuto in una variabile sembra variare in un modo inaspettato, la cosa non sembra preoccupare il programmatore C.

Ma il compilatore ideale dovrebbe controllare tutti i tipi definiti o al limite non controllarne nessuno?
In un linguaggio come il Modula 2, il compilatore controlla in fase di compilazione tutti gli errori di tipo, e non permette che si continui a lavorare se il problema non è stato risolto in uno dei due modi che vi indicherò.

Il primo modo consiste nel correggere l'errore riscontrato nel tentativo di assegnamento incompatibile.

Il secondo modo consiste nel costringere esplicitamente il tipo (operazione chiamata 'casting' in C), ad accettare esattamente l'operazione che si vuole eseguire senza rilevare errori. In altre parole, il compilatore vi costringe a prendere atto delle conseguenze che avranno le vostre azioni, lasciandovi il controllo del codice, invece di restare alla mercè del compilatore. In questo modo è un po' meno facile scontrarsi con un bug nascosto ad applicazione completata. E' stato detto che "tutto è possibile in C, fino al run time", e questa affermazione racchiude un senso di verità.

In fondo, considero l'operato di Kernighan & Ritchie un lavoro intelligente, senza sconfinare però nel mondo dei grandi sistemi e dei programmi applicativi. La generazione di codice contenente dei bug nei grandi sistemi operativi conferma che il C è un linguaggio adatto per i piccoli progetti di programmazione. A parte il campanilismo, i dati non possono rimanere "nascosti", né i moduli possono essere trattati indipendentemente se essi causano cambiamenti nell'uso di altri moduli.

Non dubito di avervi un po' innervosito con le mie affermazioni, e sarebbe realmente molto interessante sapere cosa vorreste dire riguardo ai vari linguaggi disponibili. Per la cronaca, i miei linguaggi preferiti sono assembler, COMAL, Modula 2 e Icon, non in ordine di preferenza, ma in rapporto alle applicazioni da realizzare.

Ma allora voi quali linguaggi usate? Perchè? Cosa avete da dire sui miei preferiti?

NOTA: ARexx è disponibile presso: William Hawes, P.O. Box 308, Maynard, MA 01754, Tel.(617)568-8695 (Rete telefonica USA).

Il prezzo è di $49.95

È IN EDICOLA
IL 1° FASCICOLO

dBIII e plus

PC MASTER

CORSO COMPLETO IN AUTOISTRUZIONE ALL'USO DEL PERSONAL COMPUTER

Questo corso in autoistruzione, fornisce al lettore tutte le conoscenze necessarie per permettere di utilizzare efficacemente dBase III e dBase III plus, i più noti e diffusi data Base. Il corso è strutturato in due momenti integrati tra loro: testo e software interattivo.

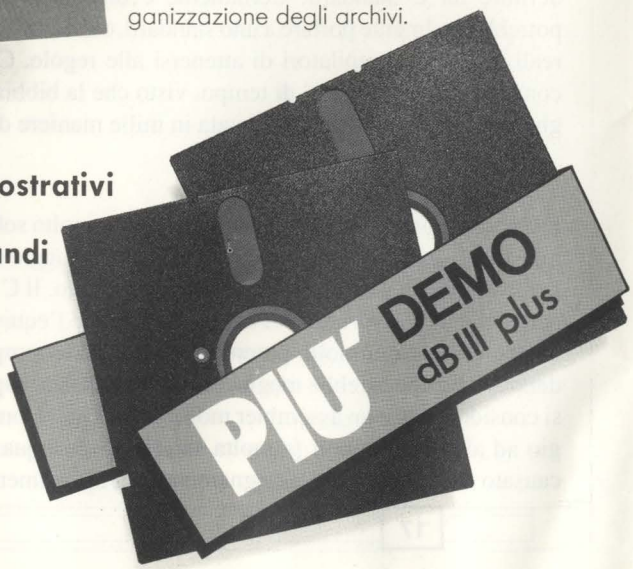
DISPONIBILE
NELLE VERSIONI
5¼" e 3½"

Il testo, con metodologia semplice e graduale, guida il lettore ad una completa comprensione e padronanza dei concetti fondamentali, permettendo l'apprendimento anche a coloro che non hanno ancora acquisito una preparazione specifica sull'argomento.

Il software, simulando le caratteristiche e le situazioni operative del dBase III e dBase III plus, permette di esercitarsi immediatamente sugli argomenti trattati, fornendo in tal modo quella interazione pratica indispensabile all'apprendimento.

Al fine di rendere la trattazione più esaustiva possibile è prevista una SEZIONE ARGOMENTI, in cui vengono trattati temi teorici relativi alla gestione delle BASI DI DATI, non strettamente legati all'uso del dBase III, ma la cui conoscenza è utile per una comprensione dei criteri più generali che stanno alla base della organizzazione degli archivi.

Con il 1° fascicolo, troverete i dimostrativi dei programmi, che permettono di analizzare in dettaglio tutti i comandi e le possibilità di questi pacchetti.



GRUPPO EDITORIALE
JACKSON

$13 Frequenti errori dei programmatori di Amiga ... e come evitarli

di Bryce Nesbitt

Bryce Nesbitt é un hacker di Berkeley, California, che ha lavorato a molti progetti commerciali su microcomputer della Commodore. Alcuni dei suoi prodotti più noti sono il 1541 Flash e l'interfaccia QuickSilver IEEE, entrambi distribuiti da Skyles Electric Works. I progetti sviluppati su Amiga includono PopToFront e altri programmi di pubblico dominio.

Questo articolo è un "conto alla rovescia" di $13 tra i più frequenti errori in cui ci si può imbattere nello scrivere software commerciale per Amiga. I colpevoli appartengono sia alle software house artigianali sia alle vere e proprie compagnie multinazionali. Aver ben presente questi potenziali problemi può notevolmente migliorare i vostri programmi.

$13> Alcuni programmi modificano con valori casuali i primi 50 byte circa della memoria. Questo può causare una miriade di sottili problemi che possono rivelarsi nient'affatto facili da identificare. In generale, la situazione si verifica a causa dell'uso di un puntatore che non è mai stato opportunamente inizializzato e che quindi finisce con l'avere valore zero.

Un programma disponibile nel pubblico dominio chiamato **MemWatch** (di John Toebes della Software Distillery) può essere posto in background a operare in maniera continuativa una scansione delle prime locazioni di memoria. Se rileva un accesso illegale, ne avverte l'utente con un messaggio. (MemWatch é disponibile sul Fish Disk #48 o sul Transactor Amiga Disk di questo numero).

Alcuni programmi non funzioneranno anche se si pone solamente un valore strano come $00DFF001 nella locazione zero.

$12> Come default, il DOS pone requester relativi a un vostro processo sullo schermo del Workbench. Se il vostro programma apre uno schermo custom, deve indirizzare su di esso il **DOS**. Questo frammento di programma mostra come fare:

```
#include "libraries/dosextens.h"
  ...
struct Process  *mproc;
struct Window   *window;
APTR            temp;

  ...
   mproc = (struct Process *)FindTask(0L);
   temp = mproc->pr_WindowPtr;
   /* salva il valore iniziale*/
  mproc->pr_WindowPtr = (APTR)window;
                        /* poni in pr_WindowPtr un
                         * puntatore a una qualsiasi
                         * finestra sullo schermo custom
                         */
  ...
  ...
  mproc->pr_WindowPtr = temp;
                        /* ripristina il valore iniziale
                         * prima di chiudere la finestra */
  ...
  CloseWindow(window);
```

Per maggiori dettagli si consulti l'AmigaDOS Technical Reference Manual.

$11> Bisogna fare molta attenzione nell'uso di tutti i messaggi "**VERIFY**" dell'Intuition. Questi, infatti, risultano piuttosto bizzarri da usare in alcune circostanze. Per i dettagli del caso si legga l'Amiga Enhancer Software Manual. Una circostanza è importante quanto basta per essere menzionata: quando è abilitata una funzione **VERIFY**, non si deve mai chiamare la funzione AutoRequest() o qualsiasi altra funzione che possa indirettamente o in qualche altro modo chiamare il DOS. Questo comprende OpenLibrary(), OpenDevice() e OpenDiskFont().

$10> Se il nostro programma tratta messaggi di tipo **RAWKEY**, é arrivato soltanto a metà strada! I "raw keycodes" forniscono solo un'informazione posizionale e non l'effettivo valore associato al tasto. Per ogni paese in cui viene venduto, l'Amiga ha una sua "keymap" specifica. I "raw keycodes" devono essere ulteriormente elaborati per ottenere ciò che veramente significano. Tale elaborazione comprende due passi:

1) Si apre il console device e si estrae il suo puntatore alla "libreria". Fatta eccezione per lo strano metodo di estrazione, il puntatore funziona nella stessa maniera di quello a una vera e propria library.

Attenzione: Alcune versioni del file "functions.h" dell'Aztec C contengono un "bug" che dà luogo a un errore di simbolo ripetuto. Per correggerlo, non dovete far altro che rimuovere la linea che contiene la parola "ConsoleDevice()".

2) Si inviano tutti i messaggi **RAWKEY** ricevuti da Intuition alla funzione DeadKeyConvert() che qui riportiamo:

```
struct ConsoleDevice   *ConsoleDevice; /* puntatore alla "libreria" */
```

```
struct IOStdReq      ioreq;      /* IOStdReq vuoto, a zero  *
...
   /* Apri una console senza connettervi Window o port */
   if (OpenDevice("console.device",-1L,&ioreq,0L))
      exit(21);
   ConsoleDevice = (struct ConsoleDevice *)ioreq.io_Device;
...
/*
 * DeadKeyConvert(). Dall'Amiga Enhancer Manual.
 *
 * Prende un messaggio Intuition RAWKEY e lo elabora.
 * Il risultato è costituito dai valori associati ai tasti o dalle
 * conversioni dei tasti ed è posto nel buffer specificato.
 *
 * Restituisce -2 se non si trattava di un messaggio RAWKEY,
 * -1 se il buffer è andato in "overflow", oppure il numero di
 * caratteri convertiti nel caso in cui tutto sia andato bene.
 */

long   DeadKeyConvert(msg,kbuffer,kbsize,kmap)
struct IntuiMessage  *msg;   /* IntuiMessage da convertire   */

UBYTE *kbuffer;          /* buffer da riempire          */
long   kbsize;           /* dimensione del buffer       */
struct KeyMap *kmap;     /* ptr a una keymap custom o zero */

{
static   struct   InputEvent                ievent   =
{NULL,IECLASS_RAWKEY0,0,0};

   if (msg->Class != RAWKEY)  return(-2);

   /* Trasferisci il contenuto dell'IntuiMessage in un InputEvent
*/
   ievent.ie_Code = msg->Code;
   ievent.ie_Qualifier = msg->Qualifier;
   /* Ottieni i codici precedenti dalla locazione puntata da IAd-
dress
    * Questo puntatore "magico" è valido fino al momento in cui
    * si replica all'IntuiMessage
    */
   ievent.ie_position.ie_addr = *((APTR *)msg->IAddress);

   return(RawKeyConvert(&ievent,buffer,kbsize,kmap));

}
```

$F> Ci sono due tipi di memoria sull'Amiga, contraddistinti da tre attributi. E' estremamente importante che questi ultimi siano impiegati appropriatamente.

CHIP è il tipo di memoria a cui possono accedere i chip custom. Alcune delle cose che devono essere allocate in CHIP ram sono:

dati relativi agli sprite
dati relativi alla grafica (comprese le immagini dei gadget)
buffer per l'audio
buffer per il device trackdisk

Se non ci si ricorda di specificare l'attributo CHIP per zone di memoria che devono contenere tali dati, il programma non funzionerà su una macchina con espansione di memoria. Un programma che contiene in un suo segmento dati relativi a gadget o immagini, deve assicurarsi che tali dati finiscano nell'appropriata classe di memoria. Questo può essere ottenuto in cinque modi:

1) Si copiano i dati in una zona di CHIP memory appena allocata.

2) Si usa un linguaggio che supporti la possibilità di specificare il tipo di memoria per i dati.

3) Per utenti del Lattice C, si impiega la utility **ATOM.**

4) Per utenti del Manx C, si fa il linking specificando l'opzione +Cd, che costringe tutti i segmenti dati ad essere caricati nella memoria CHIP.

5) Per correggere programmi già esistenti, si usa la utility **Fix-Hunk** reperibile sul Fish disk #36 o sul Transactor Amiga Disk #1, che contiene i programmi relativi a questo numero della rivista.

FAST è il tipo di memoria a cui i chip custom NON possono accedere. Nel caso in cui sia presente, diviene la zona in cui per default vengono ospitati il vostro programma, i vostri dati, lo stack, le chiamate ad AllocMem() e tutto ciò che non richieda CHIP memory. Dal momento che alcuni Amiga non hanno memoria FAST, non si dovrebbe mai specificare l'attributo FAST.

PUBLIC è forse l'attributo più frainteso. Esso può venir combinato con qualsiasi altro flag e specifica che la memoria non può essere spostata o resa in qualsiasi maniera inaccessibile. Poiché gli Amiga della prima generazione non hanno la capacità di fare queste cose, il bit per tale attributo è stato riservato per ragioni di compatibilità.
Qualsiasi blocco di memoria che viene usato da interrupt o che verrà impiegato da più task deve essere allocato con questo bit settato.
Si ricordi a tal proposito che i "device" sono task indipendenti e, quindi, le strutture impiegate nell'I/O o per comunicare con i device vanno allocate in memoria di tipo PUBLIC.

Nota: Poiché lo stack rappresenta memoria privat a è illegale allocare temporaneamente nello stack una struttura che richiede memoria
PUBLIC.

CLEAR specifica che il blocco di memoria venga posto a zero

prima di essere concesso. Questo attributo può essere combinato con tutti gli altri flags.

LARGEST é un attributo utile solo nella chiamata alla funzione AvailMem(). La funzione, in tal caso, ritorna la dimensione del più grande blocco di memoria libera. (A causa del multitasking, il valore restituito fornisce solo un'idea della dimensione massima del blocco che può essere realmente allocato.)

Esempi:

```
AllocMem(MEMF_CHIP|MEMF_CLEAR|MEMF_PUBLIC);
/* Trova memoria CHIP e pubblica, azzerandola prima di fornirla
*/

AllocMem(MEMF_CLEAR);   /* Trova memoria di qualsiasi
tipo e la azzera */

AllocMem(0L);           /* Senza particolarità, fornisce
memoria di tipo qualsiasi */
```

$E> Stiamo attenti ai seguenti "bugs":

-Chiamando **WindowToFront()**, mentre l'utente ha agganciato un'icona, si blocca la macchina.

-La funzione **ScrollRaster()** (usata verso destra o sinistra) in una SuperBitmap window "mangia" memoria se non c'é un TmpRas precedente.

-Il **serial.device** va in crash se i caratteri arrivano alla porta seriale con un baud rate notevolmente diverso da quello previsto.

-Si può lavorare con **rp_Mask** solo quando i layer sono "locked".

-Talvolta le chiamate **Delay(0)** o **WaitForChar(X,0)** possono mandare in crash.

-argc e argv sono scambiati fra di loro nei programmi lanciati da Workbench quando il linking é stato fatto con alcune versioni di **Astartup.obj**.

-Le funzioni relative all'uso dei semafori hanno l'interfacciamento con il C difettoso. Un esempio di correzione é disponibile presso la Commodore.

-L'uscita del comando **Text()** viene troncata secondo la larghezza del rastport anche quando la scrittura é iniziata prima del bordo sinistro del rastport.

-Stiamo attenti a usare **ReportMouse()!** L'ordine in cui vengono specificati gli argomenti dipende dal linguaggio impiegato:

Aztec C - ReportMouse(Window,Boolean);
Lattice C - ReportMouse(Boolean,Window);
Assembler - Window in d0, Boolean in a0.

-CINIT e UCopperListInit() non allocano una UCopList se viene loro passato un puntatore nullo. Dobbiamo prenderci personalmente cura di allocare un buffer di classe: MEMF_CHIP|MEMF_PUBLIC|MEMF_CLEAR di 12 bytes e passarne il puntatore. Comunque, FreeVPortCopList() si occuperà di deallocare tale area per nostro conto.

$D> Per mantenere la compatibilità con gli Amiga che hanno come processori un 68010, un 68020 o un 68030, assicuriamoci di:

-non usare mai l'istruzione assembly "**MOVE <SR>,EA**". Utilizziamo la funzione dell'Exec **GetCC()** per esaminare i "condition codes" del processore in maniera compatibile.

-non utilizzare gli otto bit superiori di un puntatore a 32 bit.

-tenere conto del fatto che ogni membro della famiglia 68000 supporta un differente stack frame: non fate dunque assunzioni sul formato dello stack senza controllare su quale **CPU** stia girando il programma.

-non avere scritto del codice che modifica se stesso. Ciò, infatti, non sarà compatibile con i sistemi futuri sotto molti aspetti.

$C> Si può aumentare l'affidabilità apparente dei propri programmi evitando di utilizzare la memoria di sistema fino a quando questa non sia più disponibile. Un semplice accorgimento per aggiungere un po' di spazio vitale consiste nel sostituire tutte le chiamate ad AllocMem() con chiamate alla funzione SafeAllocMem() che riportiamo qui sotto.

Nel caso dovessimo informare l'utente che non c'é più memoria libera, ma non ci fosse memoria sufficiente per usare un requester o un alert, potremmo semplicemente mettere il messaggio **"*** MEMORIA ESAURITA ***"** (magari in rosso) nel titolo della nostra window o del nostro screen. Questa operazione non richiederebbe alcuna allocazione dinamica.

```
#include "exec/memory.h"

...
#define PANIC_FACTOR_CHIP   25000L

...
APTR   SafeAllocMem(size,flags)
long   size;
long   flags;
{
register   APTR   p;


 if (p = (APTR)AllocMem(size,flags))
 if (AvailMem(MEMF_CHIP) < PANIC_FACTOR_CHIP)   {
       FreeMem(p,size);
       return(0);
    }  /* Non c'é memoria; niente da fare! */
 return(p);
```

}

$B> Facciamo attenzione ai valori restituiti! I programmi andranno molto probabilmente in crash se la segnalazione di un errore viene ignorata e/o la risposta a una condizione anomala non funziona come previsto. Errori di questo tipo sono assolutamente troppo frequenti, soprattutto con l'allocazione di memoria.

$A> Siamo gentili con coloro che usano i nostri programmi: mettiamo almeno un briciolo di "drag bars" e di "page gadgets" su tutti i nostri schermi custom. Implementiamo il tasto **HELP,** soprattutto se abbiamo già un help on-line.
Se apriamo uno string gadget in cui l'utente debba scrivere qualcosa, utilizzeremo la funzione **ActivateGadget()** della V1.2 per risparmiargli la noia di dovervi clickare dentro. Se normalmente programmiamo con lo stack avente una dimensione diversa da quella di default, controlliamo che il programma funzioni anche con uno stack normale. Imitiamo la sistemazione dei gadget degli AutoRequest(): "Cancel" o "No" a destra, "Retry" o "Yes" a sinistra.

Proviamo a far girare il nostro programma con il font **Topaz-60** selezionato da Preferences. Se il programma non può lavorare con uno qualsiasi, specifichiamo almeno con quale font o con quale insieme di font attributes il programma funzioni.

$9> Se siamo interessati solo alla posizione attuale del mouse, non dovremmo rispondere a ogni mouse event, ma solo al più recente. Un buon esempio di questa tecnica può essere ritrovato nel capitolo del manuale di Intuition che descrive l'**IDCMP**.

$8> Se ci arriva un messaggio di un tipo, classe, codice che non conosciamo, ignoriamolo. Non è detto che se qualcosa non è nè A nè B, debba essere necessariamente C; qualcuno un giorno potrebbe aggiungere nuovi tipi o classi e procurare grossi problemi alla nostra applicazione.

$7> Il sistema operativo dell'Amiga é in costante evoluzione. Se utilizziamo una qualsiasi delle nuove funzioni, avremo la responsabilità di controllare il numero della versione del sistema operativo. Ecco una lista completa:

0: qualsiasi
30: V1.0
31: V1.1
32: V1.1 (con PAL)
33: V1.2
34: V1.2 (con boot da hard-disk o da rete)

La maniera più semplice per accertarsi che il programma stia girando su una versione sufficientemente recente del sistema operativo, consiste nello specificare un valore minimo quando si aprono delle librerie. Visto che il KickStart V1.1 è ormai completamente obsoleto, è meglio specificare almeno la versione 33 nei nuovi programmi.

$6> Se apriamo uno schermo custom, ricordiamoci che la larghezza e l'altezza dello schermo dipenderanno dalla configurazione che l'utente impiega. Il programma di utilità **MoreRows** può essere usato su qualunque Amiga per aumentare l'area di schermo disponibile. Le macchine **PAL** hanno un display che è intrinsecamente più alto di 56 linee rispetto a quello **NTSC**. Si noti, infine, che i prossimi Amiga renderanno disponibili risoluzioni ancora maggiori.

Con la V1.2 ci sono diverse possibilità per impiegare l'eventuale spazio aggiuntivo a disposizione:

1) Mettere la costante **STDSCREENHEIGHT** nella variabile NewScreen.Height. Questo fa sì che lo schermo si apra magicamente al suo massimo. A questo punto, se vogliamo sapere quanto é alto in definitiva il nostro schermo, dovremo esaminare la struttura Screen.

2) Le variabili pubbliche GfxBase->**NormalDisplayRows e NormalDisplayColumns** contengono le dimensioni dello schermo del Workbench non interallacciato. Queste ci permetteranno di estendere la larghezza del nostro schermo fino a raggiungere quella del Workbench, ma fate attenzione al fatto che se allargate oltre le 640 colonne, gli sprite potrebbero risultare non più disponibili (tranne il pointer del mouse). Potremo anche scoprire l'aspect ratio (rapporto fra l'altezza e la larghezza di un singolo pixel) dello schermo esaminando **GfxBase->NormalIDPMX** e **NormalIDPMY**.

3) La funzione di Intuition **GetScreenData()** copia i parametri dello schermo in un buffer. Successivamente potremo utilizzare queste variabili per aprire il nostro schermo.

GetScreenData() può anche essere utilizzata per aprire finestre che non coprano la barra del titolo appartenente allo schermo sottostante. La soluzione più semplice è quella di sottrarre l'altezza della nostra finestra da quella dello schermo e aprire quindi la finestra alla locazione così ricavata. Come ultima possibilità potremo aprire la nostra finestra un pixel più in basso rispetto allo schermo, per permettere all'utente di accedere alla barra di spostamento dello schermo e ai suoi gadget di profondità.

Il programma **Morerows** è disponibile sul disco Fish #54 oppure sul disco di Transactor relativo a questo numero della rivista.

$5> E' impossibile creare un AutoRequest() che abbia solo possibilità di risposta positiva. Mettere a zero la variabile NegativeText è un errore.

$4> I campi **MaxWidth** e **MaxHeight** della struttura NewWindow vengono inizializzati spesso con costanti arbitrarie come 640 e 200. In molti casi questo è sbagliato.
La disponibilità di schermi più larghi del normale è molto frequente, addirittura comune, fra gli utenti di Amiga.Settiamo quindi questi campi con il valore massimo che il nostro programma può supportare, anche se questo valore è maggiore del più grande schermo che abbiate mai visto. Se non c'é ragione per limitare l'espansione di una finestra, mettiamo entrambe queste variabili a -1 (o ~0).

$3> Usiamo il programma **PM** che si trova nel disco **Extras** V1.2. Questo monitor di processi ci mostrerà istantaneamente se il nostro programma sta utilizzando abusivamente la CPU mentre dovrebbe attendere qualche evento esterno.

$2> Esiste un flag per il controllo del refresh nella struttura **NewWindow,** che dovrebbe essere utilizzato in molti casi. Perfino le finestre **SMART_REFRESH** possono generare dei refresh se possiedono un gadget di dimensionamento.

Se non abbiamo scritto del codice apposta per gestire queste situazioni, DOBBIAMO usare il flag **NOCAREREFRESH**. Se invece esiste il codice per gestire il refresh, assicuriamoci di utilizzare le chiamate a **Begin/EndRefresh()**. Se nessuno di questi due accorgimenti verrà rispettato, Intuition si troverà in un limbo non definito e rallenterà le operazioni per tutte le finestre dello schermo.

Il popolare programma "PowerWindows" è uno dei principali imputati che non utilizzano questo bit quando invece dovrebbero.

1> Troppi programmi presentano delle disfunzioni nella gestione della memoria. Facciamo questa prova: lanciamo il nostro programma, poi usciamo.

Controlliamo la memoria disponibile. Lanciamo ancora il nostro programma, poi usciamo. Adesso verifichiamo che la memoria disponibile sia esattamente la stessa. Se c'é una qualsiasi differenza, vuole dire che il programma contiene un errore. Lanciando il Workbench con "LoadWB -debug" avremo a disposizione un menu supplementare invisibile, che sarà di aiuto nel rimuovere le librerie ed i font residenti utilizzati, per non influenzare il computo della memoria libera.
Questa prova è particolarmente efficace per scoprire problemi che altrimenti verrebbero ignorati. Effettuatela sempre sia da **CLI** sia da Workbench.

Se si verifica un ammanco nella quantità di memoria disponibile, controllate che tutta quella che viene allocata venga poi deallocata, e che qualsiasi cosa aperta venga poi chiusa. Le funzioni del **DOS** sono particolarmente insidiose, in quanto alcune restituiscono dei lock. Tre indizi che rivelano sicuramente la presenza di un problema sono:

1) Perdita di memoria.

2) L'icona di un disco non "sparisce" quando il disco viene rimosso.

3) Un file o una directory generano un errore 121 "oggetto in uso", se viene fatto un tentativo di cancellarli.

$0,$FFFF e oltre?

La mia lista è finita. Se avete un vostro elenco degli errori di programmazione preferiti, che pensate debbano essere resi di pubblico dominio, spartitelo con gli altri.

Entriamo in nuove dimensioni Allocazione di memoria e liste: un'introduzione

di Rob Peck

L'utilizzo delle liste dell'Exec aiuta a fornire ai vostri programmi lo spazio per crescere!

Rob Peck è l'autore dei manuali Amiga ROM Kernel e Amiga Hardware e del libro "Programmers' Guide To The Amiga (SYBEX, 1987). Può essere raggiunto presso: DATAPATH, P.O. Box 1828, Los Gatos, CA, 95031-1828.

Avete mai creato un programma per gestire un quantitativo specifico di dati, solo per scoprire poi che avreste voluto utilizzarlo con più elementi di quanti ne avevate originariamente previsti? Questo normalmente significa che dovrete modificare il programma per aumentare le dimensioni degli array che utilizzate per immagazzinare i dati. Invece di fare questo, tuttavia, potreste considerare l'utilizzo di dati creati dinamicamente, permettendo al programma di gestire una dimensione qualsiasi degli array. Fino a quando ci sarà memoria sufficiente per gestire il tutto, il programma non dovrà essere più cambiato.

Questo articolo presuppone che abbiate una familiarità con la programmazione in BASIC, e che abbiate già visto qualche programma C nei libri o nelle riviste dedicate ad Amiga. A meno che non abbiate studiato il linguaggio C, quei listati saranno stati probabilmente di difficile comprensione. Per introdurre i concetti di memorizzazione dinamica dei dati cominciamo con una spiegazione delle strutture dati in C, quindi mostreremo come queste strutture dati vengono allocate dinamicamente, vedremo come poter costruire liste di strutture dati e infine come tenere tutto sotto controllo.

Come si farebbe in BASIC

Cominciamo pensando a come normalmente si gestiscono i dati quando si programma in BASIC. Supponiamo che vogliate creare una raccolta di dati sulle persone, magari una mailing list. Se avete 100 persone in questa lista potreste creare un programma che definisce le seguenti variabili:

```
DIM NOME$(100), INDIRIZZO$(100)
DIM PAESE$(100), STATO$(100), CAP(100)
```

Per lavorare con questi dati nel vostro programma, dovreste usare una variabile INDICE per riferirvi a un particolare elemento di ciascun array, assegnando valori agli elementi di ogni array man mano che vi spostate al loro interno. Per esempio:

```
INDICE = INDICE + 1

NOME$(INDICE) = "Mario Rossi"
INDIRIZZO$(INDICE) = "via Lario 123"
```

e così via.

Perchè cercare un'altra strada?

Prima di imparare il Pascal e il C, pensavo che tutti utilizzassero questo metodo per organizzare gruppi di dati in una serie di array indicizzati. Ciò valeva per il FORTRAN e quindi lo trasportai anche in BASIC. Perchè qualcuno avrebbe dovuto farlo in un'altra maniera?

Ho invece imparato che ci sono molte ragioni perchè si possa desiderare di farlo in altri modi. L'organizzazione dei dati in blocchi in relazione diretta, invece che in diversi array separati fra loro, ne rende più facile la manipolazione: è più facile pensarli come un'entità. Se si devono gestire gruppi di dati è molto facile lavorare con array composti da questi oggetti. Ogni elemento nell'array sarà composto da diversi elementi più piccoli, ognuno posto in una posizione ben determinata all'interno del proprio gruppo.

Come ficcarsi nei guai con un numero di elementi fisso

I comandi DIM che abbiamo visto prima impongono un limite fisso sulla quantità di dati che il programma può gestire. Se si sono già creati 100 elementi e si ha bisogno a questo punto di aggiungerne un'altro, occorre modificare il programma per consentire l'uso di un maggior numero di elementi.
Alcuni linguaggi di programmazione permettono di creare elementi in maniera dinamica. Il linguaggio chiede al sistema operativo di riservare una zona di memoria abbastanza grande da contenere la definizione completa di un elemento, per poi comunicare dove si trova questa zona. Possiamo aggiungere un altro elemento a un gruppo che stiamo formando, semplicemente copiando i dati nella zona di memoria appena allocata.

Questo è esattamente quello che faremo utilizzando le funzioni per l'allocazione di memoria disponibili sull'Amiga procurando lo spazio necessario a ogni nuovo elemento, durante la sua creazione. Costruiremo una struttura dati. Sceglieremo cioè un nome e lo utilizzeremo per rappresentare il modo in cui è organizzato un particolare gruppo di dati. Successivamente vedremo come si possa chiedere al sistema operativo di riservare la memoria con la quale dobbiamo lavorare. Infine aggiungeremo questo nuovo elemento a una lista di elementi che è possibile manipolare in qualsiasi modo si desideri.

Inizieremo col definire una struttura dati che è strettamente equivalente al precedente esempio in BASIC. Anzichè tentare di simulare le strutture dati in BASIC, utilizziamo direttamente il C.

Qualsiasi cosa sia racchiusa dalla coppia di caratteri "/*" e "*/" è un commento che viene interamente ignorato dal compilatore C. I commenti sono esattamente come il comando REM del BASIC, ma possono occupare più linee se il programmatore lo desidera. Una nota a margine: quando programmo, spesso inserisco così tanti commenti che la gente si lamenta di non riuscire più a trovare il codice vero e proprio. Sigh... non posso accontentare tutti.

```
/* assumiamo che questo si trovi in un file chiamato "etichetta.h" */

struct Etichetta {
  struct Node Lnode;  /* collega gli elementi */
                      /* occupa 14 byte */
  char nome[30];      /* riserva 30 byte per nome$ */
  char indirizzo[50]; /* e 50 byte per indirizzo$ */
  char paese[20];     /* e 20 byte per la città */
  char stato[2];      /* e due per lo stato */
  LONG cap;           /* 4 byte per il valore intero del cap */
};
```

Questa viene chiamata "dichiarazione" della struttura dati Etichetta. Una struttura dati è una organizzazione di dati alla quale si può dare un singolo nome. Si possono costruire strutture dati differenti a seconda dell'applicazione che si deve sviluppare.

La precedente dichiarazione comincia con questa linea:

```
struct Etichetta {
```

Per il linguaggio C significa che si sta definendo una singola struttura dati alla quale ci si riferirà in seguito con il nome "Etichetta". Le parentesi graffe indicano il punto iniziale e finale di uno specifico gruppo di elementi.

In molti casi, quando vedrete definizioni di strutture dati tipo quelle che si possono trovare nei file "include" di Amiga (quelli del compilatore C i cui nomi finiscono con ".h"), saranno fatte in questo modo:

```
struct nome {
/* qualcosa qui */
};
```

Ciò significa che il file sta solamente DEFINENDO la struttura dati e che il compilatore C non riserverà alcuna locazione di memoria per essa nel vostro programma, a meno che non si utilizzi effettivamente un elemento di dati di quel tipo. Questo file può considerarsi come un dizionario nel quale il compilatore cercherà la definizione di una struttura dati quando (e solo se) questa definizione è necessaria. A volte, invece, si possono trovare dichiarazioni in questa forma:

```
struct nome altro_nome = {
/* qualcosa qui */
};
```

Notate il segno di uguale (=) in quest'altro tipo di dichiarazione. Quando è presente l'uguale, il compilatore sa che non è una definizione, ma che si tratta di un oggetto per il quale il compilatore deve riservare una zona di memoria per immagazzinarvi i dati.

La seconda linea nella struttura dati "Etichetta" definita prima, è un'altra struttura dati chiamata Node (Nodo). Il C permette di incorporare strutture dati all'interno di altre strutture dati. Sorvoliamo su questo, per il momento, in quanto ha a che fare con il formare liste di elementi di dati; torneremo sull'argomento più tardi.

Gli elementi successivi nella dichiarazione di Etichetta sono:

```
char nome[30];
char indirizzo[50];
e così via...
```

Una dichiarazione di tipo "char" comunica al compilatore di riservare uno spazio in memoria per immagazzinare dei caratteri (come lettere e numeri). Così una dichiarazione come char[30] significa: "riserva 30 locazioni di memoria per immagazzinare i caratteri che costituiscono la parte relativa al nome di Etichetta".

Questo è simile al comando DIM del BASIC, in quanto pone un limite fisso alla quantità di memoria riservata. Un avvertimento: in molti casi il compilatore C non impedirà di scrivere in questa zona una stringa (per esempio) così lunga da fuoriuscire dallo spazio riservato. In effetti questo è spesso uno dei motivi che causa le visite del Guru: qualche programma è andato a scrivere inavvertitamente nella parte sbagliata della memoria.

L'ultima parte della dichiarazione di Etichetta è:

```
LONG cap;
```

Si è scelto di utilizzare una variabile LONG (intero lungo) invece di una stringa di caratteri. Una variabile LONG è un numero intero grande, il cui valore occupa 4 byte di memoria. Se si fosse specificato un intero di tipo SHORT (corto), avremmo potuto utilizzare un cap con valore massimo 32767. Il tipo USHORT avrebbe permesso di utilizzare un cap che arrivava a 65535. Così abbiamo dovuto utilizzare un valore di tipo LONG per avere abbastanza spazio per memorizzare un cap che può arrivare fino a 99999.

Così adesso disponiamo di una struttura dati definita come "Etichetta". Vediamo come può essere utilizzata nei programmi.

Supponiamo di avere un particolare gruppo di persone interessate ai biglietti d'auguri e un altro gruppo interessato a quelli da visita. Includiamo la dichiarazione di Etichetta come parte del programma in linguaggio C con:

```
#include "etichetta.h"
```

Se il file "etichetta.h" contiene la definizione della struttura "Etichetta", questo comando rende la definizione disponibile per il vostro programma. Successivamente si può comunicare al com-

pilatore che si sta lavorando con due gruppi differenti di persone, mediante una dichiarazione simile alla seguente:

```
struct Etichetta auguri[100];
sruct Etichetta visita[100];
```

Il compilatore C riserverà lo spazio per 100 strutture dati delle dimensioni di Etichetta chiamate "auguri" e riserverà 100 strutture dati delle dimensioni di Etichetta da chiamare "visita". Ogni struttura Etichetta occupa 120 locazioni di memoria - 14 per il "Lnode", 30 per il nome, 50 per l'indirizzo, 20 per la città, 2 per lo stato, e 4 per il codice di avviamento postale.

Le dichiarazioni usate per auguri e visita sono proprio come i comandi DIM del BASIC e costringerebbero quindi il programma a usare solamente quel numero massimo di elementi.

Trovare un elemento

Durante la costruzione di una struttura dati in memoria, il compilatore mette gli elementi sempre in locazioni di memoria consecutive. Per localizzare un particolare elemento (membro) di una struttura, il compilatore utilizza la definizione si è fornita per calcolare l'indirizzo di memoria nel quale si trova l'elemento. Nell'esempio precedente, se il primo elemento dell'array "auguri" si trova alla locazione 10000 (decimale), allora le prime 14 locazioni di memoria, partendo da quella locazione, saranno riempite con i dati relativi alla parte della struttura chiamata Lnode. L'indirizzo del primo carattere del nome sarà quindi posizionato a 10014 decimale (le locazioni dalla 10000 alla 10013 compresa sono occupate dalla parte del Node).

E poi?

Fino a questo punto si è vista solamente l'allocazione statica di strutture dati: abbiamo cioè dichiarato due array, auguri e visita, ognuno dei quali fa parte del gruppo di elementi di dati definito come "Etichetta". Questo metodo è ancora limitato, in quanto il numero degli elementi a cui si può accedere è definito al momento nel quale il programma viene compilato.

Dal momento che si vuole lasciare la porta aperta per aggiungere nuovi elementi, occorre allocare gli elementi dinamicamente anzichè staticamente. Vedremo ora come si possa chiedere al sistema operativo di allocare un blocco di memoria e come lo si possa sistemare all'interno di una lista di blocchi posta sotto il nostro controllo.

Allocazione dinamica

L'Amiga fornisce una funzione chiamata AllocMem che viene usata per richiedere un blocco di memoria al sistema operativo. Se la chiamata a questa funzione ottiene in risposta un valore diverso da zero, il sistema operativo ha effettivamente allocato un blocco di memoria per l'uso esclusivo del richiedente. Su Amiga ciò significa anche che il sistema operativo si scorderà completamente dell'esistenza di quel blocco di memoria. Quando il programma termina, occorre preoccuparsi di restituire quell'area di memoria al sistema operativo in modo che possa essere successivamente utilizzata da un altro task. Più avanti daremo un'occhiata più approfondita alla deallocazione della memoria.

Ecco un esempio che mostra come utilizzare la funzione AllocMem. Verrà inoltre introdotto un altro nuovo concetto per questo articolo: la variabile puntatore. Un puntatore è una variabile che contiene l'indirizzo di qualcosa. Bisogna dichiarare a che tipo di oggetto questa variabile deve puntare (a una struttura, per esempio), così che il compilatore sappia come sfruttare il puntatore per riferirsi a quell'oggetto.

```
struct Etichetta *Lpointer;

/* dichiara un puntatore a una struttura Etichetta */

/* ...più avanti nel programma */

Lpointer = (struct Etichetta *)AllocMem(sizeof(struct
Etichetta),MEMF_CLEAR);
```

Questa scrittura equivale a dire: "Dammi un blocco di memoria che abbia le dimensioni della struttura Etichetta, e metti l'indirizzo della locazione di memoria più bassa di quel blocco nel puntatore chiamato Lpointer". La notazione:

```
(struct Etichetta *)
```

viene chiamata "operatore di cast". Quando si tenta di assegnare una cosa a un'altra, il compilatore C controlla che i due oggetti siano dello stesso tipo oppure siano compatibili. Se non lo sono, il compilatore emette un messaggio di avvertimento e potrebbe non terminare la compilazione del programma.

Il cast di tipo è necessario perchè la funzione AllocMem restituisce un tipo di puntatore differente. Il cast dice al compilatore che non deve preoccuparsi del tipo di valore restituito da AllocMem e di trattarlo semplicemente come se fosse un puntatore a una struttura di tipo Etichetta. A questo punto, quando il compilatore vedrà l'operatore di assegnamento, riterrà di effettuare l'assegnamento di due oggetti dello stesso tipo.

Il parametro chiamato MEMF_CLEAR indica che la routine di allocazione della memoria dovrà riempire il blocco di memoria di zeri prima di restituirne l'indirizzo.

Teniamo tutto sotto controllo

Adesso che abbiamo ottenuto un blocco di memoria, come facciamo a tenerne traccia? Ricordate che dovremo restituirlo quando il programma terminerà, altrimenti il sistema operativo non potrà ri-assegnarlo dopo che il nostro programma avrà terminato. Si potrebbe aggiungere questo blocco di memoria a una lista utilizzabile per rappresentare il nostro gruppo di persone. Vediamo come si può creare una lista chiamata Auguri:

```
struct List Auguri; /* dichiara una struttura List */
/* ...più avanti nel programma */
NewList(&Auguri); /* passa l'indirizzo della lista a NewList
                   * per permetterne l'inizializzazione  */
```

E' stato scritto "più avanti nel programma" perchè in C le dichiarazioni dei tipi delle variabili deve essere effettuato per prima cosa, mettendo le linee di istruzioni eseguibili in seguito.

Adesso è arrivato il momento di tirare in ballo la parte "Lnode" della nostra struttura dati. Tutte le routine che gestiscono le liste di sistema si aspettano che qualsiasi oggetto che gli venga passato si riferisca a un blocco di memoria strutturato come un Node. Un Node contiene i dati che permettono la creazione di liste, come il puntatore all'elemento precedente nella lista, il puntatore all'elemento successivo nella lista, e un numero di priorità che viene utilizzato dal gestore della lista per inserire l'elemento in una certa posizione.

Non abbiamo bisogno di effettuare altre operazioni su questa parte Node se vogliamo semplicemente aggiungere il nostro blocco di memoria nella lista. Dal momento che MEMF_CLEAR è stato utilizzato, tutti i membri nella parte Node della struttura dati contengono zero, e verranno trattati di conseguenza.

Ora aggiungiamo nella lista Auguri l'elemento appena allocato con:

```
AddTail(&Auguri,Lpointer);
```

AddTail è una funzione che aggiunge degli elementi alla fine di una lista. Accetta come argomenti due puntatori o indirizzi: il primo è l'indirizzo della lista nella quale bisogna inserire l'elemento, il secondo è l'indirizzo dell'elemento da aggiungere. La "e commerciale" (&) davanti alla parola Auguri indica al compilatore di utilizzare l'indirizzo della struttura dati (la List) Auguri come primo parametro. Dal momento che Lpointer contiene l'indirizzo dell'elemento, questa parte è già a posto.

Adesso vediamo come porre i dati nella struttura usando un puntatore. Se Lpointer contiene l'indirizzo del primo elemento di una struttura dati Etichetta, allora l'indirizzo del primo carattere del nome è:

```
&Lpointer->name[0]
```

Questo è l'indirizzo (grazie al "&") di un elemento contenuto nella struttura dati chiamata "Etichetta", poiché Lpointer è stato dichiarato come puntatore a una struttura dati Etichetta. La freccia in avanti ("->") significa che ci si riferisce a uno specifico membro della struttura, in questo caso all'array nome. Noi vogliamo l'indirizzo dell'elemento 0 dell'array, il primo carattere. In C gli indici degli array cominciano con l'elemento 0 e vanno fino a "dimensione_array - 1", anzichè partire da 1 ed arrivare fino a "dimensione_array".

Dal momento che ci sono numerosi elementi in questa struttura dati, sarebbe più semplice leggere il programma se potessimo usare dei nomi significativi per i singoli elementi, anzichè usare la notazione "&Lpointer..." e cosè via. Sfruttiamo allora ancora un'altra caratteristica standard del compilatore C: il preprocessore. Il preprocessore permette di definire dei termini, utilizzando il comando "#define", che, quando vengono incontrati in fase di compilazione, sono sostituiti con la definizione che è stata fornita. Questa caratteristica rende i nostri programmi più facili da leggere, senza precludere per questo che il C sappia esattamente quali variabili si intende utilizzare.
Le nostre definizioni per i puntatori possono essere:

```
#define NOMEA       &Lpointer->nome[0]
#define INDIRIZZOA  &Lpointer->indirizzo[0]
#define PAESEA      &Lpointer->paese[0]
#define STATOA      &Lpointer->stato[0]
#define CAPA        &Lpointer->cap
```

Diversamente dall'esempio in BASIC, non abbiamo più una specifica variabile INDICE da utilizzare. Dobbiamo invece contare sul valore, che deve essere corretto, contenuto in Lpointer: quello cioè di una delle strutture dati Etichetta che abbiamo creato. Se Lpointer contiene un valore appropriato, allora gli elementi definiti NOMEA, INDIRIZZOA e così via possono essere utilizzati come se contenessero l'indirizzo dell'elemento nome, dell'indirizzo e così via.

Riempiamo la struttura dati

In C, per copiare le stringhe da una locazione all'altra, non è assolutamente possibile utilizzare l'assegnamento diretto, utilizzando cioè semplicemente il segno di uguale (come si potrebbe fare in BASIC). Bisogna invece usare una funzione, fornita in una 'libreria' di funzioni assieme al compilatore, per effettuare la copia. Solitamente si usa "strcpy", che accetta un indirizzo di destinazione e uno di provenienza come suoi parametri. Se si è allocata una struttura Auguri, ed Lpointer contiene un valore valido (diverso da zero), allora si può scrivere,

```
strcpy(NOMEA,InputString);
```

per copiare la stringa di ingresso nella porzione nome della struttura dati. Il preprocessore C esegue la sostituzione per la definizione NOMEA durante la compilazione, e stabilisce il corretto indirizzo a partire dal quale bisogna copiare la stringa. Alla fine di questo articolo, si trova un esempio molto ridotto nel quale il programma pone in una lista di strutture dati solo il nome e il numero di telefono. Questo esempio non utilizza nessuna di quelle funzioni sofisticate di gestione delle liste per mettere gli elementi in sequenza; inoltre, non essendo negli obbiettivi di questo articolo, non salva nemmeno i dati in un file. E' solamente un semplice esempio della creazione e cancellazione dinamica di strutture dati che utilizza le più semplici funzioni per la gestione delle liste.

Perchè mi dovrebbero interessare le liste?

In questo articolo ho fornito una piccola introduzione alle liste,

sperando di stimolare il vostro appetito nell'approfondire l'argomento. Praticamente tutti i dati trattati dall'Exec, il programma di controllo del sistema multitasking di Amiga, sono mantenute in apposite liste.

Viene mantenuta una lista delle zone di memoria libere che possono essere allocate dai programmi in esecuzione. La funzioni che gestiscono l'allocazione della memoria lavorano prendendo un blocco da questa lista e restituendo il puntatore al programma che lo deve utilizzare.

Ci sono liste dei device (periferiche) tramite le quali l'AmigaDOS riesce a comunicare con apparati esterni o virtuali, liste di librerie di routine richiamabili dai propri programmi, liste di task pronti ad essere eseguiti, liste di task che aspettano che succeda qualcosa prima di poter essere eseguiti nuovamente e liste di porte (in senso software) alle quali possono essere mandati messaggi scambiati tra task diversi in esecuzione.

Si potrebbe dire: molte liste per molti fini.
Se avete trovato questo articolo interessante, sfogliate qualcuno dei libri sul sistema operativo di Amiga. Vi diranno di più.

```
/* rubricatel.c - esempio di strutture dati */

/* nota: lanciate questo programma da CLI:

     1> rubricatel

NON digitate "run rubricatel"
clude "exec/lists.h"
#include "exec/memory.h"
#include "stdio.h"

extern APTR AllocMem();
extern struct Node *RemHead();

struct PhoneEntry {
  struct Node Pnode;
  char  name[41];
  char phonenum[31];
};

struct PhoneEntry *pe;
struct PhoneEntry *nextpe;

#define NAMEP  &pe->name[0]     /* indirizzo della parte
  quit =  0;       /* non uscire presto        */

nome  */
#define NUMP   &pe->phonenum[0] /* indirizzo della parte
numero */

main()
{
  struct List Plist;
  char buffer1[256]; /* per inserire il nome     */
char buffer2[256]; /* per inserire il numero   */
int c, quit;
char mode;       /* determina cosa fare      */

NewList(&Plist);  /* inizializza la lista     */

printf("Lista vuota, inserisci dati? (y/n) ");
gets(buffer1);
c = buffer1[0];
if (c == 'n') exit(0); /* niente dati: fine  */

mode = 'a';      /* inizia aggiungendo       */

  while(1)    /* loop esterno: leggi e mostra i
dati */
  {
    switch(mode)

  {
  case 'a':     /* modo: aggiungi */
    while(1)   /* ripete all'infinito fino al termine */
    {
      printf("Inserisci il nome (max 40 caratteri)\n");
      printf("o 'Q' per uscire dal modo inserimento\n");
      printf("quindi premi RETURN\n");
      gets(buffer1); /* legge una stringa */
      if(strlen(buffer1) == 1 && buffer1[0] == 'Q') break;
      /* fai qualcos'altro se e' finito l'inserimento */

      printf("Inserisci il numero di telefono; quindi premi
     RETURN\n");
      gets(buffer2);
      pe = (struct PhoneEntry *)
        AllocMem(sizeof(structPhoneEntry),MEMF_CLEAR);

      if(pe != NULL)
      {
        buffer1[40] = '\0';  /* tronca la stringa di ingresso */
        strcpy(NAMEP, buffer1);
        buffer2[30] = '\0';  /* anche questa, in caso che... */
        strcpy(NUMP,  buffer2);
        AddTail(&Plist, pe); /* aggiungi alla fine della lista */
      }
    /* esegue all'infinito, fino ad un 'Q' "break" */
    }
    printf("Ecco i dati inseriti finora:\n\n");

    /* passaggio a catena attraverso la lista */
    pe=(struct PhoneEntry *)Plist.lh_Head;
    while (pe->Pnode.ln_Succ != NULL)
    {
    printf("%s : %s\n",NAMEP,NUMP);

    pe=(struct PhoneEntry *)pe->Pnode.ln_Succ;
    /* punta a quello successivo */
    }
    break;

  case 'd':    /* modo cancella */
    pe = (struct PhoneEntry *)Plist.lh_Head;
```

```
    while (pe->Pnode.ln_Succ != NULL)
    {
      /* punta a quello successivo */
      nextpe=(struct PhoneEntry *)pe->Pnode.ln_Succ;
      printf("Registrazione attuale:\n");
      printf("%s : %s\n",NAMEP,NUMP);
      printf("La cancello? (y/n)");
      gets(buffer1);
      if(buffer1[0] == 'y')
      {
        Remove(pe);
        FreeMem(pe, sizeof(struct PhoneEntry));
      }
      pe = nextpe;  /* riferisci su quello successivo */
    }
    break;

  case 'q':   /* modo esci */
    quit = 1;
    break;

  default:    /* in tutti gli altri casi */
    break;
  }   /* fine dell'enunciato switch */

  if (quit != 0) break;  /* esci dal loop esterno */

  printf("Aggiungi (a), Cancella (d), Esci (q)\n");
  printf("(a/d/q)");
  gets(buffer1);
  mode = buffer1[0];
 }   /* fine del loop esterno all'infinito */

cleanup: /* salta qui quando pronto ad uscire */

 while(1)
 {
  /* tirale fuori dalla lista, una alla volta */
  pe = (struct PhoneEntry *)RemHead(&Plist, pe);
  if(pe != NULL)
    FreeMem(pe, sizeof(struct PhoneEntry));
  else
    break;
 }
}   /* fine di main */
```

(segue da pag. 13)

Comandi dell'AmigaDOS V1.3

Sono supportati anche i livelli negativi di priorità.

TYPE

TYPE [FROM] <file> [[TO] <file>] [OPT HEX=H|NUMBER=N]

Ora il comando TYPE fornisce un warning se il file di destinazione esiste già. Inoltre sono stati aggiunti i sinonimi HEX e NUMBER di H e N.

VERSION

VERSION [<nome libreria|nome device> [<versione> <revisione>]]

Il comando VERSION può essere usato per verificare il numero di versione di una libreria, di un device o del disco Workbench. VERSION può anche essere usato per verificare che un certo oggetto abbia numero di versione uguale o superiore (quindi sia più recente) a quello specificato. Se non viene specificato nulla, VERSION mantiene la sua funzione normale: visualizza le versioni del Kickstart e del Workbench attualmente in funzione.

[1.3] XICON

XICON

XICON è usato per eseguire file di comandi CLI tramite una icona. Prima dell'esecuzione del file di comandi viene aperta una finestra tramite la quale viene effettuato l'eventuale input e output. Per usare XICON si deve prima creare una icona "project" per il file di comandi. Poi usando "Info" del menu del Workbench si deve porre come tool di default "C:Xicon"

Riflessioni sul Workbench

di Rob Peck

Cos' è e come possiamo utilizzarlo

Rob Peck è l'autore dei manuali Amiga ROM Kernel e Amiga Hardware e del libro "Programmers' Guide To The Amiga" (SYBEX, 1987). Può essere raggiunto presso: DATAPATH, P.O. Box 1828, Los Gatos, CA, 95031-1828.

Il Workbench è la parte più visibile dell'interfaccia utente di Amiga. Il fatto che sia semplicemente un programma applicativo in grado di lanciare altri programmi lo rende molto interessante.

E' facile, per un programmatore, creare delle routine esterne per il proprio programma che lo possano avviare correttamente, indipendentemente dal fatto che questo sia stato lanciato dal Workbench piuttosto che da CLI. In questo articolo viene analizzato il metodo corretto per adattare un programma in linguaggio C in modo che sia utilizzabile in entrambi gli ambienti di lavoro. Per fornire maggiori informazioni, farò riferimento ad alcuni programmi che utilizzano queste tecniche. Naturalmente il metodo "ufficiale" è indicato nel capitolo del Workbench dell'Amiga ROM Kernel Manual.

Se un utente si imbatte in un programma che è stato progettato esclusivamente per uno dei due ambienti di lavoro (environment), può servirsi di alcuni programmi di utilità di pubblico dominio che permettono di adattare quel programma all'altro ambiente di lavoro, e viceversa.

In questo articolo verrà esaminato "il programma" Workbench. Cominceremo con alcune routine che l'Amiga mette a disposizione come interfaccia per la programmazione del Workbench. Daremo anche un'occhiata a qualche utilità di pubblico dominio che gli utenti hanno creato per rendere più facile l'interfacciamento col Workbench, o per rendere possibile il passaggio da Workbench a CLI e viceversa.

C'è un Workbench screen senza il Workbench

Quando il sistema riparte da zero, viene creato uno schermo Workbench, anche se il Workbench vero e proprio non è ancora presente. Potete verificare questo fatto personalmente, compiendo le seguenti operazioni: formattate (FORMAT) ed installate (INSTALL) un nuovo disco, senza copiarvi niente sopra (niente Statrtup-Sequence). Inserite questo disco nel drive interno e resettate la macchina. Quello che vedrete dopo il reset è un CLI che somiglia molto al display di qualsiasi altro computer.
Adesso riducete questa finestra usando il gadget di ridimensionamento, e spostate il CLI in basso, in modo da scoprire la barra del titolo dello schermo. A questo punto troverete il Workbench screen, ma senza icone e nessun menu; in pratica, niente Workbench. Il Workbench screen è solamente il posto dove viene costruito il Workbench.

Con l'AmigaDOS 1.2, quando la vostra Startup-Sequence esegue il comando "LoadWB", il vero codice del Workbench apre sul Workbench screen una di quelle che vengono chiamate finestre di sfondo (backdrop window). Le finestre di sfondo sono uniche, in quanto nessun'altra finestra potrà essere mandata sotto quella finestra all'interno di uno stesso schermo.

Il Workbench disegna le sue icone all'interno della finestra di sfondo, e può utilizzare le funzioni standard di Intuition dedicate alla gestione delle finestre (finestra sopra, finestra sotto) con tutte quelle che verranno aperte sullo schermo del Workbench. Dal momento che le altre finestre rimangono sempre sopra quella di sfondo, anche quando vengono spinte sotto, le icone rimangono sempre sotto qualsiasi finestra.

Il Workbench riceve messaggi

La finestra di sfondo del Workbench può ricevere messaggi relativi ai tasti del mouse e ai suoi spostamenti, esattamente come può la vostra finestra. In questa maniera il Workbench può selezionare un menu, o annullare la selezione di un gadget, semplicemente controllando i movimenti del mouse ed i 'click' effettuati dall'utente.

Le icone visibili sullo schermo del Workbench sono in realtà dei Gadget di Intuition che mandano messaggi addizionali al Workbench.
Intuition può comunicare al Workbench che un'icona è stata selezionata, o che è stata attivata (doppio click), oppure che è stata trascinata e lasciata da qualche altra parte sullo schermo.

Le icone del Workbench possono avere due forme

Con la versione 1.1 del Workbench, quando un'icona veniva "afferrata", il cursore del mouse diventava un oggetto che doveva rappresentare l'icona selezionata. Quell'oggetto era uno sprite, come lo è la normale freccia del mouse.
La versione 1.2 del Workbench si avvantaggia di un'altra funzione grafica di Amiga: il "Bob".
Un Bob è un elemento grafico che può essere utilizzato con le funzioni di animazione incorporate in Amiga. L'immagine dell'icona viene copiata nella struttura di un Bob e le routine di animazione vengono usate per muovere l'immagine sullo schermo, seguendo gli spostamenti del mouse.

Mentre l'immagine originale dell'icona è disegnata nella finestra di sfondo, rimanendo così sotto tutte le altre finestre, l'immagine del Bob viene visualizzata direttamente sullo schermo.
Questo rende possibile lo spostamento del Bob che contiene l'immagine dell'icona in qualsiasi parte dello schermo, passando sopra qualunque finestra, indipendentemente dal loro numero.

Il Workbench, qualche volta, esclude gli altri task

Qualsiasi altro task che tenti di disegnare nello schermo del Workbench mentre l'icona (Bob) viene trascinata in giro, viene automaticamente messo a dormire (questo avviene direttamente all'interno delle routine grafiche di Amiga), per fornire uno sfondo stabile sul quale muovere l'icona. Potete osservare questo fenomeno facendo partire qualche dimostrazione di grafica dinamica, come i programmi "Lines" o "Boxes", e prelevando quindi un'icona per muoverla da qualche altra parte; l'aggiornamento delle immagini in quelle finestre dinamiche si ferma finché non avrete terminato di spostare l'icona.

Questo succede perché il sistema deve salvare i dati contenuti nell'area dove verrà disegnato il Bob, quindi disegnare il Bob, per poi ripristinare quell'immagine quando il Bob verrà mosso da qualche altra parte. Se la zona sotto il Bob venisse cambiata mentre il Bob è fermo, l'immagine così creata non corrisponderebbe più a quella salvata e in definitiva il mouse si lascerebbe dietro una traccia di "sporcizia".

Solamente un altro programma applicativo

Eccoci al dunque. Il Workbench: un programma applicativo proprio come gli altri, che definisce il suo ambiente operativo, comprese le cose da fare quando le icone vengono spostate dall'utente. Con la versione 1.2 del Workbench non possiamo "dire al Workbench che abbiamo appena creato un'icona" o chiedergli di depositare un'icona all'interno di una nostra finestra così semplicemente come la lascerebbe cadere in una delle sue (per esempio nella finestra di un cassetto, o nella finestra o nell'icona di un disco, oppure nella finestra o nell'icona della pattumiera). Il Workbench controlla personalmente questi oggetti e sa come interagire con essi, ma non è stata documenta alcuna interfaccia (quantomeno finora) per permettere ai programmi esterni di controllarlo direttamente.

Il Workbench, comunque, può utilizzare le icone da noi create per lanciare i nostri programmi; le routine presenti nella ICON.LIBRARY rendono facile la creazione di queste icone. Così, fino a quando Hackbench (un rifacimento di pubblico dominio del Workbench) non sarà migliorato a sufficienza o fino a quando lo stesso Workbench non diventerà interattivo con gli altri programmi, dovremo rassegnarci al fatto che l'utente debba compiere la maggior parte di interazione con esso, senza poterlo sfruttare per creare i prodotti che circolano attualmente.

Vediamo adesso come i nostri programmi possano riconoscere se sono stati lanciati da Workbench o da CLI, in modo che possano partire senza problemi da entrambi gli ambienti. Vedremo poi alcune delle routine presenti nella ICON.LIBRARY, in modo da capire come sono state utilizzate per adattarsi al Workbench in alcuni programmi liberamente distribuibili.

Come trovare i nostri argomenti

Se un programma è stato progettato per partire sia da Workbench sia da CLI, deve essere in grado di ricevere i suoi argomenti da entrambi gli ambienti. Tanto per fare un esempio, ipotizziamo che il nome del programma sia TEST. Se fatto partire da CLI, l'utente potrebbe avere digitato:

```
1> TEST argomento1 argomento2
```

Il punto di entrata del nostro programma C, in corrispondenza della funzione 'main', sarebbe scritto in questo modo:

```
main(argc,argv)
int argc;
char **argv;
{
/* il programma inizia qui */
```

In questo esempio 'argc' avrebbe un valore uguale a 3. Gli argomenti sarebbero una serie di stringhe di caratteri, formattate come se avessimo dichiarato un array del tipo:

```
char *argv[] = {
"TEST",
"argomento1",
"argomento2"
};
```

Così il nostro programma può esaminare sia il nome col quale è stato lanciato (qualche volta lo stesso programma può fare cose diverse, a seconda del nome che avete usato per lanciarlo), sia il contenuto degli argomenti, uno alla volta.

Se il programmaè stato chiamato da Workbench, 'argv' non ha significato, e 'argc' sarà uguale a zero. In questo caso un puntatore generato dal vostro codice di inizializzazione (quello che ogni programma C contiene dopo il link) conterrà il valore di un puntatore esterno chiamato "WBenchMsg", che punta a sua volta agli argomenti che il Workbench ha preparato per il vostro programma. Questi argomenti rappresentano una o più selezioni che l'utente ha operato da Workbench (utilizzando la selezione estesa tramite il tasto SHIFT per coinvolgere altre icone) prima di attivare, con il doppio click o con la selezione di OPEN dal menu del Workench, l'icona del nostro programma. In questo caso il programma diventa:

```
extern struct WBStartup *WBenchMsg;

main(argc,argv)
int argc;
char **argv;
{
  if(argc) /* vero se argc diverso da zero */
  {
    PartitoDaCLI(argc,argv);
  }
  else
  {
     PartitoDaWB(WBenchMsg->sm_NumArgs,WBenchMsg-
>sm_ArgList);
```

```
}
}
```

Il valore contenuto in 'sm_NumArgs' rappresenta il numero di argomenti, proprio come 'argc' contiene gli argomenti ricevuti da CLI.

Il valore in 'sm_ArgList' punta all'inizio dell'array che contiene gli argomenti ricevuti dal Workbench. Ecco come apparirebbe la lista di argomenti al vostro programma se, per esempio, l'utente avesse selezionato due icone, una per argomento1 e una per argomento2, selezionando infine ed attivando l'icona TEST:

```
struct WBArg arg[3];
```

Internamente ogni struttura WBArg consiste in un puntatore (un BPTR in realtà, ma questo non ha importanza nel nostro caso) a un Lock, e in un puntatore (un puntatore normale, questa volta!) a una stringa contenente un nome:

```
struct WBArg {
BPTR wa_Lock;
char *wa_Name;
};
```

Proprio come avviene per gli argomenti ricevuti da CLI, il primo WBArg nella lista (arg[0]) rappresenta l'icona del programma, mentre gli argomenti successivi rappresentano le altre icone che l'utente ha scelto (probabilmente icone di tipo TOOL), nell'ordine nel quale sono state selezionate.

Per aprire il file associato a un'icona, prima bisogna spostarsi nella sua directory. Il file associato all'icona deve essere, naturalmente, nella stessa directory. Per spostarsi in quella directory si usa la routine 'Current Dir' contenuta nella dos.library:

```
struct Lock *newlock, *oldlock;

newlock = arg[2].wa_Lock;
oldlock = CurrentDir(newlock);
```

Adesso possiamo aprire il file, utilizzando la 'Open' o 'fopen' o qualunque altra funzione C che serva allo scopo. Per usare, ad esempio, la funzione Open dell'AmigaDOS il codice diventerebbe:

```
struct FileHandle *fh;
fh = Open(arg[2].wa_Name,MODE_OLDFILE);
```

Le funzioni della Icon.library

Il file 'LIBS:icon.library', presente sul disco Workbench, contiene alcune funzioni che i programmi menzionati in seguito utilizzano per manipolare le icone. Queste funzioni sono:

GetDiskObject - legge un'icona
PutDiskObject - scrive un'icona
FreeDiskObject - segue la 'GetDiskObject' per liberare la memoria utilizzata per immagazzinare l'icona letta da disco.

Ancora una volta, la cosa piacevole delle funzioni citate é che non bisogna conoscere il formato dei file icona per poterle utilizzare. E' sufficiente manipolare le strutture dati che riceveremo da 'GetDiskObject' e così via. (Infatti la Commodore non documenta il contenuto esatto di queste strutture dati. Sul prossimo numero di questa rivista, comunque, troverete l'articolo "A proposito dei .info", scritto da Betty Clay proprio per fare luce su questi misteri. NdT)
La funzione 'GetDiskObject' restituirà il puntatore alla struttura 'DiskObject', che definisce un'icona. Uno dei campi di questa struttura è un puntatore ai 'ToolTypes' dell'icona. I ToolTypes sono stringhe di caratteri associate a ciascun icona che possono essere lette dall'applicazione. I ToolTypes di qualsiasi icona possono essere facilmente modificati dall'utente, selezionando il comando INFO dal menu del Workbench. Vengono usati per fornire informazioni al programma riguardo opzioni, valori di default, ecc., e trasportano queste informazioni in modo standard (seguono spiegazioni più dettagliate).

FindToolType: esamina un array di stringhe per trovare un ToolType che combaci con quello che state cercando, e restituisce un puntatore al primo carattere di quel ToolType, oppure zero, se non trova quello desiderato.
MatchToolValue: dopo avere utilizzato 'FindToolType', possiamo esaminare ulteriormente la stringa che ci viene restituita, per vedere se quella stringa contiene un determinato elemento.
Queste due funzioni sono utili per lavorare con il ToolType, ma possiamo utilizzarle anche per qualcosa di diverso dalle icone, se desideriamo. 'FindToolTypes' lavora su un array di puntatori a stringhe, al quale possiamo pensare come se fosse definito in questa maniera:

```
char *tt_strings[] = {
"PRIMOOGGETTO=stringa del primo oggetto
"SECONDOOGGETTO=altra stringa",
"TERZO=questolquelloled.altre.cose",
""
};
```

L'ultima linea in questa pseudo-definizione è importante: una stringa di lunghezza zero è sempre l'ultimo elemento in un array di ToolTypes. La presenza della stringa nulla comunica alle routine del Workbench e della icon.library che i ToolTypes sono finiti. Questo non è stato detto nel ROM Kernel Manual.
Per usare la funzione 'FindToolTypes', dobbiamo passargli il puntatore al primo elemento nell'array di stringhe ToolTypes e il puntatore alla stringa da cercare. Vediamo un esempio che utilizza l'array definito precedentemente:

```
valore=FindToolType(tt_strings,"TERZO");
```

La 'FindToolTypes' trova qualcosa che dice "TERZO=", cosi' restituisce l'indirizzo di memoria al quale si trova il primo carattere che segue il segno di uguale. Funzionalmente (non in C) è come se avessimo scritto:

```
valore="questolquelloled.altre.cose"
```

Le barre verticali hanno un significato speciale per 'MatchToolValuè. Quando vogliamo assegnare più di un valore a un unico oggetto nel ToolTypes, mettiamo una barra verticale per separare i vari valori. A questo punto possiamo utilizzare 'MatchToolValué per vedere se qualcuno dei valori combacia con la stringa che stiamo cercando. Questi sono i risultati delle chiamate a MatchToolValue utilizzando la variabile 'valore' definita in precedenza:

risultato=MatchToolValue(valore,"questo");
restituisce VERO.
risultato=MatchToolValue(valore,"quello");
restituisce VERO.
risultato=MatchToolValue(valore,"TEST");
restituisce FALSO.

Programmi che usano la icon.library

A questo punto, se sappiamo di essere stati lanciati da Workbench, sapremo anche che ci saranno delle informazioni addizionali disponibili per noi nel file che contiene i dati dell'icona (icon file), oppure nel messaggio di inizializzazione (startup message) che abbiamo ricevuto dal Workbench. John Toebes, della Software Distillery, ha scritto alcuni programmi che utilizzano queste informazioni addizionali fornite dal Workbench. Lavorare con le icone del Workbench e con la icon.library è abbastanza facile. Comunque se siete particolarmente interessati a questo argomento, vi consigliamo di esaminare il codice sorgente dei programmi di John per avere la possibilità di apprendere esattamente come leggere, scrivere e interpretare le informazioni contenute nelle icone.

Tutti i sorgenti dei programmi qui menzionati si trovano sui dischi FISH numero 12 e 43. Per vostra comodità le copie di questi sorgenti sono disponibili anche nel disco di Transactor di questo numero. Alla fine di questo articolo è stato inserito un programma esemplificativo che prova due funzioni: 'ReadInfoFile' che carica un icon file in memoria, 'WriteInfoFile' che lo riscrive di nuovo sul disco. Queste sono delle estensioni delle funzioni 'GetDiskObject' e 'PutDiskObject' alle quali bisogna fornire anche il percorso (path) per raggiungere la directory che contiene l'oggetto.

Icone alternate

Il Workbench può visualizzare una immagine differente per un'icona selezionata, rispetto a quella visibile quando l'icona non è selezionata. Alcuni hanno creato coppie di immagini per l'icona Trashcan (una pattumiera chiusa e una aperta), per l'icona Drawers (un cassetto chiuso e uno aperto) e per altre icone di programmi, dove l'immagine, quando viene selezionata, è sostituita da scritte di copyright. Per aggiungere un'immagine alternata a una icona già esistente, bisogna:

1. Leggere l'icon file e salvare tutti i puntatori della struttura dati che si intende cambiare.

2. Settare i flag dell'icona (gg_Flags) a GADGHIMAGE.

3. Settare gg_SelectRender in modo che punti a una struttura Image che descrive l'immagine alternata e dove posizionare il lato sinistro e il lato superiore della stessa, in relazione alla posizione dell'icona. Inoltre occorre specificare le dimensioni, altezza e larghezza, e le zone di memoria in cui si trovano i bitmap dell'immagine.

4. Scrivere su disco l'icon file.

5. Rimettere le cose come erano originariamente nella struttura dati dell'icona.

6. Liberare la zona di memoria dell'icona.

Il programma SETALTERNATE (Fish #12) aggiunge un'immagine alternata a un'icona, accettando come argomenti i nomi di una icona primaria e di una secondaria da mettere insieme. Entrambe le icone devono essere state create precedentemente. Questo rende tutto più facile perché a questo punto basterà leggere gli icon file in memoria (utilizzando le funzioni della icon.library), modificare qualche flag, salvare qualche puntatore e quindi salvare la icona risultante.

E' necessario salvare i valori dei puntatori in quanto tutte le routine della icon.library allocano della memoria per le varie strutture dati che ciascuna utilizza. In pratica, un icon file letto da 'GetDiskObject' crea una serie di strutture dati in memoria. Dopo aver salvato i puntatori creati dalla icon.library, potremo tranquillamente modificarne i valori, ad esempio per puntare alle nostre immagini e ai nostri array di stringhe, per poi riscrivere la versione modificata del file su disco. Ma dopo queste modifiche dobbiamo ripristinare i puntatori originali, che avevamo precedentemente salvato, per permettere al sistema di liberare con 'FreeDiskObject' la stessa memoria che era stata allocata in origine. Il programma esemplificativo che accompagna questo articolo mostra come salvare e ripristinare questi puntatori.

Stringhe ToolTypes

Ecco una caratteristica del Workbench poco utilizzata. Le stringhe 'ToolTypes' sono accessibili dall'ambiente Workbench quando l'utente utilizza la INFO.LIBRARY (usando il comando INFO del menu del Workbench). Probabilmente le stringhe ToolTypes non sono usate spesso dai programmatori perchè l'utente può modificarle e quindi è difficile fare affidamento sul fatto che l'informazione rimanga inalterata, come dovrebbe essere. A ogni modo, io sono riuscito a trovare un loro possibile utilizzo e John Toebes ne ha escogitato un altro.

Ho modificato una versione del programma di pubblico dominio SpeechToy (Fish #5) in modo che legga il proprio icon file (nel caso sia stato fatto partire da Workbench) e che esamini le stringhe ToolTypes. Queste conterranno i valori di default che il programma dovrà utilizzare. I valori di default sono quelli che l'utente ha scelto l'ultima volta che il programma é stato eseguito. Invece di usare ogni volta i default di sistema, ho utilizzato queste stringhe per descrivere le variabili PITCH = <qualcosa>,RATE = <qualcosa>,SEX = 0 (per il maschio) ed altre ancora.

John Toebes ha utilizzato questa caratteristica in un programma chiamato IconExec (Fish #12) che permette di creare un'icona contenente una sequenza di comandi CLI da eseguire uno alla volta. La tecnica utilizzata nel programma non segue le 'raccomandazioni ufficiali' (Amiga ROM Kernel Manual, capitolo sul Workbench) per le stringhe di tipo tool (che dovrebbero essere costruite come: "QUALCHECOSA = QUALCHEVALORE"). Il programma funziona comunque bene e viene accettato dalla icon.library, cosi' penso che dovremmo poter utilizzare questa tecnica tranquillamente. John ha anche scritto un programma di accompagnamento, chiamato SetWindow, che permette di stabilire le dimensioni di default per la finestra CLI creata da IconExec (invece di averla sempre a pieno schermo).

L'altra strada

Ci sono alcuni programmi che sono stati progettati per essere lanciati esclusivamente da Workbench, e che non forniscono alcuna interfaccia per il CLI. Per risparmiare memoria, alcune persone potrebbero volere lanciare tutti i programmi esclusivamente da CLI. Il programma WBrun (Fish #43) è stato creato per soddisfare questa esigenza. WBrun, scritto sempre da John Toebes, ricostruisce l'ambiente di lavoro del Workbench, senza bisogno di caricare il Workbench stesso.
Se, come si va dicendo, è possibile che il Workbench venga trasferito da ROM a disco in una delle future revisioni del sistema operativo di Amiga, questo programma diventerebbe assai utile. Come John fa notare nella documentazione di questo programma, WBrun è lungo meno di 3.700 byte. Se un utente vuole fare girare programmi che utilizzano il Workbench senza caricare il Workbench, questa é la strada da seguire.

Sommario

Benché l'utente occasionale possa anche non notarlo, il Workbench è proprio una delle dimostrazioni più appariscenti delle capacità multitasking della macchina. Possiamo lanciare un programma semplicemente clickando due volte sulla sua icona, e il Workbench è ancora lì, pronto ad interagire con noi, mentre l'altro programma sta girando felicemente in una finestra da qualche parte o nel suo schermo privato. Il Workbench screen può essere spostato sopra tutti gli altri schermi utilizzando la combinazione di tasti Amiga sinistro-N (Commodore-N su alcuni Amiga 500 e 2000), oppure spinto sul fondo con Amiga sinistro-M (Commodore-M), una volta che avremo terminato di usarlo.

Il piccolo programma che ho inserito alla fine dell'articolo parte solo da CLI (sigh - non si può avere ogni cosa).

Ritengo, comunque, che i programmatori dovrebbero supportare contemporaneamente sia il CLI che il Workbench nei loro programmi, per dare all'utente di Amiga la maggiore flessibilità possibile. Dopo tutto, la flessibilità è uno dei punti di forza di Amiga; sfruttiamola come si deve.

```
/* ReadInfoFile è la funzione che legge un icon file da disco
descritta nel ROM Kernel Manual */

/* WriteInfoFile è la sua controparte, creata in questa occasione
*/

/* ReadInfoFile è stata modificata rimuovendo gli errori tipografici che si trovano sul ROM K.M. Inoltre, anziché utilizzare i lock standard del Workbench, questo programma usa i lock standard dell'AmigaDOS sull'oggetto o sulla sua directory, passandoli alle funzioni del Workbench. */

/* wbenchdemo.c */


#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dosextens.h>
#include <workbench/workbench.h>
#include <workbench/icon.h>

/* ReadInfoFile:
 * input = directory e nome del file .info da leggere.
 * output = puntatore a struttura DiskObject, se il file viene
caricato
 *          con successo, altrimenti NULL.
 */

LONG IconBase;
extern struct Lock *Lock();

struct DiskObject *
ReadInfoFile(icondir, filename)
char *icondir;     /* directory che contiene l'icona */
char *filename;    /* nome dell'icona */
{
  LONG olddir;
  struct Lock *lock;

  struct DiskObject *diskobj;

  diskobj = NULL;   /* aspettiamoci il peggio */

  lock = Lock(icondir, ACCESS_READ);
  if(lock == NULL)
  {
    goto end;   /* non riesco a trovare la directory */
  }
  olddir = CurrentDir(lock);  /* spostiamoci nella directory dell-
'icona */

  diskobj = GetDiskObject(filename); /* adesso la possiamo leggere */

   /* Avremmo anche potuto fornire a GetDiskObject() tutto il
path per
  raggiungere il file, ma sembra piu' facile in questa maniera */

  CurrentDir(olddir);  /* ritorniamo nella directory originale */
  UnLock(lock);      /* libera il lock sulla directory dell'icona */
end:
  return(diskobj);  /* se tutto va bene restituisce il puntatore alla
zona
```

```
            * di memoria allocata dall'icon.library... ricordatevi
            * di usare FreeDiskObject quando avete finito. */
} /* end of ReadInfoFile */

/***************/


/* WriteInfoFile:
 * input = directory, nome e puntatore dell'icona da scrivere su disco
 * output = zero se tutto è andato bene, altrimenti un numero diverso
 */


long
WriteInfoFile(icondir, filename, diskobj)
char *icondir;
char *filename;
struct DiskObject *diskobj;
{
  LONG olddir;
  struct Lock *lock;
  long status;

  status = -1;  /* aspettiamoci il peggio */
  lock = Lock(icondir, ACCESS_READ);
  if(lock == NULL)
  {
    goto end;   /* non riesco a trovare la directory */
  }
  olddir = CurrentDir(lock);   /* spostiamoci nella directory dell'icona */
  status = PutDiskObject(filename, diskobj);
  if(status == 0)
  {
    status = IoErr();  /* sentiamo cosa dice AmigaDOS... */
  }
  CurrentDir(olddir);
  UnLock(lock);
end:
  return(status);
} /* end of WriteInfoFile */


/***************/


/* Questo programma parte solo da CLI, in modo da potere stampare eventuali messaggi di errori nel CLI dalla quale è stato lanciato.
Il formato della linea di comando è: fixicon <dir> <name>, dove <dir> è la directory nella quale risiede l'icona, <name>è il nome dell'icona (senza .info). Il programma legge l'icon file (se possibile), DISTRUGGE il vecchio array ToolTypes e lo sostituisce con uno nuovo. */

/* fixicon.c */

main(argc, argv)
int argc;
char **argv;
{
  struct DiskObject *dj;
  long i, success;
  char **tt;
  char *newarray[2];

  newarray[0] = "TOOLTEST=coronato_da_successo";
  newarray[1] = "";  /* la stringa lunga zero segnala la fine */

*/

*/
  IconBase = OpenLibrary(ICONNAME,1);
  if(IconBase == NULL)  return(NULL);  /* non trovo la libreria */

  dj = ReadInfoFile(argv[1], argv[2]);

  if(dj == NULL)
  {
    printf("ReadInfoFile ha fallito!\n");
    exit(10);
  }
  else
  {
    tt = dj->do_ToolTypes;

  for(i = 0; ;i++)  /* fino a quando non troviamo la stringavuota */
    {
      if(strlen(tt[i]) == 0)
        break;
      else
        printf("La Toolstring e`: %s\n",tt[i]);
    }
    tt = dj->do_ToolTypes;  /* salva il puntatore al vecchi array */

    dj->do_ToolTypes = newarray;
  }
  success = WriteInfoFile(argv[1], argv[2], dj);

  printf("Il valore di success è: %ld  e quello di ioerr() è: %ld\n",
  success, IoErr());

  dj->do_ToolTypes = tt;  /* ripristina il vecchio puntatore */

  FreeDiskObject(dj);
  CloseLibrary(IconBase);

} /* end of main */
```

Virus e antivirus: la guerra continua

di Leonardo Fei

Leonardo Fei studia ingegneria elettronica al Politecnico di Milano e si diverte a programmare su Amiga nel tempo libero. Attualmente è impegnato nell' edizione Italiana di Transactor for the Amiga e nello sviluppo di nuovi programmi di utilità. Il suo indirizzo è: via A. Fava 6, 20125 Milano.

La guerra fra virus e antivirus viene combattuta con armi e trucchi sempre più sofisticati; ormai ci sono parecchi virus in circolazione. Alcuni sono solo delle versioni modificate del primo virus SCA, o di quello BYTE BANDIT, mentre iniziano ad arrivarne alcuni completamente nuovi. Anche il numero dei programmi antivirus cresce, ma non tutti sono completamente sicuri contro tutti i tipi di virus che si possono trovare nel bootblock di un disco (per bootblock intenderemo l'insieme dei blocchi 0 e 1, quelli che contengono l'identificatore e il codice di boot di un disco che può essere caricato direttamente quando la macchina chiede il Workbench).

La storia

La prima generazione di antivirus controllava il bootblock confrontandone il contenuto con una lista, contenuta all'interno del programma antivirus, per vedere se corrispondeva a un dato tipo di virus o meno. La lista dei virus da cercare era quindi limitata ai tipi conosciuti nel momento in cui l'antivirus era stato scritto. Questi programmi diventavano inutilizzabili non appena compariva una nuovo virus, in quanto non veniva riconosciuto. Inoltre, questi antivirus erano programmi a sè stanti, che andavano caricati espressamente ogni volta che si desiderava controllare un disco, ma non proteggevano l'utente durante il normale utilizzo del computer. In questo modo era abbastanza facile dimenticarsi di controllare un disco, permettendo al virus di propagarsi nuovamente su tutti i dischi appena "ripuliti" e quindi considerati "sicuri". Un programma abbastanza conosciuto che funziona in questa maniera si chiama "VCheck", scritto da Bill Koester (membro del CATS, il supporto tecnico della Commodore Amiga americana).

Quando diventò evidente che prima o poi sarebbero sempre apparsi nuovi virus, cambiò l'approccio al problema, portando alla seconda generazione di antivirus. Probabilmente il programma più conosciuto di questa categoria é "VirusX" di Steve Tibbet. Questo programma gira in background e controlla automaticamente tutti i dischi che vengono inseriti in qualsiasi drive. La nuova idea di questo programma consisteva nel controllare il contenuto del bootblock, non solo confrontandolo con un elenco di virus conosciuti, ma, qualora ciò non avesse dato risultati, segnalando l'assenza del bootblock standard. In questo modo il programma era capace non solo di riconoscere i virus noti, ma anche di intercettare qualsiasi bootblock non standard che avrebbe potuto contenere un nuovo tipo di virus. L'unico inconveniente, a mio giudizio, consiste nel fatto che il controllo dei virus viene fatto durante il funzionamento normale della macchina, quando cioè i virus sono innocui in quanto il boot è già stato effettuato. Proprio durante quel boot un virus potrebbe essersi caricato in memoria.

Quando ho iniziato a pensare a un programma antivirus, cosa avvenuta prima di avere la possibilità di vedere VirusX, ho affrontato il problema con in maniera diversa.

Per me il problema non era quello di esaminare i dischi quando venivano inseriti in un drive, ma solo quando questi potevano effettivamente rappresentare un pericolo, cioè durante il boot. Il boot da disco è una specie di collo di bottiglia, di passaggio forzato attraverso il quale i virus devono passare per diventare attivi e quindi pericolosi. Se gli impediamo di entrare nella nostra macchina durante il boot, non dovremo più preoccuparci di loro anche se sono ancora fisicamente presenti su alcuni dei nostri dischi. Possiamo infatti con tutta tranquillità inserire dischi infetti in qualsiasi drive, purché il boot sia già stato fatto, senza correre ALCUN pericolo che i virus vengano caricati e attivati.

Un altro punto molto importante consisteva nella capacità del programma di bloccare non solo i virus attualmente conosciuti, ma anche qualsiasi versione futura, senza dovere essere continuamente aggiornato. Considerando questo aspetto risultò subito chiara l'inutilità di incorporare nel programma una lista di virus da cercare, in quanto questa sarebbe presto diventata obsoleta. Virtualmente si possono infatti codificare un numero infinito di virus nel bootblock; è unico, invece, il bootblock standard, quello "ufficiale" creato dal comando INSTALL del workbench! L'unico modo, quindi, di rendere il programma assolutamente privo da problemi di invecchiamento, consisteva non nel cercare determinati virus, ma nel verificare la presenza del bootblock standard; qualsiasi bootblock non standard avrebbe dovuto essere considerato come un potenziale pericolo.

C'era un solo modo per mettere in pratica tutto questo: modificare il modulo di bootstrap contenuto nel Kickstart, quello che carica il bootblock dal drive interno e che gli passa il controllo. Così ho localizzato il punto in cui quel modulo passa il controllo al bootblock caricato da disco e ho sostituito quell'istruzione con un salto al mio programma. La prima prova del programma fu eseguita modificando il Kickstart su disco per l'Amiga 1000, in seguito venne creata la versione che funziona anche con il Kickstart in ROM di Amiga 500/2000.

Guardian v1.1 vide la luce alla fine di febbraio e cominciò a "viaggiare" alla fine di marzo, mentre la versione 1.2 fu terminata i primi giorni di giugno, appena in tempo per il Commodore Show di Londra.

Vediamo come funziona

Guardian non sottrae neanche un singolo ciclo macchina dal tempo normalmente a disposizione dei programmi applicativi, perché è attivato solamente durante il boot. Non verranno quindi controllati i dischi inseriti dopo il boot (potete usare VirusX insieme a Guardian per ottenere questo controllo addizionale, ma notate che VirusX aggiungerà un nuovo task nella lista dei processi e aggiungerà anche un piccolo "tempo morto" al processo di riconoscimento di un disco ogni volta che ne verrà inserito uno), in quanto, come spiegato prima, a questo punto non rappresenteranno piu' alcun pericolo. Prima o poi verrano usati per fare il boot e allora Guardian li bloccherà, permettendoci di rimuovere il virus.

Il bootstrap carica il bootblock in memoria, controlla che il suo ID sia uguale a "DOS\0", controlla il checksum, quindi salta al codice dell'antivirus. Guardian controlla il bootblock appena caricato da disco confrontandolo con quello standard, contenuto all'interno dello stesso Guardian. Il bootblock standard è fatto in questa maniera:

```
        BOOT000:
DC.B    'DOS',0                 ;bb_id
DC.L    0                       ;bb_chksum
DC.L    $370                    ;bb_dosblock

        BOOT00c:
lea     BOOT026(PC),a1      ;a1 -> 'dos.library'
jsr     FindResident(a6)        ;cerca il DOS
tst.l   d0                      ;trovato ?
beq.s   BOOT022                 ;no, esci
move.l  d0,a0                   ;si, puntatore al DOS in A0
move.l  $16(a0),a0              ;a0 = rt_init ingresso DOS
moveq   #0,d0                   ;pulisci il flag di errore
        BOOT020:
rts                             ;restituisce il controllo al bootstrap

        BOOT022:
moveq   #-1,d0                  ;setta il flag di errore
bra.s   BOOT020                 ;ed esci

        BOOT026:
DC.B    'dos.library',0,0,0
```

E' sufficiente controllare solo questa parte del bootblock, in quanto se questa è standard, sarà impossibile passare il controllo a qualsiasi altra cosa che non sia l'AmigaDOS. Tutto quello che potrebbe esserci nella parte rimanente del bootblock verrà ignorato.

"ATTENZIONE !! QUESTO BOOTBLOCK NON E` STANDARD !! Possibile **VIRUS !!** Devo passargli il controllo ?"

Se il codice del bootblock differisce anche solamente di un singolo byte, viene visualizzato un alert per avvertire l'utente di una possibile minaccia contenuta nel bootcode e viene fornita una visualizzazione ASCII del bootblock sospetto, per aiutarci a identificarlo. Potremo riconoscere un virus dalla presenza di stringhe di testo come **"SCA!SCA!SCA!"** o **"VIRUS BY BYTE BANDIT"**, ma ricordiamo che circolano anche virus anonimi, senza un messaggio contenuto dentro di essi. Dovremo quindi essere sempre molto cauti, anche quando non vedremo queste scritte.

L'utente può scegliere se passare comunque il controllo al bootblock caricato, oppure se eseguire quello standard contenuto nel codice di Guardian. Viene data l'opportunità di eseguire il bootblock caricato, perché esistono diversi bootblock non-standard che non contengono alcun virus. Questi sono presentazioni-boot, caricatori veloci, boot-menu e altri bootblock personalizzati che possiamo trovare su dischi commerciali e non. Dovremo eseguirli, se vorremo che quei dischi funzionino in maniera appropriata.

Se decideremo di passare il controllo al bootblock caricato, il colore dello schermo diventerà rosso, per ricordarci che abbiamo scelto una strada che potrebbe rivelarsi pericolosa. Viene fatta una copia della struttura dati dell'Exec prima di eseguire il bootcode. Dopo che il controllo è stato restituito dal bootcode a Guardian, questa copia è confrontata alla attuale struttura dati dell'Exec.

"ATTENZIONE !! ExecBase alterato durante il boot ! VIRUS probabilmente attivo !! Ripristino vecchio ExecBase ?"

Se queste due copie differiscono anche di un solo byte, viene visualizzato un alert e noi possiamo decidere se rimettere la vecchia copia di ExecBase al suo posto, annullando così i cambiamenti fatti dal bootcode. Se riceveremo questo alert, potremmo avere eseguito il virus "BY BYTE BANDIT". Ripristinando la vecchia copia dell'ExecBase cancelleremo le modifiche fatte dal virus al vettore interrupt di Vertical Blanking e rimuoveremo il suo vettore dalla tabella dei moduli residenti. In questo modo il virus non potrà bloccare il nostro computer (perché è stato rimosso dal vettore di VBlanking) e non sopravvivera` al prossimo reset (perché é stato rimosso dalla tabella ResModules). E` rimasto tuttavia ancora un problema. Durante la sua esecuzione, questo virus modifica la tabella degli offset della libreria "trackdisk.device" in modo che il virus stesso venga attivato ogni volta che il computer accede a un nuovo disco (ogni volta che viene eseguito un comando di lettura/scrittura a partire dal blocco 0). Così i nostri dischi verranno infettati dal virus, semplicemente inserendoli in uno qualsiasi dei drive. Per risolvere anche questo problema è necessario resettare il computer. Questa operazione costringe il sistema operativo a ricostruire la tabella degli offset, rimuovendone anche le ultime tracce dalla memoria.

Anche il vecchio virus della "SCA" causerà la visualizzazione di questo alert, perché cambia il contenuto del vettore CoolCapture, contenuto nell'ExecBase. Basterà in questo caso il semplice ripristino del vecchio ExecBase per rimuovere completamente nante non rovinerà la sua area di memoria. Un'etichetta sullo schermo di boot (la mano con il disco del Workbench) informerà l'utente se e quale versione di Guardian sia correntemente attiva.

questo virus dalla memoria, senza bisogno di altre operazioni.

Se decidiamo di non ripristinare il vecchio ExecBase, Guardian potrebbe essere disattivato e rimosso dalla tabella ResModules (il virus **"BY BYTE BANDIT"** lo farà sicuramente, mentre il virus della "SCA" rimarrà in coabitazione pacifica) e noi saremo responsabili di quello che potrebbe accadere in seguito.

Se invece decidiamo di non passare il controllo al bootblock caricato, il colore dello schermo diventerà bianco e ci verrà offerta l'opportunità di installare il disco con il bootblock standard.

"Sostituisco questo **BOOTBLOCK** con uno standard ?"

Se il bootblock contiene un virus potremo rimpiazzarlo con quello standard.

ATTENZIONE! *Non installate il disco originale, a meno che non abbiate una copia di sicurezza, o a meno che non siate assolutamente sicuri di quello che state facendo. Alcuni dischi commerciali sono dotati di bootblock non-standard (caricatori veloci, introduzioni, ecc.) e potreste non essere più in grado di utilizzarli, una volta che il loro bootblock originale fosse stato sostituito da quello standard.*

"Il disco è protetto. Devo riprovare ?"

Un alert addizionale viene visualizzato nel caso che il disco sia protetto dalla scrittura.

"PERICOLO !!! - Non riesco a riscrivere il bootblock!"

Una nuova funzione è stata aggiunta a Guardian in questo punto. Dopo che il bootblock standard è stato scritto sul disco, il bootblock appena creato viene ricaricato in memoria e viene confrontato con quello standard. Naturalmente dovrebbero risultare uguali. Se non lo sono, vuole dire che è successo qualcosa di grave al trackdisk.device. Molto probabilmente abbiamo lanciato Guardian con il virus di **"BYTE BANDIT"** già in memoria, oppure non abbiamo resettato il computer dopo avere passato il controllo allo stesso virus e dopo avere ripristinato il vecchio ExecBase. Questo è un alert senza uscita, quindi non avremo altra possibilità se non quella di forzare un cold-reset del computer premendo uno qualsiasi dei tasti del mouse.

PERFAVORE NON RESETTATE CON CTRL-AMIGA-AMIGA A QUESTO PUNTO !
Questo lascerebbe il virus libero di tramare nell' ombra !

Un cold-reset viene ottenuto modificando opportunamente la parte iniziale della struttura dati dell'Exec, senza poi ricalcolare il corrispodente checksum. Durante il successivo reset, il computer é costretto a ricostruire tutte le strutture interne, eliminando così qualsiasi virus (ma anche lo stesso Guardian) dalla memoria. Dovremo inserire un disco ** SICURO ** (sicuramente non infetto dal virus) nel drive interno PRIMA di premere uno qualsiasi dei tasti del mouse, perché il caricamento del bootblock avverrà appena qualche istante dopo. Ricordate che il bootblock del disco che avete nel drive interno quando ricevete questo alert non è stato sostituito con quello standard. Se lasciate quindi questo disco nel drive interno, il virus si caricherà in memoria ancora una volta, perché Guardian è stato disattivato e rimosso durante il cold-reset. La cosa migliore, in questi casi, è di inserire il ** SICURO ** disco originale del Guardian (non avete mai disattivato la sua protezione contro la scrittura, vero ?!!) nel drive interno e premere uno qualsiasi dei tasti del mouse per resettare il computer.

Caricamento e utilizzo di Guardian

Ancora una volta, la cosa migliore consiste nell'utilizzare il ** SICURO ** disco di Guardian per il primo boot, appena il computer è stato acceso (e dopo che il disco del Kickstart è stato caricato sull'A1000, naturalmente!), prima di inserire QUALSIASI altro disco bootabile nel drive interno. Se non rimuovete MAI la protezione in scrittura di questo disco, sarete sempre sicuri che NESSUN virus potrà installarvisi, così quando sarete in dubbio riguardo ad altri dischi, dovrete semplicemente spegnere il computer ed utilizzare questo per il primo boot. Ci sono stati pettegolezzi e dicerie riguardo un fantomatico nuovo virus che sarebbe capace di scrivere anche su un disco protetto. Questo è assolutamente IMPOSSIBILE. L'ultimo controllo sulla protezione di un disco viene fatto direttamente dalla meccanica del floppy drive e non c'è possibilità di ingannarla. E' possibile solamente costringere il computer (il software) a credere che il disco non sia protetto. In questo modo potreste "eseguire" operazioni di scrittura su un disco protetto e il floppy drive si comporterebbe come se stesse effettivamente scrivendo su quel disco, ma alla fine dell'operazione ritrovereste i contenuti originali assolutamente intatti. Questo potrebbe quindi unicamente servire a fare qualche scherzo agli amici, ma niente di più.

Dopo avere rimosso i virus dai vostri dischi, potete copiare Guardian nelle loro directory C e richiamarlo dalle loro startup-sequence.

Guardian deve essere lanciato per primo nella startup-sequence, perché utilizza un metodo di installazione particolare. Quando viene chiamato, controlla nella tabella ResModules per vedere se vi è già presente. Se questo non è vero (come nel caso sia stato lanciato per la prima volta), Guardian si installa nella memoria e resetta il computer, per costringere la routine di reset a ricostruire nuovamente la tabella ResModules.

Da questo momento, non dovrete più preoccuparvi di lanciare ancora Guardian, perché è montato in un modulo residente e quindi la routine di reset si occupa di riattivarlo automaticamente dopo un reset o un crash di sistema. Guardian sopravviverà a qualsiasi numero di reset o crash, fino a quando il computer non sarà forzato a eseguire un cold-reset (provocato volontariamente dallo stesso Guardian o involontariamente da un programma malfunzionante, oppure anche dalla presenza di espansioni hardware non completamente conformi alle specifiche di autoconfigurazione ufficiali) o fino a quando un programma malfunzionante

non rovinerà la sua area di memoria. Quando Guardian è già attivo e viene lanciato nuovamente, visualizza un testo di copyright e ritorna immediatamente il controllo alla CLI dal quale è stato lanciato.

Il flag -a

Questa è una nuova funzione della versione 1.2. Guardian, normalmente, si installa in maniera "gentile", preservando i vettori che dovessero essere contenuti in KickTagPtr e KickMemPtr, ma non può ovviamente distinguere fra quelli "buoni" e quelli "cattivi". Il nuovo ram disk (RAMB0) che troviamo sul disco del Workbench 1.3, ad esempio, inserisce un nuovo vettore in ciascuno dei suddetti membri della struttura dell'Exec. Questi vettori sono quindi "buoni".

D'altra parte invece, i vettori creati dal virus "BY BYTE BANDIT", sono da considerarsi "cattivi" in quanto responsabili delle nefande azioni del virus stesso (per non parlare della maniera molto maleducata con la quale il virus installa di prepotenza i suoi vettori, cancellando tutti quelli che dovessero essere già presenti !). Se non volete che il contenuto di questi due membri venga preservato, potete usare il vettore -a (Arrabbiato) quando lanciate Guardian per la prima volta. Questo lo costringerà a ripulire questi vettori prima di installarsi, disattivando gli altri programmi che verranno eliminati dalla memoria durante il successivo reset. Se Guardian è già montato e viene rilanciato con il flag -a, muoverà i suoi vettori in cima alla lista e rimuoverà tutti gli altri. Notate che gli altri programmi, legati ai vettori appena rimossi, non verranno disattivati e rimossi prima del prossimo reset.

Generalmente non è necessario usare il flag -a e non è nemmeno consigliabile se state utilizzando qualcosa come il RAMB0 device o altri programmi che utilizzano la tecnica dei moduli residenti per sopravvivere ai reset.

Una situazione abbastanza comune nella quale è necessario utilizzare il flag -a è questa: Guardian non è attivo e voi fate il boot con un dischetto infetto dal virus "BY BYTE BANDIT". Il virus si attiva, quindi la startup-sequence è eseguita e Guardian viene lanciato. Se non avete usato il flag -q (ne parleremo in seguito), riceverete l'alert sui vettori di Interrupt. Sostituiteli con i valori di default. Poi lanciate Guardian di nuovo, questa volta utilizzando il flag -a, mettete un disco ** SICURO ** nel drive interno e resettate con CTRL-AMIGA-AMIGA.

Il flag -q

Un'altra nuova funzione della versione 1.2 è la capacità di verificare la tabella dei vettori di interrupt per scoprire valori non-standard e di controllare che i vettori di reset-capture (cattura del reset) siano vuoti.

Normalmente questa operazione di sicurezza viene eseguita ogni volta che Guardian viene lanciato, ma può essere disinserita utilizzando il flag -q (Quieto). Il controllo di queste zone è stato implementato perché sono molto critiche e sono utilizzate dai virus per le loro operazioni, la prima (i vettori di interrupt) dal virus "BY BYTE BANDIT", la seconda (i vettori di reset-capture) dal virus della "SCA".

Se avete uno di questi virus già in memoria quando lanciate Guardian, riceverete uno dei due alert e avrete la possibilità di ripristinare i valori di default nella tabella dei vettori di interrupt, o di azzerare i vettori di reset-capture. Potete tenere sott'occhio questi vettori lanciando Guardian di tanto in tanto, senza usare il flag -q. Se avete lanciato un programma che ne utilizza qualcuno (esempio il RAMB0 device), potete costringere Guardian a ignorare la situazione utilizzando il flag -q, altrimenti riceverete un alert ogni volta che Guardian verrà lanciato.

"ATTENZIONE !!! - I vettori di Reset Capture non sono vuoti ! Devo ripulirli ?"

Se siete infetti dal virus della "SCA", potete facilmente eliminarlo ripulendo i vettori di Reset Capture.

"ATTENZIONE !!! - I vettori di Interrupt non sono standard ! Devo sostituirli con i valori di default ?"

Se siete infetti dal virus "BY BYTE BANDIT", potete scegliere di ripristinare i vettori di interrupt standard, ma dopo questa operazione non sarete necessariamente al sicuro. Questo dipende se avete lanciato Guardian con il flag -a o meno.

Se avete usato questo flag, quando riceverete questo alert sostituite i vettori, poi Guardian rimuoverà il vettore del virus in KickTagPtr, si installerà e (se lanciato per la prima volta) resetterà il computer, costringendolo a ricostruire le tabelle di offset. In questo modo il virus sarà rimosso anche dal trackdisk.device.

Se non avete usato questo flag, il vettore residente del virus verrà conservato e quindi il parassita risulterà attivo. Se questo dovesse accadere, dovrete spegnere il computer e fare il primo boot con un disco sicuro, oppure potrete rilanciare Guardian con il flag -a e immediatamente dopo resettare il computer con CTRL-AMIGA-AMIGA per eliminare il virus dal trackdisk.device.

Il flag -k

Una nuova funzione della versione 1.2 è il flag -k. Se per qualche ragione (incompatibilità ? altamente improbabile !) desiderate liberarvi della presenza di Guardian, potete farlo utilizzando il flag -k. Il modulo residente del programma verrà rimosso dalla lista e la sua memoria sarà disponibile dopo il prossimo reset. Utilizzando il flag -k rimuoverete qualsiasi versione di Guardian attualmente attiva. Se avete in memoria la versione 1.1 e volete rimpiazzarla con la nuova v1.2, non avete bisogno di utilizzare questo flag. Basterà semplicemente lanciare la versione 1.2. L'ultima versione rimuoverà le precedenti e ne libererà la memoria. Per favore notate che non potete fare l'inverso, cioè lanciare la

v1.1 con la v1.2 in memoria, perché questo porterà a un ciclo di reset senza fine. Se vi imbattete in questa situazione, estraete il disco del boot dal drive interno e resettate con CTRL-AMIGA-AMIGA. Non dimenticatevi di sostituire la vecchia versione di Guardian con quella nuova, in tutti i vostri dischi.

I flag speciali per il Kickstart 1.3 (v34.5)

Se state utilizzando il Kickstart 1.3 (v34.5), potrete avvantaggiarvi di due flag speciali e di quattro hot-key. Con il Kickstart 1.3 (v34.5) potete fare il boot, non più solo da floppy disk, ma anche da hard disk e da ram disk (RAMB0). Ma, se volete fare il boot da ram disk, dovete mettere nel drive interno un floppy disk non installato, oppure dovete tirare fuori ogni volta, durante il boot, quello installato. Guardian vi permette di scegliere se il bootstrap dovrà testare per prima cosa la presenza del floppy disk oppure la presenza del ram disk.
Normalmente il bootstrap tenta di fare il boot da floppy disk. Se non è possibile, tenta di fare il boot da ram disk e se anche questo fallisce, appare la schermata con la richiesta di inserire il disco del Workbench.

Il flag -r

Se lanciate Guardian con il flag -r, quest'ordine di tentativi verra' cambiato. Il bootstrap tenterà di fare il boot prima da ram disk, poi da floppy, ed eventualmente richiederà l'inserimento di un disco.

Il flag -f

Potete usare il flag -f per riportare le cose alla normalità: prima prova da floppy, poi da ram.

Hot keys (tasti speciali)

Se avete selezionato il boot da ram disk e avete bisogno di forzarlo per una volta da floppy disk, potete utilizzare uno dei due tasti appositamente dedicati. Appena la luce del LED di alimentazione smette di lampeggiare durante il processo di reset, lo schermo diventa di colore grigio chiaro e poi bianco.

Non appena diventa bianco potete premere il tasto AMIGA sinistro per forzare il boot da floppy disk. Se invece premete il tasto ALT sinistro, la schermata di boot con l'etichetta di Guardian verrà visualizzata fino a quando non rilascerete il tasto e il boot avverrà anche in questo caso da floppy.

Se, al contrario, avete selezionato il boot da floppy disk e avete bisogno di forzarlo da ram disk, premete il tasto AMIGA destro. Premendo invece il tasto ALT destro verrà visualizzata la schermata di boot, come per il tasto ALT sinistro e il boot avverrà in questo caso da ram disk.
Le hot-key ALT sinistro/destro sono state implementate per permettervi di controllare la presenza dell'etichetta di Guardian sulla schermata di boot.

Notate che quando parliamo di "forzare il boot da ..." vogliamo dire che il bootstrap tenterà di fare il boot da quel particolare device PER PRIMO. Se questo non sarà possibile, tenterà comunque di fare il boot dagli altri device disponibili.

Una parola finale sul Kickstart 1.3 (v34.5)

Non sono sicuro se questa versione del Kickstart sarà quella definitiva o meno, ma dal momento che è ormai molto diffusa fra gli utenti di Amiga 1000, ho tarato questa versione di Guardian per funzionare con essa.

Una nuova versione verrà preparata non appena sarà disponibile la versione ufficiale del Kickstart 1.3.

Una parola finale su Guardian v1.2r

Per darvi un maggiore livello di sicurezza nei confronti dei virus, è stato creato Guardian v1.2r, che si installa direttamente sul disco del Kickstart, al posto della mai utilizzata funzione Debug().

In questo modo non dovrete preoccuparvi del primo boot e delle altre cose relative. Se possedete un Amiga 1000, potete utilizzare "Guardian_creator" per modificare una copia del disco del Kickstart originale.
Basta semplicemente lanciare questo programma e seguire le istruzioni che vi verranno fornite.
Adesso potete utilizzare questo Kickstart modificato al posto di quello originale.

Non potrete usare piu' i flag -a, -k, -f e -r, perché sono implementati nella routine di startup di Guardian v1.2, ma potrete sempre usare le hot-key che sono controllate dal bootstrap stesso. Se lanciate Guardian_creator per modificare un disco del Kickstart 1.3 (v34.5) vi verrà chiesto di stabilire da quale dei due device, il Floppy disk o il Ram disk, bisogna tentare per primo il boot. Questa scelta verrà "impressa" nel codice della versione Kickstart-residente di Guardian e verrà usata ogni volta che caricherete il Kickstart modificato, fino a quando non userete nuovamente Guardian_creator su quel disco.

Vi suggerisco di settare e sfruttare il boot da ram disk, in quanto é molto più veloce e di usare le hot-key ALT/AMIGA sinistro quando avete bisogno di partire da floppy disk. Mentre utilizzate un Kickstart modificato con Guardian_creator, potete comunque lanciare Guardian v1.2 nelle vostre startup-sequence, per controllare i vettori di interrupt e di reset-capture.

Guardian v1.2 funziona su A500/1000/2000 (v1.2r funziona solo su A1000), con le release 1.2 e 1.3 del Kickstart (v33.180 e v34.5).

Per ottenere il disco contenente Guardian v1.2 e Guardian _creator (v1.2r) si vedano le modalità d'ordine contenute all'interno di questa rivista.

Concetti di programmazione su Amiga

di Chris Zamara e Nick Sullivan

Amiga è una macchina complessa, ma la sua programmazione può risultare più semplice di quanto si potrebbe pensare.

Le fondamenta

In questo articolo sulla programmazione di Amiga tenteremo di rimuovere alcuni dei più comuni ostacoli che scoraggiano molti potenziali programmatori. Parleremo principalmente del linguaggio C, con qualche riferimento all'assembly del 68000. Non tenteremo di insegnarvi a programmare in C o in assembly, in quanto pensiamo che ciò possa essere fatto solo con un buon libro. Il nostro obiettivo è solamente quello di insegnarvi a programmare l'Amiga utilizzando uno di questi due linguaggi come strumento. Anche se il C e l'assembly non sono i vostri linguaggi favoriti, dovreste almeno familiarizzarvi quanto basta per seguirne il codice sorgente e gli esempi; la maggior parte della documentazione e delle spiegazioni sul Kernel di Amiga utilizzano esempi scritti in C. La conoscenza di un pò di assembly può essere utile per ottimizzare il codice compilato. Oltre a un buon libro sul C ("Linguaggio C" di Brian Kernighan e Dennis Ritchie, Gruppo Editoriale Jackson, codice 541P) sarà anche necessaria la documentazione ufficiale di Amiga: Rom Kernel, Dos e Intuition manual, che si trovano purtroppo solo in edizione originale americana, editi dalla Addison-Wesley.

Se pensate che la programmazione di Amiga sia terribilmente complicata e siete spaventati dal numero di manuali che occorrono per padroneggiarla, ecco un piccolo segreto nel quale potreste essere interessati: programmare l'Amiga è FACILE. Per quanto possa sembrare difficile all'inizio, la verità è che il sistema operativo fa talmente tante cose per noi, che l'eseguire complicate operazioni molto spesso si riduce al semplice riempire un formulario, inserendo i valori desiderati in una struttura di sistema, quindi nel passarlo all'apposita routine che farà tutto il lavoro. In questo articolo verranno trattati tre argomenti che possono risultare particolarmente confusi ai nuovi programmatori di Amiga: le strutture, i file include e le librerie.

Le strutture

I programmatori C sono familiari con il concetto di struttura, che può essere pensata come uno strano array nel quale possono essere contenute variabili di differenti tipi. Questa sezione dell'articolo può contenere concetti già conosciuti da coloro che programmano in C, ma vengono qui riproposti perché sono particolarmente importanti sull'Amiga. Il sistema operativo fa largo utilizzo delle strutture come mezzo per passare pacchetti di dati fra diverse funzioni senza essere costretto a dichiarare ogni loro componente come variabili separate.

Consideriamo, per esempio, la funzione OpenWindow contenuta nell'Intuition.library. Per aprire una finestra è sufficiente passare a questa funzione il puntatore a (cioè l'indirizzo di) una struttura "NewWindow" che contiene le informazioni necessarie riguardanti la finestra che intendiamo aprire. La funzione OpenWindow e il nostro programma conoscono la definizione (chiamata anche "template") della struttura NewWindow, in quanto l'hanno letta in uno dei file include standard (trattati nella sezione successiva di quest'articolo). La funzione ci restituisce il puntatore a una copia della struttura "Window", che il nostro programma può utilizzare per conoscere i dati della finestra che è stata appena aperta (la sua larghezza, ecc.) e come argomento per chiamare altre funzioni di Intuition. Le strutture sono anche usate estensivamente nel sistema operativo come mezzo di comunicazione fra task, anche se non dovremo preoccuparci di questo aspetto per lo sviluppo di un buon numero di applicazioni.

É importante avere ben chiara in mente la differenza fra la dichiarazione di una struttura e l'esemplare di quella struttura che viene realmente utilizzato. Per esempio, possiamo definire una struttura "foo" in questo modo:

```
struct foo {
 char name[20];
 unsigned int RefNumber;
 struct foo *NextGuy; /* puntatore a un'altra struttura */
};
```

A questo punto "struct foo" diventa un tipo di variabile che può essere utilizzato nelle dichiarazioni. In questo modo, se vogliamo dichiarare due strutture di tipo "foo", potremo scrivere:

```
struct foo Acquirente, Cliente;
```

Adesso "Acquirente" e "Cliente" sono strutture di tipo "foo". Sottolineiamo che foo è la definizione della struttura, mentre Acquirente e Cliente sono due esemplari reali della struttura foo. Essi esistono veramente in memoria, occupano uno spazio, mentre la definizione della struttura può essere pensata come un "libretto di istruzioni" che spiega al C come costruire e utilizzare ogni copia della struttura che verrà successivamente dichiarata e utilizzata. Ci si può riferire ai singoli membri di Acquirente e Cliente utilizzando l'operatore punto "." come mostrato di seguito:

```
n = Cliente.RefNumber;
```

Le definizioni delle strutture di sistema sono generalmente contenute nei file include che il nostro programma utilizza, mentre le

strutture in sè sono dichiarate nel nostro programma. Per utilizzare un nostro esemplare della struttura NewWindow chiamandolo "MyNewWindow", dichiareremo:

```
struct NewWindow MyNewWindow;
```

Un'altra importante distinzione da fare è la differenza che esiste fra il puntatore a una struttura e la struttura stessa. Le funzioni che passano gli argomenti attraverso le strutture, in realtà utilizzano i puntatori a queste strutture. Non bisogna fare confusione quando si opera contemporaneamente con le strutture e con i relativi puntatori, in quanto gli elementi di queste strutture vanno indirizzati in maniera differente a seconda di quale dei due casi si presenta. Abbiamo dichiarato prima una struttura NewWindow chiamata MyNewWindow; dichiariamo adesso un puntatore a una struttura di tipo "Window" e chiamiamolo "MyWindowPtr":

```
struct Window *MyWindowPtr;
```

Per accedere a un membro di MyNewWindow scriviamo:

```
MyNewWindow.LeftEdge = 100;
```

Per accedere invece a un membro di MyWindowPtr scriviamo:

```
x = MyWindowPtr->Width;
```

L'operatore freccia (->), costruito con i caratteri "meno" e "maggiore", viene utilizzato per accedere all'elemento di una struttura della quale abbiamo il puntatore, mentre l'operatore punto (.) viene impiegato quando utilizziamo la struttura stessa. Nell'esempio di foo abbiamo visto che una struttura può contenere puntatori ad altre strutture; questa possibilità viene usata spesso per creare liste concatenate di strutture dello stesso tipo.

In assembly le strutture vengono simulate definendo il nome di ogni elemento come un offset (distanza) dall'inizio della struttura alla quale appartiene. Queste definizioni, come già succedeva per quelle C, sono contenute nei file include. (La maniera nella quale sono fatte queste definizioni è molto interessante, in quanto simulano la sintassi C sfruttando le macro; vale la pena di dare un'occhiata a qualche file include per vedere come sono fatte.) Viene definito anche il numero di byte che la struttura occupa, in modo che si possa allocare lo spazio nei nostri programmi per tutte le strutture di cui abbiamo bisogno. Questo spazio e il nome che gli diamo formano la struttura vera e propria e il suo nome. Possiamo accedere a un elemento della struttura utilizzando il nome dell'elemento (definito nei file include) come un offset dal nome della struttura. Per esempio, per "dichiarare" una struttura NewWindow di nome MyWindow, riserviamo lo spazio sufficiente (probabilmente alla fine del nostro programma) in questo modo:

```
MyWindow   DS.B  nw_SIZE
```

Adesso possiamo mettere l'indirizzo della struttura in un registro:

```
LEA  MyWindow,a2
```

e immagazzinare un valore in un membro della struttura come segue:

```
MOVE  #100,nw_Width(a2)
```

Questo basta per le strutture; passiamo adesso a un altro argomento!

I file include

Una cosa alla quale si fa fatica ad abituarsi quando si programma per la prima volta su Amiga è rappresentata dalla moltitudine di istruzioni INCLUDE che bisogna sempre mettere all'inizio di ogni file sorgente. Su altri sistemi si possono scrivere molti programmi senza includere altro che "stdio.h". Su Amiga è una storia diversa, come si può verificare osservando il codice sorgente di un qualsiasi programma scritto specificamente per il nostro computer o esaminando il labirinto di file contenuti nella directory INCLUDE di un disco di sviluppo C.

I file include che bisogna richiamare nel nostro programma dipendono largamente da quali librerie e device utilizziamo. Per esempio, se si utilizza la libreria di Intuition (parleremo delle librerie più avanti), bisognerà includere il file "intuition/intuition.h". Se non siamo sicuri su quali file vadano inclusi o meno, possiamo utilizzare il metodo che adesso spiegheremo per tentare di compilare e di effettuare il link di un programma. Quando riceviamo errori indicanti che una certa struttura, costante o macro non è definita, useremo il comando SEARCH per localizzare il suo nome nella directory include, quindi INCLUDEremo il file nel quale verrà trovata. (Alternativamente si potranno utilizzare delle apposite tabelle che contengono l'elenco in ordine alfabetico di tutte le strutture, costanti e macro insieme ai file nei quali si trovano. Quelle stampate nei libri Amiga ROM Kernel Manual Vol 1 e 2 si riferiscono alla vecchia versione 1.1 del sistema operativo. Meglio utilizzare quelle contenute nei file costantemente aggiornati e pronti per essere stampati, che vengono pubblicati periodicamente nei dischi di Fred Fish, disponibili presso la redazione. N.d.T.) I nomi dei file include finiscono sempre in ".i" per l'assembly o in ".h" (come header, intestazione) per i programmi C. Le informazioni contenute nei due tipi sono fondamentalmente le stesse. Vi troveremo i seguenti oggetti:

Definizioni (di costanti)
Queste stabiliscono valori strettamente legati al sistema operativo per rappresentare modi, condizioni di errore, flag e così via. Quando apriamo una finestra attraverso Intuition, per esempio, una delle cose che possiamo specificare é la serie di gadget di sistema (il gadget di dimensionamento, la barra di trascinamento, ecc.) che vogliamo utilizzare. Possiamo quindi scrivere una linea di questo tipo:

```
newwindow.flags = WINDOWDRAG | WINDOWCLOSE;
```

che equivale allo scrivere:

```
newwindow.flags = 0x0002 | 0x0008;
```

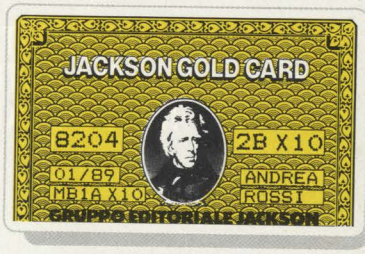

INFORMATICA 60
SETTIMANALE
Anche le reti locali nel mirino di Olivetti
Wyse Technology obiettivo Italia
MECCANICA OGGI
SETTIMANALE 1
SETTIMA
ELETTRONICA · AUTOMAZIONE
La fabbrica elettronica au
BIT
SPECIALE UNIX
VETRINA
Tutto su AppleFest
Compaq a 25 MHz
INFORMATICA
elettronica OGGI
Speciale: SOFTW DI SIMU
STRUMENTAZIONE & MISURE OGGI
PC Floppy
PC MAGAZINE
Summa, gestionale per tutti
PC WORLD
PC MAGAZINE
Amiga Transactor
Trasmissione Dati
Telecomunicazioni
SUPER COMMODORE 64 e 128
CON DISCO
ROOT RACE
OIL DEFENSE
SKETCH PAD PLUS
ML CLONER
BASICALLY MUSIC
OMPUTER GRAFICA
APPLICAZIONI
AMIGA MAGAZINE
CONTIENE:
CONTIENE DUE FLOPPY DISK CON PROGRAMMI
Compu Scuola
La rivista di informatica nella didattica per la scuola italiana
NUMERO SPECIALE
dedicato agli Atti di
SCUOLA 2000
olivetti PRODEST
USER
LA PRIMA E UNICA RIVISTA INDIPENDENTE PER GLI UTENTI PC 128-PC 128 S-PC 1
PC 1
Penna, penna, come te non c'è nessuno Xtree
PC 128 S
Sistema musicale Le risposte di Eliza
PC 128
Sistemi di numerazione Quadrato cinese
IL L.M. DEL PC 128
PC GAMES
IN QUESTO NUMERO:
CONCENTRATION • FREESTYLIN
TIC TAC TOE • ULTIME NOVITA'
CALCIO INDOOR
N.39 SETTEMBRE '88
ELET
Realizzazioni pratiche
Tachimetro per bicicletta
Audiometro
Controller per impianti di riscaldamento
Voltmetro digitale per MSX
Accordatore d'antenna

ABBONAMENTO JACKSON = SERVIZIO COMPLETO

Da quest'anno l'abbonamento alle riviste Jackson offre una serie innegabile di vantaggi e servizi: anzitutto lo sconto eccezionale del 40% sul prezzo di copertina, pressochè doppio rispetto al passato, che Jackson ha voluto proporre ai lettori

per celebrare il decimo anno di attività. Inoltre, abbonarsi a Jackson garantisce l'accesso a una rete multinazionale di informazioni, grazie al recente accordo azionario con la VNU Business Press Group, maggiore editore tecnico internazionale del settore. Ma c'è di più: **la Jackson Gold Card,** per l'identificazione immediata del codice abbonamento, sarà recapitata gratuitamente agli abbonati e permetterà al titolare di usufruire di molteplici servizi gratuiti quali: sconto del 20% fino al 28/2/1989 e del 10% dopo tale data, sul prezzo di copertina di libri e software Jackson, per acquisti effettuati direttamente dall'editore, oltre a una serie di sconti per acquisti vari presso librerie, computershop e altri esercizi convenzionati in tutta Italia.

In più, il titolare di **Jackson Gold Card** potrà ottenere sconti sui corsi di formazione della Jackson S.A.T.A., la scuola Jackson di Alte Tecnologie Applicate, oltre all'abbonamento gratuito a 6 numeri di uno (a scelta) dei tre settimanali Jackson: "E.O. News Settimanale di Elettronica", "Informatica Oggi Settimanale" o il nuovissimo "Meccanica Oggi", annunciato per l'inizio del 1989.



Infine, l'abbonato ha diritto all'invio personalizzato e riservato dei cataloghi libri e della nuova rivista **"Jackson Preview Magazine",** con l'annuncio di tutte le novità editoriali Jackson.

GRUPPO EDITORIALE JACKSON

PRIMO NELLA BUSINESS-TO-BUSINESS COMMUNICATION

1° PREMIO

HONG KONG • BANGKOK • SINGAPORE

UN FANTASTICO VIAGGIO
IN ESTREMO ORIENTE
IN COLLABORAZIONE CON:

CENTRO TURISMO

swissair

Sheraton

I LEADER PER UN VIAGGIO DI SUCCESSO

2° PREMIO

UN COMPUTER
AMIGA 2000

6°

UN COMPUTER

dal 7° al 10° PREMIO

4 PACCHETTI SOFTWARE
"Commodore Software by CTO"

ABBONAMENTO JACKSON = FORTUNA STREPITOSA

AUT. MIN. RICH.

Abbonarsi alle riviste Jackson significa leggere il meglio, risparmiando il 40%, in informatica, elettronica e nuove tecnologie, ma soprattutto partecipare al grande concorso Jackson riservato agli abbonati, con la possibilità di vincere premi favolosi.

REGOLAMENTO DEL CONCORSO

1 – Il Gruppo Editoriale Jackson S.p.A. promuove un concorso a premi in occasione della Campagna Abbonamenti 1988/1989.
2 – Per partecipare è sufficiente sottoscrivere, entro il 31.3.1989, un abbonamento a una delle 30 riviste Jackson.
3 – Sono previsti 10 favolosi premi da sorteggiare fra tutti gli abbonati.
4 – Primo premio: un viaggio per due persone in Estremo Oriente, che prevede: passaggi aerei Swissair, pernottamenti in Hong Kong, Bangkok e Singapore, presso gli hotel: Royal Orchid Sheraton e Sheraton Towers della catena Sheraton Hotel, nonchè escursioni in luogo nelle tre suddette località.
Gli altri nove premi consistono rispettivamente (in ordine di esposizione) in:
1 computer Amiga 2000 completo di unità centrale con 1 MB di memoria, dischetto da 3" 1/2, tastiera, mouse, sistema operativo e monitor a colori 1084.
1 personal computer PC 20 III SERIE completo di unità centrale con 640 KB di memoria, dischetto da 5" 1/4, hard disk da 20 MB, mouse 1352, sistema operativo MS-DOS 3.20 monitor monocromatico e tastiera.
1 personal computer PC1 completo di unità centrale con memoria 512 KB, dischetto da 5" 1/4, tastiera, monitor monocromatico, sistema operativo MS-DOS 3.20 e GW-Basic.
1 computer Amiga 500 con 512 KB Ram e 256 KB Rom di memoria, sistema operativo e monitor a colori 1084.
1 computer "nuovo C64" completo di manuali e sistema operativo.
Dal settimo al decimo premio incluso, n. 4 pacchetti "Commodore Software by CTO".
5 – Gli abbonati a più di una rivista avranno diritto, per l'estrazione, all'inserimento del proprio nominativo tante volte quante sono le testate sottoscritte.
6 – L'estrazione dei 10 premi in palio avverrà presso la sede del Gruppo Editoriale Jackson entro il 30.5.1989.
7 – L'elenco dei vincitori, ad estrazione avvenuta, pubblicato su almeno 10 delle riviste Jackson. La vincita inoltre sarà pubblicata con lettera raccomandata a ciascuno dei sorteggiati.
8 – I premi verranno messi a disposizione degli aventi diritto entro 30 giorni dalla data dell'estrazione, ad esclusione del primo premio, il quale dovrà essere effettuato, compatibilmente con la disponibilità dei posti, in un periodo da definirsi, entro il 31.12.1989.
9 – Le spese di vitto relative al viaggio, nonchè l'eventuale controllo di manutenzione extra garanzia per i personal computer Commodore, saranno a carico dei rispettivi vincitori.
10 – I dipendenti, i familiari, i collaboratori del Gruppo Editoriale Jackson sono esclusi dal concorso.

Sempre quest'anno, il concorso abbonamenti Jackson prevede un primo premio veramente eccezionale: la possibilità di esplorare il misterioso Estremo Oriente, in un viaggio che unisce il fascino di una tradizione millenaria ad uno sviluppo tecnologico senza precedenti.

Il viaggio, di oltre dieci giorni per due persone, è studiato nei minimi dettagli, per offrire il miglior comfort e le migliori ospitalità ed è garantito da tre leader di primissimo livello: Acentro Turismo di Milano, Swissair e Sheraton Hotels.

Non solo. Ad altri nove abbonati fortunati, il Gruppo Editoriale Jackson, in collaborazione con Commodore Computer e CTO, riserva altri premi eccezionali, dalla più completa gamma di computer di successo: un favoloso personal computer Amiga 2000, un Commodore PC20 III serie, un Commodore PC1, un Amiga 500 e un nuovo C64, in palio dal secondo al sesto estratto. Quattro pacchetti "Commodore Software by CTO" saranno inoltre sorteggiati dal settimo al decimo premio.

Partecipare al concorso è semplice: basta abbonarsi a una o più tra le riviste Jackson (chi si abbona a più riviste ha, naturalmente, più possibilità di vincita), utilizzando la speciale Cartolina/Questionario, già predisposta e affrancata, da compilare in ogni sua parte e restituire all'editore.
Affrettatevi! Abbonatevi per vincere!

Commodore
LEADER IN PERSONAL COMPUTER

GRUPPO EDITORIALE JACKSON
1 PRIMO NELLA BUSINESS-TO-BUSINESS COMMUNICATION

ABBONAMENTO JACKSON = RISPARMIO ECCEZIONALE

Abbonarsi è semplice: basta compilare in ogni sua voce la speciale Cartolina/Questionario già predisposta e affrancata e rispedirla all'editore.

Per il versamento dell'importo dell'abbonamento, utilizzate, preferibilmente l'apposito modulo di C.C.P. già predisposto e allegato alla rivista.

SERVIZIO QUALIFICAZIONE LETTORI

SPECIALE: PER CHI ACQUISTA LE RIVISTE JACKSON IN EDICOLA

Da quest'anno il Gruppo Editoriale Jackson ha predisposto uno **Speciale Servizio di Qualificazione Lettori e Abbonati,** che prevede l'assegnazione di una serie di dati relativi agli interessi specifici di ognuno, per poter offrire un servizio adeguato alle reali esperienze di aggiornamento del lettore.

Tutti i lettori interessati allo **Speciale Servizio Qualificazione Lettori,** e quindi anche i non abbonati, devono restituire, compilata nella parte **Qualificazione lettori,** la **Cartolina Questionario** già predisposta e affrancata.

Per chi la spedirà, il Gruppo Editoriale Jackson garantisce fin d'ora **GRATUITAMENTE:**

- Jackson Silver Card, che offre tutti i vantaggi della Gold Card, esclusi gli sconti sui libri riservati agli abbonati.

- Invio gratuito del **Catalogo Generale Libri Jackson.**
- Invio gratuito della **Jackson Preview Magazine.**
- **Abbonamento gratuito a sei numeri,** a scelta tra le seguenti riviste settimanali: E.O. News Settimanale - Informatica Oggi Settimanale - Meccanica Oggi (pubblicato da febbraio '89)

Area	Testate	Numeri Anno	Tariffa abbonam.	Tariffa intera
Elettronica e automazione	EO News Settimanale	40 + 6 omaggio	£. 59.500	£. 100.000
	Elettronica Oggi	20	£. 60.500	£. 100.000
	Automazione Oggi	20	£. 60.000	£. 100.000
	Meccanica Oggi	40 + 6 omaggio	£.59.000	£. 100.000
	Strumentazione e Misure Oggi	11	£ 39.000	£. 66.000
Informatica e Personal Computer	Informatica Oggi Settimanale	40 + 6 omaggio	£. 61.000	£. 100.000
	Informatica Oggi mese	11	£. 33.500	£. 55.000
	BIT (quindicinale da Gennaio)	20	£. 48.000	£. 80.000
	PC Magazine	11	£. 32.500	£. 55.000
	PC Floppy	11	£. 79.500	£. 132.000
	Computergrafica e applicazioni	11	£. 39.500	£. 66.000
	Trasmissione dati e Telec.	11	£. 34.000	£. 55.000
	Compuscuola	10	£. 24.500	£. 40.000
Tecnologie e mercati	WATT (quindicinale da Gennaio)	20	£. 36.500	£. 60.000
	LAB. NEWS	10	£. 30.000	£. 50.000
	Industria Oggi	11	£. 34.500	£. 55.000
	Media Production	11	£. 46.500	£. 77.000
	Strumenti musicali	11	£. 32.000	£. 55.000
Hobby e Home Computer	Fare Elettronica	12	£. 36.000	£. 60.000
	Amiga Magazine disk	11	£. 92.500	£. 154.000
	Amiga Transactor	6	£. 25.500	£. 42.000
	Commodore Professional 64/128 disk	11	£. 85.000	£. 143.000
	Commodore Professional 64/128 cass.	11	£. 59.500	£. 99.000
	Supercommodore 64/128 disk	11	£. 79.000	£. 132.000
	Supercommodore 64/128 cassetta	11	£. 49.500	£. 82.500
	Olivetti Prodest User	6	£. 18.000	£. 30.000
	PC Software	11	£. 66.000	£. 110.000
	PC Games 5 1/4"	11	£. 93.000	£. 154.000
	PC Games 3 ½	11	£. 99.500	£. 165.000
	3 ½ Software	11	£. 99.000	£. 165.000

Lo sconto del 40% è stato calcolato, in certi casi, arrotondando le cifre in modo da differenziare le tariffe di ciascuna rivista per esigenze di gestione.

GRUPPO EDITORIALE JACKSON

PRIMO NELLA BUSINESS-TO-BUSINESS COMMUNICATION

o direttamente:

```
newwindow.flags = 0x000a;
```

I valori numerici di WINDOWDRAG e WINDOWCLOSE sono definiti, insieme a un gran numero di altre cose, dalle istruzioni #define contenute nel file "include/intuition/intuition.h". Volendo, avremmo potuto scrivere direttamente i numeri nel listato del nostro programma, ma ciò sarebbe stato stupido; i nomi di questi flag sono molto più facili da ricordare e rendono i nostri programmi indipendenti dalle revisioni del sistema operativo.

Macro: queste mettono a disposizione un metodo veloce e comodo per incorporare nei nostri programmi pezzi di codice utilizzati ripetutamente. Un classico esempio é rappresentato dalla macro MAX:

```
#define MAX(x,y) ((x)>(y)?(x):(y))
```

Alcuni file include contengono definizioni di macro per facilitare operazioni specifiche di Amiga, come quella di modificare i bit delle variabili di sistema. Vediamo un esempio tratto dal file "include/graphics/gfxmacros.h":

```
#define SetWrMsk(w,m) {(w)->Mask = m;}
```

Si potrebbe facilmente digitare nel sorgente il codice finale, eliminando il bisogno di usare una macro semplice come questa (ad es. rp->Mask = x; invece di SetWrMsk(rp,x);), ma questo non garantirebbe ai nostri programmi la compatibilità con le revisioni future del sistema operativo. I manuali di Amiga raccomandano sempre di utilizzare le macro standard anziché sostituirle manualmente con il codice reale.

Strutture: la definizione di strutture è uno degli scopi principali dei file include. Il sistema operativo fa largo uso delle strutture per gestire ogni tipo di informazione necessaria a conoscere lo stato attuale della macchina. Nella maggior parte dei casi i nostri programmi comunicano con le routine del sistema operativo per mezzo di strutture conosciute a entrambi.

Typedef (definizioni di tipo): nel file include "include/exec/types.h" possiamo trovare un gran numero di istruzioni typedef che definiscono i tipi di variabili comunemente usati. Queste istruzioni hanno due scopi: rendere i nostri programmi più facilmente trasportabili da un sistema all'altro e più comodi da scrivere. Vediamo due esempi presi dal file menzionato prima:

```
typedef short SHORT;
typedef unsigned char *STRPTR;
```

Se, per esempio, dichiariamo una variabile come SHORT (16 bit) e vogliamo in seguito trasportare il programma su un compilatore che utilizza short a 8 bit, potremo semplicemente cambiare l'enunciato typedef senza essere costretti a cambiare tutte le dichiarazioni di variabili. La seconda istruzione typedef ci aiuterà a scrivere programmi più corti e più leggibili, dal momento che possiamo dichiarare puntatori a stringhe scrivendo:

```
STRPTR x,y;
```

invece che essere costretti ad utilizzare:

```
unsigned char *x,*y;
```

Commenti: i file include di Amiga sono pieni di commenti che spiegano la composizione di tutte le strutture e in molti casi anche il loro utilizzo. Si possono ricavare molte informazioni preziose leggendo questi file. Per risparmiare spazio prezioso sul nostro disco di sviluppo, possiamo rimuovere questi commenti (a questo scopo sono disponibili alcuni programmi di pubblico dominio), ma dovremmo tenere a portata di mano una copia dei file originali per qualsiasi riferimento potesse essere necessario a chiarire qualche concetto espresso poco chiaramente nei manuali.

Le librerie

Il termine "library" (libreria) può generare confusione su Amiga, in quanto ne esistono due tipi ben diversi. Ci sono le librerie standard che vengono utilizzate durante il link di qualsiasi programma C o assembly. Per esempio, bisogna effettuare il link con una di queste librerie per includere nel nostro programma quelle funzioni standard come printf, strcmp, ecc. Con il compilatore C della Lattice per Amiga dobbiamo utilizzare "c.o" per rendere disponibili queste funzioni, mentre l'equivalente del compilatore Aztec C della Manx si chiama "c.lib". Questo significato della parola "library' dovrebbe già essere familiare a coloro che hanno avuto eperienze di programmazione C o assembly su altri computer.

Ma proprio per metterci in crisi (andiamo, non pretenderete che sia COSI' semplice, vero?), esiste un altro tipo di librerie su Amiga. Per accedere a una qualsiasi delle routine del sistema operativo dobbiamo aprire una libreria. Ne sono disponibili parecchie, ciascuna delle quali copre una differente caratteristica del sistema operativo. Un esempio è costituito dalla libreria grafica della ROM, la quale contiene al suo interno tutte le primitive per disegnare linee, stampare testo, colorare zone, ecc. Il programmatore di Amiga ha a disposizione più di una dozzina di librerie di questo tipo; alcune di queste sono in ROM, altre sono contenute nella directory "libs" del disco Workbench, ma sono tutte utilizzate nella stessa maniera se il programmatore richiede una libreria che non è presente in ROM, il sistema operativo chiederà di inserire il disco con il quale è stato effettuato il boot, qualora questo non sia già nel drive.

Per potere utilizzare una qualsiasi delle routine di sistema, dovremo "aprire" (open) la libreria nella quale si trova la funzione desiderata. La procedura di apertura di una libreria restituisce il puntatore alla base della libreria "library base", che consiste nell'indirizzo in memoria al quale si trova una tabella (jump table) contenente gli indirizzi di tutte le routine all'interno di quella libreria. La base di una libreria non è costante: quelle che si

trovano su disco (disk-based libraries) possono essere caricate in qualsiasi indirizzo al momento del loro utilizzo, mentre quelle contenute in ROM (ROM-based) possono trovarsi in posizioni differenti in ogni nuova versione del sistema operativo. Non dovremo mai preoccuparci della posizione della base della libreria: in C basterà aprire quella desiderata utilizzando la funzione "OpenLibrary" per poterne chiamarne tutte le routine esattamente come faremmo con qualsiasi altra funzione. In assembly è un pò più complicato, quindi ne rimanderemo la discussione a un'altra volta.

Per prima cosa esamineremo come vengono lanciate le funzioni della libreria. Non c'è bisogno di sapere tutte queste cose per potere utilizzare una routine di sistema, ma parecchi programmatori preferiscono sapere cosa stia realmente accadendo, anziché limitarsi a utilizzare un incantesimo solo perché funziona.

Quando chiamiamo la OpenLibrary, le passiamo come argomento il nome della libreria che desideriamo utilizzare (ad esempio "graphics.library") e la versione minima del sistema operativo richiesta dal nostro programma per girare correttamente (zero, se la versione non ha importanza). Ci verrà restituito il puntatore alla base della libreria. A questo punto il programma può chiamarne qualsiasi routine utilizzando gli offset dalla base. Come si ottengono i valori di questi offset? Sono definiti nella libreria utilizzata durante il link ("amiga.lib" con l'assembler e il compilatore standard). Nelle locazioni di memoria sotto la base della libreria si trova la jump table con gli indirizzi di tutte le routine contenute, proprio come la Kernal jump table del Commodore 64/128. Sopra la base si trova invece una struttura che contiene i dati globali necessari alle routine della libreria.

A questo punto avreste potuto notare che c'è qualcosa che non va. Abbiamo detto che per poter utilizzare una funzione qualsiasi di sistema, bisogna aprire la libreria che la contiene. Abbiamo anche detto che per aprire una libreria bisogna utilizzare la routine OpenLibrary. "Hmmmmm, ma allora come si fa ad aprire la libreria che contiene la OpenLibrary?". Bella domanda. Questa funzione è contenuta in "exec.library", che può essere rintracciata grazie all'unica locazione di memoria fissa dell'intero sistema operativo. La locazione 000004 contiene il puntatore a "ExecBase", la base della libreria dell'Exec.

Quando programmiamo in C, la libreria dell'Exec viene automaticamente aperta senza bisogno del nostro intervento (lo stesso avviene per la libreria del DOS), ma in assembly dovremo utilizzare la locazione 4 per procurarci il puntatore a ExecBase. Una volta ottenuto, potremo utilizzare l'offset (il cui valore verrà definito durante il link) per aprire la libreria.

Quando riceviamo il puntatore alla base della libreria dalla OpenLibrary, dobbiamo assegnarlo a una variabile avente un nome ben specifico, che permetterà alle routine contenute in "amiga.lib" di conoscere la dislocazione della base della libreria. Anche il nostro programma potrà utilizzare questa variabile, dal momento che punta alla struttura della libreria (ricordiamo che la jump table è sotto la base della libreria, mentre la struttura è sopra).

I nomi di tali variabili per alcune delle librerie di sistema sono:

Libreria	Nome della base
exec.library	ExecBase
dos.library	DosBase
intuition.library	IntuitionBase
graphics.library	GfxBase
layers.library	LayersBase
mathffp.library	MathBase
translator.library	TranslatorBase

Dal momento che la base della libreria viene utilizzata come puntatore alla struttura della libreria stessa, dovremo dichiararla come tale (la struttura della libreriaè stata definita, come al solito, in uno dei file include che abbiamo richiamato all'inizio del nostro programma). Dal momento che viene utilizzata per aprire tutti i tipi di librerie disponibili, ognuna avente la propria definizione di struttura, bisognerà dichiarare la funzione OpenLibrary come se restituisse un generico puntatore. Poiché la variabile che contiene la base della libreria è stata dichiarata come puntatore alla struttura della libreria, dovremo effettuare l'operazione di "cast" nell'assegnamento della funzione OpenLibrary. L'esempio seguente mostra come aprire una libreria, in questo caso quella di Intuition:

```
#include <intuition/intuition.h>  /* per la libreria di Intuition */
#include <exec/types.h>  /* per le definizioni dei tipi */
/* dichiariamo la OpenLibrary come restituente un puntatore */
/* (APTR è definito in "exec/types.h" come generico puntatore *
APTR OpenLibrary();
/* dich. la base della libreria come puntatore alla struttura della
lib. */
struct IntuitionBase *IntuitionBase;
main()
{
  /* apre la libreria ed effettua cast sul risultato */
Intuition Base = (struct Intuition Base *)

OpenLibrary("intuition.library",0L);
  /* controlla se OpenLibrary ha fallito per qualche motivo */
  if (IntuitionBase == NULL)
    exit(0L);  /* se ha fallito usciamo dal programma */


   DoStuff();  /* la nostra funzione che utilizza le chiamate a
Intuition */
  /* dobbiamo chiudere ogni libreria aperta */
  CloseLibrary(IntuitionBase);
}
```

Non è strettamente necessario chiudere le librerie contenute in ROM; se non vengono chiuse invece quelle normalmente contenute sul disco Workbench e caricate in RAM durante il loro utilizzo, il sistema operativo non potrà eliminarle dalla memoria se avrà bisogno di spazio. Dal momento che tutte le librerie sono fatte nella stessa maniera e non vogliamo preoccuparci di scoprire quali sono in ROM e quali no (anche dal momento che una libreria che OGGI è in ROM, DOMANI, in una nuova versione del sistema operativo, potrebbe non esserlo più! N.d.T.), i nostri programmi dovranno sempre chiudere tutte le librerie che sono state aperte.

...E di cosa sono fatti i piccoli font?

di Betty Clay

Tutto quello che avete sempre desiderato conoscere sui filedi definizione dei font

Betty Clay èun'insegnante di matematica ("Ritirata finalmente!", dice) per professione e una studiosa di computer della Commodore per vocazione. Betty può essere raggiunta via Compuserve (74145,657), QuantumLink (bjc), o via posta al 1322 South Oak Street, Arlington, Texas, 76010.

Se esaminiamo la directory FONTS: sul nostro disco Workbench, vedremo un certo numero di directory (una per ogni font presente su disco, come Opal e Garnet) e, sotto i nomi delle directory, un gruppo di file con gli stessi nomi ma con l'aggiunta dell'estensione ".font".

Spingendosi più in dettaglio, guardando in una di quelle directory (per esempio la directory Garnet), troveremo due file con degli strani nomi: '9' e '16'.

Cosa sono tutti questi file e cosa significano? Come è strutturato un font sull'Amiga?

I file .font

Questi file sono, in realtà, le intestazioni dei font e ne contengono solo una descrizione. Se è disponibile solamente una misura o uno stile per un font, la sua intestazione (header) contiene esattamente 264 byte. Possiamo vedere il contenuto di questo file in forma esadecimale utilizzando questo comando:

type fonts/garnet.font opt h

Questa è una parte di quello che vedremo:

```
0F000002 6761726E 65742F31 36000000
```

Seguirà un gran numero di zeri. Il significato di questi numeri può essere interpretato molto facilmente:

0F00 indica che questo file è proprio l'intestazione di un font.

0002 significa che ci sono due definizioni (dimensioni o stili) per questo font.

La parte rimanente del file descrive il primo dei font Garnet, chiamato Garnet 16.

6761726E 6574 rappresentazione ASCII di "Garnet".

2F31 36000000 il numero 2F rappresenta '/' mentre 3136 è la rappresentazione ASCII di '16', l'altezza in pixel di questo font.

Quindi ci saranno 244 byte di zeri e alla fine della descrizione troveremo queste informazioni:

0010 $10=16, l'altezza del font in pixel.

00 lo stile del font. Settando opportunamente i bit di questo byte possiamo definire lo stile del font, come sottolineato (underlined), grassetto (bold) o italico (italic). Nel nostro esempio non viene dichiarato alcuno stile particolare, ma in caso contrario, quando lo stile viene dichiarato nel font in questo modo, non potrà più essere cambiato tramite (per esempio) il menu del Notepad.
62 i flag di questo font. In questo caso il 6 indica che è un font 'disegnato' anziché 'costruito' e che è proporzionale anziché a larghezza fissa. Il 2 indica che è un font caricato da disco. Una lista completa delle definizioni di questi flag si trova nell'header file "graphics/text.h" contenuto nella directory INCLUDE del compilatore C.
Nel file "Garnet.font" i dati appena esaminati sono seguiti immediatamente dalla descrizione dell'altro font "Garnet/9". Questa è esattamente la stessa vista prima, con un'unica eccezione: la penultima word, la dimensione verticale (Ysize) o altezza in pixel del font Garnet/9 è 0009.
Ci sarà una descrizione simile, ognuna contenente lo stesso tipo di informazioni, per ogni dimensione o stile in ciascuna 'famiglia' di font. Il file "nome_del_font.font" può diventare molto lungo se ci sono svariate dimensioni o stili nella stessa famiglia.
E tutti quegli zero? Sono solamente di riempimento, messi lì per riservarci tutto lo spazio di cui potremmo avere bisogno (fino a 256 byte) per specificare all'AmigaDOS l'intero percorso (path) che deve seguire per localizzare il nostro font.

Quei file con dei numeri come nome...

Utilizzando ancora il font Garnet come esempio, la cosa che esamineremo successivamente è la sottodirectory chiamata "Garnet". Per raggiungerla chiediamo della directory "Fonts:Garnet". Lì troveremo due file chiamati '16' e '9'.
La definizione del font è contenuta in questi file chiamati con l'altezza in pixel del font stesso. Il loro contenuto può essere estremamente complesso. Nel caso di un font a larghezza fissa è necessario un numero di informazioni abbastanza ridotto, ma un font proporzionale occupa quattro tabelle separate con le informazioni richieste per la sua completa descrizione.
La prima parte di qualsiasi file di definizione contiene una struttura node (nodo) che permette al font di essere messo in relazione al resto del sistema, uno spazio che verrà riempito con il nome del font non appena terminato il suo caricamento e una struttura dati di tipo "TextFont" contenente le informazioni richieste dal sistema per gestire il font. Per esempio, la struttura TextFont contiene un contatore indicante il numero dei task che

stanno usando quel font, contatore che verrà aggiornato mentre il font è mantenuto in memoria e aggiornato ogni volta che un nuovo task comincerà a utilizzarlo. Non ci sarà permesso di rimuovere un font fino a quando tutti non avranno terminato di utilizzarlo, n potremo cambiare l'assegnamento della directory Fonts: ("assign fonts: ...") se qualche task sta ancora utilizzando un qualsiasi font nell'assegnamento corrente (fino a quando cioè, questo contatatore sarà stato decrementato a zero). In questo caso riceveremo un messaggio di errore "Can't cancel fonts" ("Non posso cancellare i font").
Esaminiamo adesso il contenuto in forma esadeciamle di quella parte del file "Garnet/16" necessaria per comprenderne il significato. Per dare un'occhiata a tutto il file potremo utilizzare il comando:

type fonts/garnet/16 opt h

Vediamo com'è organizzato:

```
0000: 000003F3 00000000 00000001 00000000
0010: 00000000 0000055E
```

Queste sono informazioni di caricamento, necessarie in quanto la definizione di un font è vista da Amiga semplicemente come un file caricabile. Il blocco viene chiamato "hunk_header" (intestazione dell'hunk) ed è identificato dal numero $000003F3. La seconda long-word indica che non è necessario aprire alcuna libreria residente quando questo file viene caricato e la terza informa che va caricato solo un hunk. Seguono due long-word che specificano la posizione del primo e dell'ultimo hunk del file nella tabella utilizzata dal caricatore di sistema (system loader). In questo caso esiste un solo hunk quindi entrambe queste long-word contengono il valore $00000000. L'ultimo numero nel blocco esprime la dimensione dell'hunk in long-word. Questo è l'unico elemento del blocco di intestazione che cambierà da un font file all'altro.

```
0010: 00000000 0000055E 000003E9 0000055E
```

Le ultime due long-word indicano che si tratta di un "hunk_code" ($3E9), al contrario di "hunk_data" o "hunk_bss" (area dati non inizializzata) e viene fornita ancora la lunghezza in long-word, $55E.

```
0020: 70004E75 00000000 00000000 0C000000
```

La prima word di questa linea, $7000, corrisponde all'istruzione in linguaggio assembly MOVEQ #0,D0 e la seconda word, $4E75, all'istruzione RTS. Queste istruzioni sono state inserite qui per sicurezza, nel caso qualcuno tenti di eseguire questo file anzich accedervi appropriatamente con una chiamata alla funzione OpenDiskFont() dell'Exec. Di seguito comincia la struttura node che verrà utilizzata per collegare il font alla lista di sistema dei font caricati da disco, che a sua volta è contenuta nella struttura della libreria DiskfontBase (la diskfont.library deve essere aperta da qualsiasi programma che intenda utilizzare i font da disco). Le prime due long-word nella struttura sono i puntatori rispettivamente al node successivo e a quello precedente nella lista; questi puntatori nel nostro caso sono nulli perché il font non è ancora stato caricato in memoria. Il byte successivo indica il tipo di node (0C = NT_FONT), quindi viene la priorità (00), e il puntatore al nome del node.
Questo puntatore ($0000001A) contiene in realtà l'offset dall'inizio dell'hunk che si trova alla locazione assoluta 0020. Quando il font viene caricato in memoria, il valore di questo puntatore viene sostituito dal sistema operativo con l'indirizzo effettivo al quale è stato caricato il font, cosa che non può essere nota anzitempo. Possiamo osservare che i dati puntati (0020 + 001A = 003A) sono nulli; consistono infatti in 32 zeri consecutivi. Questo spazio è riempito dalla routine OpenDiskFont quando il font viene caricato.
Dopo il puntatore al nome del node troviamo un numero che identifica i font file (0F80), un numero di revisione (0001) e una long-word vuota che servirà al DOS per immagazzinare l'indirizzo del segmento dati del font dopo il caricamento. Questo ci porta a 003A, dove iniziano i 32 byte riservati al nome del font.
A 005A si trova un'altra struttura node, identica a quella appena discussa. Questo node, che servirà a collegare il font alla lista di sistema di tutti i font attivi (non solo caricati da disco), fa parte di una struttura message dell'Exec, che a sua volta è il primo membro della struttura DiskFont per questo font. Quando il font viene chiuso dal suo ultimo utilizzatore e non è quindi più richiesto da alcun programma, il sistema operativo risponderà alla porta indicata in quel messaggio, per segnalare che il font può essere rimosso dalla memoria.
L'ultimo oggetto nella struttura message, la lunghezza del messaggio, si trova a 006C. Questa dimensione ammonta a $1508 byte per comprendere tutti i dati del font: la tabella con la forma dei caratteri e le tre tabelle associate. Se state pensando di costruirvi un font file, tenete presente che il programma esemplificativo contenuto nel ROM Kernel Manual stabilisce le dimensioni di questo messaggio in maniera incorretta con la linea:

```
DC.W  fontEnd-font      ;mn_Length
```

La versione corretta dovrebbe essere:

```
DC.W  fontEnd-fontData  ;mn_Length
```

```
0030: 001A0F80 00010000 00000000 00000000
0040: 00000000 00000000 00000000 00000000
0050: 00000000 00000000 00000000 00000000
0060: 00000C00 0000001A 00000000 15080010
```

La parte più interessante del file inizia al byte $006E. Inizia qui la vera definizione del font, con il rimanente della struttura TextFont, quella parte che viene dopo il messaggio dell'Exec, come documentato nell'include file graphics/text.h (o graphics/text.i per le definizioni in linguaggio assembly). La struttura inizia con la dimensione verticale (Ysize), l'altezza del font espressa in pixel:

```
0060: 00000C00 0000001A 00000000 15080010
```

Garnet/16 ha un'altezza di $10 = 16 pixel.

```
0070: 00620014 000B0001 000020FF 0000006E
```

I bit che definiscono lo stile contengono $00 (0 per il font

normale) mentre i bit di flag, $62, indicano che questo è un font "disegnato", che è proporzionale e che viene caricato da disco.

```
0070: 00620014 000B0001 000020FF 0000006E
```

La dimensione orizzontale (Xsize) del font è $14 (decimale 20). Questa è la larghezza nominale, ma siccome stiamo studiando un font proporzionale, ogni carattere avrà la sua larghezza definita in una tabella di spaziatura.

```
0070: 00620014 000B0001 000020FF 0000006E
```

La linea di base del font è posizionata a $000B (11) linee dalla sua cima. Ogni carattere è posizionato sullo schermo in relazione a questa linea, che dovrebbe essere la linea inferiore dei caratteri che non hanno discendenti.

```
0070: 00620014 000B0001 000020FF 0000006E
```

Il font verrà disegnato in grassetto, in modo che appaia più scuro.

```
0070: 00620014 000B0001 000020FF 0000006E
```

In questo spazio Amiga tiene conto del numero di task che stanno utlizzando questo font una volta che esso sia stato caricato in memoria. Il contatore viene incrementato e decrementato man mano che i vari task aprono e chiudono il font.

```
0070: 00620014 000B0001 000020FF 0000006E
```

Questo font inizia con il carattere di spazio, il cui valore ASCII è $20 (32) e termina al carattere che ha un valore di $FF, decimale 255. L'ultimo carattere appartiene alla serie (set) di caratteri alternati. Molti font non includono così tanti caratteri; la maggior parte inizia con lo spazio e finisce dopo la 'z' minuscola o, a volte, quattro caratteri oltre, dopo la '~' (tilde) che ha il valore ASCII $7E, decimale 126.

```
0070: 00620014 000B0001 000020FF 0000006E
```

Le informazioni relative alla mappa a bit (bitmap) di questo font iniziano all'offset $6E + $20 = $8E (le vedremo più avanti).

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

Questo è il Modulo, il numero di byte di cui il puntatore deve muoversi per andare da una riga di pixel a quella successiva all'interno dello stesso carattere. In questo caso ci sono $00E0 (240) byte nella mappa a bit per ciascuna riga dell'intero set di caratteri. Più avanti nell'articolo troveremo una spiegazione più dettagliata su questo punto.

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

Questo puntatore chiamato CharLoc individua una tabella nella quale sono contenute le posizioni dei dati per ogni singolo carattere, relative all'inizio della mappa a bit. Nel nostro caso la tabella CharLoc inizia a $E6E (3694) byte dall'inizio dell'hunk_code a 0020.

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

Questa long-word individua la posizione della tabella CharSpace contenente lo spazio che deve avere ogni carattere. Se il font non fosse proporzionale, ma la maggior parte lo sono, questa tabella non sarebbe richiesta! La tabella CharSpace per questo font si trova a un offset di $11F2 (4594) byte.

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

La tabella di Kerning per il font Garnet/16 é situata all'offset $13B4 (5044). Su Amiga la parola "kerning" non assume il suo significato standard, in quanto indica il numero di pixel da lasciare in bianco prima di iniziare a stampare il carattere. Nel suo uso ortodosso, kerning indica lo spostamento a sinistra di lettere come la V o la W quando sono stampate vicino a una lettera con un'inclinazione a loro parallela, come la A, rendendo il testo più piacevole a vedersi.

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

In questo punto inizia la bitmap che contiene le immagini dei caratteri.

Cosa significa "impaccamento di bit"?

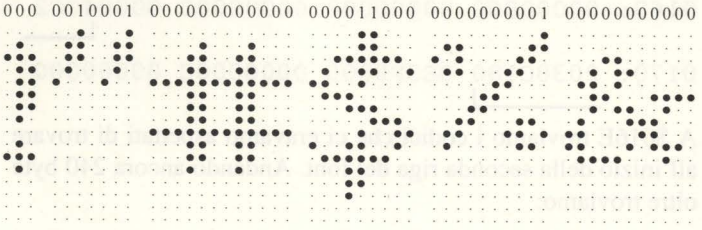
Le descrizioni dei caratteri per i font di Amiga sono contenute in una tabella che inizia con i dati per disegnare la prima riga superiore di un carattere, seguita dalla prima riga del secondo e così via fino all'ultimo carattere, senza spazi fra l'uno e l'altro. Immediatamente di seguito ci sono i dati che descrivono la seconda riga di ciascun carattere, poi sarà la volta della terza, della quarta e così via, fino ad esaurire il numero di righe che costituiscono il font (altezza del font in pixel). Nel caso di Garnet/16 il primo carattere è uno spazio largo 8 pixel. Dal momento che per gli spazi esiste la tabella separata, questo carattere viene omesso dalla bitmap e si comincia con il carattere successivo, il '!'. Ecco i bit che descrivono l'immagine della prima riga dei primi sei caratteri:

```
000001000100000000000000001100000000000010000000000...
```

(segue)

Dopo che è stata scritta tutta la prima riga dell'intero font, inizia la seconda riga. Per rendere questi caratteri più visibili li indichiamo qui sotto con '.' al posto di 0 ed con '●' al posto di 1. In realtà nel file non ci sono spazi fra un carattere e l'altro, ma noi li abbiamo introdotti per rendere i singoli caratteri più leggibili.

```
000 0010001 000000000000 000011000 0000000001 00000000000
```



Guardando attentamente potremo riconoscere che questi rappresentano i caratteri '!', doppio apice, '#','$','%' e '&'; quelli cioé con i valore ASCII $21,$22,$23,$24,$25 e $26.

Adesso cambieremo questi digit binari nei corrispondenti esadecimali, ricordando che per fare questo dobbiamo combinare quattro digit binari per ogni digit esadecimale. Dividendo le colonne a gruppi di quattro, ogni colonna di ciascun gruppo contenente un uno rappresenterà 8, 4, 2, o 1:

```
% 0000 0100 0100 0000 0000 0000 0011 0000 0000
0000 1000 0000 0000
```

significa:

```
$ 0  4  4  0  0  0  3  0  0  0  8  0  0
```

In questo modo le nostre informazioni impaccate a bit possono essere trasformate in formato esadecimale e l'inizio di ogni riga del font può essere scritta come segue:

```
Riga  1: $0440003000800
Riga  2: $4CC22030C1800
Riga  3: $ECC660FD430C0
Riga  4: $68866132C6160
Riga  5: $713FFF200C340
Riga  6: $600661C01818F
Riga  7: $6006607030164
Riga  8: $4006601C63238
Riga  9: $003FFC26C6E18
Riga 10: $400662648362E
Riga 11: $E00661F8003C4
Riga 12: $4004406000000
Riga 13: $0000006000000
Riga 14: $0000004000000
```

Le righe 15 e 16 sono tutte a zero. Utilizzando il modulo, menzionato precedentemente, sappiamo che la Riga 2 inizia a $E0, o 240 byte dopo l'inizio della Riga 1 e che la Riga 3 inizia 240 byte dopo l'inizio della Riga 2 e così via. Adesso lo controlliamo nella stampa esadecimale:

```
0080: 00E00000 0E6E0000 11F20000 13B40440
0090: 00300080 04000000 00000004 00000000
```

Questa è la rappresentazione esadecimale della prima riga dei primi sei caratteri che abbiamo disegnato. La parte rimanente della prima riga continua fino a che non raggiungiamo il byte $016D, 240 byte più avanti nel file.

```
0160: 00000000 00000000 00000000 00004CC2
0170: 2030C180 0E3F0000 0000000C 00000000
```

A $016E troviamo i codici che ci eravamo aspettati di trovare all'inizio della seconda riga del font. Andando ancora 240 byte oltre troviamo:

```
0240: 00000000 1C007060 0001C000 0000ECC6
0250: 60FDA30C 06418000 00000018 00000000
```

Abbiamo trovato gli stessi digit che si trovano nella nostra tabella precedente. Così abbiamo visto come sono immagazzinate nel file le varie righe di dati che servono a disegnare il font. Ogni 240 byte inizia una nuova linea, fino a quando non siano state immagazzinate tutte le sedici linee che descrivono interamente il font. Non è necessario stampare qui tutto il file, ma se desiderate, potete osservarlo interamente sul vostro schermo (TYPE fonts:garnet/16 opt h).

Continuando il nostro studio saltiamo adesso a $09E0, dove troviamo la tabella CharLoc. Esaminando il Garnet/16 con un font editor possiamo vedere che i primi sei caratteri del font occupano, rispettivamente, 3 bit, 7 bit, 12 bit, 9 bit, 10 bit e 11 bit. Nella tabella CharLoc troviamo le coppie di word che indicano, per ciascun carattere, l'inizio dei dati nella tabella e la larghezza in bit:

```
0E90: 00000000 00030003 0007000A 000C0016...
```

0000 0003 indicano che il primo carattere inizia all'offset (in bit) zero dall'inizio della tabella CharLoc ed è largo tre bit. 0003 0007 indicano che il secondo carattere inizia al terzo bit ed è largo 7 bit. 000A 000C indica che il terzo carattere inizia al bit 10 (3 + 7) ed è largo $C (12) bit. La tabella continua, descrivendo fino a un massimo di 255 caratteri nel font, nel nostro caso 255 - 32 = 223.

Ma la spaziatura è importante!

E infatti è vero. L'Amiga fornisce un'altra tabella in cui viene registrato lo spazio disponibile per ogni carattere. Nella sezione che precedeva la descrizione del font abbiamo trovato il puntatore "CharSpace". Localizziamo adesso nel font file la tabella nella quale sono elencati questi spazi. Esaminando il font con un editor appropriato possiamo vedere che la larghezza in pixel riservata al carattere spazio è otto, quattro per '!', otto per il doppio apice, tredici per '#', sette per '$', undici per '%', e dodici per '&'. Saltando alla posizione $1212 nel file troviamo questa tabella:

```
1210: 00000008 00040008 000D0007 000B000C
1220: 00050007 0007000A 00080005 00090005
```

Questa è la rappresentazione esadecimale del numero esatto di spazi previsti e la tabella continua mostrando la larghezza di tutti i caratteri definiti per questo font.

Ma il kern non è un tipo di uccello?

Il testo di tipografia che possiedo dice che il kern è la parte di una lettera che sporge sopra il corpo del tipo. Nell'Amiga, comunque, il kern rappresenta il numero di bit vuoti che vanno inseriti prima che il carattere venga disegnato. La routine di stampa del testo, prima che qualsiasi carattere venga visualizzato, controlla quanti bit di spazio deve inserire prima di ogni carattere e solo successivamente lo cerca nella mappa a bit e lo stampa.

Abbiamo visto che all'inizio di questo file c'é un puntatore che

(segue a pag. 82)

LA TUA COLLANA PER CAPIRE, IDEARE, PROGETTARE

LIBRI DI BASE ELETTRONICA

ELETTRONICA IN AUTO

L'elettronica mantiene uno stretto rapporto con l'automobile, sin dall'istante della progettazione. In questa guida la presentazione dei circuiti elettronici in dotazione sulle vetture in circolazione e su quelle che lo saranno nel prossimo futuro. Un argomento di grande interesse e di estrema utilità.

AMPLIFICATORI OPERAZIONALI

Gli OP-AMP per la loro caratteristica di poter "eseguire operazioni", sono oggi impiegati e applicati in diversi settori dell'elettronica, nell'audio e alta fedeltà, nelle apparecchiature di laboratorio, nell'elettronica di commutazione. Questa guida ti permetterà di familiarizzare con questo componente elettronico moderno.

Se hai l'esigenza di conoscere per costruire tutto sull'elettronica, il Gruppo Editoriale Jackson ti propone i nuovi: "Libri di Base Elettronica", 20 preziose guide attraverso circuiti, componenti, grafici, fotografie e soprattutto innumerevoli idee per scatenare la tua fantasia con progetti collaudati e di immediata realizzazione.

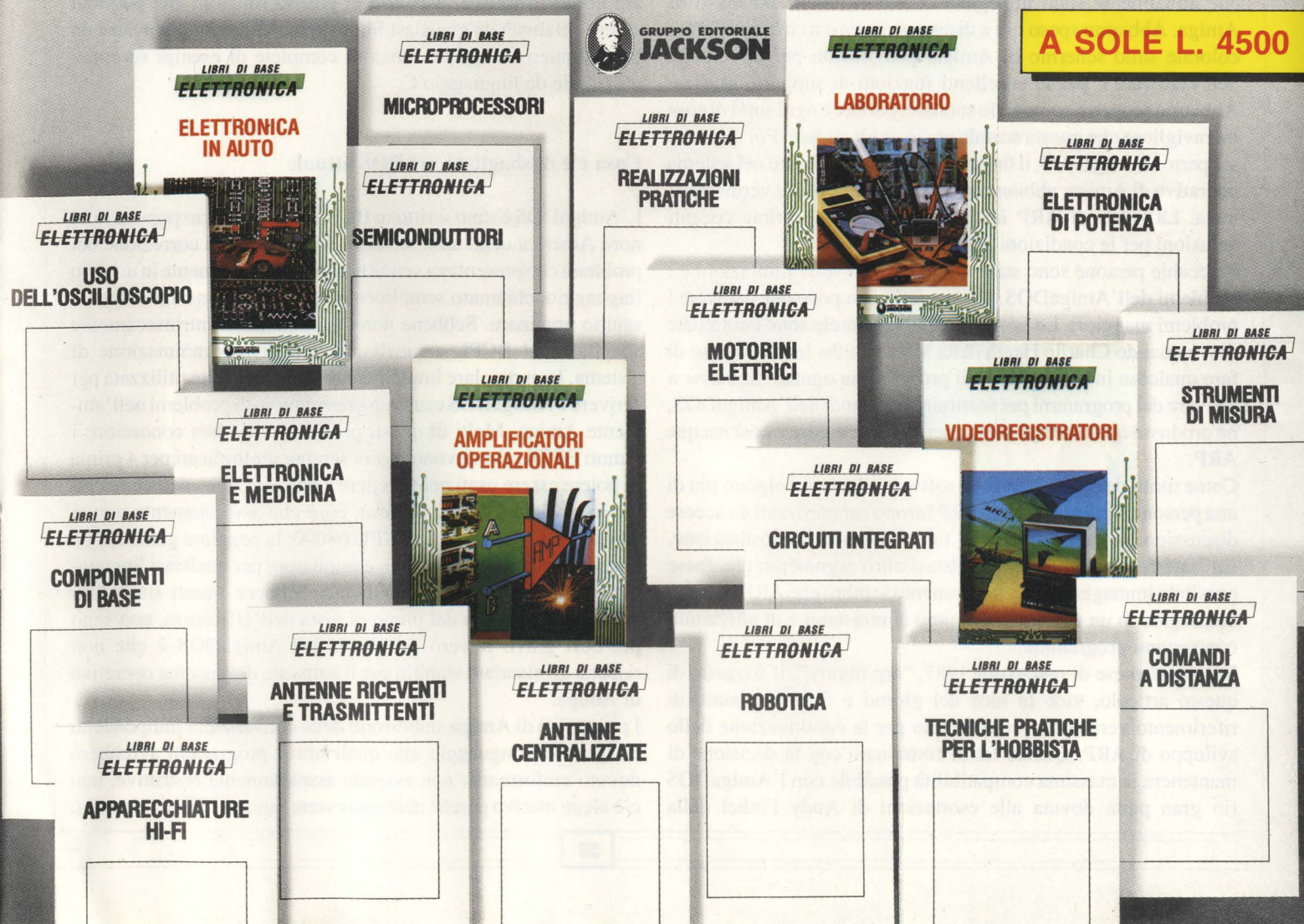
VIDEOREGISTRATORI

In questo libro viene fornita un'idea di base sui differenti tipi di magnetoscopi esistenti in commercio, con particolare riguardo ai sistemi domestici da mezzo pollice e ai sistemi industriali U-Matic. Un intero capitolo è dedicato al montaggio, con consigli sulla manutenzione e sulla conservazione delle attrezzature.

LABORATORIO

Sei un appassionato di laboratorio, oppure miri a diventare un professionista del settore? Ecco un testo in grado di chiarirti le idee, verificare i tuoi progetti o insegnarti a regolare le apparecchiature già costruite. Un tester ed un saldatore per iniziare e poi la tua fantasia...

QUESTO MESE IN EDICOLA



Visita guidata del programmatore alla "arp.library": Storia, funzioni e futuro

di Scott Ballantyne

...Come potete immaginare, sostituire l'AmigaDOS è un'impresa monumentale...

Scott Ballantyne è un programmatore il cui interesse attuale risiede nello sviluppo di sistemi operativi e di linguaggi di programmazione. Scott è ben conosciuto come autore del famoso programma Blazin' Forth per il Commodore 64. Vive a Manhattan e può essere raggiunto via CIS:70066,603 o BIX come sdb.

Una breve storia di ARP

Il progetto ARP (AmigaDOS Replacement Programs, programmi sostitutivi dell'AmigaDOS) ha ormai più di un anno di vita, ma la vera storia del progetto inizia in realtà nel momento in cui l'Amiga è stato immesso sul mercato. Come molti di voi, coloro che attualmente sono impegnati con ARP erano entusiasti di Amiga. Abbiamo speso ore a disegnare linee e a costruire figure colorate sullo schermo di Amiga, gongolando per la velocità dell'hardware e per le eccellenti funzioni di supporto grafico. Abbiamo scoperto come farlo suonare, parlare e ogni sorta di cose meravigliose che questa scatola era in grado di fare. Poi abbiamo scoperto l'AmigaDOS, il linguaggio di livello più alto nel sistema operativo di Amiga; abbiamo scoperto il proverbiale verme nella mela. La storia di ARP inizia in realtà con le prime cocenti delusioni per le condizioni del DOS di Amiga.

Parecchie persone sono state molto loquaci sulle limitazioni e i problemi dell'AmigaDOS, ma si è fatto ben poco per risolvere i problemi maggiori. Lo scontento e le lamentele sono continuate fino a quando Charlie Heath della Microsmiths Inc. ha deciso di fare qualcosa in proposito. Egli propose che ognuno iniziasse a scrivere dei programmi per sostituire i comandi dell'AmigaDOS, ne produsse qualcuno lui stesso per avviare le cose e così nacque ARP.

Come molti dei grandi progetti software che coinvolgono più di una persona, i primi giorni di ARP furono caratterizzati da accese discussioni su come andavano fatti i comandi di sostituzione, sull'interfaccia utente e su qualsiasi altro argomento che fosse possibile immaginare. Per un momento sembrò che ARP dovesse naufragare in un oceano di opinioni contrastanti e di altrettanto contrastanti programmi.

Intorno al mese di marzo del 1987, "arp.library", il soggetto di questo articolo, vide la luce del giorno e fornì un punto di riferimento veramente straordinario per la continuazione dello sviluppo di ARP. Questo fatto, combinato con la decisione di mantenere la massima compatibilità possibile con l'AmigaDOS (in gran parte dovuta alle esortazioni di Andy Finkel della Commodore), diede ad ARP un impeto e una direzione di marcia che sono state mantenute fino a questo momento.

Come potete immaginare, la sostituzione dell'AmigaDOS è un'impresa monumentale che richiederà un certo tempo per essere completata. La versione su cui si basa quest'articolo, pubblicata all'AmiEXPO nell'ottobre 1987, consiste di un gran numero di comandi sostitutivi largamente compatibili con i corrispondenti comandi dell'AmigaDOS e di una libreria condivisa costruita seguendo le regole di Amiga. Tutti questi comandi messi insieme vi faranno risparmiare 20k di spazio sul disco e miglioreranno notevolmente l'ambiente di lavoro dell'Amiga CLI, fornendo comandi più veloci, più piccoli e molto più potenti. In aggiunta a tutto questo, la libreria (chiamata abbastanza logicamente arp.library) fornisce una serie di potenti funzioni che possono essere chiamate da qualsiasi linguaggio. Vedremo più avanti un buon numero di queste funzioni complete di esempi su come utilizzarle da linguaggio C.

Cosa c'è di sbagliato nel DOS attuale

L'AmigaDOS è stato scritto in BCPL, un linguaggio popolare nel nord America degli anni sessanta. In seguito alla correzione dei problemi che presentava venne trasformato lentamente in un altro linguaggio, chiamato semplicemente C, che avrete sicuramente sentito nominare. Sebbene non ci sia niente di intrinsecamente sbagliato nel BCPL come linguaggio di programmazione di sistema, la particolare implementazione che è stata utilizzata per scrivere l'AmigaDOS causa un gran numero di problemi nell'ambiente Amiga. Molti di questi problemi sono ben conosciuti: i temuti BPTR, che devono essere sempre moltiplicati per 4 prima di potere essere usati per accedere a una locazione della macchina; lo stack che cresce all'insù, cosa che assolutamente non si adatta bene alla famiglia di CPU 68000; la peggiore generazione di codice mai effettuata da un compilatore per qualsiasi linguaggio che abbia visto personalmente. Sebbene questi siano tutti problemi molto seri dal punto di vista dell'efficienza, non sono poi così gravi. Il vero problema dell'AmigaDOS è che non rispetta gli standard stabiliti per il software del sistema operativo di Amiga.

I progettisti di Amiga stabilirono delle convenzioni indipendenti da qualsiasi linguaggio alle quali tutti i programmi avrebbero dovuto uniformarsi; non essendo assolutamente restrittive, non c'è alcun motivo perché debbano essere ignorate. Ad evidenza di

questo fatto Amiga è stato programmato con successo in linguaggi molto differenti fra di loro come il C, Modula 2, Basic, Lisp, Icon, Forth, Fortran e BCPL. Benché tutti questi linguaggi possano chiamare una qualsiasi funzione delle librerie di sistema contenute in Amiga, non è altrettanto possibile chiamare una funzione della libreria privata BCPL, perché questa non aderisce alle convenzioni stabilite. Come risultato esiste una larga porzione di codice nella ROM di Amiga che solamente i programmi in BCPL possono sfruttare.
Ancora peggio di questo è il fatto che le convenzioni per l'inizializzazione (startup) di un programma su Amiga sono complicate dalla necessità di mantenere due serie differenti di convenzioni sull'uso dei registri e dello stack: una per il BCPL e una per il resto del mondo. Ciò ha causato gravi problemi nella pratica, in modo particolare quando è necessario o conveniente che un programma carichi e ne lanci un altro. Quest'ultima è una tecnica di programmazione molto utile in un sistema multi-tasking, ma è ancora scarsamente utilizzata su Amiga, anche quando potrebbe esserlo con grande vantaggio, perché nessuno può essere sicuro se il programma che sta per essere caricato sia di tipo BCPL o "normale". Se si riceve quello sbagliato succede un disastro.
L'AmigaDOS ha anche qualche problema dal punto di vista del supporto a livello di sistema del multitasking. Alcune delle strutture dati globali dell'AmigaDOS non sono state ben progettate per un uso condiviso. In particolare, la lista DevInfo dell'AmigaDOS non è assolutamente protetta dall'essere calpestata contemporaneamente da più di un programma.
Dal punto di vista dell'utente, i programmi di comando in BCPL dell'AmigaDOS hanno una serie di problemi completamente diversi: sono inconsistenti nella sintassi di utilizzo, forniscono una potenza insufficiente per una macchina della qualità di Amiga e sono troppo grandi e troppo lenti.

Cosa fa ARP per risolvere questi problemi

ARP corregge le mancanze sopraelencate in vari modi differenti. I comandi sostitutivi sono scritti riga per riga in linguaggio assembly per risolvere con eleganza il problema delle dimensioni e della velocità. Abbiamo fatto anche ogni sforzo per rendere ogni comando il più consistente e potente possibile.
Sostituendo il contenuto della directory C, abbiamo fatto un gran passo avanti nell'eliminare la necessità di startup speciali per i registri e lo stack del BCPL. I programmi che girano nell'ambiente ARP, girano in un ambiente che segue gli standard di Amiga quindi sanno cosa aspettarsi. Questo fatto aumenterà di molto l'affidabilità dei programmi scritti per Amiga e dovrebbe anche ridurne i tempi di sviluppo e le dimensioni.
I comandi sostitutivi contenuti nella directory C di ARP si servono largamente della "arp.library", una libreria che fornisce le funzioni necessarie per scrivere facilmente ed efficientemente programmi in stile AmigaDOS, costruita seguendo gli standard Amiga. Dal momento che questa è una libreria standard, le sue funzioni possono essere chiamate in ogni momento da qualsiasi linguaggio. Attualmente vengono fornite le librerie di interfaccia e il codice di startup per i compilatori C della Manx e della Lattice, così come alcuni programmi di interfaccia per il Modula II. Quindi non c'è più bisogno di una libreria speciale in BCPL nella ROM.
Inoltre arp.library fornisce una quantità considerevole di funzioni "extra", alcune delle quali sono state progettate per risolvere i problemi dell'AmigaDOS, come quello riguardo la lista di device. La gestione delle risorse (pensiamo che questa sia una delle cose più importanti su Amiga), le variabili di ambiente (environment variables) e un File Requester completo di tutto, sono altre caratteristiche uniche e importanti dell'arp.library.

Usiamo arp.library da C

Sono possibili due metodi per incorporare arp.library nei nostri programmi C. Vengono fornite due librerie per il link (gli utenti del Manx hanno anche le versioni a 32 bit) e due header file: arpfunctions.h e arpbase.h. Di tutto questo è disponibile anche il codice sorgente per poter apportare modifiche in grado di soddisfare le nostre personali esigenze e preferenze.
Le due librerie C sono chiamate a.lib e arp.lib. A.lib contiene tutte le routine di collegamento che permettono al vostro compilatore di interfacciarsi con arp.library. Arp.lib contiene, oltre alle stesse routine, anche il codice di startup specifico di arp.library. Questo programma di inizializzazione apre la libreria e ne utilizza le routine di allocazione di memoria e gli analizzatori (parser) di argomenti, per processare le linee di comando nella maniera standard dell'AmigaDOS. Ne vengono anche gestiti i caratteristici punti di domanda interattivi "?", oltre alle possibilità fornite da ARP di ottenere aiuto supplementare per le linee di comando.

Usiamo il codice di inizializzazione: "Startup.c"

Il programma contenuto nel listato 1, "Startup.c", dimostra come possiamo utilizzare il codice di startup fornito da ARP per scrivere un programma che riceva dei comandi dall'ambiente CLI e che presenti il familiare formato d'uso (template) e le caratteristiche funzioni di aiuto all'utente dell'AmigaDOS. Compiliamo il programma eseguiamo il link con arp.lib (dovremo effettuare il link anche con le librerie che accompagnano il compilatore) e giochiamoci un pò per avere un'idea di come funziona.
Il codice di startup utilizza il parser contenuto in ARP e conosciuto come GADS() per analizzare gli argomenti della linea di comando. GADS() sta per Get Arguments Dos Style (leggi gli argomenti come fa il DOS) e questo è essenzialmente quello che fa. Possiamo lasciare a GADS() e al codice di startup il pesante compito di analizzare la linea di comando, dopo averla divisa nelle sue parti componenti, stabilendo un appropriato template dei comandi semplicemente mettendo la corrispondente stringa di caratteri nella variabile globale CLI_Template; inoltre possiamo aggiungere una linea di aiuto supplementare utilizzando la variabile CLI_Help.

GADS() può gestire correttamente anche gli eventuali escape, racchiusi tra apici, forniti dall'utente, oltre a processare alcuni errori contenuti nella linea di comando, mentre il codice di startup visualizza un messaggio che descrive l'errore e si preoccupa di interrompere l'esecuzione del programma, risparmiandoci gran parte del lavoro normalmente richiesto per gestire gli errori. Se non stabiliamo alcun valore per le variabili CLI_Templates o

CLI_Help verranno utilizzati i valori di default.
Tutto quello che dovremo fare consiste nel costruire un template appropriato. Sebbene soltanto ognuno possa singolarmente decidere qual'è la forma migliore per il template dei comandi dei suoi programmi, ecco qualche buona indicazione:

1) Non usare spazi bianchi (tab e spazi) nelle parole chiave (keyword).
2) Usare il simbolo "=" per fornire forme abbreviate per gli switch o le opzioni, per esempio:

"FROM/A,E=EVERYTHING/S,Q=QUIET/S"

3) Il template precedente permetterà all'utente di digitare una E per rappresentare lo switch EVERYTHING e una Q per QUIET.
4) Generalmente, se il programma richiede nomi di file come argomenti, proviamo ad utilizzarne quanti più possiamo (almeno 11 o addirittura 15). Possiamo specificare la presenza di nomi di file senza dover dare una descrizione per ognuno di essi, semplicemente usando le virgole da sole:
",,,"

Il template precedente permette di utilizzare fino a quattro volte lo stesso comando con destinazioni separate.
Per maggiori informazioni si può esaminare il programma di esempio e la pagina del manuale dedicata al parser di argomenti GADS() contenuti nella documentazione per i programmatori (Programmer's Docs) di ARP che si trova sul disco di Amiga Transactor relativo a questo numero.

L'utilizzazione di GADS() (attraverso il codice di startup fornito, oppure anche direttamente) ci permetterà di analizzare facilmente le linee di comando e di fornire un'interfaccia uniforme all'utente del nostro programma, abituato ormai ai template dei comandi di Amiga.

Un altro vantaggio derivante dall'uso del codice si startup fornito é il libero accesso a IntuitionBase e a GfxBase. L'arp.library apre queste librerie come parte del proprio startup e queste variabili sono i loro puntatori che rimangono sempre validi durante la vita del nostro programma. Il codice di startup contenuto in arp.lib copia questi puntatori nelle variabili riservate al nostro uso. Inoltre, otterremo un programma di dimensioni inferiori a 3000 byte. Lo stesso programma compilato con il codice di startup standard del C e utilizzante la funzione standard del C printf(), anziché quella dell'arp.library Printf(), risulterà di oltre 6000 byte (queste dimensioni sono state ricavate utilizzando il compilatore della Manx).

Usiamo ARP senza startup: "Nostartup.c"

Abbiamo la possibilità di effettuare il link con il codice di startup standard, detto anche codice di "runtime", fornito insieme al compilatore C, oppure di aprire arp.library da soli. Un tipico esempio viene dato in "Nostartup.c" (listato 2).

Degno di nota è il metodo utilizzato per ottenere da ArpBase i puntatori alla intuition.library e alla graphics.library.
Questo programma dovrebbe essere linkato prima con a.lib e poi con le altre eventuali librerie necessarie.

Termine di un programma che usa arp.library

L'arp.library fornisce la propria routine di uscita per i programmi, chiamata ArpExit(). Questa funzione rende nuovamente disponibili tutte le risorse il cui utilizzo è stato sottoposto a controllo da parte delle routine di resource tracking contenute nell'arp.library e chiude questa libreria in vece nostra.
Ad ogni modo, parlando in generale, non si dovrebbe usare ArpExit() direttamente da un programma C. La maniera migliore e più sicura di uscire da un programma C che sta utilizzando ARP, non importa con quale versione dei file .lib sia stato effettuato il link, è quella di utilizzare la funzione exit() del nostro compilatore. Se non stiamo utilizzando il codice di startup fornito con ARP, dovremo chiamare la funzione CloseLibrary(ArpBase). Se lo stiamo invece utilizzando, allora NON possiamo chiamare CloseLibrary(ArpBase).
Le seguenti considerazioni riguardano l'interazione con le librerie per il link contenute nel dischi del nostro compilatore C. Queste librerie potrebbero avere richiesto e ottenuto l'utilizzo di alcune risorse che la routine exit(), si suppone, dovrebbe liberare. Se eseguiamo immediatamente ArpExit() queste risorse non verranno più restituite. Se il programma è stato lanciato da Workbench, inoltre, è importante che il codice di uscita risponda con Reply() al messaggio di startup del Workbench stesso. Niente di tutto questo accadrà se chiamiamo ArpExit() (o se chiamiamo la funzione Exit() dell'AmigaDOS).
Nel caso invece si utilizzi del codice di startup personalizzato, senza eseguire il link con alcuna libreria C, allora potremo tranquillamente usare ArpExit().
Useremo il codice di startup fornito da ARP in tutti i programmi di questo articolo, dal momento che le sue dimensioni sono molto ridotte, che può utilizzare tutte le funzioni di ARP oltre a tutte le funzioni standard del C, che può utilizzare l'argument parser di ARP e dal momento che partirà ugualmente bene sia da Workbench sia da CLI. Assumeremo inoltre che questo sia il metodo scelto in generale per sviluppare programmi su Amiga.

Il succo di arp.library: le funzioni di utilità

ARP mette a disposizione diverse funzioni che svolgono compiti utili, ma non molto eccitanti, in maniera altamente efficiente. Se avete già programmato in C, molte di queste vi saranno già familiari, anche se bisogna fare attenzione, dal momento che il loro comportamento potrebbe essere sottilmente diverso da quello che vi aspettate. Sotto alcuni aspetti queste funzioni sono più significative per i programmatori che utilizzano l'assembly, perché rendono disponibile un mezzo di facile utilizzo per svolgere compiti come il confronto di stringhe e la formattazione del video, che normalmente richiedono subroutine scritte ad hoc. Dal momento che queste funzioni sono già abbastanza familiari, ne accenneremo solo brevemente. Il loro nome è generalmente simile a quello delle analoghe funzioni contenute nelle librerie del C. Per evitare possibili collisioni con c.lib, la prima lettera dei nomi contenuti nell'arp.library è maiuscola.
Arp.library contiene l'equivalente delle usatissime funzioni C Printf(), FPrintf() e Puts(). Qualora fosse necessario visualizzare un output formattato, si dovrebbe assolutamente considerare l'utilizzo delle funzioni contenute nell'arp.library, dal momento

che il tipico equivalente generato dal compilatore C aumenta le dimensioni del programma di parecchie migliaia di byte. L'unica cosa che le routine dell'arp.library non fanno è quella di visualizzare correttamente i numeri in virgola mobile. Nel caso sia necessario trattare questo tipo di valori, bisognerà rimanere legati alle versioni generate dal compilatore. Un altro vantaggio nell'usare le funzioni contenute nell'arp.library consiste nel fatto che sono "sicure". Chiamando la fuzione Printf() o Puts() da un programma lanciato da Workbench, saremo felici di scoprire che queste non si impiantano. La stessa cosa non vale per gli equivalenti generati dal compilatore C, come molti hanno scoperto a loro spese.
Bisogna fare attenzione quando si passano dei valori da formattare a Printf(), per il fatto che il C convertirà gli argomenti interi nel tipo integer di default. Questo potrebbe causare alcuni problemi se non facciamo attenzione. Per esempio, il seguente comando Printf() funzionerà correttamente solo utilizzando il compilatore C della Manx settato per gli interi da 16 bit:

```
Printf("Questo dovrebbe visualizzare 10 = %d",10);
```

Utilizzando il Lattice o lo switch +L del Manx dovremo usare la seguente forma per l'espressione precedente:

```
Printf("Questo dovrebbe visualizzare 10 = %ld",10);
```

Ciò accade perché il compilatore converte l'argomento da integer a long e le routine dell'arp.library non sono a conoscenza del cambiamento; assumono infatti che voi siate sicuri del tipo di argomenti che gli state passando.
Le altre funzioni di utilità stile C dell'arp.library comprendono routine di confronto delle stringhe (Strcmp e Strncmp), di conversione minuscolo/ maiuscolo (Toupper), di conversione da stringhe in interi (Atoi) e una routine non ricorsiva di quick sort (QSort) veloce come la luce. Da notare il fatto che le routine di confronto delle stringhe sono state progettate per funzionare con l'AmigaDOS e quindi non distinguono fra maiuscole e minuscole. Tutte queste funzioni sono utilizzate ogni giorno da programmatori C e queste implementazioni sono molto veloci e molto efficienti; inoltre, essendo contenute in una libreria condivisa, mantengono ridotte le dimensioni del nostro programma in maniera notevole.

L'arp.library non è priva di glamour: FileRequest()

Quello contenuto nell'arp.library è una versione ampiamente migliorata del file requester di Charlie Heath, che è stato utilizzato in molti dei primi e più importanti prodotti per l'Amiga, fra i quali TxEd della Microsmiths o prodotti della Aegis come il Draw e l'Images, é sempre stato considerato come uno dei migliori file requester per l'Amiga e la sua inclusione nell'arp.library lo rende disponibile al solo costo di una chiamata di funzione.
E' molto facile usare il file requester: bisogna inizializzare una struttura molto semplice chiamata, abbastanza ragionevolmente, struttura FileRequester e passarne il puntatore alla funzione FileRequest(). Se l'utente seleziona il gadget CANCEL, il valore restituito dalla funzione è nullo, altrimenti consiste in un puntatore al campo che contiene il nome del file nella struttura FileRequester. Questa è sua definizione, contenuta in arpbase.h:

```
struct FileRequester {
  BYTE *fr_Hail;   /* testo di saluto */
  BYTE *fr_File;   /* puntatore al nome del file */
  BYTE *fr_Dir;    /* puntatore al nome della directory */
  struct Window *fr_Window; /* puntatore alla finestra o NULL
*/
  SHORT fr_Flags;    /* non utilizzato in questa release */
  VOID (*fr_WildFunc)(); /* non utilizzato in questa release */

  VOID (*fr_MsgFunc)(); /* non utilizzato in questa release */

};
```

Le variabili commentate con "non utilizzato in questa release" sono riservate per future espansioni e dovrebbero essere messe a zero. La variabile fr_Hail punta al testo che verrà visualizzato nel corpo del requester. Questo può essere il nome del nostro programma, delle istruzioni, o qualsiasi altra cosa.
Le variabili fr_File e fr_Dir dovranno puntare a degli array di caratteri che serviranno alla funzione FileRequest() per immagazzinare la directory e il file scelti dell'utente. Se riceviamo un valore non nullo in ritorno da FileRequest(), dovremo guardare in queste variabili per trovare le scelte dell'utente. L'array fr_File dovrà essere abbastanza grande da contenere il nome più lungo possibile per un file, ovvero dovrà essere lungo FCHARS+1. L'array fr_Dir dovrà invece essere abbastanza grande da poter contenere il pathname più lungo che potrà capitare. E' saggio essere generosi nell'allocare lo spazio per questo array..
Gli array fr_File e fr_Dir possono essere inizializzati con dei valori di default. Quando il FileRequester verrà attivato, questi saranno la directory esaminata e il file visualizzato nel box di selezione del nome. Un esempio di come utilizzare FileRequest() é contenuto in "FileRequester.c" (listato 3).
Bisogna notare che FileRequest() esaminerà le directory utilizzando dei caratteri jolly (wildcard). Per esempio, se l'utente ha specificato il nome di un drawer come "DF0:C/E*", il file requester mostrerà solo i file contenuti in DF0:C che cominciano con la lettera "E". In questo caso, se l'utente seleziona un file, la stringa della directory che ci verrà restituita conterrà i caratteri wildcard.
Notiamo anche che è possibile ricevere il nome del file, vuoto. Questo può accadere se l'utente attiva il gadget OK senza prima selezionare il nome di un file. Il nostro programma dovrebbe essere progettato in modo da gestire in maniera ragionevole questa eventualità. Potete farvi una chiara idea di come utilizzare il file requester giocherellando con il demo "FileRequester.c" e osservando le cose che restituirà.

Lasciamo che arp.library si occupi delle nostre risorse

Uno dei grandi e irrisolti problemi della programmazione di Amiga consiste nel liberare le risorse allocate. L'arp.library contiene le funzioni che ci permetteranno di tenere sotto controllo le più comuni risorse allocate mediante la semplice chiamata di una funzione e ci permetterà inoltre di accedere direttamente al meccanismo di tracciamento in modo da poter controllare quello

che più preferiamo.
L'idea di base dietro il tracciamento delle risorse (resource tracking) è molto semplice. Quando si chiama per la prima volta una routine di tracking, viene creata una lista di risorse riservata al nostro programma (ResList, parlando in ARPese). La risorsa che viene allocata viene concatenata in questa lista. Ulteriori chiamate alle routine di tracking continuano a concatenare i nodi delle risorse nella ResList del nostro task. Ad un certo punto il nostro programma terminerà e chiamerà CloseLibrary(ArpBase). Durante questa fase (o se venisse chiamata ArpExit()) arp.library si accorgerà delle risorse allocate e non più utilizzate e le libererà per noi prima di terminare l'esecuzione. Quindi impiegando il codice di startup contenuto in arp.lib, tutto quello che dobbiamo fare per uscire dopo avere liberato tutte le risorse allocate si riduce a chiamare la funzione exit().

La parte supplementare alla fine di questo articolo contiene informazioni più dettagliate sul resource tracking.

Alla ricerca di pattern nel filesystem

Chi avesse già utilizzato i comandi di sostituzione di ARP, sarà già a conoscenza del fatto che una delle capacità più potenti e piacevoli del nuovo set di comandi consiste nella possibilità di usare i wildcard o i pattern in comandi come Type, Rename e Protect; la maggior parte dei comandi ARP, infatti, supportano i wildcard. Non ne viene supportato solo l'intero set dell'AmigaDOS, ma anche altri wildcard standard ai quali molti di noi sono stati abituati su altri computer. Non dovremo neanche fare una scelta in quanto potremo utilizzare quelli con i quali ci sentiamo più a nostro agio. Utilizzando le funzioni di ricerca per le directory contenute in ARP potremo facilmente aggiungere questo tipo di interpretazione del pattern ai nostri programmi.
La parte supplementare alla fine di questo articolo contiene informazioni più dettagliate sul pattern matching.

Utilizziamo le liste DA per saperne di più sui device

Si presentano molte occasioni nelle quali è necessario sapere i vari dettagli riguardanti la specifica configurazione hardware sulla quale il nostro programma sta girando. Per esempio, il nostro programma potrebbe avvantaggiarsi di floppy drive addizionali, oppure potrebbe richiedere che vengano fatti specifici assegnamenti logici, o addirittura potrebbe avere bisogno di controllare se uno specifico volume sia attualmente presente in un drive o meno.
Tutte queste informazioni sono attualmente contenute in una lista lunga e complicata chiamata DevInfo. Non è un lavoro semplice quello di decifrare gli oggetti contenuti nella lista DevInfo, ma il grosso problema nell'utilizzarla consiste nel fatto che non esiste una protezione multitasking per questa lista: un programma potrebbe alterarla nello stesso momento in cui noi potremmo tentare di capire cosa vi é contenuto.
L'arp.library fornisce la funzione AddDADevs() che crea una lista DA per qualsiasi categoria di device possa interessarci. Per esempio, potremmo chiedere a AddDADevs() quali device a disco sono presenti e essa costruirà una lista di tutte le periferiche che supportano un filing system. Oppure potremmo chiederle una lista di tutti i volumi conosciuti; verrà creata allora una lista di tutti i volumi e verrà indicato (nel campo de_Type) se sono attualmente inseriti in un drive o meno.
Un esempio di come utilizzare la AddDADevs() è contenuto nel programma "Devices.c" (listato 4). Questo visualizza semplicemente le periferiche trovate. L'utilizzo della AddDADevs() per scoprire in quale ambiente stiamo lavorando é di gran lunga preferibile al fatto di inserire in precedenza nel programma tutti i nomi dei device conosciuti, oppure all'assumere che ognuno abbia un solo disk drive.

Le funzioni di basso livello simili al DOS

L'arp.library contiene funzioni che rendono molto più semplice accedere a caratteristiche normalmente difficili da raggiungere dell'AmigaDOS e in alcuni casi contiene delle routine che sembrano mancare nella implementazione corrente dell'AmigaDOS. Esempi del primo tipo di funzioni sono quelle routine che aiutano ad inizializzare e a spedire i pacchetti del DOS (DOS packets) e le funzioni che generano un pathname completo partendo dal lock di un file. Esempi del secondo tipo di funzioni sono quelle routine chiamate per effettuare assegnamenti logici e quelle utilizzate per lanciare un altro programma dopo avere creato i suoi file handle di ingresso e uscita.

Spedizione di pacchetti

A un livello molto basso delle sue operazioni, l'AmigaDOS usa i pacchetti per trasferire l'informazione fra i vari processi. Per esempio, quando utilizziamo la funzione Write() dell'AmigaDOS per mandare dei dati a un file, la chiamata alla funzione Write() viene tradotta in un pacchetto che viene mandato al processo del file system che controlla il nostro file. A turno, il file system restituirà il pacchetto al mittente (il nostro processo) e il nostro programma potrà continuare l'esecuzione. Normalmente non c'è ragione di spingersi così in basso nella programmazione del DOS, ma ci sono alcune cose che possono essere fatte solo a questo livello. Mettere la console nel modo RAW ed eseguire operazioni di I/O su file in maniera asincrona sono due esempi. (Per maggiori informazioni sui pacchetti e sulle operazioni di I/O asincrone si legga l'articolo di Matt Dillon in questo stesso numero, "L'interfaccia a pacchetti per l'AmigaDOS, in C").
L'arp.library contiene due funzioni per gestire i pacchetti. La prima, di nome SendPacket(), spedisce un pacchetto a uno specifico handler e poi aspetta che venga mandato indietro. Ci restituisce quindi il campo Result1 del pacchetto e mette a disposizione il campo Result2, ottenibile chiamando IoErr() oppure leggendo il registro D1 se stiamo scrivendo in linguaggio assembly. Ecco un esempio di come utilizzare SendPacket() per settare la console in modo RAW:

```
ULONG args[7];  /* argomenti del pacchetto */
args[0] = -1;       /* setta il modo RAW */
res1=SendPacket(ACTION_SCREEN_MODE,
args,FindTask(0L)->pr_ConsoleTask);
```

La seconda funzione serve ad inizializzare pacchetti standard ed é molto utile per implementare comunicazioni a pacchetti asincroni. Altre funzioni che hanno a che fare con lo spedire e il ricevere pacchetti sono CreatePort() e DeletePort(), che sono essenzialmente le solite funzioni della libreria C dello stesso nome.

Assegnamenti logici e variabili di ambiente

Gli assegnamenti logici sono un elemento utile nell'ambiente di Amiga ed è quindi un fatto particolarmente infelice e strano che l'AmigaDOS non fornisca una funzione per fare questo da programma. ARP viene in aiuto con la funzione chiamata Assign(). Il suo utilizzo é estremamente semplice. Per esempio, per assegnare al disco nel drive uno il nome DODAH scriveremo:

```
Return = Assign("DODAH","DF1:");
```

Dovremo poi controllare il valore restituito (i valori di ritorno per la funzione Assign() sono definiti in arpbase.h). Per cancellare un assegnamento scriveremo invece:

```
Return = Assign("DODAH",0L);
```

Si noti che anche il tentativo di cancellare un assegnamento può fallire; dovremo quindi controllare il valore restituito anche da questo tipo di chiamate. Un errore comune che si ripete frequentemente nell'utilizzo della funzione Assign() consiste nel tentare di cancellare un assegnamento in questo modo:

```
Return = Assign("DODAH","");
```

Sfortunatamente, questo finisce per fare riferire DODAH alla directory corrente.
La funzione Assign() è abbastanza sicura e facile da usare. Possiamo fare ripetutamente lo stesso assegnamento senza effetti collaterali e si può anche cambiare l'assegnamento di un nome logico senza averlo cancellato in precedenza.
Sebbene gli assegnamenti logici siano convenienti per trattare con le directory e con i file, essi non possono risolvere tutti i problemi. Un meccanismo comunemente presente nei sistemi operativi moderni è costituito dalle variabili di ambiente (environment variable). Sfortunatamente le variabili di ambiente non erano parte del progetto originale dell'AmigaDOS. Forse si é pensato che gli assegnamenti logici potessero fare tutto, ma nella pratica questi non si sono rivelati abbastanza flessibili.
Consideriamo il caso di un assembler o di un compilatore che deve guardare in numerose directory per trovare i suoi header file. Attualmente è possibile scrivere una cosa di questo tipo:

```
Assign INCLUDE: SYS:INCLUDE
```

e progettare l'assembler in modo che utilizzi sempre il nome logico INCLUDE per cercare i suoi header file. Ma cosa succederebbe se volessimo cercarli in più di una directory? Il meccanismo dell'assegnamento a questo punto crolla. Questo compito, invece, potrebbe essere svolto molto semplicemente con le variabili di ambiente. Infatti il compilatore e l'assemblatore della Manx utilizzano proprio queste ultime. Nota per gli utenti Manx: dal momento che la Manx è stata la prima a impiegare le variabili di ambiente su Amiga, l'implementazione delle stesse nel progetto ARP è stata fatta mantenendo la compatibilità con le prime. Possiamo quindi utilizzare le funzioni e il comando Set di ARP in concomitanza con il software Manx.
Come altro esempio dell'utilità delle variabili di ambiente, consideriamo il problema di mandare delle informazioni non appartenenti al file system a uno o più programmi. Per esempio, un programma di copia potrebbe esaminare la variabile di ambiente chiamata OVERWRITE. Se questa contenesse il valore VERO, il programma potrebbe scrivere su un file già esistente; se FALSO il programma si rifiuterebbe. Possiamo usare le variabili di ambiente anche per trasmettere informazioni numeriche ai programmi, per stabilire valori di default di ogni tipo, ecc. Utilizzando le variabili di ambiente possono essere trasferiti molti tipi di informazioni per le quali gli assegnamenti logici, che rimangono comunque utili nel loro campo, non erano stati progettati.
L'utente può modificare il valore di una variabile di ambiente utilizzando il comando Set, la cui sintassi é:

```
Set VAR=VAL VAR2=VAL2
```

Questo darà a VAR il valore VAL e a VAR2 il valore VAL2. Le variabili di ambiente possono essere rimosse omettendone il valore:

```
Set VAR= VAR2=
```

Possiamo accedere a queste variabili anche utilizzando le funzioni Getenv() e Setenv(). Tutti i valori delle variabili di ambiente sono memorizzati come stringhe ASCII e a noi é lasciata l'incombenza di effettuare le analisi e le conversioni (per esempio per i dati numerici) che dovessero essere necessarie. Come esempio del loro utilizzo, vediamo Setenv():

```
Setenv("AMIGA","VA MEGLIO CON ARP");
```

Questa chiamata setta una variabile di ambiente chiamata AMIGA al valore "VA MEGLIO CON ARP". Utilizziamo Getenv() per leggere il valore di questa variabile:

```
Getenv("AMIGA",buffer,255L);
```

Viene riempito "buffer" (un array di caratteri) con la stringa assegnata ad AMIGA. Getenv() restituisce anche un puntatore al valore della variabile. Questo puntatore individua la copia privata di questa variabile appartenente ad arp.library, quindi non bisogna mai accedere a quest'area senza prima chiamare Forbid(). Normalmente questo non sarà necessario e noi potremo utilizzare l'oggetto restituito da Getenv() per sapere se è stato definito un qualche valore per la variabile in questione (nessun valore é stato definito se viene restituito NULL).

Lanciare altri programmi dal nostro programma.

L'arp.library mette a disposizione la funzione SyncRun(), che carica e lancia un programma e ne restituisce il codice di uscita

oppure un errore indicante che il programma in questione non è stato trovato o che non è stato possibile caricarlo. Per convenzione, quando i programmi terminano viene restituito uno zero se non ci sono stati problemi, oppure un valore maggiore di zero se l'esecuzione è stata interrotta in maniera anormale. SyncRun() ci restituirà questo valore. Se riceviamo un codice negativo di errore da SyncRun() significa che c'è stato un problema nel caricare il programma. Per esempio, il file potrebbe non essere stato trovato, o potrebbe non esserci stata sufficiente memoria libera.
Si possono specificare gli handle di input/output da utilizzare come default per il nuovo programma. Fornendo il valore NULL per questi handle, il nuovo programma erediterà quelli del programma chiamante. Ecco un esempio del suo utilizzo:

```
rc = SyncRun("Echo","Testing 123",0L,0L);
if (rc>0)
  Printf("Echo terminato in modo anormale, rc = %ld",rc);
else if (rc<0)
  Printf("Non riesco a trovare o a caricare Echo");
```

La funzione SyncRun() è certamente il metodo più sicuro per questo tipo di lavori. Per prima cosa salva i dati privati del nostro task, in modo che sia possibile utilizzare tranquillamente i puntatori privati nel blocco di controllo del nostro task, senza preoccuparsi del fatto che potrebbero essere alterati dal programma che stiamo per lanciare. SyncRun(), inoltre, lavora correttamente con le funzioni di resource tracking per proteggere il ResList del nostro processo da possibili interferenze provocate dal nuovo task.
I programmi BCPL hanno dei bug che gli impediscono di ricevere gli argomenti che vengono loro passati da SyncRun(). Questi programmi non riusciranno ad abortire SyncRun(), ma si comporteranno come se non gli fosse stata passata alcuna linea di comando. Quando i programmi BCPL si uniranno a tutti gli altri nel seguire le convenzioni per l'inizializzazione di un programma, saranno anch'essi in grado di ricevere gli argomenti della linea di comando da SyncRun().

I bug di ARP:

Nessuna visita guidata del programmatore sarebbe completa senza comprendere una o due segnalazioni di errore. Sono emersi i seguenti problemi nella prima release di ARP:

FreeAccess(), GetAccess(), Getenv(), TRACK_GENERIC

Queste funzioni non funzionano nella release 31.0; funzionano regolarmente invece nella release 31.1 che troverete nel disco di Amiga Transactor associato a questo numero della rivista.

CloseWindowSafely() e TRACK_WINDOW

Se apriamo una finestra con OpenWindow(), mettendo NULL nella variabile IDCMPFlags, la funzione CloseWindowSafely() andrà in GURU. Questa funzione si può chiamare solo se avremo abilitato IDCMP (caso molto comune).
Ci sono una serie di piccoli bug aggiuntivi che sono stati già eliminati nella versione 31.1. I programmatori che utilizzano ARP dovrebbero usare quest'ultima versione per pubblicare il loro software.
Questi sono gli unici bug conosciuti nella prima versione di arp.library.

Fine della visita: il futuro di ARP.

Questo conclude il nostro tour dell'arp.library. Non abbiamo potuto menzionare tutte le funzioni di ARP, ma abbiamo cercato di parlare di quelle che si suppone possano essere di immediata utilità per i programmatori, in termini di ridurre le dimensioni dei programmi e permettere di fare cose che prima erano assai difficili o assolutamente impraticabili.
Continueremo ancora per un certo tempo a lavorare su ARP. L'ultima versione (v1.1) è stata messa in distribuzione il 5 marzo 1988 e contiene quasi tutti i comandi di sostituzione per la directory C (con pochissime eccezioni), alcuni sostanziali miglioramenti e le correzioni degli errori elencati in precedenza. ARP v1.1 è disponibile sul disco di Transactor per questo numero. Stiamo lavorando ad una nuova Shell, che dovrebbe sostituire il CLI in una prossima versione, oltre all'ampliamento del linguaggio di comandi del DOS (conosciuto come script o batch language). Charlie Heath della Microsmiths rimane il coordinatore centrale del progetto ARP. Egli sta inoltre raccogliendo tutti i nominativi degli utenti e dei rivenditori che utilizzano ARP e renderà disponibile questo elenco ai programmatori che utilizzano ARP nei loro prodotti. Dal momento che gli utenti di ARP sono, tendenzialmente, coloro più interessati al software di qualità, questa mailing list dovrebbe fornire un aiuto prezioso a tutti. Il futuro ultimo di ARP rimane naturalmente nelle mani degli utenti, dei programmatori e della Commodore. L'accoglienza iniziale riservata ad ARP dagli utenti è stata molto positiva. Speriamo che la Commodore inizi a fornire ARP sul disco Workbench ed eventualmente in ROM. Sebbene sia troppo presto per dire qualcosa di definitivo su questo fatto, abbiamo lavorato assai duramente per soddisfare le richieste di Andy Finkel della Commodore, il quale è stato di grande aiuto durante lo sviluppo di ARP. Con la rimozione della dipendenza di Amiga dalla interfaccia BCPL, ARP ha rimosso l'ostacolo più grande ai futuri sviluppi del sistema operativo. Fornendo funzioni chiamabili dall'utente come GADS() e il file requester FileRequest(), abbiamo anche contribuito in maniera significativa alla standardizzazione su Amiga di entrambe le interfacce d'utente CLI e Workbench. ARP, per inciso, è stato offerto gratuitamente alla Commodore e questo dovrebbe rimuovere un ostacolo che comunemente si frappone all'aggiunta di nuovo software: il costo. Speriamo che le "grandi potenze" riconoscano una cosa valida quando la vedono e che decidano di sfruttarla.
Penso di poter parlare anche a nome degli altri sviluppatori di ARP e dei suoi nuovi utenti, dicendo che, qualsiasi cosa accada, é valsa la pena di fare tutto questo duro lavoro. E' veramente bello, grazie alle dimensioni ridotte dei programmi ARP, poter mettere altro materiale sul mio disco di lavoro e sul disco Workbench ed è una gioia poter utilizzare le wildcard con programmi come Type e Protect. Come programmatore di Amiga, arp.library mi mette a disposizione una maggiore funzionalità e programmi più ridotti. Per quanto mi riguarda, Amiga va meglio con ARP.

Come tenere sotto controllo le risorse con ARP (resource tracking)

Le funzioni di ARP che ottengono e mantengono sotto controllo determinate risorse sono:

ArpOpen()	Ottiene e controlla i file
ArpLock()	Ottiene e controlla i lock
ArpDupLock()	Duplica un lock e lo controlla
ArpAlloc()	Ottiene memoria PUBLIC, la inizializza a 0 e la controlla
ArpAllocMem()	Ottiene memoria come la funzione dell'Exec e la controlla

Le funzioni di tracking non utilizzano direttamente i puntatori alle risorse, anche se questo è il valore restituito dalle funzioni appena elencate. Viene utilizzato invece un oggetto chiamato Tracker. Un Tracker è la porzione utilizzabile di un nodo di tracking allocato per la nostra risorsa. E' possibile costruire il proprio Tracker nel caso ce ne fosse bisogno, ma per la maggior parte delle applicazioni questo non dovrebbe essere necessario e non verrà trattato in questo articolo.
Sarà necessario utilizzare il tracker se si vorranno liberare singolarmente le risorse precedentemente ottenute.E' sufficiente esaminare la variabile globale LastTracker, immediatamente dopo la chiamata a ciascuna delle funzioni sopraelencate, per ottenerne il relativo tracker. Questo valore potrà essere copiato in un posto sicuro, per potere essere successivamente riutilizzato, a discrezione dell'utente, per liberare le risorse chiamando direttamente la funzione di basso livello FreeTrackItem(). Vediamo un esempio:

```
struct DefaultTracker *ftrack;

if (File = ArpOpen("FileName",MODE_OLDFILE))
  ftrack = LastTracker;
else
{
  Puts("Il file non si é aperto!");
  exit(20);
}

/* qui il resto del programma e poi liberiamo l'oggetto */
FreeTrackItem(ftrack); /* chiude il file */
```

Nel caso più semplice usciremo normalmente dal programma e lasceremo che arp.library chiuda il file per noi.
Il meccanismo di tracking di arp.library è in grado di controllare e rilasciare molte altre cose, oltre a quelle indicate nelle funzioni sopraelencate. Quelle funzioni sono state create apposta perché le risorse da loro gestite sono utilizzate così comunemente da risultare conveniente una sola chiamata di funzione per ottenere contemporaneamente sia la risorsa, sia il relativo tracker.
Per controllare risorse diverse da quelle viste prima bisogna ottenere un tracker, poi la risorsa, quindi installare la risorsa nel tracker. Il prossimo esempio mostra come utilizzare le funzioni di tracking per controllare una finestra di Intuition:

```
struct Window *Window;
struct DefaultTracker *mytracker;

if (mytracker = GetTracker(TRAK_WINDOW))
{
  if (Window = OpenWindow(&nw))
  {
mytracker->dt_Object.dt_Resource = (CPTR)Window;
mytracker->dt_Extra.dt_Window2 = NULL;
  }
  else
  {
  CLEAR_ID(mytracker);
  Puts("Non riesco ad aprire la finestra!");
  exit(20);
  }
}
else
{
  Puts("Non riesco ad ottenere il tracker!");
  exit(20);
}
```

Il codice precedente mostra quale sia l'approccio generale da utilizzare per tutte le chiamate della funzione GetTracker().
Per ottenere un tracker basta chiamare GetTracker(ID), mettendo al posto di ID una delle TRAK_ID che si trovano in arpbase.h.
Se l'allocazione del tracker ha successo si potrà tentare di ottenere la risorsa desiderata. Si procederà quindi ad inizializzare DefaultTracker nella maniera appropriata per questa risorsa. Normalmente questo vuole dire copiare il puntatore alla risorsa nella variabile dt_Object. Per alcune risorse potrebbe essere richiesta un'azione supplementare sulla variabile dt_Extra. Le finestre, ad esempio, sono delle risorse che richiedono un valore nel campo dt_Extra. Naturalmente ogni risorsa che richiede questo campo supplementare lo utilizzerà in maniera diversa dalle altre. Nel caso delle finestre, il campo dt_Extra viene utilizzato per segnalare se più di una window stia utilizzando lo stesso IDCMP. Se ciò risulta vero, l'IDCMP della finestra non verrà chiuso.
Se la risorsa non viene allocata, utilizzeremo la macro CLEAR_ID() contenuta in arpbase.h per cancellare il campo ID del tracker e, successivamente, usciremo dal programma. Il tracker sarà rilasciato automaticamente (questa è la ragione per cui di norma è più conveniente ottenere il tracker prima di ottenere la risorsa).
Utilizzando l'id TRAK_GENERIC, è possibile anche specificare una funzione che verrà chiamata quando arriverà il momento di liberare le risorse. Questo ci permette di controllare altre risorse di sistema, strutture software da noi progettate e tutto ciò che desideriamo.

Le funzioni che controllano le risorse hanno anche la capacità di liberarle selettivamente se il sistema viene sovraccaricato. Utilizzando la funzione FreeAccess(), segnaleremo le risorse delle quali possiamo eventualmente fare a meno nel caso che la memoria disponibile scarseggi. Per vedere se una tale risorsa sia ancora disponibile e in caso affermativo per bloccarla in memoria, si utilizzerà la funzione GetAccess(). Vediamo un tipico esempio:

```
struct DefaultTracker *dt;
BYTE *Buffer;

if (Buffer = (BYTE *)ArpAlloc(GROSSO_VALORE))
  dt = LastTracker;
else
  exit(20);

/* Adesso lavoriamo con il buffer e quando abbiamo finito
diciamo al sistema che ci piacerebbe tenerlo, ma solamente se non
sottrae spazio ad altre risorse */

FreeAccess(dt);

/* Quando, più avanti, vogliamo lavorare ancora con il buffer,
dobbiamo prima vedere se è ancora al suo posto e, se questa con-
dizione è verificata, bloccarlo in memoria mentre lo usiamo,
perché non vogliamo che se ne vada mentre siamo nel bel mezzo
del suo utilizzo! */

Buffer = (BYTE *)GetAccess(dt);
if (Buffer != NULL)
{
  /* qui facciamo altre cose e poi */
  FreeAccess(dt);
}
else
{
 /* il buffer non è più disponibile. Possiamo tentare di riallocar-
lo, possiamo liberare il nodo tracker oppure possiamo andarcene
*/
}
```

Le chiamate a FreeAccess() e GetAccess() possono essere annidate. Questo ci permetterà di condividere le risorse, fra task, processi o moduli. Ognuno può usare FreeAccess() o GetAccess() su una risorsa che non verrà rilasciata dal sistema operativo fino a quando tutti i task non avranno finito di utilizzarla.
Per ottenere una nuova lista di risorse per il nostro task è anche possibile chiamare le funzioni di tracking di livello più basso, CreateTaskResList() e FreeTaskResList(). In genere questo avviene automaticamente quando chiamiamo una delle routine di tracking, ma può essere anche determinato manualmente. Una possibilità interessante che ne deriva consiste nel poter annidare le liste di risorse. Possiamo scrivere una funzione che all'inizio chiami la CreateTaskResList() e alla fine la FreetaskResList() per liberare solamente le risorse allocate da quella funzione (e da tutte le funzioni che ha chiamato).

Le wildcard e il pattern matching di ARP

Le funzioni che esaminano le directory utilizzano una struttura dati chiamata Anchor (àncora). Non èè necessario impiegare direttamente queste strutture, ma è necessario sapere che vengono utilizzate al nostro posto da queste funzioni e che è necessario rilasciarle esplicitamente alla fine della ricerca. Si possono liberare tutte in una volta chiamando la funzione FreeAnchorChain(), oppure si possono utilizzare le funzioni di tracking, come descritto in precedenza, per farlo automaticamente all'uscita del programma.
Per ottenere l'Anchor iniziale dovremo allocare e inizializzare una struttura di controllo chiamata AnchorPath. Questa struttura contiene fra i suoi elementi la struttura BCPL FileInfoBlock, così dovremo assicurarci che l'intera AnchorPath sia allineata su longword. La maniera più facile di realizzare questa condizione consiste nell'usare la funzione ArpAlloc(). La struttura AnchorPath, definita in arpbase.h, consiste in:

```
struct AnchorPath {
  struct Anchor *ap_Base;
  struct Anchor *ap_Last;
  LONG  ap_BreakBits;
  LONG  ap_FoundBreak;
  ULONG ap_Length;     /* dimensione di ap_Buf */
  struct FileInfoBlock ap_Info;
  BYTE  ap_Buf;
};
```

Possiamo leggere il nome del file nella struttura ap_Info (vedere il file dos.h) oppure possiamo utilizzare le funzioni di pattern matching per ricostruire l'intero path in ap_Buf. Per compiere quest'ultima operazione, dobbiamo allungare la struttura AnchorPath in modo che possa contenere un buffer di maggiori dimensioni e poi dobbiamo mettere le dimensioni di questo buffer in ap_Length. Se ci serve solo il nome del file, che possiamo ottenere dalla struttura ap_Info, non dovremo estendere la struttura AnchorPath e metteremo zero in ap_Length. Una maniera semplice e conveniente per allargare ap_Buf consiste nel dichiarare una nostra struttura AnchorPath come segue:

```
struct UserAnchor {
  struct AnchorPath ua_AP;
  BYTE  moremem[255];  /* buffer esteso */
};
```

Dopo avere allocato una struttura UserAnchor, utilizzando ArpAlloc(), avremo a disposizione un buffer ap_Buf di 256 byte. Se le funzioni di pattern matching trovano un nome composto da un numero di caratteri maggiore di quello contenuto in ap_Length restituiscono un ERROR_BUFFER_OVERFLOW.

Le funzioni di pattern matching di ARP sono molto veloci, ma le ricerche in una directory molto grande possono richiedere un certo tempo. Possiamo permettere ad un utente di interrompere un processo di ricerca inizializzando i bit della variabile ap_BreakBits con i valori dei messaggi dai quali vogliamo lasciarci interrompere. Questi bit di segnalazione sono definiti nel file dos.h e corrispondono alla pressione dei tasti CTRL C D E o F, o all'utilizzare il programma break per ottenere lo stesso effetto dei tasti su un task che gira in background. Quando le routine di pattern matching ricevono il segnale dell'utente, ci restituiscono immediatamente ERROR_BREAK con il valore del segnale di interruzione nella variabile ap_FoundBreak. Questo permette di rispondere in maniera differente alle diverse sequenze di controllo, così come permette di stabilire a quali rispondere e quali ignorare. Normalmente si dovrà rispondere almeno a CTRL C:

```
ap_BreakBits |= SIGBREAKF_CTRL_C;
```

Una volta ottenuta e inizializzata la struttura AnchorPath, procederemo come segue:

1) Chiameremo FindFirst("Pat",AnchorPath) per trovare il primo file o la prima directory che combacia con "Pat". Se la FindFirst() restituisce zero, potrebbero esserci altri file o directory con quel nome all'interno della directory esaminata.
2) Per trovare anche i file o le directory successive chiamate "Pat", utilizzeremo FindNext(AnchorPath) fino a quando restituirà un valore diverso da zero.
3) Chiameremo poi FreeAnchorChain(AnchorPath) e poi ci occuperemo del codice di errore che ci viene restituito (e che potrebbe non rappresentare affatto un errore).
Entrambe le funzioni FindFirst() e FindNext() restituiscono un valore diverso da zero quando incontrano un errore. Se questo accade, dovremo prima liberare la AnchorChain e poi gestire l'errore. Questo valore potrebbe anche significare che non ci sono più file o directory con quel nome (ERROR_NO_MORE_ENTRIES), oppure potrebbe indicare una richiesta di interruzione da parte dell'utente (ERROR_BREAK), oppure ancora potrebbe segnalare un errore di overflow delle I/O o del buffer. Il programma "WildDemo1.c" nel listato 5 mostra come gestire queste situazioni.

Liste ordinate

Come avrete sicuramente notato se avete lanciato il programma WildDemo1, le funzioni di ricerca delle directory restituiscono i file nell'ordine nel quale vengono trovati, che potrebbe non essere affatto l'ordine nel quale le vorremmo vedere. Per creare liste ordinate, arp.library mette a disposizione la funzione AddDANode(), la quale crea liste di nodi DirectoryEntry percorribili in una sola direzione. Queste strutture sono costruite come segue:

```
struct DirectoryEntry {
  struct DirectoryEntry *de_Next;
  BYTE  de_Type;
  BYTE  de_Flags;
  BYTE  de_Name[1];
};
```

Per ora non utilizzeremo la variabile de_Flags in quanto è riservata per future espansioni. Da notare che le liste di nodi DirectoryEntry sono percorribili in una sola direzione, come le liste dell'AmigaDOS, in quanto non sono concatenate con una doppia serie di puntatori, come invece sono organizzate le liste dell'Exec. Le liste a doppia catena sono migliori quando si ha bisogno di accedere ai nodi della lista in modo random, non sequenziale. Nel caso di liste ordinate, dove si parte in testa e si finisce in coda, le liste con una doppia catena di puntatori occuperebbero dello spazio supplementare che non verrebbe comunque utilizzato.
La funzione AddDANode() aggiunge un nodo alla lista rispettando l'ordine alfabetico basato sull'array de_Name (verrà automaticamente allocata la memoria supplementare necessaria per questo array), inoltre riordina la lista utilizzando il campo de_Type come seconda chiave di ordinamento.
Entrambi questi metodi vengono utilizzati nella seconda versione del programma WildDemo (listato 6). In questo caso, anziché visualizzare i nomi dei file man mano che vengono trovati, li aggiungiamo a una lista con la funzione AddDANode(). Dal momento che desideriamo avere due liste separate, una contenente i nomi delle directory e una contenente i nomi dei file, specificheremo due differenti seconde chiavi di ordinamento. Siccome la chiave che rappresenta le DIRECTORY consiste in un valore numerico inferiore rispetto a quella che definisce i FILE, le directory appariranno per prime nella lista ordinata alfabeticamente poi seguirà la parte ordinata dei file.
Dopo avere costruito la lista, visualizzeremo quello che troviamo spostandoci sequenzialmente di un nodo alla volta. Utilizzeremo ancora la seconda chiave di sort, questa volta per determinare se stiamo visualizzando dei file o delle directory.
Alla fine libereremo tutta la memoria e tutti i nodi della lista chiamando la funzione FreeDAList().

Come liberare determinati nodi nella lista DA

E' possibile liberare anche solo una parte della lista DA, dal momento che FreeDAList() richiede come argomento il puntatore ad un nodo (invece del puntatore all'inizio della lista, detto "listhead"). Per esempio, anziché liberare tutta la lista, avremmo potuto rimuovere solo tutti i nodi associati ad un file scrivendo:

```
FreeDAList(Nodo_del_primo_elemento_File);
```

Si potrebbe presentare anche la necessità di liberare singoli nodi nella lista DA. A questo scopo utilizzeremo la funzione DosFreeMem() contenuta in arp.library, in quanto i nodi della lista DA vengono allocati dalla DosAllocMem() (ecco da dove vengono le lettere DA). I nodi della lista DA sono caratterizzati dall'avere dimensioni variabili. Le funzioni DosAllocMem() e DosFreeMem() usano un semplice sistema di controllo a tabelle, sistema già utilizzato dall'AmigaDOS, per tenere sotto controllo blocchi di memoria di dimensioni variabili. Utilizzando il metodo dei nodi a dimensione variabile, arp.library è in grado di allocare la memoria per un nodo con una sola chiamata all'apposita funzione. In questo modo non solo si rendono le cose più semplici, ma si previene anche un'eccessiva frammentazione della memoria, cosa che potrebbe creare seri problemi con liste di questo tipo.
A titolo di esempio su come liberare i nodi di una lista DA, ecco come sarebbe stato scritto il loop per visualizzare i file nel programma "WildDemo2.c", se avesse anche dovuto liberare ogni nodo dopo averlo utilizzato:

```
for (tmp = de->de_Next; de; tmp = de->de_Next)
{
  columnize(de->de_Name);   /* usalo */
  DosFreeMem(de);           /* liberalo */
  de = tmp;                 /* prendi il prossimo nodo */
}
```

Notiamo che AddDANode() utilizza nodi di tipo DirectoryEntry, mentre FreeDAList() no. FreeDAList() può essere utilizzata per

Mem() e contengano nella prima variabile il puntatore al nodo successivo.

Listato 1: "Startup.c"

```
/* Startup.c — Esempio di utilizzo del codice di startup per l'analisi
 *    degli argomenti.
 *    -=+SDB+=-
 *
 * Copyright (c) 1987, Scott Ballantyne,
 * Usate e abusate a vostro piacere.
 *
 * Compilate normalmente ed effettuate il link con
 * arp.lib e c.lib.
 *
 * Manx:
 * cc startup.c
 * ln startup.o -larp -lc ; link con arp.lib
 *
 * Lattice:
 * lc startup.c
 *  blink lib:c.o startup.o to startup lib lib:arp.lib lib:lc.lib lib:amiga.lib
 *
 * Notate che quando utilizzate il codice di startup di ARP, non c'è bisogno
 * di aprire o chiudere arp.library.
 *
 * Ancora meglio, IntuitionBase e GfxBase verranno inizializzati *per voi;
 * non c'è bisogno di aprire o chiudere neanche queste librerie.
 *
 */

#include <exec/types.h>
#include "libraries/arpbase.h"   /* il nostro header standard */
#include "arpfunctions.h"   /* dichiarazioni di funzioni */

/* Mettete in CLI_Template il template che intendete utilizzare,
 * altrimenti verrà utilizzato quello standard.
 * Nota: riceverete un avvertimento dal linker nel nostro caso, in
 * quanto il "_CLI_Template" viene definito qui una seconda *volta
 * Questo significa semplicemente che state sostituendo il template
 * di default già definito nel codice di startup. Non preoccupatevi.
 */

char *CLI_Template = "FROM/A,TO,OPT/K,COSMIC/S";

/* L'analizzatore di argomenti contenuto in arp.library (GADS) *metterà
 * queste cose nei posti giusti all'interno dell'array argv. Il codice
 * di startup è abbastanza intelligente da creare un array abbastanza lungo.
 *
 * Ecco un metodo per leggere gli argomenti. Quello più efficiente *dipende
 * dal template utilizzato, ovviamente.
 */

#define ARG_FROM  1
#define ARG_TO    2
#define ARG_OPT   3
#define ARG_COSMIC  4

/* Notate che GADS() metterà sempre gli argomenti richiesti nei *punti
 * giusti dell'array. Provate a dare questi input a Startup:
 *
 * 1> Startup FROM file1 TO file2 OPT abc COSMIC
 * 1> Startup From File1 file2 COSMIC
 * 1> Startup Cosmic opt abc TO FILE1 FROM FILE2
 *
 * E per provare cosa succede in caso di errore nella linea di comando:
 *
 * 1> Startup TO File2
 * 1> Startup From File1 File2 Opt
 * 1> Startup From File1 File2 Opt abc Cosmic karmic
 *
 */

/* Mettete in CLI_Help la stringa di aiuto che volete fornire all'utente
 * se questi batte un punto di domanda dopo avere letto il template.
 * Se definite alcuna stringa, verrà fornita quella standard.
 *
 * 1> Startup ?
 *  FROM/A,TO,OPT/K,COMIC/S: ?  <- visualizza la stringa CLI_Help.
 */

char *CLI_Help = "Questo è un demo. Rilassatevi.";

/*
 * Il codice di startup rivelerà eventuali errori restituiti da GADS() *e
 * visualizzerà il messaggio di errore appropriato (provatene a *fare qualche
 * errore ed osservate i risultati).
 *
 * Gli argomenti /A vengono gestiti in questa maniera:
 *
 *  Se l'utente NON fornisce alcun argomento, è compito vostro *recuperare
 *  in qualche maniera gli argomenti.
 *
 * Se l'utente ha fornito qualche argomento, ma ha omesso quelli *necessari
 *  (/A), GADS() se ne accorgerà e il codice di startup si fermerà.
 *
 * Come ultima cosa, le funzioni Printf() e FPrintf() appartengono ad
 * arp.library e funzionano esattamente come quelle normali "C" (quelle
```

```
 * iniziano con le minuscole: printf() e fprintf() ), ma occupano
 * molto
 * meno spazio.
 */

VOID main(argc, argv)
int argc;
char **argv;
{
  if (argc < 2)  /* Ci sono gli argomenti richiesti?  */
  {
        Puts("Uso: FROM <this> TO <that> OPT <abc>
COSMIC<ally>");
    exit(20);
  }
  /* Sappiamo che il prossimo c'è, perché è di tipo /A */
  Printf("FROM argument is \"%s\"\n", argv[ARG_FROM]);
  /* Controlla gli altri */
  if ( argv[ARG_TO] )
    Printf("TO argument is \"%s\"\n", argv[ARG_TO]);

  if ( argv[ARG_OPT] )
   Printf("OPTions specified are \"%s\"\n", argv[ARG_OPT]);

 /* Gli argomenti spefificati come switch non hanno senso come
  * dtringhe.
  * Se lo switch è stato utilizzato dall'utente, la sua posizione
  * nell'array di pointer è occupata da -1L, altrimenti è 0L.
  */

  if ( argv[ARG_COSMIC] )
     Puts("Il programma procede cosmicamente! (COSMICal-
ly)");
  else
    Puts("Lo switch COSMIC non è stato utilizzato");

  /* A questo punto continuate con il resto del programma. */
}
```

Listato 2: "Nostartup.c", un esempio di come utilizzare arp.library senza il codice di startup contenuto in arp.lib. Effettuate il link con a.lib e poi con le altre librerie necessarie. Notate che arp.library deve essere aperta esplicitamente, mentre le librerie di Intuition e della grafica vengono aperte automaticamente e i loro puntatori sono disponibili.

```
/* Nostartup.c — Esempio di come usare ARP senza usare il suo
 * codice di startup.
 *
 * Notate la facilità di conoscere IntuitionBase e GfxBase.
 *
 * Copyright (c) 1987, Scott Ballantyne
 * Usate e abusate a vostro piacere.
 *
 * Manx:
 * cc nostartup.c
 * ln nostartup.o -la -lc  ;per il link con a.lib
 *
 * Lattice:
 * lc nostartup.c
 * blink lib:c.o nostartup.o to nostartup lib lib:a.lib lib:lc.lib
lib:amiga.lib
 *
 * -+=SDB+=-
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <arpfunctions.h>

struct ArpBase *OpenLibrary();
struct ArpBase *ArpBase;   /* Variabile di base della nostra
libreria  */

/* Possiamo usare intuition e la grafica gratis */

struct Library *IntuitionBase, *GfxBase;

/* Esempio di come aprire ARP */

VOID main()
{
  char  buffer[MaxInputBuf];

  if (!(ArpBase = OpenLibrary(ArpName, ArpVersion)) )
    exit(20);
  /* Arp è aperto, adesso leggiamo intuitionbase e graphicsbase
*/
  GfxBase = ArpBase->GfxBase;
  IntuitionBase = ArpBase->IntuiBase;

  Printf("Digita qualcosa: ");
  ReadLine(buffer);
  Printf("Hai battuto \"%s\"\n", buffer);

 /* Questa è la sequenza consigliata per terminare un programma
  * quando
  * usate un ARP da "C". E' meglio non usare ArpExit().
  */
  CloseLibrary(ArpBase);
  exit(0); /* Oppure cadi giù alla fine del mondo */
}
```

Listato 3: "FileRequest.c"

```
/* FileRequest.c — Esempio di come usare la funzione FileRe-
quest() di ARP.
 *
 *                    -=+SDB+=-
 *
 * Copyright (c) 1987, Scott Ballantyne
 * Usate e abusate a vostro piacere.
 *
 * Eseguite il link con arp.lib
 *
```

```
 */

#include <exec/types.h>
#include "libraries/arpbase.h"          /* il nostro header standard
*/
#include "arpfunctions.h"  /* dichiarazioni di funzioni */

/* Inizializzamo una struttura da passare alla funzione File
 *Request().
 *
 * Dobbiamo fornire un buffer per i nomi del file e della directory.
 *
 * Potete fornire entrambi un file ed una directory di default inizializzando
 * gli array con una stringa prima di passare la struttura a FileRequest().
 *
 * In questo esempio, inizializziamo solo la stringa della directory.
 */

#define MAXPATH                 ( (FCHARS * 10) + DSIZE
+ 1)      /* Dimensioni del path */

BYTE Filename[FCHARS + 1];
BYTE Directory[MAXPATH] = "DF0:";

struct FileRequester FR = {
        "Salve! Cliccate un pò!!!",/* Testo di saluto */
        Filename,                  /* filename array */
        Directory,                 /* directory array */
        NULL,  /* Window, NULL == workbench */
        NULL,  /* Flags, non usato in questa versione */
        NULL,  /* Puntatori di funzione, non usati in questa
versione */
        NULL,  /* "" */
};

VOID main()
{
        char *Selection;

        if ( Selection = FileRequest( &FR ) )
        {
            if ( *Selection )          /* Controlla il filename */
                Printf("Filename = %s ", Filename);
            else
                Printf("Nessun Filename selezionato, ");
                Printf("Directory = %s\n", Directory);
        }
        else
            Puts("L'utente ha cancellato il Requester!");
}
```

Listato 4: "Devices.c"

```
/* Devices.c — Demo di arp.library che utilizza le liste DA per
sapere
 *                quali device sono collegati al sistema -
=+SDB+=-
 *
 * Copyright (c) 1987, Scott Ballantyne
 * Usate e abusate a vostro piacere.
 *
 * MANX:
 * cc Devices.c
 * ln Devices.o -larp -lc
 *
 * Lattice:
 * lc Devices.c
 * blink lib:c.o Devices.o to Devices lib lib:arp.lib lib:lc.lib
lib:amiga.lib
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <arpfunctions.h>

struct DirectoryEntry *DeviceList = NULL; /* List header */

#define ARG_DEVICES   1
#define ARG_DISKS     2
#define ARG_VOLUMES 3
#define ARG_ASSIGNS   4

char  *CLI_Template  =  "DEVICES/S,DISKS/S,VOLUMES/
S,ASSIGNS/S";
/* Esempio di un messaggio di aiuto esteso  */
/* Sarebbe bello se altri programmi aumentassero questo tipo di
help */

#ifdef AZTEC_C /* Il Lattice non si comporta bene con le stringhe lunghe */
char *CLI_Help =
"Visualizza i device collegati a questo sistema.\nScegliete:\n\
\tDEVICES — Per vedere tutti i device (default)\n\
\tDISKS   — Per vedere tutti i device basati su disco\n\
\tVOLUMES — Per vedere tutti i volumi\n\
\tASSIGNS — Per vedere tutti gli assegnamenti logici\n\n\
Potete selezionarne una qualsiasi combinazione.  Notate che\n\
DEVICES DISKS è come dire semplicemente DISKS\n";
#else
char *CLI_Help = "Visualizza i device collegati a questo
sistema:\n\
\tDEVICES — Per i device\n\
\tDISKS   — Per i device a disco\n\
\tVOLUMES — Per i volumi\n\
\tASSIGNS — Per gli assegnamenti\n\nO qualsiasi combinazione a scelta\n\n";
#endif

VOID main(argc, argv)
int argc;
char **argv;
{
    LONGBITS WhichDevices = 0;
    LONG  numdevs;
```

```
    register struct DirectoryEntry *dev;

    if (argc < 2 )  /* set default */
        WhichDevices |= DLF_DEVICES;

    /* Scopri quali switch sono stati utilizzati, se lo sono stati */


    if ( argv[ARG_DEVICES] )
        WhichDevices |= DLF_DEVICES;
    if ( argv[ARG_DISKS] )
        WhichDevice |= ( DLF_DEVICES| DLF_DISKONLY
            );
    if ( argv[ARG_VOLUMES] )
        WhichDevices |= DLF_VOLUMES;
    if ( argv[ARG_ASSIGNS] )
        WhichDevices |= DLF_DIRS;

    /* Adesso esamina i device richiesti */
  if ( (numdevs = AddDADevs( &DeviceList, WhichDevices ))
  ==0)
    {
        Puts("Non ho trovato niente");
        exit(0);
    }
    Printf("Ho trovato %ld device\n", numdevs);
    /* Adesso visualizza il nome e il tipo del device. */

    for ( dev = DeviceList; dev; dev = dev->de_Next)
    {

        Printf("%-32s", dev->de_Name);
        switch (dev->de_Type)
        {
                case DLX_DEVICE:
                        Puts("Device residente");
                        break;
                case DLX_VOLUME:
                        Puts("Volume [montato]");
                        break;
                case DLX_UNMOUNTED:
                        Puts("Volume [non montato]");
                        break;
                case DLX_ASSIGN:
                        Puts("Assegnamento logico");
                        break;
                default:
                        Puts("Incontrato device misterio
                        so!");
        }
    }
    FreeDAList(DeviceList);
    }
```

Listato 5: "WildDemo1.c"

```
/*
 * WildDemo1.c — Dimostrazione dell'uso delle routine di
"wildcard"
 *      dell'arp.library. Questo programma simula una sorta di
comando dir.
 *
 * -+=SDB=+-
 *
 * Copyright(c) 1987, Scott Ballantyne
 * Usate e abusate a vostro piacere
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include<libraries/dos.h>
/*Per ERROR_NO_MORE_ENTRIES, ^C, ecc. */
#include <arpfunctions.h>

/* Una struttura AnchorPath leggermente estesa */
/* Estendiamo la struttura ancora in questa maniera perché
useremo le
 * funzioni di pattern matching per costruire i path name completi.
 * Se non intendete usare questa possibilità, allora non avete alcun
 * bisogno di estendere la struttura.
 */

struct UserAnchor {
   struct AnchorPath ua_AP;
    BYTE    moremem[255];   /* semplice modo di estendere
ap_Buf[] */
};

/* Le nostre stringhe di help o di template */

char *CLI_Template = "Pattern";
char *CLI_Help = "Produce una lista di file o di directory con la
wildcard";

main(argc,argv)
int argc;
char **argv;
{
   struct  UserAnchor *Anchor;
   char    *pat;
   LONG    Result;
   if  (argc < 2)
      pat = "*";  /* Tutti i file o le dir nella directory corrente */
   else
      pat = argv[1];

 /* Poiché una struttura AnchorPath contiene una struttura BCPL
  * FileInfoBlock, dobbiamo sempre assicurare che la Anchor
    Path sia
  * allineata sulla longword. Un semplice modo per far questo e
    per
  * tracciarla consiste nel chiamare ArpAlloc().
  */

 if(Anchor=(structUserAnchor
 *)ArpAlloc((ULONG)sizeof(*Anchor)) )
```

```
    {

      Anchor->ua_AP.ap_Length = 255;
      /* Vogliamo che venga costrruito un intero path */
       /* Interromperemo l'esecuzione sia sul control-C sia */
       /* sul control-D, solo per mostrare come si fa     */

      Anchor->ua_AP.ap_BreakBits |= SIGBREAKF_CTRL_C;
      Anchor->ua_AP.ap_BreakBits |= SIGBREAKF_CTRL_D;

    }
    else
    {

      Puts("No memory!");
      exit(20);
    }

    Result = FindFirst(pat,Anchor);

    while (Result == 0)
    {

    Printf("%s",Anchor->ua_AP.ap_Buf);  /* Stampa il nome */

      if (Anchor->ua_AP.ap_Info.fib_DirEntryType >= 0)
        Puts("\t(dir)");    /* Stampa il tipo */
      else
        Puts("");
      Result = FindNext(Anchor);
    }

    /* Libera la catena di Anchor costruita dalle funzioni qui
    sopra */

    FreeAnchorChain(Anchor);

    /* Infine, controlliamo il risultato - se è
    ERROR_NO_MORE_ENTRIES,
     * allora stiamo uscendo regolarmente, altrimenti è un'inter-
ruzione
     * o un effettivo errore di qualche tipo.
     */

    if (Result == ERROR_BREAK)  /* Control-C */
    {
    if (Anchor->ua_AP.ap_FoundBreak&
      SIGBREAKF_CTRL_C)
        Puts("***Interruzione a causa di '^C' ");
      else
        Puts("***Interruzione a causa di '^D' ");
    }
    else if (Result == ERROR_OBJECT_NOT_FOUND)
    {
      Printf("Non riesco a trovare %s\n",pat);
      exit(20);
    }
    else if (Result == ERROR_BUFFER_OVERFLOW)
    {
      Puts("Avrei dovuto allocare un buffer più grosso.
Scusa!");
      exit(20);
    }
    else if (Result != ERROR_NO_MORE_ENTRIES)
    {
      Puts("Ha avuto luogo un errore di I/O!");
      Printf("%ld\n",Result);
      exit(20);
    }
    /* altrimenti... */
    exit(0);
}
```

Listato 6: "WildDemo2.c"

```
/*
 * WildDemo2.c -  Cerca i pattern nel filesystem, separandoli
in directory
 * file che vengono ordinate separatamente utilizzando le liste
DA.
 *
 * -+=SDB=+-
 *
 * Copyright (c) 1987, Scott Ballantyne
 * Usate e abusate a vostro piacere.
 *
 * Manx:
 * cc WildDemo2.c
 * ln WildDemo2.o -larp -lc
 *
 * Lattice:
 * lc WildDemo2.c
 * blink lib:c.o WildDemo2.o to WildDemo2 lib lib:arp.lib
lib:lc.lib lib:amiga.lib
 */

#include <exec/types.h>
#include "libraries/arpbase.h"
#include <libraries/dos.h>
#include "arpfunctions.h"


struct DirectoryEntry *FileList = NULL;
/* Head della lista DA */


/* Chiavi secondarie di ordinamento per la lista per visualizza-
re prima
 * le directory.
 */

#define DIRECTORY      0L
#define FILE           1L

/* template e stringa di help */
char *CLI_Template = "Pattern";
```

```
char *CLI_Help = "Visualizza directory e file utilizzando le
wildcard";

BOOL lflag;       /* Cosmetics */

VOID columnize();
VOID main(argc, argv)
int argc;
char **argv;
{
    struct AnchorPath *Anchor;
    register struct DirectoryEntry    *de; /* servirà più tardi */
    register LONG fcount = 0, dcount = 0;
    LONG      key;
    char      *pat;
    LONG      Result;
    if (argc < 2)
        pat = "*";          /* tutti i file/dir nella directory
                               corrente*/
    else
        pat = argv[1];
        /* Get our AnchorBase */

        if (Anchor=(structAnchorPath*)ArpAlloc(
          (ULONG)sizeof( *Anchor )) )
    {
        Anchor->ap_Length = 0;
        Anchor->ap_BreakBits |= SIGBREAKF_CTRL_C;
    }
    else
    {
        Puts("Non c'é memoria!");
        exit(20);
    }

    Result = FindFirst( pat, Anchor);

    while ( Result == 0 )
    {
        if (Anchor->ap_Info.fib_DirEntryType >= 0)
        {
            key = DIRECTORY;
            dcount++;
        }
        else
        {
            key = FILE;
            fcount++;
        }

        if ( !AddDANode( Anchor->ap_Info.fib_FileName,
          &FileList, 0L, key))
        {
            Puts("Non c'é; memoria!");
            FreeDAList( FileList );
            exit(20);
        }
        Result = FindNext( Anchor );

    }
    /* libera la AnchorChain costruita in precedenza */
    FreeAnchorChain( Anchor );
    if (Result == ERROR_BREAK)          /* Control 'C' */
        Puts("***Break");
    else if (Result == ERROR_OBJECT_NOT_FOUND)
    {
        Printf("Non ho trovato %s\n", pat);
        exit(20);

    }
    else if (Result != ERROR_NO_MORE_ENTRIES)
    {
        Printf("ERROR #%ld\n", Result);
        exit(20);
    }
    /* Adesso percorriamo la lista DA, visualizzando i file
       ordinati */
    de = FileList;
    Printf("%ld Directories:", dcount);
    if (de->de_Type == DIRECTORY)  /* fino a quando
                                      abbiamo directory */

    {
        lflag = 1;
        for ( ; de ; de = de->de_Next)
        {
            if (de->de_Type != DIRECTORY)
                break;
            columnize(de->de_Name);
        }
    }
    /* adesso occupiamoci dei file */
    Printf("\n%ld Files:", fcount);
    if ( de )       /* ancora altri elementi */
    {
        lflag = 1;
        for ( ; de ; de = de->de_Next)
            columnize(de->de_Name);
    }
    Puts("");
    FreeDAList( FileList );
    }
/* Routine per stampare i nomi su due colonne */
VOID columnize(name)
char *name;
{

    if (lflag)
        Printf("\n\t");
    Printf("%-32s", name);
    lflag = !lflag;
}
```

Breakpoint

di Victor A. Wagner,Jr.

Questo è il primo di una serie di articoli sull'arte e la scienza del debugging del software. Cominciamo con uno sguardo agli anni bui...

Vic Wagner cominciò a interessarsi di computer nel 1965 mentre era in servizio di leva, lavorando a uno studio di simulazione di volo digitale dell'aeronautica statunitense. Dal 1966, ritornando un civile, ha lavorato con molti costruttori di minicomputer nell'area del software di base per applicazioni realtime. Nei giorni feriali, dalle 8 alle 5, Vic lavora al Technical Support line (il servizio di supporto telefonico) della Computer Automation Inc., e gestisce 3 sistemi software realtime. La sera e nei fine settimana, passa gran parte del suo tempo a parlare con Taarna (il suo Amiga 1000) e a scrivere programmi su CIS AmigaForum. Vic probabilmente è conosciuto nell'area Amiga per la sua profonda conoscenza del ben noto programma di debugging MetaScope, pubblicato dalla Metadigm Inc.

Era la notte della vigilia del Beta
E tutto in casa,
Non una creatura era agitata.
Nemmeno il mio mouse.
I breakpoint erano settati
Nel programma con attenzione,
Per timore che il Guru
Presto sarebbe tornato.

Se il mio mouse non si muove, è quasi certamente segno che il guru mi visiterà da un momento all'altro, non importa quanto sia stato attento. Bene, sembra proprio che abbia intenzione di cominciare questa sessione di debugging, perciò diamo inizio alle danze.

Prima di entrare in dettaglio a parlare del debugging su Amiga, mi piacerebbe raccontare una breve fiaba di computer. Capire da dove veniamo ci può aiutare a capire dove siamo e perché siamo qui. Alcune delle tecniche usate agli albori dell'informatica sono tuttora applicabili. Altre, pur essendo ancora in voga, sembrano risentire del peso degli anni (e probabilmente sarebbe ora di mandarle in pensione).

Prima di tutto, analizziamo la parola in sè : "Debug". Da dove proviene questo termine? Ho sentito la storia che sto per raccontarvi 20 anni fa, e recentemente lessi un'intervista dell'Ammiraglio Grace Hopper in cui la raccontava tale e quale: deduco di conseguenza che abbia per lo meno un fondamento di verità.

Pare che uno dei primi computer realizzati lavorasse ottimamente, finché improvvisamente cominciò a comportarsi in modo strano. Alcune volte i programmi di test funzionavano correttamente, altre volte no. Coloro che lavoravano con la macchina cominciarono a strapparsi i capelli. La causa di questo strano comportamento venne infine scoperta: una tarma era andata a finire in uno dei relè (ricordatevi che era uno dei primi computer), causando un guasto saltuario. Siccome rimuovendo l'insetto (bug) si risolse il problema, qualcuno pensò di coniare il termine "debugging".

Se tutto ciò che si dovesse fare per eliminare i bug fosse spruzzare una bella bomboletta di insetticida (incrementando il buco di ozono!), per poi insinuarsi pazientemente nella macchina per ripulirla dai bug morti con un aspirapolvere, oggi il debugging sarebbe un'occupazione di minor rilievo, legata al controllo degli insetti e non menzionata nelle gerarchie aziendali. I debugger (intesi come persone) diverrebbero probabilmente dei pariah a causa del massiccio uso di pesticidi e questo articolo non sarebbe certo stato necessario. Ma tante cose sono cambiate da quando al primo computer è stato fatto il debug: non solo per il fatto che i computer siano diventati più piccoli e non ci si può infilare dentro carponi (anche se le tarme sono insetti molto piccoli e si intrufolano dappertutto); Il più grande cambiamento si è verificato con l'avvento del "software".

I computer di un tempo erano a logica cablata: erano cioè costruiti per svolgere un solo lavoro. Se i computer non svolgevano bene il proprio lavoro, significava che c'era un guasto e perciò occorreva ripararlo. Trovare il componente guasto poteva a volte essere (e lo era spesso) un lavoro lungo e tedioso. Sono stati inventati degli strumenti allo scopo di ritrovare questi guasti: l'onnipresente oscilloscopio e il più recente analizzatore logico sono i principali ferri del mestiere.

La nascita del software

Ben presto ci si rese conto che costruire un nuovo computer ogni volta che si presentava un problema diverso da risolvere, costituiva un grande sforzo per le risorse della razza umana. Si è capito che invece di immagazzinare nella memoria di un computer solo dati, era possibile immagazzinare anche rappresentazioni e parti di algoritmo. Ciò che ai giorni nostri viene chiamato computer, era chiamato al tempo computer digitale programmabile. Con l'avvento del computer digitale programmabile, quindi, si scoprì l'esistenza di un'altro tipo di errori: gli errori software.

C'è molta differenza tra il trovare errori hardware e il trovare errori software. Quando uno scopritore di bug hardware ricerca un guasto, sa che quell'hardware un tempo funzionava: ciò che ha sempre funzionato ora si è guastato. Il software, d'altra parte, non si può guastare: se si trova un bug nel software, è perché è sempre esistito. Un'altra distinzione si ritrova nella relativa complessità degli oggetti da correggere o da riparare. La mia esperienza sia in campo hardware che in campo software, mi conduce alla conclusione che il software ora supera l'hardware in quanto a complessità; occorrono quindi degli strumenti per affrontare la situazione.

I primi programmi per computer non potevano servirsi nè di compilatori e nemmeno di assemblatori. I programmatori prima stabilivano quali bit dovevano essere posti in memoria e poi li inserivano utilizzando, spesso, deviatori mossi a mano. Poi ese-

guivano il programma e confrontavano i risultati con le laboriose risposte generate manualmente. Se erano fortunati, ogni cosa andava come previsto; se c'erano errori, occorreva controllare la memoria per assicurarsi che tutte le istruzioni fossero corrette. Se queste operazioni non davano esito positivo, i programmatori dovevano impostare la macchina ad eseguire una istruzione alla volta e a registrare tutte le informazioni in risultati intermedi, cercando di risolvere il problema in questo modo. Ma far lavorare una CPU passo-passo, significa sprecare una notevole quantità di tempo, e questo risultava costoso, specialmente con i vecchi computer. Si dovevano cercare anche altre strade per permettere al programmatore di trovare gli errori senza occupare per lungo tempo la macchina.
Il dump della memoria (ovvero la lista di tutto il contenuto della memoria stampato su carta) è stato il primo strumento di debugging. I dump potevano essere prodotti con il minimo sforzo (non dimenticate che a quei tempi una macchina da 16 kbyte era una macchina molto potente). Armato del dump, il programmatore doveva poi spendere ore, e a volte giorni, seduto alla sua scrivania a risolvere il problema. La tecnica è ancora in auge in alcuni centri di calcolo: immergersi nel profondo di un dump è, a quanto si dice, una dei momenti del debugging più tediosi.
Vi rivelo però che un tempo il debugging non era un problema reale: erano così tanti gli sforzi necessari per realizzare un programma che il tempo necessario per il debugging era relativamente poco. A onor del vero, il grande sforzo necessario per la creazione del programma non permetteva al programmatore di distinguere la fase di creazione dalla fase di debugging: veniva più che altro vista come una verifica delle scorrettezze di progetto e implementazione.
Il grande problema ora era creare programmi in maniera più efficiente.

E poi venne l'assembler

Alcune persone geniali, non ne ricordo il nome, osservarono che il computer risultava ottimale nello svolgere operazioni metodiche e tediose; in fondo quell'aggeggio aveva anche buona memoria. Partendo da queste non troppo rimarchevoli affermazioni, giunsero a una rimarchevole realizzazione: l'assembler. L'assembler permetteva al programmatore di concentrarsi su ciò che il computer doveva fare, piuttosto che sul tentativo di ricordare, ad esempio, che un istruzione ADD si realizza con i bit 0,3,4 e 7 posti a uno. Quando qualcun'altro osservò che potevano essere dati dei nomi anche a delle locazioni di memoria (e altre cose del genere), saltò fuori l'assembler simbolico. Improvvisamente, furono creati una pletora di programmi. Il debugging non era considerato ancora un problema realmente serio, poichè questi primi programmi erano ancora piuttosto semplici rispetto a quelli di oggi. I programmi semplici possono spesso essere corretti semplicemente leggendo il listato con molta attenzione; se si fallisce, si chiede aiuto a qualcun'altro che leggerà il listato al nostro posto. Se anche questo fallisce, beh, c'è sempre come ultimo rimedio il dump della memoria, ore tarde e litri di caffè.

Alcuni programmi di quei tempi lasciavano già presagire come si sarebbe potuto realizzare poi un debugger. Questi programmi permettevano essenzialmente il controllo e la modifica di locazioni di memoria (generalmente in diversi formati). Ai tempi avevano assolto il compito assegnatogli, ma oggi è molto improbabile che qualcuno li possa considerare dei debugger. E' importante ricordare che i dispositivi di memorizzazione magnetici non erano disponibili. Molti programmi sorgenti erano tenuti su nastri o schede perforate. I programmi eseguibili era più comodo tenerli su nastri perforati, poiché le schede perforate risultavano molto costose.
Tenere i programmi eseguibili su nastro perforato non era una brutta idea, ma considerate i problemi associati a un sorgente su nastro di carta. Scrivere un programma su nastro era laborioso: il nastro doveva essere copiato con i cambiamenti inseriti; per lo meno con le schede perforate era necessario solo sostituire quelle modificate. Una modifica comportava la creazione di un altro programma su nastro avente circa le dimensioni del nastro precedente. I programmatori erano restii a operare cambiamenti sul codice sorgente ogni volta che veniva ritrovato un bug, perché l'intero ammasso di carta sprecata nelle modifiche avrebbe presto avvolto il loro ufficio: per ovviare all'inconveniente, "rattoppavano" l'eseguibile con le loro routine di utilità e creavano un nuovo eseguibile con le stesse routine. Una delle più importanti caratteristiche di queste utility era l'abilità di portare il contenuto della memoria su nastro di carta (o su schede) in una forma adatta al ricaricamento una volta che queste venivano nuovamente modificate.
La tipica "ultima versione" era generalmente un nastro contenente il codice sorgente di recente produzione, un codice prodotto dall'assemblatore e marcato "LATEST" (ovvero "ULTIMO"), scarabocchiato con un evidenziatore in questo o quel punto, ad indicare le ultime toppe inserite nell'eseguibile.
Quando il listato era troppo rovinato o non c'era più spazio ai margini per poter scarabocchiare, i nostri eroi, seppur riluttanti, sostituivano il codice sorgente e lo riassemblavano.
Naturalmente i nuovi listati venivano marcati come "LATEST" con grandi e rassicuranti lettere. Vi chiederete a questo punto perché non usare la data posta in cima al listato per tenere traccia dell'ultima versione. Beh, un tempo non esistevano troppi sistemi operativi, e gli assemblatori erano generalmente dei programmi separati. Prima veniva avviata la macchina, dopodichè si caricava ed eseguiva l'assemblatore.
Ciò significa che non esistevano date in cima al listato. Se vi chiedete ora perché non si scriveva la data a mano sul listato, tutto quello che posso dirvi è: "non lo so".
Un programmatore che doveva avviare una sessione di debug, per eseguire tutte le operazioni necessarie, seguiva una sorta di training autogeno del tipo:

1) Recupera il listato appena assemblato e il nastro oggetto.
2) Estrai gli oggetti per ognuna delle subroutine di libreria che si intendono usare.

Questo può richiedere qualche spiegazione: i linker erano particolarmente rari, così ogni programma o subroutine era assemblato in locazioni di memoria assolute. Gli assegnamenti di queste locazioni erano lasciate al capo progetto, che manteneva una mappa di memoria contenente tutti i moduli che si volevano inserire, nonchè gli entry point per tutte le routine, gli argomenti passati e l'uso delle risorse.

3) Carica il programma di utilità, e poi tutti i moduli per costruire l'intero programma.

4) Se sei intelligente, fatti una nuova copia su nastro della configurazione corrente della memoria, prima che qualcosa salti in aria.
5) Tieni conto di qualsiasi locazione di memoria aggiuntiva richiesta dai programmi di utilità.
6) Se la macchina su cui stai lavorando dispone di un "Address Halt" hardware, selezionalo all'indirizzo dove desideri che la macchina si fermi per poterne osservare lo stato. Se la macchina non ha un "Address Halt", inserisci un'istruzione di "Halt" nel punto in cui vuoi che si fermi.
7) Salta all'indirizzo da cui vuoi far partire l'esecuzione e attendi che la macchina si metta in halt.
8) Se la tua macchina non ha l'Address Halt, reinserisci l'istruzione che hai tolto per poter inserire l'"Halt".
9) Controlla tutti i registri (tutte le macchine disponevano di console sulle quali si potevano esaminare i contenuti dei registri), nonché il contenuto di alcune locazioni di memoria, e scegli se proseguire o ricominciare.
10) Molte macchine disponevano di una funzione "Single step", affinchè il computer potesse eseguire una sola istruzione alla volta. La funzione "Going on" consisteva invece nell'operazione di single step e nel controllo dei registri automatico per verificare l'operato del programma.
11) Una volta trovato un errore nel programma, vai all'indirizzo di partenza del programma di utility, modifica la memoria e ritorna al punto 4. Se il tempo a disposizione sulla macchina è quasi scaduto, raccogli le tue cose e torna in ufficio. E, non dimenticare di segnare le modifiche operate alla memoria sul listato, così un giorno potrai modificare il sorgente e riassemblarlo!

Single step - Un balzo da giganti?

Da allora sono successe parecchie cose; non so quale sia avvenuta prima, perciò andrò un po' a naso. Qualcuno notò che parecchio del tempo utilizzato nel testing/debugging veniva passato a pigiare il tasto di "Single step", ad annotare i registri, a ripigiare il "Single step"...fino alla nausea. Dato che questa è ovviamente un'operazione ripetitiva, e i computer sono bravi a eseguire operazioni ripetitive, fu scritto un programma che imitasse questo comportamento. Il programma in oggetto, denominato "Trace program", al tempo simulava l'esecuzione di ogni istruzione e stampava il contenuto dei registri dopo che ogni istruzione simulata veniva eseguita.
Apparvero molti Trace program con ogni sorta di caratteristiche. alcuni visualizzavano l'istruzione che veniva eseguita nello stesso formato riconosciuto dall'assemblatore; altri visualizzavano solo i registri il cui contenuto veniva modificato. In altri ancora si dovevano specificare quali indirizzi o serie di indirizzi erano importanti ed essi avrebbero stampato informazioni relative solamente a quegli indirizzi. Ben presto questi programmi divennero più popolari degli hula hoop. I manager li amavano: infatti i programmatori non sarebbero rimasti più a lungo seduti alla console a sprecare tempo prezioso: un computer medio di allora costava almeno quanto lo stipendio annuale di dieci programmatori e il mantenimento ne valeva almeno altri quattro. La tipica sessione di debug divenne così: carica il programma, imposta le operazioni dell'utility, controlla che la stampante stampi parecchie centinaia di pagine, riprenditi la tua roba e torna alla scrivania.
Poiché i Trace program non erano in grado di prendere decisioni riguardanti ciò che era stato trovato in alcuni punti della loro esecuzione, è mia opinione che molto tempo di elaborazione e molta carta stampata andavano persi. La mia esperienza personale con questo tipo di debugging non mi convinse del tutto, poiché ciò che avevo bisogno di vedere non era l'intero tracciato delle operazioni eseguite, ma soltanto una o due delle diverse centinaia di pagine. Uscire dalla sala computer con tutta quella carta sotto il braccio dava però un certo tono, e le ore che si sarebbero trascorse sottolineando copiosamente in rosso e nero il listato, avrebbero fatto sembrare chiunque molto indaffarato; in effetti lo era, ma non sono troppo sicuro che fosse tempo produttivo.
Può darsi che io non abbia mai imparato a usare i Trace program; può anche darsi che io non ne abbia mai avuto uno buono. Ma so di aver visto dei programmi di debug che correggevano un altro programma, che a sua volta correggeva ... beh, avete capito. Ragazzi, quanta carta sprecata, e quanto tempo. In ogni modo, essi tenevano il programmatore fuori dai guai e indubbiamente portavano molto denaro all'economia nazionale. Sono sicuro che sia i costruttori di stampanti, sia i fornitori di carta amassero i programmi di debug.

Un'altra innovazione: la rilocabilità

Con l'andar del tempo, una nuova parola cominciò a farsi strada nel non troppo coscienzioso mondo dei computer. Un nuovo termine: rilocabile. Notate come fà schioccare bene la lingua: RI-lo-CA-bile; proprio piacevole a dirsi.
Se non eri nel giro, non ne capivi il significato. I non programmatori rimanevano estasiati nell'ascoltare tutti i termini "magici" che i programmatori pronunciavano. Tutto ciò toccava profondamente l'ego dei programmatori, tanto che molti di noi si montarono la testa. In molti casi, lo stereotipo del programmatore prima-donna non era troppo lontano dal vero.
Ad ogni modo, mentre la rilocabilità permetteva di scrivere programmi più velocemente, i suoi effetti in fase di verifica non risultavano troppo entusiasmanti. Ho menzionato prima il fatto che gli assembler erano realizzati con indirizzi assoluti: ciò significa che l'indirizzo di ogni modulo assembly era conosciuto nel momento in cui veniva eseguito l'assemblatore. Uno dei problemi che più comunemente si riscontravano nella fase di mapping della memoria consisteva nel fatto che i moduli risultavano più grandi dello spazio ad essi allocato, cosicchè alcuni o tutti i pezzi dovevano essere spostati nuovamente. Quando questo accadeva, i moduli che erano stati referenziati nelle parti spostate, dovevano essere riassemblati. Risultato: molto lavoro per i poveri programmatori. Il codice rilocabile aveva i presupposti per risolvere tutti i problemi, e in alcuni casi li risolse veramente. Ad essere sinceri, i problemi di allocazione degli indirizzi vennero risolti, ma ad un prezzo. Per poter usare questa nuova ed eclatante caratteristica, occorreva:

1) Un nuovo assembler rilocabile
2) Un caricatore-linker rilocabile

Ma che cosa è il linker? Beh, quando gli indirizzi di tutti i moduli del sistema erano conosciuti, era possibile inserire un comando nell'assembler che recitava così: "la routine trigonometrica del

calcolo del seno è all'indirizzo 123", indicando all'assemblatore il modo in cui chiamare la routine per il calcolo del seno. Tutto ciò generalmente si risolveva in una cosa del tipo:

```
SENO EQU 123
```

Era, ovviamente, compito del programmatore controllare che la routine per il calcolo del seno si trovasse effettivamente all'indirizzo 123 dopo che il programma veniva caricato in memoria. A questo punto esisteva un metodo differente per dire all'assemblatore dove ritrovare la routine per il seno.
Un qualcosa come:

```
EXTERNAL SENO
```

Questa istruzione indicava all'assemblatore di partecipare a una congiura con il "linking loader rilocabile" per assicurarsi che il corretto indirizzo del seno sarebbe stato inserito in tutti i luoghi appropriati, affinché le istruzioni riferite al calcolo del seno fossero eseguite correttamente. Tutto questo è effettivamente un buon metodo, ma come fa il caricatore a sapere dove si trova esattamente la routine del seno? Nel codice assembly che conteneva la routine SENO, si sarà certamente trovata un'istruzione del tipo:

```
NAME SENO
```

e più avanti, nello steso programma:

```
SENO EQU $
```

A coloro che non conoscono la ragione per cui il simbolo "$" rappresenti la locazione corrente, dico che è solo un gioco di parole. Qualcuno da qualche parte stava scrivendo un assemblatore e aveva bisogno di un simbolo per la locazione corrente (current location):

"Current" -> "Currency" -> "Currency symbol" -> "$" (Il gioco di parole sta nel fatto che currency significa valuta e, come tutti sanno, il simbolo della valuta americana e il $).

Ma torniamo ai linking loaders rilocabili. Il caricatore poteva ritrovare la routine SENO per poi mettere a posto tutti i programmi che la referenziano. Un loader particolarmente intelligente poteva anche decidere se un certo modulo assembly fosse o meno da utilizzare, basandosi sul fatto che i suoi entry point fossero richiamati in qualche parte del programma. Così si poteva caricare il programma e si erano furbescamente evitati i problemi associati all'indirizzamento assoluto. Ora, come era possibile ritrovare il punto in cui doveva essere presente una istruzione? Il nostro listato assembly, che usava un tempo dei bei indirizzi sui margini, ora partiva perennemente dall'indirizzo 0 (che quasi sicuramente non era l'indirizzo in cui veniva caricato il programma). La risposta a questo problema era un listato chiamato "load map", generato dal caricatore, contenente tutti i simboli e gli indirizzi che erano stati loro assegnati. Se si era fortunati, il listato era disponibile sia in ordine alfabetico che in ordine numerico. A questo punto si ritornava al vecchio procedimento utilizzato prima dell'esistenza dei caricatori, all'uso del "single step" o all'uso dei Trace program, dopo aver fatto tutti i conticini per stabilire dove si trova effettivamente in memoria la mia locazione 123. La realizzazione dei programmi era certamente divenuta più facile, ma la verifica era rimasta pressoché invariata, eccetto per coloro che venivano considerati dei maghi della numerazione ottale (proprio quella che veniva usata a quei tempi). Insomma, ci si era già mossi, ma non si era ancora del tutto partiti. Ora che era più facile realizzare i programmi, venivano costruiti più computer e anche la produzione di software era aumentata: si era verificato un boom della elaborazione di informazioni e dello sviluppo di programmi. Il debugging non veniva ancora concepito come problema di primaria importanza, essendo sempre confrontato sempre con il problema generale di realizzare un programma: Progettazione, Implementazione, e verifica (guarda caso la verifica veniva sempre scritta con l'iniziale minuscola). La gente pensava che, se un problema fosse stato espresso in modo più semplice, la verifica sarebbe andata via liscia di conseguenza. Si percepiva che il grande problema non era la progettazione del programma, ma l'inclinazione umana a sbagliare nella trascrizione del progetto in quelle buffe e piccole istruzioni di cui il computer non poteva fare a meno per poter eseguire un lavoro.

E finalmente, un linguaggio ad alto livello

Un'altra persona geniale saltò fuori con l'idea di scrivere un programma che avrebbe tradotto automaticamente le espressioni matematiche in istruzioni per computer.Prima di allora quasi tutti i programmi che erano stati scritti non erano troppo bravi nel masticare numeri, e quindi questa sembrò una buona idea. Nacque così il FORTRAN che stava per, ovviamente, FORmula TRANslator. Wow, ora si sarebbero potuti scrivere molti programmi! E si potevano scrivere anche molto velocemente! Non ci si sarebbe dovuti preoccupare della progettazione ma solo dell-'implementazione corretta, poichè il computer avrebbe generato da solo le istruzioni giuste!

WOW!

Come avrete certamente capito, tutto questo non aveva realmente risolto il problema. Semplicemente si cambiava la percezione di dove si trovasse effettivamente il problema. Ora qualcuno cominciava a scrivere "Verifica" con la lettera maiuscola. Qualcun'altro cominciò a scrivere programmi che avrebbero aiutato altri a correggere i loro programmi; era stato aggiunto dell'hardware specifico per facilitare le operazioni di verifica: era quasi l'alba. Chiunque poteva scrivere un programma, ma scrivere un programma corretto era un'altra cosa.Entreremo in alcuni dettagli sullo stato attuale del debugging nel prossimo numero. Nell'attesa, eccovi un quesito facile facile: secondo voi quale tecnica di debugging venne applicata con l'introduzione del FORTRAN? Questa tecnica è in uso ancora oggi e si ritrova spesso su macchine con linguaggi ad alto livello. Un altro suggerimento: noi tutti usiamo questa tecnica.
Nel prossimo numero tratteremo dei breakpoint, da cui prende il nome l'articolo, e vedremo quale profondo effetto ha sulle tecniche di debugging. Vedremo anche quale radicale cambiamento ha portato l'uso del video nella programmazione (Progettazione, Implementazione...e Verifica).

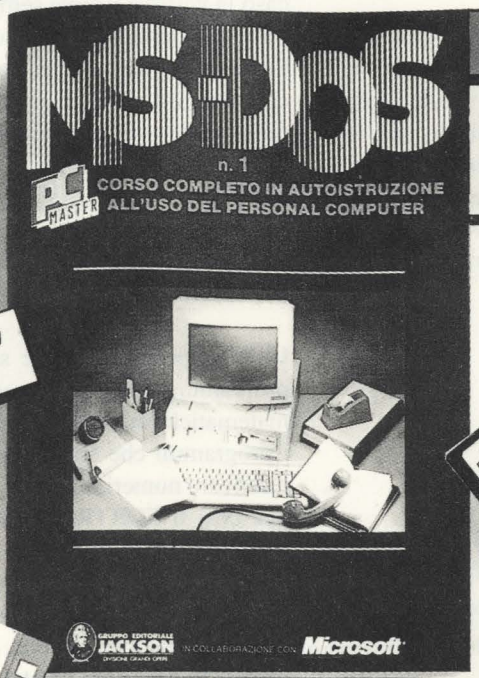
MS-DOS

PC MASTER

CORSO COMPLETO IN AUTOISTRUZIONE ALL'USO DEL PERSONAL COMPUTER

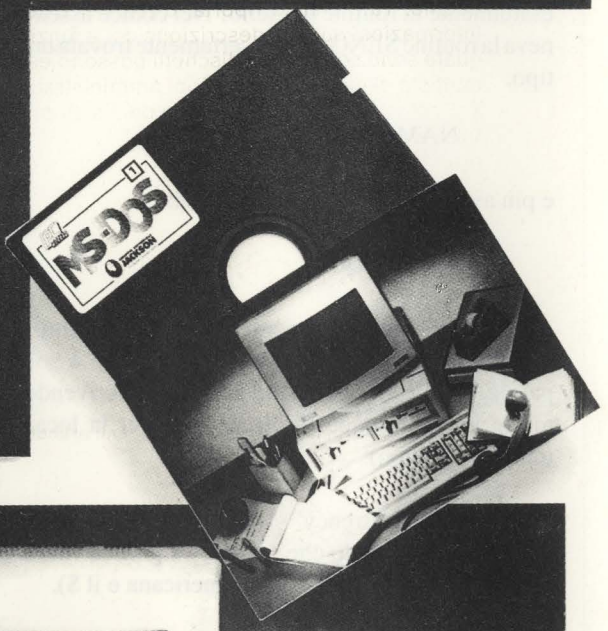
NOVITÀ IN EDICOLA

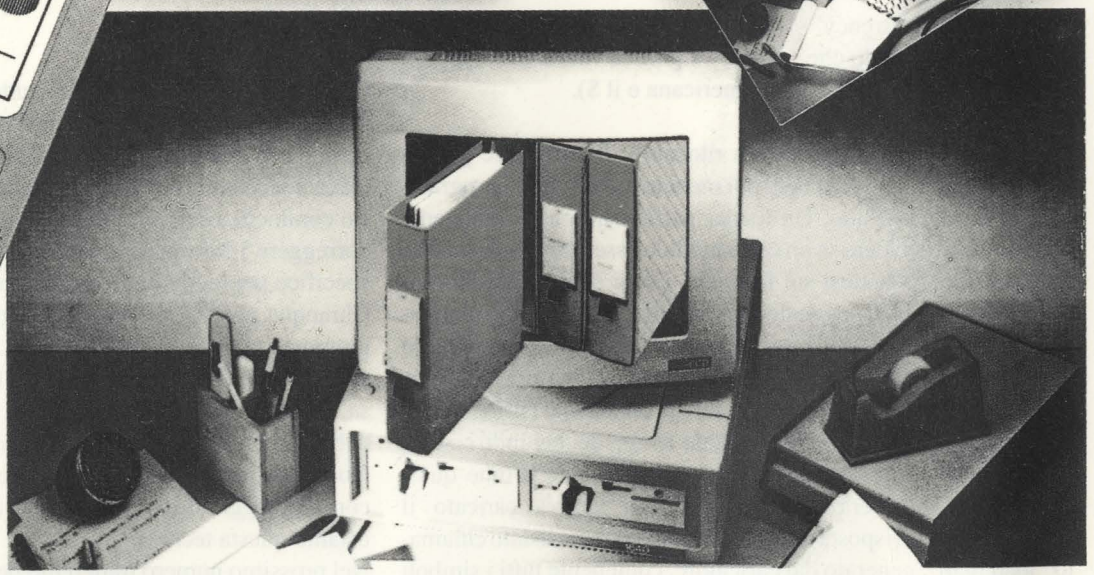
8 FASCICOLI CON FLOPPY DISK DA 3½"



IN EDICOLA LA RISTAMPA

8 FASCICOLI CON FLOPPY DISK DA 5¼"





GEJ0016

GRUPPO EDITORIALE JACKSON IN COLLABORAZIONE CON **Microsoft®**

Richiedi anche in EDICOLA **3½" MS-DOS 2 SOFTWARE** la prima e unica rivista dedicata ai possessori di PC (con dischetto) MS-DOS da 3½"

I Servizi
di

Amiga Transactor offre una serie di servizi per agevolare i propri lettori nel reperimento di software e materiale utile alla programmazione.
È disponibile l'intera libreria di dischetti di pubblico dominio curata da Fred Fish. Ogni dischetto contiene numerosi programmi e utility, spesso corredati da listati sorgenti e commenti degli autori. Ogni dischetto è fornito di una etichetta sulla quale sono riportati i titoli e, salvo rari casi, la descrizione dei programmi contenuti. Le etichette, per venire incontro alle esigenze di quanti possedessero già i Fish Disk, sono disponibili anche separatamente.
Per districarsi fra le centinaia di programmi disponibili nei dischi di Fred Fish, è stato creato un apposito catalogo di 20 pagine. Tale elenco riporta, divisi per categoria e in ordine alfabetico, tutti i programmi presenti, completandoli con informazioni quali la descrizione della funzione, l'autore, il numero di versione, la disponibilità del sorgente e il disco nel quale sono contenuti. I dischetti possono essere ordinati contrassegnando i numeri desiderati, purché la quantità sia un multiplo di cinque. Per ragioni amministrative saremo costretti a non accettare ordini che non soddisfino questa regola.
Tutto il materiale, a esclusione dei listati pubblicati sulla rivista, viene fornito nella versione originale americana, senza traduzione o modifiche.
A ogni numero della rivista corrisponde un disco chiamato "AmiTrans Disk" che contiene tutti i listati pubblicati su quel numero, nonché i corrispondenti eseguibili e tutti gli altri programmi di pubblico dominio menzionati negli articoli.
I "Transactor Classic Disk", fino ad oggi ne sono stati preparati due, raccolgono programmi selezionati appositamente per i programmatori.
Uedit è il text editor preferito dai programmatori, per le sue potenzialità e la sua facilità d'uso.

BUONO D'ORDINE

Completa il buono d'ordine (o una sua fotocopia) e spedire in busta chiusa a: I servizi di Transactor per Amiga - Via Rosellini, 12 - 20124 Milano

Desidero ricevere i seguenti articoli; contrassegnare con una X i numeri di Fish disk desiderati (a gruppi di 5)

Nota: I dischi 57, 80 e 88 non sono disponibili

☐ 1	☐ 11	☐ 21	☐ 31	☐ 41	☐ 51	☐ 62	☐ 72	☐ 83	☐ 94	☐ 104	☐ 114	☐ 124	☐ 134	☐ 144
☐ 2	☐ 12	☐ 22	☐ 32	☐ 42	☐ 52	☐ 63	☐ 73	☐ 84	☐ 95	☐ 105	☐ 115	☐ 125	☐ 135	☐ 145
☐ 3	☐ 13	☐ 23	☐ 33	☐ 43	☐ 53	☐ 64	☐ 74	☐ 85	☐ 96	☐ 106	☐ 116	☐ 126	☐ 136	☐ 146
☐ 4	☐ 14	☐ 24	☐ 34	☐ 44	☐ 54	☐ 65	☐ 75	☐ 86	☐ 97	☐ 107	☐ 117	☐ 127	☐ 137	
☐ 5	☐ 15	☐ 25	☐ 35	☐ 45	☐ 55	☐ 66	☐ 76	☐ 87	☐ 98	☐ 108	☐ 118	☐ 128	☐ 138	
☐ 6	☐ 16	☐ 26	☐ 36	☐ 46	☐ 56	☐ 67	☐ 77	☐ 89	☐ 99	☐ 109	☐ 119	☐ 129	☐ 139	
☐ 7	☐ 17	☐ 27	☐ 37	☐ 47	☐ 58	☐ 68	☐ 78	☐ 90	☐ 100	☐ 110	☐ 120	☐ 130	☐ 140	
☐ 8	☐ 18	☐ 28	☐ 38	☐ 48	☐ 59	☐ 69	☐ 79	☐ 91	☐ 101	☐ 111	☐ 121	☐ 131	☐ 141	
☐ 9	☐ 19	☐ 29	☐ 39	☐ 49	☐ 60	☐ 70	☐ 81	☐ 92	☐ 102	☐ 112	☐ 122	☐ 132	☐ 142	
☐ 10	☐ 20	☐ 30	☐ 40	☐ 50	☐ 61	☐ 71	☐ 82	☐ 93	☐ 103	☐ 113	☐ 123	☐ 133	☐ 143	

DESCRIZIONE

☐ Fish Disk	Lit. 35.000 per gruppo di cinque
☐ Etichette	Lit. 30.000 l'intera serie (fino al Fish 146)
☐ Catalogo	Lit. 15.000 aggiornato fino al Fish 138
☐ AmiTrans Disk #1	Lit. 15.000
☐ Transactor Classic Disk #1	Lit. 15.000
☐ Transactor Classic Disk #2	Lit. 15.000
☐ Uedit V2.4	Lit. 63.000

Tutti i prezzi sono da intendersi IVA inclusa e spese di spedizione comprese.

Cognome .....

Nome .....

Via .....

Cap. ..... Città .....

Prov. ..... Tel. .....

Firma.....

(se minorenne quella di un genitore)
Gli ordini non firmati non verranno evasi.

L'interfaccia a pacchetti dell'AmigaDOS, in C

di Matthew Dillon

Usare il DOS a livello di pacchetto fornisce nuove possibilità al programmatore

Matt Dillon è senior di Ingegneria Elettrotecnica a Berkeley. A cominciare dal suo primo microcomputer - un PET Commodore - ha lavorato su vari progetti software e hardware. Ha iniziato a lavorare su Amiga quasi due anni fa (ma la sua attenzione si appuntò su questa macchina non appena ne lesse le specifiche tecniche) ed è conosciuto nell'ambiente degli utenti di Amiga per la sua famosa "DOS Shell" e per il suo text editor "DME".

Il funzionamento interno dell'AmigaDOS è rimasto un mistero per molto tempo e la maggior parte delle discussioni tecniche su tale argomento sono state, fino a poco tempo fa, di competenza non del normale programmatore, ma di veri "Guru" della programmazione. Speriamo di sovvertire questo stato di cose presentando un articolo che non solo costituisca una buona introduzione al funzionamento dell'AmigaDOS, ma che comprenda anche alcune applicazioni pratiche in forma di programmi C. Questo articolo, in particolare, tratta l'interfacciamento con l'AmigaDOS dal punto di vista del programma applicativo e non esaurisce assolutamente l'argomento.

Prima vediamo qualche nota generale. L'AmigaDOS risulta piuttosto strano in quanto è stato scritto nel linguaggio BCPL. Benché si possa parlare molto bene del BCPL per certi versi, rimane il fatto che non si integra molto bene con il resto del sistema operativo di Amiga. Più specificamente, i puntatori, le stringhe e le librerie sono trattate in maniera differente rispetto al linguaggio C. Un puntatore BCPL (BPTR) è sempre allineato sulla longword; in effetti non c'è scelta, perché i puntatori BCPL sono dati dall'indirizzo effettivo diviso 4. Le seguenti macro possono essere impiegate per convertire i puntatori C in BPTR e viceversa:

```
#define BTOC(bptr)    ((long)(bptr)<<2)
#define CTOB(cptr)    ((long)(cptr)>>2)
```

Le stringhe BCPL (BSTR) non assomigliano per nulla alle stringhe C. Il primo carattere della stringa è la sua lunghezza: un valore unsigned da 0 a 255. I successivi N bytes costituiscono la stringa vera e propria, la cui fine non è delimitata da alcun carattere speciale. Le stringhe BCPL sono pertanto limitate a una lunghezza massima di 255 caratteri. In molti casi si parla di BPTR a BSTR (ovvero di puntatori BCPL a stringhe BCPL).

Le librerie BCPL sono poi la cosa peggiore; in effetti, costituiscono ancora una zona per lo più inesplorata. Fortunatamente non ce ne occuperemo affatto. E' sufficiente osservare che la dos.library (l'interfaccia C al BCPL dell'AmigaDOS) deve compiere operazioni piuttosto strane per "appiccicare" l'AmigaDOS al resto del sistema operativo.

Nonostante l'AmigaDOS sia, per ammissione comune, una specie di schifezza, i concetti su cui si basa sono validi e molto flessibili. L'aspetto più importante dell'AmigaDOS è che ogni device driver (come CON:, DF0: e RAM:) è costituito da un processo separato. Le applicazioni e il DOS comunicano usando (sorpresa!) le message port messe a disposizione dall'Exec, per inviare pacchetti avanti e indietro. La comunicazione sembra sincrona solo perchè l'interfaccia presente nella dos.library, che nasconde tutte queste operazioni, dopo avere creato e inviato un pacchetto per nostro conto, aspetta che sia restituito dal device driver.

Se inviamo un pacchetto noi stessi, invece di lasciare che il DOS si occupi di queste quisquilie, potremo fare cose che non risulterebbero possibili usando le routine del DOS di più alto livello. In questo articolo vengono proposti due esempi: scrittura asincrona, in cui vengono trasferiti dei dati a un file mentre il programma continua l'esecuzione del codice e l'ottenimento del puntatore alla finestra utilizzata come console da un processo.

La procedura generale per inviare un pacchetto è molto semplice:

(A) Trovare l'handler a cui si intende inviare il pacchetto

(B) Costruire un pacchetto DOS

(C) Specificare il tipo del pacchetto e riempire anche gli altri campi

(D) Inviare il pacchetto al processo che costituisce l'handler

(E) Attendere che venga restituito il pacchetto

(F) Esaminare il valore restituito nel campo dp_Res1

(A) Gli handler sono generalmente individuati dal puntatore alla message port del loro processo. I campi pr_ConsoleTask e pr_FileSystemTask presenti nella struttura del processo puntano alle message port rispettivamente dell'handler della console e dell'handler del file system. I membri pr_CIS e pr_COS sono BPTR a strutture di tipo FileHandle. Il campo fh_Type nella struttura FileHandle punta alla message port del processo che controlla quel file handle (struttura che descrive un file). La funzione DeviceProc() presente nella dos.library restituisce un

puntatore alla message port del processo associato al device specificato come argomento. Ad esempio:

```
struct MsgPort *port = DeviceProc("DF0:");
```

Si noti che non è possibile ottenere un puntatore a un handler per device come CON: o RAW: tramite DeviceProc(), perchè possono essere presenti diversi processi CON: o RAW: in uno stesso momento.

(B) Si vedano gli esempi riportati oltre. La costruzione è estremamente semplice, ma non standard.

(C) Si vedano, anche per questo aspetto, gli esempi che seguono. Il campo dp_Type contiene un valore che corrisponde al tipo del pacchetto (ACTION_WRITE, ACTION_INHIBIT, e così via). I campi dp_Arg1,2,3..7 contengono gli argomenti relativi al tipo del pacchetto. Ad esempio, ACTION_READ e ACTION_WRITE richiedono tre argomenti:

dp_Arg1 : il campo fh_Arg1 nel descrittore del file (File Handle)
dp_Arg2 : un puntatore al buffer (senza limitazioni di allineamento)
dp_Arg3 : la dimensione del buffer in byte

ACTION_INHIBIT viene in genere mandato a un "DFx:" e richiede un argomento che impedisca o permetta al device driver l'accesso al disco:

dp_Arg1 : DOS_TRUE (-1L) o DOS_FALSE (0L)

Ci sono circa 25 tipi di pacchetti supportati, ma occorre osservare che non tutti i device driver li supportano al completo.

(D) L'invio del pacchetto è realizzato per mezzo di PutMsg() che rende accessibile il pacchetto alla message port (E) dell'handler; molto banale, in effetti. Se si intende compiere tale operazione in maniera sincrona (cioè attendendo che il pacchetto venga rimandato indietro), si può usare la message port del proprio processo come port di risposta. Diversamente va specificata una port privata come port di risposta.

(F) Il risultato è contenuto nel campo dp_Res1 del pacchetto. Nella maggior parte dei casi un DOS_FALSE (0) significa che ha avuto luogo un errore. Se effettivamente si è verificato un errore, il suo codice è disponibile nel campo dp_Res2 del pacchetto.

Esempio #1

Questo esempio dimostra la possibilità di compiere operazioni di scrittura asincrone con un qualunque descrittore di file. In altre parole, si invia un pacchetto di tipo WRITE al processo che si occupa del FileHandle in questione, senza attendere che venga restituito. L'interfaccia con i messaggi non è complicata, ma richiede una preparazione un po' contorta della struttura associata al messaggio, operazione che comunque è mostrata in questo esempio. Ci sono poi delle limitazioni intrinseche del DOS di cui bisogna tener conto:

(1) Dobbiamo usare una port di risposta privata, in modo tale da non confondere il DOS
(2) Non si può avere più di un pacchetto, relativo a uno stesso descrittore di file, che attenda la risposta pena un crash del sistema

Quest'ultima restrizione non è un bug del DOS, ma solo una sua limitazione. In effetti questo significa che se stiamo scrivendo in modalità asincrona su un certo descrittore di file, non possiamo fare altre chiamate al DOS concernenti quel descrittore di file.

Si ricordi che il buffer impiegato in una chiamata a WriteAsync() non dovrebbe essere modificato se non dopo una successiva chiamata a WriteAsync(). Si ha dunque bisogno di due buffer perchè si deve effettivamente operare un double-buffering per le operazioni di scrittura: si scrive nel buffer che non sta per essere trasferito su disco, quindi si invia quel buffer e si scrive nell'altro. Il valore di ritorno non è disponibile fino alla prossima chiamata a WriteAsync(), mentre EndAsync() restituisce il valore di ritorno relativo alla scrittura che ha avuto luogo con l'ultima chiamata a WriteAsync().

```
/*
 * ESEMPIO1.C      SCRITTURA ASINCRONA
 *
 * Questo programma fornisce un esempio di scrittura asincrona
 * usando i
 * pacchetti: scrive su un file e contemporaneamente stampa un
 * messaggio su
 * schermo.
 *
 * NOTA: In questo esempio si assume che gli interi siano lunghi
32 bits.
 *      Il programma va lanciato da CLI o da una shell.
 */


#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>

#define BTOC(bptr)    ((long)(bptr) << 2)
#define CTOB(cptr)    ((long)(cptr) >> 2)

typedef struct FileHandle     FH;
typedef struct StandardPacket STDPKT;
typedef struct MsgPort        MSGPORT;
typedef struct Task           TASK;

extern BPTR Open();
extern TASK *FindTask();
extern char *AllocMem();

static char Buf1[8192];
static char Buf2[4096];
```

```
main()
{
   BPTR fh = Open("DF0:test", MODE_NEWFILE);
   /* Utilizziamo DF0: */
   short i;
   long res;

   if (!fh)
   {
      puts("Non posso aprire DF0:test");
      exit(20);
   }
   InitAsync(fh);
            /* metti tutto nel Buf1 */
   WriteAsync(Buf1, sizeof(Buf1));
   printf("WriteAsync(buf,%ld) #1 ha terminato\n",
   sizeof(Buf1));
            /* metti tutto nel Buf2 */
   for (i = 0; i < 10; ++i)
   {
      puts("Guarda mamma... sto scrivendo e parlando contem-
poraneamente.");
      Delay(10);
   }
   res = WriteAsync(Buf2, sizeof(Buf2));
      printf("WriteAsync(buf,%ld) #2 ha terminato; risultato
%ld\n",
         sizeof(Buf2), res);
   res = EndAsync();
   printf("EndAsync() ha terminato; risultato %ld\n", res);
   Close(fh);
}

/*
 * Routine per l'interazione asincrona col DOS. Le routine in
questo esempio
 * possono trattare solamente un unico descrittore di file in
modalità
 * asincrona. Le operazioni da compiere sono le seguenti:
 *
 * (1) Inizializzare il descrittore di file con InitAsync(fh)
 * (2) Chiamare il numero desiderato di volte
   WriteAsync(buf,bytes)
 * (3) Rilasciare il descrittore di file con EndAsync()
 *
 * Si badi che non vanno fatte altre chiamate al DOS
*concernenti lo stesso
 * descrittore di file durante tali operazioni.
 *
 * WriteAsync() restituisce il valore di ritorno relativo
*al precedente
 * pacchetto di scrittura. Può essere inviato un NULL
*come buffer per far
 * sì che la operazione di scrittura in corso venga completata.
 * EndAsync() restituisce il valore di ritorno dell'ultimo
*pacchetto di scrittura.
 *
 * Si osservi che WriteAsync() ferma l'esecuzione
 *esclusivamente quando la
 * precedente operazione di scrittura non ha ancora avuto termine.
 */

static FH     *Fh;       /* Il file handle              */
static STDPKT *Packet;   /* standard DOS message & packet
*/
                         /* (DEVE essere allineato su longword) */
static MSGPORT *RPort;   /* standard EXEC message port  */

static char Pending;     /* flag di 'operazione in atto'  */

InitAsync(fh)
BPTR fh;
{
   Fh = (FH *)BTOC(fh);
      Packet = (STDPKT *)AllocMem(sizeof(STDPKT),
MEMF_CLEAR | MEMF_PUBLIC);
   RPort  = (MSGPORT *)AllocMem(sizeof(MSGPORT),
MEMF_CLEAR | MEMF_PUBLIC);
   RPort->mp_Node.ln_Type = NT_MSGPORT;
   RPort->mp_Node.ln_Name = "Async";
   RPort->mp_Flags  = PA_SIGNAL;
   RPort->mp_SigBit  = AllocSignal(-1);
   RPort->mp_SigTask = FindTask(NULL);
   NewList(&RPort->mp_MsgList);
}

WriteAsync(buf, bytes)
char *buf;
int bytes;
{
long result;
   if (Pending)
   {
      WaitPort(RPort);  /* attende la restituzione del packet */
      GetMsg(RPort);    /* rimuove il packet dalla porta      */
      Pending = 0;
   }
   result = Packet->sp_Pkt.dp_Res1;
   if (buf)
   {  /* Il DOS richiede che venga posto un puntatore al
       * pacchetto nel campo ln_Name del messaggio e un
       * puntatore al messaggio nel campo dp_Link del
       * pacchetto.
       */
      Packet->sp_Msg.mn_Node.ln_Name = (char *)&(Packet-
      >sp_Pkt);
      Packet->sp_Pkt.dp_Link = &Packet->sp_Msg;
      Packet->sp_Pkt.dp_Port = RPort;
      Packet->sp_Pkt.dp_Type = ACTION_WRITE;
      Packet->sp_Pkt.dp_Arg1 = Fh->fh_Arg1;
      Packet->sp_Pkt.dp_Arg2 = (long)buf;
      Packet->sp_Pkt.dp_Arg3 = bytes;
      PutMsg (Fh->fh_Type, Packet);
      Pending = 1;
   }
   return(result);
```

```
}

EndAsync ()
{
  long result;
  if (Pending)
  {
    WaitPort(RPort);
    GetMsg(RPort);
    Pending = 0;
  }
  result = Packet->sp_Pkt.dp_Res1;

  FreeSignal(RPort->mp_SigBit);
  FreeMem(RPort, sizeof(MSGPORT));
  FreeMem(Packet,sizeof(STDPKT));
  return(result);
}
```

Esempio #2

Il secondo esempio dimostra come estrarre il puntatore alla Window di Intuition utilizzata da un certo device come console, inviando un pacchetto speciale a quel device.

```
/*
 *  ESEMPIO2.C  LEGGE IL PUNTATORE A UNA
 * FINESTRA "CON:" TRAMITE IL
 *          PACCHETTO DISK_INFO
 *
 * NOTA: In questo esempio si assume che gli interi siano lunghi
32 bits.
 *      Il programma va lanciato da CLI o da una shell.
 *
 * La funzione principale chiama GetWindow() per avere il
puntatore alla
 * finestra utilizzata come console dal processo corrente, poi fa
alcune
 * cose divertenti con quella finestra.
 *
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>

#define BTOC(bptr)    ((long)(bptr) << 2)
#define CTOB(cptr)    ((long)(cptr) >> 2)

typedef struct Task                TASK;
typedef struct Process        PROC;
typedef struct StandardPacket      STDPKT;
typedef struct MsgPort          MSGPORT;
typedef struct Window           WIN;
typedef struct InfoData         INFODATA;

extern TASK *FindTask();
extern char *AllocMem();
extern WIN  *GetWindow();
extern long  OpenLibrary();
long IntuitionBase, GfxBase;
main()
{
  WIN *win = GetWindow();    /*  il cuore      */
  int i, j, bt, bb, bl, br;   /*  variabili varie */

  /*
   *  Divertiamoci con la finestra
   */

  if (!win)
  {
    puts("Niente Window!");
    exit(1);
  }
  IntuitionBase = OpenLibrary("intuition.library", 0);
  GfxBase = OpenLibrary("graphics.library", 0);
  if (IntuitionBase && GfxBase)
  {
    printf("WINDOW=%08lx. Width=%ld Height=%ld\n",
        win, win->Width, win->Height);
    bt = win->BorderTop;
    bb = win->BorderBottom;
    bl = win->BorderLeft;
    br = win->BorderRight;

    for (i = bl; i < win->Width - br; i += 16)
    {
          j = win->Height-bb - i*(win->Height-bt-bb)/(win-
>Width-br);
      Move(win->RPort, i, bt);
      Draw(win->RPort, bl, j);
    }
  }
  else
    puts("Non riesco ad aprire le librerie");

  if (IntuitionBase)
    CloseLibrary(IntuitionBase);
  if (GfxBase)
    CloseLibrary(GfxBase);
}

/*
 * GETWINDOW()
 *
 * Restituisce il puntatore alla finestra usata dalla
 * console del processo
 * corrente. Possiamo usare il message port del
 * nostro processo come port di
 * risposta poiché si tratta di un pacchetto
 * sincrono (cioè aspettiamo che
 * ci torni indietro il risultato).
 * ATTENZIONE: questa routine non controlla
 * che la console del processo
```

```
 * corrente sia realmente un 'console device' vero e proprio.
 *
 * Viene inviato un pacchetto di tipo DISK_INFO
 * al 'console device'.
 * Benchè questo tipo di pacchetto venga solitamente
 * impiegato per ottenere
 * informazioni relative ai dischi e quindi
 * normalmente inviato a 'disk
 * devices', esso viene riconosciuto dal 'console device'
 * che in risposta
 * pone un puntatore alla finestra nel campo
 * id_VolumeNode della struttura
 * InfoData. Viene anche posto un puntatore alla unità
 * di console nel campo
 * id_InUse della struttura InfoData.
 */

WIN *
GetWindow()
{
PROC    *proc;
STDPKT  *packet;
INFODATA *infodata;
long    result;
WIN     *win;
   proc = (PROC *)FindTask(NULL);
   if (!proc->pr_ConsoleTask)
      return(NULL);
   /* NOTA: Dal momento che il DOS richiede che
    * sia il pacchetto
    * sia la struttura InfoData siano allineati sulla longword,
    * non è possibile allocarli come variabili globali o sullo
    * stack (che è invece allineato sulla word). AllocMem()
    * restituisce sempre puntatori a blocchi di memoria allineati
    * sulla longword.
    */

   packet  = (STDPKT  *)AllocMem(sizeof(STDPKT),
         MEMF_CLEAR | MEMF_PUBLIC);
   infodata = (INFODATA *)AllocMem(sizeof(INFODATA),
         MEMF_CLEAR | MEMF_PUBLIC);
   packet->sp_Msg.mn_Node.ln_Name =
   (char *)&(packet->sp_Pkt);
   packet->sp_Pkt.dp_Link = &packet->sp_Msg;
   packet->sp_Pkt.dp_Port = &proc->pr_MsgPort;
   packet->sp_Pkt.dp_Type = ACTION_DISK_INFO;
   packet->sp_Pkt.dp_Arg1 = CTOB(infodata);
   PutMsg(proc->pr_ConsoleTask, packet);
   WaitPort(&proc->pr_MsgPort);
   GetMsg(&proc->pr_MsgPort);
   result = packet->sp_Pkt.dp_Res1;
   win = (WIN *)infodata->id_VolumeNode;
   /* nota: anche id_InUse contiene un puntatore alla console *
   FreeMem(packet , sizeof(STDPKT));
   FreeMem(infodata, sizeof(INFODATA));
   if (!result)
      return(NULL);
   return(win);
}
```

(segue da pag. 54)

E di cosa...

aiuta Amiga a trovare la tabella di kerning. Era l'ultima cosa prima dell'inizio della mappa a bit dei caratteri. Nella definizione in linguaggio C della struttura TextFont questo puntatore è chiamato "tf_CharKern".
Utilizzando un font editor possiamo osservare che i sei caratteri all'inizio di Garnet/16 hanno un valore di kerning uguale a: zero per lo spazio, uno per '!', zero per il doppio apice, zero per '#', uno per '$', ed ancora zero per '%' e per '&'.
Dal momento che le indicazioni per la spaziatura richiedono due byte per ogni carattere, dobbiamo muoverci avanti nel file di 223 * 2 = 446 byte per trovare la tabella di kerning alla linea $13D0, cominciando dalla terza word:

```
13D0: 00090000 00000001 00000000 00010000

13E0: 00000002 00010001 00000001 00010001
```

Le prime due word in $13D0 e $13D2 completano la tabella di spaziatura, con nove bit per il carattere $FF (255) e 0 bit per il carattere fittizio che verrà stampato se l'utente chiede un carattere che non è presente in questo font. A $13D4 troviamo la tabella di kerning che comincia con zero per lo spazio, uno per '!', zero per il doppio apice, zero per '#', uno per '$', zero per '%' e zero per '&'. Il carattere successivo ha due bit di kerning e così via fino alla fine del font. Quando il sistema operativo visualizza un carattere deve seguire una procedura di questo tipo:
1) Dato il codice del carattere (ad esempio 65 per 'A), guarda nella tabella di kerning il numero di pixel da lasciare in bianco prima di iniziare a stampare.
2) Adesso guarda nella tabella CharLoc per avere l'offset, all'interno della mappa a bit, per la prima riga di pixel di quel particolare carattere e per saperne la larghezza in pixel (la massima larghezza visibile del carattere).
3) Trasferisce i pixel dalla mappa a bit sullo schermo usando il blitter (per ogni bit-plane nello schermo).
4) Esamina la tabella di spaziatura per sapere la larghezza totale del carattere (la parte visibile più lo spazio su entrambi i lati) e sposta il cursore orizzontalmente di quella quantità.

Hunk_end

Lo studio è concluso. Un font di Amiga consiste in un file descrittivo chiamato "nome_del_font.font", di una directory con lo stesso nome e di uno o più font file chiamati come la loro altezza in pixel. Questi ultimi contengono una struttura node, una struttura messagge, un gruppo di puntatori a varie parti del file, la descrizione dettagliata di ciascun carattere sotto forma di mappa a bit, una tabella CharLoc, una tabella di spaziatura e una tabella di kerning. Tutto questo è seguito da una breve sezione di informazioni utilizzate internamente dal sistema operativo, inclusa la tabella di rilocazione per i sei riferimenti rilocabili contenuti nel file (due riferimenti al nome e uno per ognuna delle quattro tabelle). Il file termina con il necessario "hunk_end" $3F2. Per essere ancora più precisi, un font è una serie di tabelle, contenenti zero e uno, che dicono al computer quali pixel accendere, quali lasciare spenti e dove metterli. E questo é quello di cui sono fatti i piccoli font.

Rappresentazione di oggetti in un sistema CAD

di Charles B. Blish

Una struttura dati per semplificare la progettazione CAD

Charles Blish è presidente e fondatore della SoftCircuits Inc., una società di sviluppo software per Amiga molto attiva in Florida. Tra i più interessanti prodotti che la SoftCircuits ha realizzato per Amiga si possono trovare: i package di CAD elettronico PCLO e PCLOplus trattati in questo articolo, Scheme, un programma di supporto al disegno di schemi elettronici, e il driver software fornito con gli hard disk della C Ltd.

In questo articolo sono trattati i metodi alternativi di manutenzione di un database ad alte prestazioni per sistemi CAD. E' auspicabile che queste informazioni tornino utili nella creazione di altri tipi di sistemi CAD specializzati, oltre a quelli che realizzano circuiti stampati (PCB).

Tipicamente, un sistema CAD deve essere di uso generale: vengono sfruttati pochi tipi di strutture che possono essere ingranditi e spostati in modo del tutto indipendente fra loro. Tipici esempi sono linee, cerchi, archi e punti. Spesso queste semplici oggetti rappresentano tutto ciò che viene messo a disposizione da un sistema CAD. Gli elementi complessi sono creati partendo da elementi più semplici con dimensioni diverse e posizionati in aree specifiche. Per disegnare un'area del progetto realizzata con questi elementi, occorre innanzitutto stabilire quali di questi elementi sono presenti in toto o in parte nell'area in questione. In seguito ogni elemento che non rientra interamente nell'area selezionata deve essere "tagliato", per mezzo di operazioni che impiegano molto tempo di elaborazione, nel punto in cui esce dallo spazio visibile dello schermo.

Questo metodo di rappresentazione, seppure semplice, comporta, oltre al tempo di elaborazione, altri inconvenienti. Mentre è abbastanza semplice mantenere un archivio di elementi base, un oggetto complesso può richiedere un gran numero di questi elementi per essere rappresentato accuratamente: conseguenze immediate di tutto ciò sono un notevole e variabile impiego della memoria necessaria alla rappresentazione di un oggetto, sequenze di disegno direttamente legate alla sua complessità e tempi di visualizzazione variabili in rapporto alla quantità di dati da rappresentare. Si possono riscontrare anche effetti sul metodo con il quale il programma può gestire i dati; ciò però non interessa direttamente l'utente ma il programmatore, che deve predisporre il CAD affinché abbia un metodo per la gestione di questi elementi.

Ciò che viene trattato in questo articolo è una metodologia di sviluppo in grado di rimuovere questi inconvenienti. L'applicazione particolare, in questo caso, è un programma per l'incisione di circuiti stampati, un'applicazione in cui linee, cerchi ed elementi base disegnati ad hoc, vengono tutti utilizzati: non è una implementazione puramente teorica, in quanto il metodo è utilizzato dal PCLO, un prodotto disponibile su Amiga sin dall'Aprile '86.

Nella fase di progettazione di un circuito stampato la capacità di elaborazione del sistema CAD (e in generale la sua velocità), è di primaria importanza. Dal momento che la progettazione di un circuito è un lavoro particolarmente complesso che richiede una continua attenzione da parte del progettista, le lunghe attese o altri inconvenienti imposti dal programma risultano estremamente negativi e influiscono direttamente sulla qualità del lavoro; per questa ragione le prestazioni sono l'obiettivo principale.

Nel corso dell'articolo, il termine "elemento" è riferito a una qualsiasi entità, come ad esempio una pista o un contatto sul circuito, mentre il termine "oggetto" verrà utilizzato per indicare un blocco grafico che da solo o insieme ad altri oggetti formerà poi un elemento.

Invece di creare elementi da primitive grafiche come linee o cerchi, è preferibile costruirli sfruttando una piccola serie di oggetti aventi le caratteristiche necessarie per rappresentare l'elemento da creare. Tutto ciò può sembrare complesso, ma in realtà le operazioni da svolgere sono molto più semplici. E' sufficiente pensare a questi elementi come alle componenti di un puzzle: ogni pezzo deve essere messo accanto ad altri per formare un tutt'uno uniforme, nel nostro caso un elemento nel database.

Invece di usare una quantità di memoria variabile per ogni oggetto, è auspicabile creare uno spazio di lavoro che consista in un array bidimensionale di celle di grandezza specifica, ognuna contenente sempre uno degli oggetti disponibili: inizialmente, ogni cella contiene un oggetto che rappresenta la condizione di "cella vuota".

Per far ciò, occorre che gli oggetti rispondano a particolari caratteristiche:

1 Devono essere particolarmente semplici da disegnare
2 Devono essere di facile memorizzazione (un byte può essere una
dimensione ideale)

3 Devono essere contenuti in una serie abbastanza limitata
4 Devono essere in grado di descrivere qualsiasi elemento si intenda creare
5 Devono essere facilmente gestibili

L'esperienza maturata nel settore ci ha portato a creare un set di 18 oggetti, illustrati in figura 1.

Questi oggetti rispondono in modo particolare alla prima caratteristica sopracitata. Ogni oggetto è rappresentato entro un area di 4x4 pixel. Gli oggetti sono disegnati solo se appaiono completamente sullo schermo, quindi le routine grafiche del programma devono solo stabilire dove disegnarli, e se inserirli nel nybble destro o sinistro del byte di destinazione sullo schermo. Ridisegnare lo schermo significa quindi esaminare tutte le locazioni che sono all'interno di una determinata finestra e tracciare sullo schermo l'immagine relativa a quella precisa area. Con uno schermo di 320x200 pixel, il lavoro viene completato quasi istantaneamente; non importa quali dati siano contenuti nell'area da visualizzare.

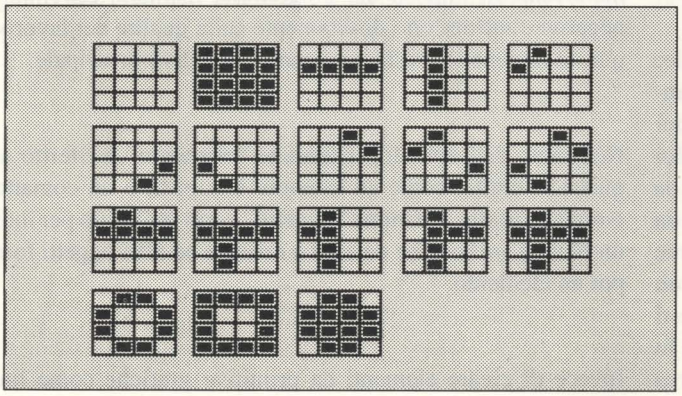


Figura 1: Il set di oggetti che formano il sistema costruttivo di PCLO. Questi oggetti possono essere liberamente combinati per costruire elementi finali. (vedi figura 2).

La figura 2 presenta un esempio di come si possano accostare gli oggetti per creare un determinato elemento del circuito. Questo diagramma presenta tre contatti connessi da una singola pista.

Con il metodo illustrato sono soddisfatte anche le caratteristiche 2 e 3, essendoci meno di 256 oggetti ed essendo quindi possibile rappresentarli con un singolo byte. Adesso possiamo stabilire lo spazio di lavoro, basandoci sulle dimensioni di questi oggetti. 256 locazioni lungo ogni parte della area di lavoro lasciano sufficiente spazio per la maggior parte delle realizzazioni di circuiti stampati, fino ad aree di 15 centimetri (12,8") di lato. Si useranno quindi 65535 byte per i dati di una superficie. Poiché è la maggior parte dei progetti richiede due superfici, occorreranno 128 kbyte in totale, e questa quantità di memoria non è certo un problema per l'Amiga.

Presupponendo che siano sempre disponibili due superfici per rappresentare gli oggetti, è possibile contenere l'intero set in 4 bit, salvando esattamente la metà della memoria richiesta, ovvero 64kbyte. Il metodo è abbastanza semplice: se si tiene presente che i contatti sono presenti su ambedue le superfici, il valore nella cella della superficie superiore può specificare uno dei 15 oggetti di tipo superficiale oppure può indicare la presenza di un contatto. A questo punto, se la prima superficie indica la presenza di un contatto, la cella corrispondente sulla seconda superficie indica il tipo del contatto stesso (solido, con foro, quadrato ecc.), altrimenti specifica un altro oggetto di tipo superficiale. Si è scelto però di non considerare questa possibilità per ragioni di prestazioni: è molto più facile, e perciò più veloce, analizzare una cella che descriva completamente sè stessa senza riferirsi a un'altra. Inoltre l'analisi di celle impacchettate in nybble anzichè in byte, comporta un tempo di elaborazione superiore.
La quarta caratteristica consiste nel potere creare tutti gli elementi di cui si ha bisogno. La definizione dei nostri oggetti è adatta per la maggior parte dei circuiti stampati. Per una piccola percentuale, tuttavia, questo metodo non offre una definizione sufficiente.

La SoftCircuit produce anche un sistema CAD più sofisticato per

la realizzazione di circuiti, il PCLOplus. Quest'ultimo usa lo stesso metodo del PCLO, ma al posto di rappresentare una singola superficie con celle da mezzo pollice quadrato, usa celle da un quarto di pollice. Ciò permette a PCLOplus di avere una definizione quattro volte superiore al PCLO originale. Le caratteristiche di PCLOplus permettono di effettuare operazioni di per sè molto complesse, come ad esempio l'inserimento di due conduttori fra contatti adiacenti su un circuito integrato, chiamato in gergo industriale, "fine-line".

Sfruttando questo metodo, si utilizzano oggetti o combinazioni di oggetti per descrivere l'elemento: la sua precisione e il suo posizionamento sono solo una funzione delle routine finali di disegno e stampa. Noi sappiamo che esiste un elemento in una cella di mezzo pollice quadrato; sono le routine finali del programma che devono pensare a ricalcolare il disegno e a rappresentare il circuito in maniera precisa, perfettamente riproducibile e funzionante.

Questa operazione comprende il posizionamento corretto delle piste, rappresentate da codici nelle celle, il rimodellamento di aree di conduttore solido cosicché non vi siano angolature troppo sottili nel lavoro finale, e il riempimento della parte interna degli angoli troppo appuntiti.

Per ciò che concerne il punto 5, cioè la manipolazione degli oggetti, è possibile, per come è concepito, utilizzare metodi già esistenti o crearne di nuovi per eseguire qualsiasi tipo di trasformazione o analisi del circuito costruito con questi oggetti. Dal canto suo, PCLO implementa operazioni di spostamento, copia, rotazione, inserimento, tracciamento ed altri ancora.
La manipolazione di una pista è più semplice quando il circuito stampato è definito come una griglia di celle come quella trattata sinora.

Nel programma PCLO sono definite alcune semplici regole che permettono di determinare il metodo di rappresentazione di un particolare elemento.
Innanzitutto, gli oggetti che costituiscono un segmento di pista sono connessi quando combaciano, in verticale, orizzontale o obliquo, pixel per pixel. In secondo luogo, gli oggetti di tipo solido e quelli che realizzano i contatti sono connessi solo quando sono adiacenti l'un l'altro orizzontalmente o verticalmente, ma non in obliquo.
Le aree solide adiacenti tra loro diagonalmente non sono nella realtà connesse e sono rielaborate nel disegno finale in modo tale da garantire che non ci sia alcun contatto.

E' possibile utilizzare questo metodo per descrivere sistemi idraulici o qualsiasi altro sistema che possa essere rappresentato in un'area bidimensionale, ammesso che possa essere descritto da un limitato numero di oggetti. Si spera quindi che queste informazioni abbiano aiutato a comprendere meglio la rappresentazione di oggetti CAD e l'implementazione di un programma di CAD elettronico, e che ciò possa essere utilizzabile nel caso qualcuno decida di sviluppare un altro sistema CAD.

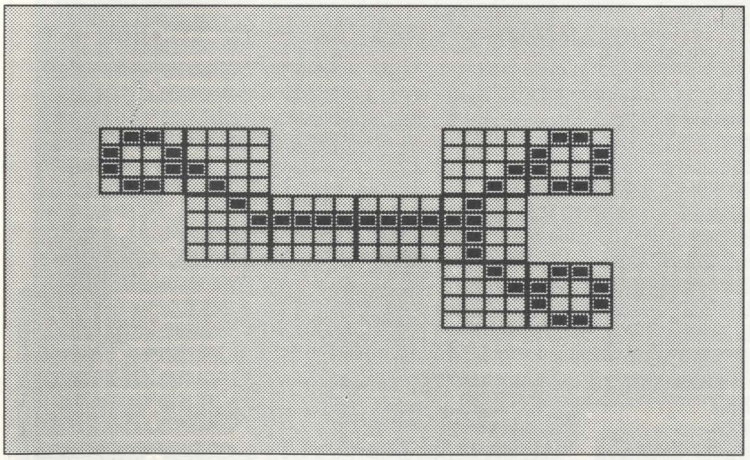
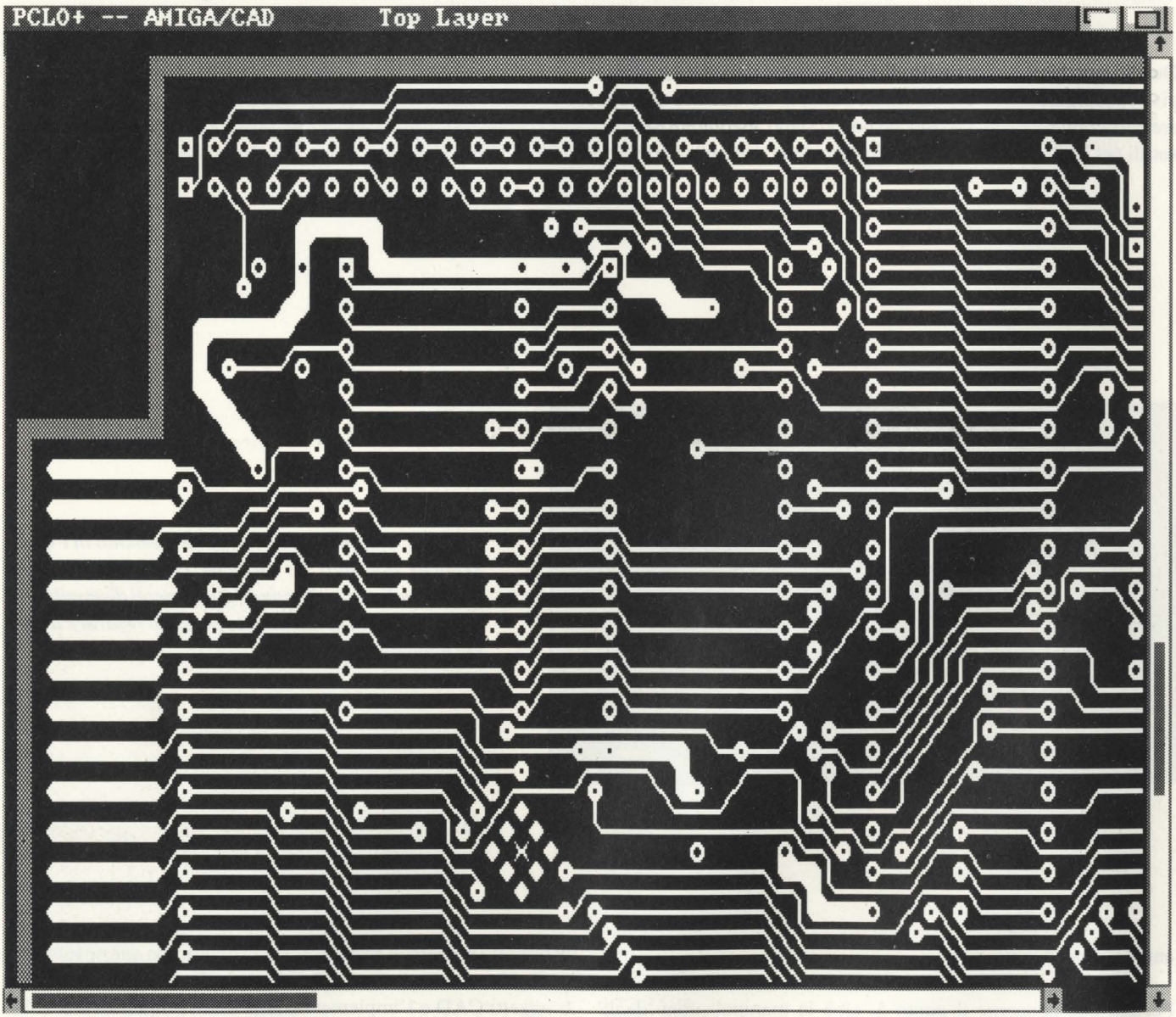


Figura 2: Rappresentazione di un elemento in PCLO. Sebbene non tutti gli elementi concepibili possono essere realizzati utilizzando gli elementi disponibili, le notevoli prestazioni ottenute possono riparare a questa mancanza.

Figura 3: Immagine IFF della disposizione su circuito stampato creata con PCLOplus, un CAD della SoftCircuit Inc. I colori nella rappresentazione originale 640x400, sono stati cambiati come segue: da giallo a bianco, da rosso a grigio chiaro, da verde a grigio scuro e con nero invariato.



PER IL TUO COMPUTER

A. Bigiarini - P. Cecioni - M. Ottolini

IL MANUALE DI AMIGA

Rivolto soprattutto ai programmatori, per saperne di più e conoscere meglio i tre modelli di Amiga e le loro ampie possibilità. Poichè vengono presentate le differenze fra i tre modelli disponibili della macchina, il libro risulta utile anche come una funzionale guida all'acquisto.

SOMMARIO

Caratteristiche generali - Grafica - Sprite - Coprocessori - Audio - Interfacciamento - Chip 8520 - Compatibilità IBM - Rom Kemel - Amiga DOS 1.1 e 1.2 - Registri dei Chip Custom - SuperDOS - ARC - SNOOP 1.0.

244 pagine Cod. CZ532 L. 39.000

R. Bonelli - M. Lunelli

AMIGA 500

GUIDA PER UTENTE

Finalmente un testo in grado di racchiudere in un'unica guida tutte le informazioni necessarie agli utenti di Amiga 500, in modo che possano comprendere tutte le possibilità del loro sistema e utilizzarlo al meglio.

SOMMARIO

Uso del mouse - Uso dei menu - Programmi del disco Workbench - Programmi del disco extras - Amiga Dos - Amiga Basic - Il Basic compilato: AC BASIC - Il True Basic.

370 pagine Cod. CC627 L. 55.000

M. England - D. Lawrence

AMIGA HANDBOOK

Un libro per conoscere l'Amiga, il nuovo computer della COMMODORE, al fine di comprendere e sfruttare al massimo tutte le potenzialità di questo sistema considerato da molti rivoluzionario.

SOMMARIO

Uno sguardo all'Amiga - Chip 68000 - Copper co-processor - Playfield e sprite - Blitter - Comunicazioni con il mondo esterno - Nucleo e Exec - Sistema operativo - Workbench e le tecniche di intuition - DOS e Command line interface - Programmi in BASIC.

204 pagine Cod. CC320 L. 35.000

RITAGLIATE E SPEDITE IN BUSTA CHIUSA

GRUPPO EDITORIALE JACKSON
Via Rosellini, 12 - 20124 MILANO

INDICARE CHIARAMENTE CODICI E QUANTITÀ DEI VOLUMI RICHIESTI

Codice	Q.tà	Codice	Q.tà	Codice	Q.tà	Codice	Q.tà

L. 4.000 per contributo fisso spese di spedizione

MODALITÀ DI PAGAMENTO

☐ Allego assegno n. _____ di L. _____ della Banca _____

☐ Ho effettuato il pagamento di L. _____ a mezzo: ☐ vaglia postale ☐ vaglia telegrafico ☐ versamento sul c/c postale n. 11666203 intestato a Gruppo Editoriale Jackson SpA Milano e allego fotocopia della ricevuta.

☐ Pagherò al postino l'importo di L. _____ al ricevimento dell'opera.

☐ Richiedo l'emissione della fattura (formula riservata alle aziende) e comunico il numero di Partita IVA _____

DATA _____ FIRMA _____

COGNOME _____

NOME _____

VIA E NUMERO _____

CITTÀ _____

CAP _____ PROV. _____

GRUPPO EDITORIALE JACKSON

Compilatore Lattice C versione 4.0

di Chris Zamara

Il primo compilatore Lattice per Amiga non ebbe concorrenti e fu scelto come il C ufficiale del sistema di sviluppo. Ciò non evitò comunque che molti si lamentassero della lentezza di compilazione e della dimensione del codice generato. Quando l'Aztec C della Manx divenne disponibile, il Lattice sembrò sfigurare al confronto. Il Manx compilava più velocemente, il suo linker era nettamente più veloce dell'Alink fornito dalla Lattice, produceva codice migliore ed era più semplice da usare. L'Aztec divenne ben presto la scelta obbligata come sistema di sviluppo di molti programmatori professionisti.

Con la nuova versione 4.00 del Lattice C, la situazione potrebbe mutare. Da quando John Toebes, famoso per la sua partecipazione alla "Software Distillery" (un gruppo di programmatori noto per aver prodotto alcuni programmi di pubblico dominio molto interessanti NdT), si è unito alla Lattice, il compilatore C ha subito cambiamenti e potenziamenti significativi per risolvere i problemi che aveva evidenziato. Se ora sia "meglio" del Manx dipende da quali siano le vostre esigenze, ma é certo che ogni punto debole che il compilatore manifestava é stato considerevolmente rafforzato e che sono state aggiunte sufficienti migliorie da rendere i programmatori C di Amiga eccitati.

In breve, i miglioramenti più significativi del Lattice rispetto alla precedente versione sono: velocità di compilazione, migliore ottimizzazione del codice, funzioni standard inglobate, chiamata diretta delle funzioni di sistema (senza usare le routine di Amiga.lib), messaggi d'errore migliorati, supporto per gli interi short (16 bit), aderenza allo standard ANSI proposto (compreso il prototyping delle funzioni), file header compressi e più funzioni di libreria. Inoltre é più semplice da utilizzare tramite il CLI, grazie alla presenza di opzioni per la compilazione e il link con un solo comando e alla possibilità di compilare i soli moduli variati dal momento dell'ultima compilazione. In più, il linker accluso è una versione migliorata di Blink della Software Distillery: supporta veri overlay, il pre-linking, l'indicizzazione delle librerie per tempi di linking più brevi e la definizione interattiva, al momento del linking, dei simboli. Il pacchetto comprende un assembler migliorato che supporta i file include standard e alcune interessanti utility di pubblico dominio.

La versione 4.00 del Lattice C è un prodotto molto importante per tutti i programmatori Amiga e di conseguenza la prova non fornisce solamente un indicazione del tipo "va bene" oppure "va male", ma spiega anche le nuove caratteristiche del compilatore in modo da aiutare a farsi un'idea precisa in proposito.

Chiamata delle routine di sistema tramite "#pragma"

Un innovativa ottimizzazione è il modo, opzionale, di chiamata delle routine di sistema. Il codice generato dal C per chiamare una funzione di sistema dell'Amiga non è mai stato efficiente poiché le routine si aspettano i propri parametri nei registri, mentre il C li passa tramite lo stack. Per superare il problema, si effettua sempre il link con una libreria (Amiga.lib) di piccole routine che interfacciano le funzioni di sistema. Esse ricavano i parametri dallo stack, li mettono nei registri appropriati e chiamano la vera funzione di sistema. Questo overhead improduttivo per ogni chiamata di sistema ha permesso ai programmatori Assembler di deridere per lungo tempo i colleghi che si servivano del C. Ora, gli utenti del Lattice 4.00 possono rendere la pariglia e dileggiare chi li ha sempre scherniti.

Una nuova direttiva per il pre-processore C chiamata "#pragma" permette di definire l'offset delle funzioni di sistema all'interno della tabella della libreria e le convenzioni dei registri per la chiamata. Attualmente non c'è bisogno di effettuare da sè le definizioni poiché i file header forniti nei dischetti contengono già le istruzioni "#pragma" per tutte le routine di sistema. Quando esiste una #pragma per una certa funzione, il compilatore genera il codice per chiamarla direttamente, senza il solito inutile overhead. Inoltre l'uso delle #pragma per tutte le routine di sistema evita di dover effettuare il link con Amiga.lib, riducendo di conseguenza il tempo complessivo di linking. Ciò può essere un vantaggio significativo se non si dispone di RAM aggiuntiva o di un hard disk e si deve effettuare il link con le librerie contenute su floppy. Il fatto che l'istruzione #pragma non è portabile tra compilatori diversi non rappresenta un problema, poiché può essere rimossa dal sorgente senza modificare nulla, solo l'efficienza. Eventualmente, per trasportare il proprio codice verso un altro compilatore, occorre solamente rimuovere una singola istruzione #include dal proprio sorgente.
Per mantenere le istruzioni #pragma sempre aggiornate con la versione corrente del Kickstart è fornita una utility chiamata "fd2pragma". Il programma legge i file ".fd" dal disco "Extras" e crea i file header contenenti gli appropriati comandi #pragma. I file .fd, usati dall'AmigaBasic per chiamare le funzioni di libreria, forniscono l'offset e le convenzioni dei registri per tutte le routine di sistema. L'uso dei file .fd per generare le istruzioni #pragma è un'ottima idea, visto che il formato dei file è conosciuto e che ognuno ha accesso ad essi: sono un'eccellente sorgente di informazioni riguardo lo stato corrente delle routine di sistema.

Prototyping delle funzioni

Un'altro aiuto alla chiamata delle funzioni giunge dal prototyping delle funzioni, una caratteristica che è conforme al proposto standard ANSI per i compilatori C. Quando si dichiara una funzione facendo uso del prototyping, oltre al nome, si indica anche il tipo di tutti i parametri che le devono essere passati. Gli stessi file header che contengono le istruzioni #pragma, possiedono i prototipi di tutte le funzioni di sistema. Un gran numero di programmi vanno in "guru meditation" perché si è passato a una routine un intero short invece di un long o perchè si é usato un numero sbagliato di parametri. L'uso dei prototipi può realmente

velocizzare lo sviluppo di programmi poiché, includendoli in un sorgente, vengono effettuate le corrette conversioni di tipo, per esempio tra short e long. Inoltre si possono ricevere messaggi di avvertimento (warning) al momento della compilazione qualora si effettui una chiamata di funzione non corretta. Per mettersi totalmente al sicuro si può utilizzare un'opzione del compilatore per generare un warning in occasione di ogni chiamata di funzioni di cui non esista il prototipo. Il prototyping è una importante caratteristica del Lattice C, poiché é un concreto vantaggio per il programmatore che il Manx non possiede.

Viene effettuato anche un altro livello di controllo delle funzioni secondo quanto dettato dallo standard ANSI: deve essere dichiarato il tipo dei valori restituiti dalla funzioni, a meno che non si tratti di un INT. Le funzioni che non possiedono al loro interno una istruzione RETURN oppure ne hanno una che non restituisce un valore devono essere dichiarate e definite come VOID, altrimenti viene generato un warning. Molti programmatori non si preoccupano di dichiarare funzioni che non restituiscono nulla (specialmente main()!) e i warning per ogni funzioni possono essere fastidiosi. La Lattice, conscia di ciò, dà la possibilità, tramite una delle opzioni che permettono piccoli scostamenti dallo standard ANSI, di sopprimere tali warning, senza per questo eliminarne altri magari più importanti. La mia prima reazione rispetto ai warning é stata negativa, ma quando ho scoperto come la Lattice mantiene la compatibilità ANSI e allo stesso tempo lascia i programmatori contenti, sono rimasto favorevolmente impressionato.

Funzioni inglobate

Le chiamate alle routine di sistema non sono le sole a essere state velocizzate; le funzioni standard usate comunemente sono ora inglobate nel compilatore (built-in), in modo che il codice generato riduca l'overhead della chiamata alla routine generale. Le funzioni disponibili come built-in sono quelle che, come dice il manuale, sono "incoraggiate dallo standard ANSI rispetto a quelle usate in precedenza": strlen, strcmp, strcpy, memset, memcmp e memcpy. Per accedere alle funzioni built-in, si deve far precedere al nome normale la stringa "_builtin_". Ciò viene fatto per mezzo di alcune #define in un file header. Anche questa volta la portabilità rispetto ad altre implementazioni di C non ne soffre, visto che il codice non deve essere modificato per usare questi built-in. Poichè funzioni come strlen() sono usate così spesso, ha senso che il compilatore le gestisca direttamente per generare codice più efficiente. Per esempio una chiamata a strlen() con una stringa costante come parametro non genera codice ma solamente una costante numerica. E' sempre possibile, se lo si desidera, chiamare le proprie funzioni invece di quelle built-in o di quelle in libreria, senza pregiudicare nulla.

File header compressi

Compilare con l'Amiga vuol dire compilare insieme al proprio sorgente file header di dimensioni non proprio limitate. Infatti, se un piccolo programma usa i file header dell'Intuition, il compilatore passa più tempo a compilare gli header che non il sorgente. Una soluzione al problema della velocità può essere la copia dei file in un RAM disk ma così si occupa troppo spazio. Una parziale soluzione è rappresentata dal formato compresso dei file header del Lattice C: i file compressi occupano meno spazio e vengono compilati più velocemente. Tutti i file header di sistema sono disponibili su un dischetto Lattice nel formato compresso ed è inoltre disponibile un programma chiamato "lcompact" per comprimere i normali file header. I file compressi aiutano certamente nello sviluppo di programmi, ma non sono una soluzione così buona come i file header pre-compilati del Manx. Criticare il compilatore Lattice perché il concorrente ha una caratteristica superiore potrebbe sembrare poco sportivo, ma, finché ci saranno solamente due compilatori C commerciali per Amiga, è inevetabile che uno sia valutato soprattutto nei confronti dell'altro; dopo tutto la vostra scelta ricadrà o sull'uno o sull'altro. Tenendo presente ciò, la mancanza nel Lattice dei file header pre-compilati (o il supporto di essi da parte del Manx) è un ottimo argomento in favore dell'Aztec C. Pre-compilando gli header che si è soliti usare, si può compilare il proprio programma senza aver bisogno dei file header, e quindi, senza attendere che vengano compilati ogni volta.

Blink

Uno dei più vitali miglioramenti non risiede nel compilatore in sè, ma è rappresentato dal linker fornito a corredo. Il Lattice originale disponeva del tremendamente lento Alink della Metacomco. Era talmente lento che la Software Distillery decise di produrre, mettendolo in pubblico dominio, un linker migliore e più veloce (nei miei test è più veloce anche del linker "ln" della Manx): il Blink. La Lattice ha ottenuto il diritto di distribuire il Blink e lo fornisce, ulteriormente migliorato e pienamente documentato nel manuale, con il nuovo C. Blink possiede diverse opzioni per creare un file eseguibile, comprese quelle per porre i blocchi DATA in chip RAM, per raggruppare tutti i blocchi CODE o DATA in un singolo blocco, per gestire gli overlay, per sopprimere le informazioni della symbol table, ecc. Una caratteristica unica è la possibilità di definire i simboli in fase di link: se Blink trova un simbolo che non può risolvere, permette all'utente di definire un valore invece di concludere l'esecuzione del linking. In questo modo, se c'è una funzione o una variabile con il nome sbagliato nel proprio sorgente, si può testare il programma comunque senza ricompilare, per poi correggere il nome alla prossima modifica. Forse non é una caratteristica vitale per un linker ma é la dimostrazione che la Lattice sta veramente tentando di accontetare ogni desiderio dei programmatori.

Con il pre-linking, si può consolidare un certo numero di file oggetto in uno solo. Tale modulo può essere poi collegato agli altri con cui si sta lavorando come se nulla fosse cambiato. Ciò evita di fare il linking di tanti file ogni volta che si sono ricompilati solo una piccola parte di sorgenti.

Le librerie di run-time che sono fornite con il C sono indicizzate in un formato che Blink riconosce e usa per velocizzare notevolmente i tempi di linking. Si può comunque effettuare il link anche

con le librerie, fornite anch'esse, nel formato normale. Questo è un vantaggio rispetto al linker Manx che può solamente usare le librerie nel proprio formato e richiede l'uso di una utility di conversione per rendere anche la libreria standard "amiga.lib" leggibile.

Installazione e uso del sistema di sviluppo

Il sistema di sviluppo è composto da quattro dischi, ma solamente due sono neccesari per iniziare a compilare; i programmi per compilare e per il linking sono nel primo disco, le librerie e i file header compressi sono nel secondo. Il modo più semplice di procedere è effettuare il boot con il disco uno e porre il disco due nel secondo drive; la Lattice raccomanda di usare un sistema con (minimo) due drive. Se si dispone di un hard disk si può eseguire un file di comandi fornito sul disco per trasferirvi le directory e i file e fare gli ASSIGN necessari.

Per accedere alle directory dei programmi, dei file header e delle librerie vengono usati (tramite ASSIGN) i device virtuali del DOS. Il sistema funziona bene, ma non è così flessibile o appariscente come le variabili di environment usate dal Manx. Le variabili di environment sono particolari variabili, affiancate al sistema operativo e contenute in una libreria speciale, il cui valore è modificabile servendosi del comando "set". Per esempio, con la variabile INCLUDE del Manx si può specificare un percorso di ricerca di più directory per i file header, mentre si può assegnere una sola directory al device virtuale INLCUDE:. Inoltre, usando gli assegnamenti richiesti per INCLUDE, QUAD, LIB e LC si preclude l'uso degli stessi nomi per qualcos'altro. Un piccolo punto, forse, ma una differenza vale pur sempre qualcosa.
Un punto forte di questa versione Lattice è la facilità di compilare e di fare il link di un programma. Il comando "lc", usato per invocare entrambi i passi di compilazione può richiamare anche il linker se viene specificata l'opzione -L. Inoltre, con un singolo comando, si può compilare un qualunque numero di file; il compilatore Manx "cc" compila un solo file e deve essere rieseguito per ogni modulo. L'intero processo di compilazione e di linking di un programma composto da diversi moduli può essere ridotto al solo comando:

Oppure, ancora più facile, si possono usare le wild card standard dell'AmigaDOS per indicare più nomi di file. Per esempio, per compilare ogni file sorgente C della directory corrente e per effettuare un link con le librerie standard in modo da creare un programma eseguibile, si può usare il comando:

lc -L #?

Ma c'è dell'altro, il potente front-end "lc" del sistema di compilazione possiede un'opzione che fornisce alcune delle funzionalità dell'utility "make". L'opzione "-M" forza la compilazione solo di quei moduli che sono più recenti dei corrispondenti file oggetto (.o). In altre parole, compilerà solamente i file che sono stati modificati dall'ultima compilazione, senza ricompilare ciò che non è necessario. Sebbene una vera utility make, che può essere acquistata separatamente sempre dalla Lattice, è molto più flessibile, questa opzione fornisce un modo semplice di sviluppare programmi costituiti da molti file sorgente.
Si può, ovviamente, compilare e fare il link con Blink separatamente, ma per sviluppare un programma senza problemi, non si può trovare nulla che batta "lc -L". Se si effettua il linking separatamente, ci si può servire del file ".lnk" creato dal compilatore; basta scrivere "Blink WITH nome_prog.lnk" e il linker riceve tutti i parametri giusti rigurdo i file oggetto, le librerie e le opzioni. Il file ".lnk" non viene rimosso dopo il linking, in modo che si possa fare un re-link utilizzandolo nuovamente.

Il comando lc è in realtà un front-end per le utility Lattice e può richiamare il programma "Object Module Librarian" per creare una propria libreria specificando l'opzione appropriata. Le opzioni fornite a lc sono passate correttamente ai vari programmi che manda in esecuzione. Una delle più importanti opzioni permette di specificare in quale tipo di memoria (chip o fast) debbano essere posti i segmenti di codice, di dati e di dati non inizializzati. La possibilità di specificare ciò tramite il compilatore per moduli selezionati è molto meglio dell'approccio Manx di usare un'opzione del linker, poiché l'ultima obbliga a mettere tutti i moduli dello stesso tipo nello stesso tipo di RAM. I dati, come quelli grafici, che devono essere usati dai chip custom dell'Amiga devono risiedere nella chip RAM. A meno che un programmatore non si assicuri specificatamente di ciò, un programma non sarà eseguito correttamente da sistemi con memoria espansa.

Il pacchetto

Il pacchetto base del compilatore Lattice 4.00 è costituito da un manuale e da quattro dischi ed è in vendita per $200 (i prezzi sono forniti in dollari non essendo al momento distribuito ufficialmente in Italia). Un completo "Lattice AmigaDOS C Development System" è disponibile per $375 e include, oltre al compilatore, le "Text Managment Utilities", un'utility "Make", il "Lattice Screen Editor" e l'eccellente debugger "Metascope" della Metadigm. A meno che non si disponga già di utility simili, il "Development System" sembrerebbe un ottimo acquisto. Questa prova, comunque si concentra sul solo compilatore.

La politica di upgrade per gli utenti registrati di versioni precedenti è ragionevole: chiunque ha acquistato un compilatore Lattice C dopo il 1 agosto 1987 ottiene un upgrade gratuito alla versione 4.00. Gli utenti della versione 3.10 ottengono l'upgrade con $45 mentre i possessori di versioni precedenti devono corrispondere $75. Si ha anche la possibiltà di fare l'upgrade del solo compilatore all'intero Development System per $250.
Il manuale spiralato è rettilineo, iniziando con l'installazione, procedendo con le caratteristiche speciali del sistema per poi finire con il materiale di riferimento di cui vivono i programmatori. Le funzioni di libreria sono divise in diverse classi: AMIGA, ANSI, LATTICE, UNIX e XENIX. Le funzioni di una certa classe sono compatibili con quello standard; per esempio ogni funzione della classe LATTICE funzionerà nello stesso modo su ogni computer per il quale la Lattice ha prodotto un compilatore.

Le descrizioni delle funzioni sono in ordine alfabetico, con un indice, e i nomi delle funzioni sono evidenziati in cima alle pagine, rendendo semplice la ricerca scorrendo velocemente i fogli. L'unico problema è che alcune funzioni sono raggruppate con altre e non possono essere ritrovate indipendentemente; per ottenere informazioni su sprintf o printf, ad esempio, occorre sapere che sono raggruppate insieme a fprintf, visto che non sono indicate in cima alla pagina o nell'indice. A parte queste pignolerie, il manuale è eccellente per qualità, accuratezza e chiarezza.

Sebbene il compilatore Lattice, diversamente dal Manx, produca codice oggetto direttamente senza passare attraverso un assembler, ne viene fornito uno da usare liberamente. Il nuovo assembler è compatibile con la sintassi standard (come nel Metacomco originale), funziona con i file include standard e può essere usato con la versione di Blink fornita. Una sezione del manuale spiega come scrivere dei moduli in assembler e come interfacciarli con i propri programmi C.

Alcune utility molto interessanti di pubblico dominio (ConMan, PopCLI e MemWatch) sono fornite nei dischetti: ConMan e PopCLI vengono installate quando si fa il boot. Potreste aver sentito parlare o avere già questi programmi, comunque ConMan permette di modificare i comandi usati in precedenza in una finestra CLI in una qualunque console, PopCLI permette di aprire una finestra CLI in qualunque momento premendo una sequenza di tasti speciale, mentre MemWatch visualizza in continuazione la parte bassa della memoria per controllare che un programma non stia facendo quattro passi dove non deve. I programmi sono disponibili tramite altri canali (BBS), ma il fatto che provengano con i dischetti Lattice permette di salvare tempo di downloading.

Vengono forniti, oltre ad alcuni programmi di esempio e ai file header commentati per C e Assembler, i listati sorgente delle routine standard di inizio e di fine del C.

Il supporto tecnico per il prodotto è fornito in tre modi: tramite computer attraverso BIX (rete informativa della rivista americana Byte, NdT) o il "Lattice Bulletin Board Service" (BBS privato della Lattice), opppure direttamente a voce per mezzo del "Technical Support Hotline" (supporto tecnico telefonico). Viene dato un credito di $10 per una eventuale registrazione a BIX oppure per un account già esistente. Il BBS Lattice è aperto 24 ore al giorno e, sebbene si possa utilizzare un qualunque programma di comunicazione per collegarsi, il programma specifico della Lattice, "SideTalk" è già configurato per usare il servizio; insieme al compilatore é fornito un coupon di $40 per l'acquisto di SideTalk. Il servizio di supporto tecnico telefonico (non è un numero con chiamata a carico del destinatario) é aperto dalle 9 del mattino alle 4 del pomeriggio (ora locale), da lunedi al venerdi. Tutte le informazioni di supporto e i coupon sono posti all'inizio del manuale in modo che possano essere trovati facilmente quando se ne presenta la necessità; naturalmente viene raccomandto di leggere bene il manuale prima di chiamare il servizio di supporto per questioni tecniche.

A confronto con il concorrente

La nuova versione è a posto per quanto riguarda le nuove possibilità operative, ma quanto è meglio riguardo alle prestazioni? Secondo quanto afferma la Lattice molto meglio. Sostengono che il benchmark Dhrystone, che per la versione 3.03 era di 460 e per la 3.10 di 730, vanta un valore di 1295 per la versione 4.00: un incremento del 70%. Di seguito sono riportati i risultati, di fonte Lattice, per i benchmark standard Dhrystone, Float e Savage a confronto con il Manx 3.40:

	Lattice 4.00	**Manx 3.40**
Dhrystone	1294 Dhrystone/sec	1010 Dhrystone/sec
Float	22,20 sec (IEEE)	98,85 sec (IEEE)
	10,16 sec (FFP)	17,60 (FFP)
Savage	47,67 sec	119,6 sec
	0,000000318 prec.	0,000109 prec.

Nei nostri test, condotti usando il programma Dhrystone del Fish Disk N.1, risultano valori differenti rispetto a quelli illustrati: il Lattice è sempre migliore ma non con un margine così elevato. Usando poi gli interi a 16 bit con entrambi i compilatori, la differenza si assottiglia ulteriormente. Non siamo comunque sicuri di quanto un singolo benchmark sia significativo. Il gioco dei benchmark è complesso, e una manciata di numeri non è sufficiente per evidenziare i punti forti e quelli deboli della generazione di codice da parte di un compilatore piuttosto che un altro.

Tenendo presente ciò, illustriamo i risultati di un altro benchmark, che non è standard, ma è ideato per mostrare le prestazioni in un specifico tipo di operazioni. La mancanza di spazio non ci permette di presentare qui il listato, ma è presente nel dischetto di questo numero. Esegue i loop di tre routine, una alla volta, e memorizza i risultati di ognuna di esse. Le routine testano operazioni comuni di aritmetica intera, puntatori e strutture, gestione di stringhe. Ecco i risultati del test, usando 10000 iterazioni:

Numero di tick

		Lattice	**Aztec**
Test:	interi	243	344
	pointer	352	407
	stringhe	1045*	1011
	Totale	1640	1762

* usando le funzioni di stringa built-in: 1032

Da questo test sembra che il Lattice sia molto meglio per quanto riguarda le operazioni intere e un po' meglio nella manipolazione di strutture e puntatori, ma il Manx è superiore nella gestione di stringhe. Usando le funzioni built-in il Lattice ha un miglioramento, ma non riesce comunque a raggiungere il risultato del Manx. Questo benchmark non si può definire esattamente scientifico,

ma va di pari passo con quelli standard, indicando migliori prestazioni globali del Lattice.

Velocità di compilazione e di link

Il vecchio Lattice era conosciuto per la lentezza in compilazione. In diversi confronti diretti usando i file header compressi, abbiamo riscontrato che la versione 4.00 ha tempi simili al Manx 3.40. Per compilare e fare il link di un tipico programma di tre moduli di circa mille linee di sorgente l'Aztec ha impiegato 1 minuto e 27 secondi, mentre il Lattice ha avuto bisogno di 1 minuto e 36 secondi. I test sono stati condotti mantenendo in RAM tutti i file necessari per minimizzare gli effetti di differenti spostamenti delle testine durente la ricerca di file su disco. Il Lattice compila più lentamente del Manx, ma Blink effettua il link più velocemente e quindi si vengono a chiudere le distanze. I risultati che si ottengono possono variare da applicazione ad applicazione ma rimangono comunque talmente simili che il tempo di compilazione non può più essere preso in considerazione per un confronto.

Usando i file header pre-compilati del Manx le differenze si fanno più drammatiche. Non è un confronto impari, visto che generalmente si usano proprio gli header pre-compilati, tanto che molti utenti dell'Aztec C non si preoccupano di includere header di sistema e compilano includendo tutti i pre-compilati. Quando si considera che un header pre-compilato di questo tipo resta in RAM mentre tutti gli header non lo fanno (tutti gli header compressi Lattice assommano a 440K, mentre tutti i pre-compilati Manx arrivano a quasi 118K), la differenza di velocità in compilazione/link aumenta.

Le dimensioni un po' troppo elevate del compilatore Lattice lo rendono più lento e causano anche altri problemi. Per compilare si ha bisogno del front-end "lc" e dei due programmi che richiama: "lc1" e "lc2". La dimensione combinata di questi programmi è di più di 200K, mentre il singolo programma "cc" del Manx occupa meno di 80K. Ciò può essere importante per quegli utenti che dispongono di solo 512K di RAM poiché sono costretti a caricare i programmi da floppy a ogni compilazione, aggiungendo una significativa quantità di tempo al processo.

Dando uno sgardo alla dimensione del codice prodotto, il Manx é sicuramente il vincitore nell'area dei programmi piccoli, producendo spesso codice inferiore del 30%. Questo ha molto a che fare con il maggior overhead introdotto dal Lattice: un programma nullo (main(){}) è lungo 3892 byte con il Lattice, e solamente 1832 con il Manx. Ciò è dovuto al fatto che il Lattice compie molto lavoro in vostro favore nel codice di startup e di exit, come fornire un requester per i CTRL-C impartiti dall'utente. Il codice di startup può essere cambiato o sostituito con la propria routine in modo che l'overhead extra sia eliminato. Anche tenendo presente l'overhead la dimensione di un programma eseguibile è comunque spesso maggiore. Lo stesso programma di 1000 linee usato nel precedente test di compilazione occupa 13584 byte con il Lattice e 9852 con l'Aztec.

Con alcuni programmi la differenza non è così marcata: la compilazione della versione 1.1 del programma "Structure Browser" di Transactor, usando gli interi a 16 bit, ha prodotto 28280 byte con il Manx contro 30456 del Lattice, solamente una differenza dell'8%. La piccola differnza in questo caso è forse dovuta al fatto che il programma è composto più da dati statici che non da codice. Tutto sommato, è indubbio affermare che il Lattice produce codice maggiore del Manx, anche usando opzioni per SMALLCODE, SMALLDATA, indirizzamento relativo al PC, interi a 16 bit e nessun controllo dello stack.

Odiamo fare generalizzazioni come queste, ma da un certo numero di test ecco le differenza tra il Manx e il Lattice: il codice Lattice viene eseguito più velocemente, il Manx produce codice minore e sono simili rispetto alla velocità di compilazione/link. Non sono sicuramente le uniche statistiche importanti per un compilatore, comunque sono fattori da tenere in considerazione.

Errori e messaggi di warning

Il Lattice è molto più pignolo dell'Aztec e fornisce warning per una varietà di cose sulle quali avreste non potuto pensare due volte. In generale ciò è ottimo, visto che i warning sono fatti apposta.
Compilando con il Lattice un programma ponderoso che era stato era stato sviluppato con il Manx, si è stati in grado di scovare alcuni bug molto difficili che non erano stati trovati nemmeno durante la fase di test! Alcuni dei warning che si ricevono rigurdano le variabili AUTO non inizializzate (un errore comune difficile da scovare), istruzioni che non hanno effetto (come scrivere c == 0 invece di c = 0), variabili che sono dichiarate ma mai utilizzate (utile per ripulire vecchi programmi) e altri. Visto che il C è così scarso nel dattare regole precise, la vigilanza del Lattice può aiutare ad evitare problemi.
D'altra parte un sentimento comune tra i programatori C più rudi é: "Se avessi voluto usare il LINT avrei digitato lint".
Io non sono un programmatore C novellino, ma se posso scovare degli errori potenziali in fase di compilazione invece che durante l'esecuzione, sono contento. Di conseguenza è opportuno guardare alla dose extra di warning del C Lattice come a un qualcosa in più.

Inevitabilmente anche la comunicazione degli errori è stata migliorata: il codice sorgente incriminato viene stampato insieme al messaggio di errore, in modo da fornire un'idea di ciò che è successo, prima che ci si rituffi a esaminare il sorgente.
Continuiamo a preferire il modo di visualizzazione degli errori una pagina alla volta del Manx, ma un CTRL-C permette sempre di interrompere la compilazione in ogni momento, fornendo anche la possibilità di riprendere da dove si è interrotto.

Una caratteristica del Manx della quale alcuni sentiranno la mancanza più di altri è la capacità di mettere codice assembler direttamente nel sorgente C per mezzo della direttiva #asm. Il Lattice può comunque effettuare il link con moduli assembler, risultando quindi non limitato in questo campo, ma presentando solo un piccolo inconveniente.

Quanto sia grave questo inconveniente dipende solamente dalle

vostre abitudini di programmazione.

Funzioni di libreria

La libreria Lattice dispone di un set di funzioni veramente ricco, comprendendo l'I/O, le routine DOS, operazioni su stringhe e interi, funzioni per l'ora e la data, allocazione di memoria, conversioni tra tipi differenti, funzioni trigonometriche e in virgola mobile, ecc.
Ci sono più di 100 funzioni, ognuna definita in modo conciso nel manuale. Molte funzioni sono presenti per mantenere la compatibilità Unix, tanto che alcune di esse eseguono operazioni che potrebbero essere tranquillamente portate a termine da funzioni AmigaDOS, rendendo semplice il trasporto di programmi Unix sull'Amiga. Senza entrare in un confronto diretto, si può comunque affermare che la libreria Lattice è molto più completa di quella Manx, specialmente nell'ambito della manipolazione delle stringhe.

Quando sarà disponibile la prossima release del compilatore Aztec C, la versione 3.6, faremo anche un confronto dettagliato delle librerie.

Il vostro vincitore?

L'ultima questione: prenderete il Lattice 4.00 oppure il Manx 3.4? O è meglio aspettare il Manx 3.6? O addirittura aspettare qualcos'altro? Fortunatamente, siete in grado di accertare le vostre necessità e fare una scelta basata su ciò che adesso sapete. Comunque non vi lasciamo da soli a questo punto, vediamo quali sono le possibilità.

Se non possedete attualmente un compilatore C e state cercando di decidere quale acquistare, vi trovate in una situazione invidiabile ma difficile. Invidiabile perché entrambi i compilatori per Amiga sono molto buoni; difficile perché prendere una decisione è veramente difficile. Il prezzo è circa lo stesso (considerando il pacchetto base del Manx) e perciò non è di nessun aiuto, nella decisione. Inoltre nessuno è migliore dell'altro su tutti i fronti e quindi non può essere scelto perchè è in assoluto il migliore.

Bisogna scendere a un livello inferiore e considerare l'anima di ognuno dei due sistemi di sviluppo. Un nuovo programmatore C, o uno che ama linguaggi che lasciano meno spazio ai possibili errori, si troverà meglio con il Lattice.

Il compilatore vi prende per mano e vi mostra gli errori anche d'impostazione, conducendovi di conseguenza a raffinare la vostra arte programmatoria. Sarete in grado di produrre programmi liberi da errori più facilmente, utilizzando in generale tempi di sviluppo minori. E' anche più semplice da usare, permettendo la compilazione di programmi composti da più moduli con un solo comando, senza aver bisogno di apprendere la strana sintassi di "Make" (che nella versione base dell'Aztec C non é comunque fornito).

Le routine standard di inizio e di fine aggiunte ai vostri programmi C li renderanno migliori, anche se occuperanno più spazio rispetto a quello che avrebbero occupato se fosero stati compilati con il Manx.

D'altra parte, se siete un programmatore C esperto, magari con qualche interesse diretto di programmazione in Assembler del 68000 e desiderate avere un sistema di compilazione che non vi disturbi con stupidi warning e che produca il codice più ridotto possibile, dovrete imboccare la strada che porta al Manx. Non vi dovrete preoccupare di quali file header avete bisogno e, se fate qualche piccolo errore nel vostro sorgente, lo verificherete quando il programma non vorrà funzionare, ma almeno non sarete infastiditi da brutti messaggi durante la compilazione. Sapete quello che fate, e se volete passare un intero short a una funzione che richiede un long, avrete quello che vi meritate. Vi piace poter ottimizzare le funzioni usate più spesso semplicemente ponendo all'interno di esse del codice direttamente in Assembler; inoltre la possibilità di dare uno sguardo al file in Assembler prodotto dal compilatore è un buon modo per trovare il codice che ha bisogno di una ottimizzazione. Trovereste il Lattice troppo stringente, troppo grande, troppo distante dal centro dell'azione.

Forse una serie di domande ancora più ardue possono provenire da cui usa correntemente il Manx: vale la pena fare un cambio? Perdereste i file header pre-compilati (che qualcuno non usa comunque) e, se foste abituati a mandare in esecuzione il compilatore e il linker dalla RAM, potreste non avere più posto in memoria. Avreste un migliore trattamento e segnalazione degli errori, un codice più veloce anche se più grosso e un maggior numero di funzioni di libreria.

E' abbastanza facile lasciare la decisione agli altri, ma noi passeremo al Lattice 4.00? Per alcune applicazioni sicuramente, ma certe realtà faranno sì che per la maggior parte manterremo il nostro Aztec C di fiducia.
Per prima cosa abbiamo molto codice sorgente che pensiamo non gradisca il trasporto: molti moduli non contengono i file header necessari perché presuppongono l'uso degli header pre-compilati. Ci sono altre sottili differenze, quali il limite di 256 byte di lunghezza delle stringhe per il Lattice, che sono sufficienti a impossibilitare la compilazione del codice esistente.

Un altro fattore che ci tiene ancorati al Manx é la promessa di un debugger a livello di codice sorgente disponibile a partire dalla versione 3.6: almeno sarà facile trovare gli errori tanto quanto l'Aztec rende facile farli. Infine, data la dimensione di alcuni nostri progetti, sarebbe difficile vivere senza l'utility "make".

L'opzione -M é bella, ma non ri-compila automaticamente i moduli i cui file header sono stati modificati e non siamo sicuri che valga la pena spendere i soldi extra necessari per il sistema "Developer" per ottenere solo il make della Lattice.

In conclusione il Lattice C 4.00 é un miglioramento tale da rendere più semplice la vita del programmatore, anche se sicuramente renderà la scelta più ardua. Questa é comunque una situazione fortunata, in quanto non é possibile fare una cattiva scelta.

LISTINO LIBRI JACKSON

CODICE	TITOLO	PREZZO	
	INFORMATICA: CONCETTI GENERALI		
511 A	COME PROGRAMMARE	15.000	
503 A	PROGRAMMAZIONE STRUTTURATA, CORSO DI AUTOISTRUZIONE	15.000	
101 H	TERMINI DELL'INFORMATICA E DELLE DISCIPLINE CONNESSE	50.000	
539 A	LOGICA E DIAGRAMMI A BLOCCHI: TECNICHE DI PROGRAMMAZIONE	40.000	
526 P	DATA BASE: CONCETTI E DISEGNO	22.500	
GYS190	TRADUTTORI DI LINGUAGGI	26.000	
G 240	PAROLE BASE DELL'INFORMATICA	8.000	
GYS245	CONCETTI DI INFORMATICA	43.000	
GYS248	DATA PROCESSING	45.000	
GY 264	DATA FILE	50.000	
GYS266	ARCHITETTURE DI SISTEMA	32.000	
GY 354	SISTEMI INTELLIGENTI	28.000	
CZ 419	ANALISI E PROGRAMMAZIONE	11.000	
158 EC	INFORMATICA DI BASE I CONCETTI FONDAMENTALI HARDWARE E SOFTWARE	55.000	
526 A	VOI E L'INFORMATICA	15.000	
100 H	DIZIONARIO DI INFORMATICA	59.000	
GY 551	I LINGUAGGI DELLA 4a GENERAZIONE	65.000	
GYS552	PRIMA DEL LINGUAGGIO LA PROGRAMMAZIONE	35.000	
GYS 559	C.S.P. - PROCESSI SEQUENZIALI	49.000	
GYS 546	ALGORITMI FONDAMENTALI	54.000	
GY 618	SISTEMI ESPERTI	28.000	
047 T	MICROPROCESSORI	14.500	
048 T	DATA BASE	14.500	
049 T	FILE	14.500	
CI 686	CAPIRE IL PERSONAL COMPUTER	35.000	
G 540	MODELLI MATEMATICI E SIMULAZIONE	56.000	
GE 688	ENCICLOPEDIA MONOGRAFICA DI ELETTR. E INF. VOLUME I	58.000	
GE 689	ENCICLOPEDIA MONOGRAFICA DI ELETTR. E INF. VOLUME II	58.000	
GY 629	SOFTWARE DI BASE - Strumenti di sviluppo	52.000	
	INFORMATICA: SISTEMI OPERATIVI		
G 223	UNIX LA GRANDE GUIDA	70.000	
GY 272	SISTEMI OPERATIVI PER MICROCOMPUTER	25.000	
GY 273	MS-DOS LA GRANDE GUIDA	45.000	
510 P	CP/M CON MP/M	29.000	
CZ 538	MS DOS 2 E 3	49.000	
G 543	XENIX	45.000	
R 588	LAVORARE CON XENIX	70.000	
GYS271	SISTEMI OPERATIVI	55.000	
R 615	I COMANDI DI XENIX MAIL	12.500	
092 D	SOFTWARE DI BASE E SISTEMI OPERATIVI	7.000	
093 D	CP/M IL "SOFTWARE BUS"	7.000	
094 D	MS-DOS E PC-DOS LO STANDARD IBM	7.000	
009 H	UNIX	8.500	
011 H	CP/M	8.500	
044 T	MS DOS	14.500	
045 T	PC DOS	14.500	
R 628	MICROSOFT OS/2	50.000	
046 T	UNIX	14.500	
MS 02 E	COFANETTO "MS-DOS" 5"¼ - Corso autoistruzione	156.000	F
R 600	MS DOS ADVANCED - Il Manuale del Programmatore	55.000	
GY 663	UNIX PROGRAMMAZIONE AVANZATA	55.000	
BY 724	GUIDA AI SISTEMI OPERATIVI	29.000	
BY 744	UNIX: CONCETTI, STRUTTURE, UTILIZZO	43.000	
R 761	MS-DOS REFERENCE GUIDE	14.500	
	INFORMATICA: LINGUAGGI		
501 A	IMPARIAMO IL PASCAL	16.000	
502 A	INTRODUZIONE AL BASIC	25.000	
500 P	PASCAL MANUALE E STANDARD DEL LINGUAGGIO	16.000	
329 A	PROGRAMMARE IN ASSEMBLER	14.000	
513 A	PROGRAMMARE IN BASIC	8.000	
514 A	PROGRAMMARE IN PASCAL	19.000	
516 A	INTRODUZIONE AL PASCAL	39.000	
517 P	DAL FORTRAN IV AL FORTRAN 77 (II ED.)	32.000	
521 A	50 ESERCIZI IN BASIC	17.000	
525 A	BASIC PER TUTTI	23.000	

CODICE	TITOLO	PREZZO	
534 A	MANUALE DEL BASIC	45.000	
509 A	LOGO: POTENZA E SEMPLICITÀ	20.500	
507 B	TUO PRIMO PROGRAMMA IN BASIC (II)	19.500	
533 A	BASIC DALLA A ALLA Z	19.000	
540 A	LINGUAGGIO ADA	19.500	
541 P	LINGUAGGIO C	25.000	
542 P	COBOL STRUTTURATO: CORSO DI AUTOISTRUZIONE	50.000	
508 P	PROGRAMMARE IN C	39.000	
G 233	COBOL PER MICROCOMPUTER	35.000	
GYS246	ESERCIZI DI FORTRAN	20.000	
GYS247	ESERCIZI IN PASCAL: ANALISI DEI PROBLEMI	29.000	
GYS254	PROGRAMMAZIONE IN LINGUAGGIO ADA	42.000	
GY 270	APL PER IL P.C. IBM	25.000	
GYS274	DAL PASCAL AL MODULA 2	26.000	
GYS311	LINGUAGGIO C IL LIBRO DELLE SOLUZIONI	24.000	
GYS328	APPLICAZIONI IN PASCAL	32.000	
GY 535	TURBO PASCAL	29.000	
G 544	"C" LIBRARY	49.000	
GYS550	PROLOG - LINGUAGGIO E APPLICAZIONE	32.000	
R 589	TURBOPASCAL - LIBRERIA DI PROGRAMMI	45.000	
042 T	LINGUAGGIO C	12.500	
108 D	FORTH ANATOMIA DI UN LINGUAGGIO	7.000	
107 D	FORTRAN E COBOL LINGUAGGI SEMPRE VERDI	7.000	
086 D	ED È SUBITO BASIC VOL. 1	7.000	
087 D	ED È SUBITO BASIC VOL. 2	7.000	
034 T	PROLOG	14.000	
035 T	LISP	12.500	
001 H	COBOL	8.500	
006 H	PASCAL	8.500	
007 H	BASIC	8.500	
010 H	FORTRAN 77	8.500	
020 H	LOGO	8.500	
022 H	FORTH	8.500	
R 612	TURBO PROLOG	50.000	
GY 626	IL MANUALE DEL PASCAL	42.000	
GY 616	DEBUGGING C	55.000	
GY 687	DALLA PROGRAMMAZIONE STRUTTURATA AL PASCAL	42.000	
GY 634	FONDAMENTI DI COMMON LISP	40.000	
	INFORMATICA: LAVORO È SOCIETÀ		
519 P	COMPUTER GRAFICA	29.000	
800 P	ODISSEA INFORMATICA	50.000	
407 H	APPLICAZIONI DEL COMPUTER NELL'UFFICIO MODERNO	23.000	
802 H	INFORMATICA MUSICALE	27.000	
802 P	COMPUTERGRAPHIA	40.000	
806 P	COMPUTER PER L'INGEGNERIA EDILE	22.000	
807 P	COMPUTER PER IL MEDICO	19.000	
CI 231	COMPUTER IMAGE	40.000	
CI 241	ODISSEA INFORMATICA STRATEGIE CULTURALI PER UNA SOCIETÀ INF.	32.000	
G 400	COMPUTER GRAPHICS E ARCHITETTURA	27.000	
PV 409	COMPUTER GRAPHICS E MEDICINA	18.000	
GY 487	MEDICO & COMPUTER	45.000	
GY 548	INFORMATICA MEDICA	65.000	
PA 685	OFFICE AUTOMATION	28.000	
RA 596	DESKTOP PUBLISHING	35.000	
050 T	WORD	14.500	
	INFORMATICA: SOFTWARE PACCHETTI APPLICATIVI		
570 P	CONTABILITÀ COL PERSONAL COMPUTER	27.000	
525 P	WORDSTAR	24.000	
546 P	MANUALE DEL DBASE II	24.000	
578 P	PC NELL'ORG. DELLE PICCOLE AZIENDE: APPL. DEL MULTIPLAN	29.000	F
PP 219	LOTUS 1-2-3: GUIDA ITALIANA ALL'USO	21.000	
G 234	RIORDINO E GESTIONE DEGLI ARCHIVI APPLICAZIONI CON PFS-FILE	30.000	
PP 255	DBASE III GUIDA ITALIANA ALL'USO	45.000	
PA 282	MODELLI DECISIONALI PER IL MANAGER	50.000	
PA 288	PIANIFICAZIONE AZIENDALE PLANNING, MARKETING STRAT., BUDGETING	35.000	F
PP 310	LA GRANDE GUIDA LOTUS A SYMPHONY	70.000	
PP 326	MULTIPLAN CORSO D'ISTRUZIONE	40.000	F

CODICE	TITOLO	PREZZO	
PP 344	FRAMEWORK II - GUIDA ITALIANA ALL'USO	27.000	
PP 351	WORD PROCESSING	27.000	
PP 467	IMPARA 1-2-3 CON LA GRANDE GUIDA LOTUS	45.000	
PP 468	CHART - CORSO ISTRUZIONE	45.000	F
PP 473	IL NUOVO 1-2-3 GUIDA ALL'USO DELLA VERSIONE ITALIANA 2 LOTUS 1-2-3	29.000	
PA 474	BILANCIO, BUDGET, CASH FLOW	40.000	F
PP 475	DBASE III - CORSO DI PROGRAMMAZIONE	23.000	
PA 476	PREVISIONE, PIANIFICAZIONE, SIMULAZIONE CON LOTUS 1-2-3	60.000	F
PV 477	GUIDA ALLA BUSINESS GRAPHIC	20.000	
PP 480	AUTOCAD	40.000	
PP 481	RBASE 5000 - GUIDA ITALIANA ALL'USO	20.000	
PP 537	IL MANUALE DI WINDOWS	60.000	
PP 539	DBASE III - TECNICHE AVANZATE DI PROGRAMMAZIONE	42.000	
PP 545	APPLICAZIONI DI DBASE III	50.000	F
PA 566	MODELLI DECISIONALI CON LOTUS 1-2-3	40.000	F
PP 577	MANUALE DBASE III PLUS	49.000	
039 T	WORDSTAR	12.500	
040 T	LOTUS 1-2-3	12.500	
043 T	WINDOWS	12.500	
PP 621	I COMANDI DI DBASE III PLUS	12.500	
095 D	GUIDA AI PACKAGE APPLICATIVI MERCEOLOGIA DEL SOFTWARE	7.000	
096 D	VISICALC GUIDA RAPIDA ALL'UTILIZZO	7.000	
098 D	WORD PROCESSING	7.000	
103 D	LOTUS 1-2-3 E SIMPHONY IL FASCINO DELL'INTEGRAZIONE	7.000	
104 D	DBASE II E III I PRINCIPI DI DATABASE	7.000	
106 D	MULTIPLAN SPREADSHEET MULTISTRATO	7.000	
110 D	PACKAGE A CONFRONTO PROVE DEI SOFTWARE PIÙ DIFFUSI	7.000	
031 T	FRAMEWORK E FRAMEWORK II	12.500	
033 T	MULTIPLAN 2.02	12.500	
036 T	SYMPHONY	12.500	
038 T	REFLEX	12.500	
027 H	EASY SCRIPT	8.500	
033 H	PAGE MAKER	8.500	
034 H	PROJECT	8.500	
035 H	RBASE	8.500	
PP 611	GUIDA ALL'USO PROFESSIONALE REFLEX	55.000	
PP 636	MANUALE DI WORD	70.000	
PP 594	GUIDA ALL'USO PROFESSIONALE DI LOTUS 1-2-3	50.000	
PP 593	VENTURA - Il grande manuale	55.000	
R 671	LINGUAGGIO C - Reference guide	12.500	
051 T	I COMANDI DI LOTUS 1-2-3 - Reference guide	12.500	
PP 581	PROGRAMMARE IN FRED	40.000	
PP 631	dBASE III PLUS - Guida uso professionale	65.000	
PP 694	PROGRAMMARE IN WINDOWS	70.000	
PA 592	GESTIONE DELLA PRODUZIONE	40.000	F
PP 727	VENTURA - REFERENCE GUIDE	14.500	
PP 700	MATEMATICA CON LOTUS 1-2-3	35.000	F
R 574	MANUALE DELLE STAMPANTI LASER	25.000	
PP 641	AUTOCAD - Il grande manuale	55.000	
PP 728	VENTURA - Fogli stile	42.000	F
PP 741	WORD 3 e 4	59.000	F
PP 642	AUTOCAD Programmazione avanzata	65.000	F
BY 707	ORACLE	75.000	
PA 771	MODELLI PER LOTUS 1-2-3	28.000	F
	PERSONAL COMPUTER		
550 D	PROGRAMMI PRATICI IN BASIC	15.000	
515 H	BASIC E LA GESTIONE DEI FILE VOL. I: METODI PRATICI	15.000	
551 D	75 PROGRAMMI IN BASIC PER IL VOSTRO COMPUTER	12.000	
552 D	PROGRAMMI DI MATEMATICA E STATISTICA IN BASIC	20.000	
554 P	PROGRAMMI SCIENTIFICI IN PASCAL	29.000	
516 H	BASIC E LA GESTIONE DEI FILE - VOL. 2	17.000	
CH 182	COMPUTER HARDWARE REALIZZ. PRATICHE PER GLI HC PIÙ DIFFUSI	18.000	
CI 187	COMPUTER L'HOBBY E IL LAVORO	12.000	
G 235	GRAFICA PER PERSONAL COMPUTER	39.000	
GE 263	METODI DI INTERFACC. PERIFERICHE	43.000	
GE 402	CORSO DI AUTOISTRUZIONE PER MICROCOMPUTER VOL. 1 + VOL. 2	35.000	
PA 406	COME GESTIRE LA PICCOLA AZIENDA CON IL P.C.	22.000	
PP 408	BUSINESS IN BASIC	23.000	
CI 412	IL COMPUTER È UNA COSA SEMPLICE	15.000	

CODICE	TITOLO	PREZZO	
CC 415	CONTROLLO DEI DISPOSITIVI DOMESTICI CON IL P.C.	23.000	
159 GC	PERSONAL COMPUTER DAL SOFTWARE DI BASE ALLE APPLICAZIONI D'UFFICIO	55.000	
R 587	HARD DISK - LA GRANDE GUIDA	75.000	
084 D	INTRODUZIONE AI PERSONAL COMPUTER VIVERE COL PC	7.000	
099 D	SCRIVERE UN'AVVENTURA, 1000 AVVENTURE COL PROPRIO PC	7.000	
100 D	GRAFICA E BASIC LE BASI DELLA COMPUTERGRAFICA	7.000	
085 D	HARDWARE DI UN PERSONAL COMPUTER DENTRO E FUORI LA SCATOLA	7.000	
101 D	GESTIONE DEI FILE IN BASIC E PASCAL VOL. 1	7.000	
102 D	GESTIONE DEI FILE IN BASIC E PASCAL VOL. 2	7.000	
113 D	DISEGNARE COL PERSONAL COMPUTER	7.000	
105 D	PERSONAL E HOME COMPUTER A CONFRONTO	7.000	
112 D	SUONO E MUSICA COL PERSONAL COMPUTER	7.000	
109 D	COSTRUIRSI UN PERSONAL DATABASE	7.000	
097 D	GUIDA ALL'ACQUISTO DI UN PERSONAL COMPUTER	7.000	
088 D	TO DO OR NOT TO DO COME AVER CURA DEL PROPRIO PC	7.000	
089 D	SOFTWARE STRUTTURATO CON ELEMENTI DI PASCAL	7.000	
090 D	DIZIONARIO DI INFORMATICA	7.000	
091 D	BASI DELLA PROGRAMMAZIONE STENDERE UN PROG. COME SI DEVE	7.000	
004 H	PROGRAMMAZIONE	8.500	
015 H	PROGRAMMI DI STATISTICA	8.500	
	PERSONAL COMPUTER: COMMODORE		
347 D	VOI E IL VOSTRO COMMODORE 64	24.000	
348 D	COMMODORE 64 - IL BASIC	28.000	
400 D	FACILE GUIDA AL COMMODORE 64	13.500	
400 B	COMMODORE 64 - FILE	19.000	
409 B	COMMODORE 64 - LA GRAFICA E IL SUONO	34.000	C
570 D	MATEMATICA E COMMODORE 64	26.500	C
350 D	LIBRO DEI GIOCHI DEL COMMODORE 64	24.000	C
575 D	TECNICHE DI PROGRAMMAZIONE SUL COMMODORE 64	16.500	
572 D	LINGUAGGIO MACCHINA DEL COMMODORE 64	35.000	F
576 D	SISTEMA TOTOMAC: LA NUOVA FRONTIERA DEL TOTOCALCIO	29.000	C
548 B	64 PERSONAL COMPUTER E C64	45.000	
SDP222	STATISTICA AD UNA DIMENSIONE CON IL C64	24.000	
CC 260	AVVENTURE (COMMODORE 64)	20.000	C
CC 320	AMIGA HANDBOOK	35.000	
CC 322	COMMODORE 128 OLTRE IL MANUALE	29.000	
CC 323	PROGRAMMI PER COMMODORE 128	29.000	C
CZ 541	128 E 64 - LE PERIFERICHE	32.000	
CC 564	MANUALE RIPARAZIONE C64	55.000	
CZ 532	MANUALE DI AMIGA	39.000	
002 H	COMMODORE 64	8.500	
CC 658	GRAFICA E SUONO PER C64 - 64PC - C128	35.000	F
CC 657	MANUALE DEL COMMODORE C64 - C64PC - C128	35.000	F
CC 627	AMIGA 500	55.000	
CC 750	C.128 LA GRANDE GUIDA	50.000	
CC 749	C.64 LA GRANDE GUIDA	50.000	
	PETER NORTON		
R 734	MANUALE DEL DOS	55.000	
R 736	INSIDE PC IBM	63.000	
R 733	HARD DISK COMPANION	60.000	
R 735	LINGUAGGIO ASSEMBLY PER PC IBM	72.000	2F
	PERSONAL COMPUTER: IBM		
564 D	PROGRAMMI UTILI PER IBM PC	19.000	
G 217	GRAFICA PER IL PERSONAL COMPUTER IBM	39.000	
GY 319	PC IBM MANUALE DEL LINGUAGGIO MACCHINA	45.000	
GY 335	MAPPING PC IBM GESTIONE DELLA MEMORIA	42.000	
PP 407	MANUALE BASE DEL PC IBM	22.000	
041 T	PC IBM	12.500	
R 609	SOLUZIONI AVANZATE PER IL PROGRAMMATORE	60.000	
CZ 751	AVVENTURE PER MS-DOS	35.000	F
RA 484	GUIDA ALLE RETI DI PC IBM	46.000	

CODICE	TITOLO	PREZZO	
	PERSONAL COMPUTER: OLIVETTI		
401 P	PRIMO LIBRO PER M24: MS DOS E GW BASIC	28.000	
401 B	OLIVETTI M10: GUIDA ALL'USO	18.000	
CL 216	BASIC IN 30 ORE PER M24 ED M20	32.000	
CZ 483	MANUALE OLIVETTI M19	42.000	
CZ 536	MANUALE PC 128 OLIVETTI PRODEST	29.000	
CZ 582	PROGR. PER PC 128 OLIVETTI PRODEST (CASS.)	27.000	
	PERSONAL COMPUTER: MSX		
CZ 181	30 PROGRAMMI PER MSX	20.000	C
417 D	MSX: IL BASIC	23.000	
CC 261	AVVENTURE (MSX)	20.000	C
CC 289	SUPER PROGRAMMI PER MSX	35.000	C
CC 336	MSX LA GRAFICA	25.000	
111 D	STANDARD MSX	7.000	
	PERSONAL COMPUTER: APPLE		
331 P	APPLE II GUIDA ALL'USO	31.000	
416 P	MACINTOSH NEGLI AFFARI: MULTIPLAN E CHART	16.500	
424 P	UN MAC PER AMICO: USO, APPLICAZIONI E PROGRAMMI PER MACINTOSH	12.000	
PP 224	MACINTOSH ARTISTA: MACPAINT E MACDRAW	16.000	
CCP277	APPLE IIC GUIDA ALL'USO	45.000	
CC 312	PROGRAMMI PER APPLE IIC	13.000	
CC 417	PROGRAMMI COMM. E FINANZIARI CON APPLE	22.000	
340 H	APPLE MEMO	15.000	
CC 576	IL MANUALE DELL'APPLE II GS	28.000	
003 H	APPLE IIE IIC	8.500	
CC 665	MICROSOFT BASIC PER APPLE MACINTOSH	32.000	F
	PERSONAL COMPUTER: ATARI - AMSTRAD - SHARP		
540 H	BASIC ATARI	18.000	
CC 330	PROGRAMMI PER AMSTRAD CPC 464 CPC 664 - CPC 6128	29.000	C
CC 331	PROGRAMMI PER ATARI 130XE	19.000	C
CC 471	MANUALE ATARI 520 ST E 1040 ST	28.000	
CC 486	WORD PROCESSING CON AMSTRAD PCW 8256/8512	35.000	
032 T	AMSTRAD PCW 8256 e PCW 8512	14.000	
028 H	AMSTRAD 464 E 664	8.500	
	COMMUNICATION E TELEMATICA		
309 A	PRINCIPI E TECNICHE DI ELABORAZIONE DATI	20.000	
518 D	TELEMATICA	28.000	
528 P	TRASMISSIONE DATI	27.000	
617 P	RETI DATI: CARATTERISTICHE, PROGETTO E SERVIZI TELEMATICI	40.000	
GYS314	ELABORAZIONE DIGITALE DEI SEGNALI: TEORIA E PRATICA	25.000	
PA 327	BANCHE DATI RICERCA ONLINE	26.000	
158 LC	COMUNICAZIONI DALLE ONDE ELETTROMAGNETICHE ALLA TELEMATICA	55.000	
CC 472	MODEM E PC USO E APPLICAZIONI	25.000	
GTS478	RETI LOCALI	44.000	
GTS479	IL MODEM - TEORIA, FUNZIONAMENTO	28.000	
R 542	TRASMISSIONE DATI PER PC	31.000	
GT 555	LA TELEMATICA NELL'UFFICIO	35.000	
R 601	COLLEGAMENTO TRA MICRO E MAINFRAME	39.000	
BT 655	MANUALE DI TV E VIDEO COMMUNICATION	45.000	
	ELETTRONICA DI BASE E TECNOLOGIA		
201 A	CORSO DI ELETTRONICA FONDAMENTALE CON ESPERIMENTI	35.000	
204 A	ELETTRONICA INTEGRATA DIGITALE	50.000	
205 A	MANUALE PRATICO DI PROGETTAZIONE ELETTRONICA	35.000	
200 A	SISTEMI DIGITALI: MANUTENZIONE, RICERCA ED ELIMINAZIONE GUASTI	28.500	
GES262	TECNOLOGIE VLSI	70.000	
GES390	ELETTRONICA INTEGRATA DIGITALE IL LIBRO DELLE SOLUZIONI	17.000	
CE 411	LA FISICA DEI SEMICONDUTTORI	10.000	
158 PC	ELETTRONICA DI BASE I FONDAMENTI DELL'ELETTRONICA ANALOGICA	55.000	
158 CC	ELETTRONICA DIGITALE VOL. 1 DALLE PORTE LOGICHE AI CIRCUITI INTEGRATI	55.000	
158 DC	ELETTRONICA DIGITALE VOL. 2 DAI BUS AI GATE ARRAY	55.000	
158 GC	ELETTROTECNICA ELETTROSTATICA ELETTROMAGNETISMO RETI ELETTR.	55.000	

CODICE	TITOLO	PREZZO
	ELETTRONICA APPLICATA	
601 B	TIMER 555	10.000
203 A	INTRODUZIONE AI CIRCUITI INTEGRATI DIGITALI	10.000
612 P	MANUALE DEGLI SCR VOL. 1	28.000
613 P	MANUALE DI OPTOELETTRONICA	15.000
614 A	FIBRE OTTICHE	15.000
GE 403	JFET MOS E DATA BOOK	20.000
GE 404	TRANSISTOR DATA BOOK	32.000
GE 405	METODI DI PROTEZIONE CONTRO LE SOVRATENSIONI	17.000
CE 413	IL MANUALE DEGLI SCR E TRIAC	15.000
CE 421	MANUALE DEI FILTRI ATTIVI	29.000
CE 423	MANUALE DEI PLL PROGETTAZIONE DEI CIRCUITI	29.000
CE 425	MANUALE DEGLI AMPLIFICATORI OPERAZIONALI	29.000
CE 429	250 PROGETTI CON GLI AMPLIFICATORI DI NORTON	39.000
CE 431	MANUALE DEI CMOS	25.000
CE 485	IL COLLAUDO DELLE SCHEDE	18.000
BE 557	I TRASDUTTORI	43.000
BT 585	FIBRE OTTICHE	29.000
BE 578	MANUALE DI ELETTRONICA	29.000
BE 558	IL MANUALE DEL TECNICO ELETTRONICO	51.000
BE 610	GUIDA ALLA STRUMENTAZIONE ELETTRONICA	34.000
BE 619	MULTIMETRI DIGITALI	42.000
BE 639	ENCICLOPEDIA DEI CIRCUITI INTEGRATI	60.000
BE 654	MANUALE DI ELETTRONICA DEL COMPUTER	20.000
701 P	MANUALE PRATICO DEL RIPARATORE RADIO TV	29.000
705 P	IMPIEGO PRATICO DELL'OSCILLOSCOPIO	17.500
615 P	PROGETTAZIONE DI SISTEMI DI ALTOPARLANTI	21.000
CE 427	L'ELETTRONICA A STATO SOLIDO	25.000
BE 718	77 SCHEDE PER IL RIPARATORE TV	40.000
BE 723	MISURE DEI CIRCUITI ELETTRONICI	26.000
BE 721	MANUALE PRATICO DI ELETTRONICA DIGITALE	26.000
BE 684	IL MANUALE DEI CMOS	35.000
BE 731	IL MANUALE DEGLI AMPLIFICATORI OPERAZIONALI	39.000
	ELETTRONICA: MICROPROCESSORI	
310 P	NANOBOOK Z80 VOL. 1	20.000
007 A	BUGBOOK VII	17.000
314 P	TECNICHE DI INTERFACCIAMENTO DEI MICROPROCESSORI	31.000
312 P	NANOBOOK Z80 VOL. III	25.000
320 P	MICROPROCESSORI DAI CHIPS AI SISTEMI	29.000
324 P	PROGRAMMAZIONE DELLO Z80 E PROGETTAZIONE LOGICA	21.500
326 P	Z80 PROGRAMMAZIONE IN LINGUAGGIO ASSEMBLY	50.000
328 D	PROGRAMMAZIONE DELLO Z80	40.000
504 B	APPLICAZIONI DEL 6502	17.000
503 B	PROGRAMMAZIONE DEL 6502	35.000
505 B	GIOCHI CON IL 6502	19.500
G 220	8086-8088 PROGRAMMAZIONE	40.000
GY 265	ASSEMBLER PER IL 68000	70.000
CE 410	IMPIEGO DELLO Z80	23.000
158 HC	MICROPROCESSORI ARCHIT. PROGR. E INTERFAC. DEI MP DA 4 A 32 BIT	55.000
013 H	ASSEMBLER 6502	8.500
016 H	ASSEMBLER Z80	8.500
021 H	ASSEMBLER 68000	8.500
025 H	ASSEMBLER 8086-8088	8.500
029 H	ASSEMBLER 80286	8.500
GE 567	80286 ARCHITETTURA E PROGRAMMAZIONE	58.000
GY 603	80386 ARCHITETTURA E PROGRAMMAZIONE	37.000
	AUTOMAZIONE	
208 A	CONTROLLORI PROGRAMMABILI	24.000
616 P	CONTROLLO AUTOMATICO DEI SISTEMI	29.500
GES251	STRUTTURA E FUNZIONAMENTO DEI CONTROLLI NUMERICI	29.000
GES252	CONTROLLI NUMERICI: PROGRAMMAZIONE E APPLICAZIONI	28.000
G 399	30 APPLICAZIONI DI CAD	29.000
G 401	CAD/CAM & ROBOTICA	28.000
CI 414	DAL CHIP ALLA ROBOTICA	15.000
GE 547	LA PROGETTAZIONE AUTOMATICA	32.000
GE 564	ROBOTICA - Fondamenti e applicazioni	38.000

CODICE	TITOLO	PREZZO	
	DIZIONARI ENCICLOPEDICI		
DS 498	FISICA	14.000	
DS 499	MATEMATICA	14.000	
DS 522	GEOLOGIA	14.000	
DS 524	ELETTRONICA	14.000	
DS 525	ASTRONOMIA	14.000	
DS 526	CHIMICA	14.000	
DS 527	RAGIONERIA GENERALE	14.000	
DS 528	RAGIONERIA APPLICATA	14.000	
DS 529	BIOLOGIA	14.000	
DS 530	MECCANICA	14.000	
DS 531	INFORMATICA	14.000	
	ARGOMENTI VARI		
704 D	MANUALE PRATICO DI REGISTRAZIONE	10.000	
706 A	COMUNICAZIONI RADIO IN MARE	18.000	
800 H	FENDER, STORIA DI UN MITO	28.000	
CZ 748	TOTOCALCIO, ENALOTTO, TOTIP	35.000	F
	SOFTWARE E MANAGEMENT TOOLS		
CZ 469	GRAFIX - DISEGNARE CON IL PC	50.000	F
TP 606	CORSO AUTOISTRUZIONE LOTUS 1-2-3 (VERS. ITALIANA) F - MS DOS	90.000	F
TY 605	CORSO AUTOISTRUZIONE SUL SISTEMA MS DOS - FLOPPY	50.000	F
TY 640	TURBO PASCAL - LIBRERIA DI PROGRAMMI	40.000	F
TP 643	CORSO AUTOISTRUZIONE LOTUS 1-2-3 (INGLESE)	90.000	F
TP 608	BUDGET STRATEGICO (LOTUS 1-2-3)	100.000	F
TP 614	GESTIONE DELLE COMMESSE DI PRODUZIONE	100.000	F
TP 623	CONTROLLO DELLE VENDITE (CON MULTIPLAN)	100.000	F
TP 625	GESTIONE DEL PERSONALE (LOTUS 1-2-3)	100.000	F
TP 677	GESTIONE DELLE COMMESSE CON MULTIPLAN 2.0	100.000	F
TP 673	PREVENTIVO E CONSUNTIVO DEI COSTI - CON LOTUS 1-2-3 VERS. 2 E MULTIPLAN 2.0	100.000	F
TP 660	1-2-3 LIBRERIA DI MACRO	60.000	F
TY 691	SUPER SCREEN - UTILITY PER I PROGRAMMATORI	50.000	F
TY 690	PC DOCTOR UTILITY - RECOVERING DEI FILE	60.000	F
TP 644	STATISTICA A UNA E DUE DIMENSIONI	100.000	F
TP 681	ANALISI ABC CON LOTUS 1-2-3	100.000	F
TP 669	GESTIONE DELLE COMMESSE CON dBASE III PLUS	100.000	F
	MARKETING & MANAGEMENT		
M 648	PROBLEMI DI MARKETING	45.000	
M 649	DISTINTA BASE	23.000	
M 650	TECNICHE DI ANALISI FINANZIARIA	52.000	
M 647	RICERCHE DI MERCATO	72.000	
	NOVITÀ NOVEMBRE '88		
BE 737	IL MANUALE DEI FILTRI ATTIVI	34.000	
R 730	TURBO C	62.000	F
GY 662	UNIX - Architettura di sistema	65.000	
PP 672	MARKETING CON LOTUS 1 2 3	41.000	F
GY 703	LINGUAGGI DI PROGRAMMAZIONE	69.000	
BE 713	80286 HARDWARE	65.000	
M 679	DIRECT MARKETING	35.000	
M 706	TECNICHE DI MARKETING	43.000	
M 726	DIRECT MAILING	35.000	

GRUPPO EDITORIALE JACKSON

F = libro con floppy
C = libro con cassette

Per le vostre ordinazioni per corrispondenza utilizzate l'apposita cedola inserita in questa rivista.

Y0008/D

SERVIZIO LETTORI

IL GRUPPO EDITORIALE JACKSON PROMUOVE OGNI GIORNO NUOVE INIZIATIVE PER FACILITARE IL CONTINUO DIALOGO CON I PROPRI LETTORI. NATURALMENTE È IMPORTANTE CHE QUESTO SCAMBIO DI INFORMAZIONI SIA RESO IL PIÙ POSSIBILE AUTOMATICO E CHE I SUOI TEMPI SIANO SEMPRE PIÙ RISTRETTI. È CON QUESTO INTENTO CHE NASCE IL SERVIZIO LETTORI JACKSON, ORGANIZZATO IN MODO DA SODDISFARE OGNI ESIGENZA, SECONDO UN SISTEMA DI CEDOLE PRECONFIGURATE, DA INVIARE AL NOSTRO SERVIZIO MARKETING. ANZITUTTO, IL SERVIZIO LETTORI JACKSON CONSENTE DI ORDINARE, UTILIZZANDO LA CEDOLA DI COMMISSIONE LIBRERIA A FIANCO, I LIBRI DEL GRUPPO EDITORIALE JACKSON SCEGLIENDO LA MODALITÀ DI PAGAMENTO PREFERITA. UN ESTRATTO CONDENSATO DEL CATALOGO LIBRI E GRANDI OPERE JACKSON È PUBBLICATO NELLE ULTIME PAGINE DI QUESTA RIVISTA; IL CATALOGO COMPLETO PUÒ ESSERE

COMUNQUE ORDINATO, UTILIZZANDO LA CEDOLA NUMERO 3: INFORMAZIONI & AGGIORNAMENTI. QUEST'ULTIMA È LA PIÙ IMPORTANTE E PERMETTE AL LETTORE DI RICEVERE, DIRETTAMENTE A CASA PROPRIA, TUTTE LE INFORMAZIONI SULLE INIZIATIVE JACKSON CHE LO INTERESSANO: CATALOGHI, LIBRI, CAMPAGNA ABBONAMENTI CORSI DELLA DIVISIONE FORMAZIONE E ALTE TECNOLOGIE S.A.T.A., COPIE OMAGGIO DI RIVISTE E FASCICOLI DI GRANDI OPERE. QUESTO SERVIZIO CONSENTE, OLTRE CHE DI RIMANERE AGGIORNATI, ANCHE DI AGGIORNARE I COLLEGHI E GLI AMICI, POICHÈ LA CEDOLA È STUDIATA ANCHE CON QUESTO INTENTO. NON PIÙ TELEFONATE LETTERE: DA OGGI È SUFFICIENTE SPEDIRE L'APPOSITO TAGLIANDO, PER OTTENERE IN BREVISSIMO TEMPO IL MATERIALE DESIDERATO.

CEDOLA COMMISSIONE LIBRI

Se desiderate ordinare libri Jackson, utilizzate questa cedola. Indicate negli appositi spazi i codici dei libri richiesti e le quantità. Precisate anche il tipo di pagamento scelto, il vostro nome, cognome, indirizzo. **Ritagliate e spedite, riportando sulla busta l'indirizzo esatto del Gruppo Editoriale Jackson.**

RITAGLIATE E SPEDITE IN BUSTA CHIUSA

MITTENTE

COGNOME _____

NOME _____

VIA E NUMERO _____

CAP _____ CITTÀ _____

PROV. _____ TEL. (_____) _____

GRUPPO EDITORIALE JACKSON
Via Rosellini, 12
20124 Milano

CEDOLA COMMISSIONE GRANDI OPERE

Se desiderate acquistare una enciclopedia o una "Grande Opera Jackson", con pagamento in un'unica soluzione oppure informazioni per l'acquisto con formula rateale a sole L. 25.000 mensili e un semplice anticipo di L. 45.000, compilate questa cedola precisando il tipo di pagamento scelto. **Ritagliate e spedite, riportando sulla busta l'indirizzo esatto del Gruppo Editoriale Jackson.**

RITAGLIATE E SPEDITE IN BUSTA CHIUSA

MITTENTE

COGNOME _____

NOME _____

VIA E NUMERO _____

CAP _____ CITTÀ _____

PROV. _____ TEL. (_____) _____

GRUPPO EDITORIALE JACKSON
Via Rosellini, 12
20124 Milano

CEDOLA INFORMAZIONI E AGGIORNAMENTI

Se desiderate ricevere rapidamente informazioni sui prodotti e attività del Gruppo Editoriale Jackson, barrate le caselle della cedola che vi interessano. **Ritagliate e spedite, riportando sulla busta l'indirizzo esatto del Gruppo Editoriale Jackson.**

RITAGLIATE E SPEDITE IN BUSTA CHIUSA

MITTENTE

COGNOME _____

NOME _____

VIA E NUMERO _____

CAP _____ CITTÀ _____

PROV. _____ TEL. (_____) _____

GRUPPO EDITORIALE JACKSON
Via Rosellini, 12
20124 Milano

SCUOLA DI ALTE TECNOLOGIE APPLICATE JACKSON S.A.T.A.

CALENDARIO CORSI 1988

Lunedì 29 agosto Inizio corso SPECIALIZZAZIONE IN AUTOMAZIONE INDUSTRIALE E ROBOTICA 175 ore (serale)
Inizio corso MICROPROCESSORI BASE 40 ore (serale)
Lunedì 5 settembre
Inizio corso CONTROLLO E PREVENZIONE DELLE PARTI ELETTRONICHE DALLE SCARICHE ELETTROSTATICHE 24 ore (intensivo)
Inizio corso PROGRAMMAZIONE IN BASIC 40 ore (intensivo)
Inizio corso UNIX, XENIX UTENTI 80 ore (serale)
Inizio corso DBIII PLUS UTENTI 24 ore (serale)
Lunedì 12 settembre
Inizio corso EMC-COMPATIBILITÀ ELETTROMAGNETICA 24 ore (intensivo)
Inizio corso PROGRAMMAZIONE IN PASCAL-TURBOPASCAL 50 ore (intensivo)
Inizio corso CASE-COMPUTER AIDED SOFTWARE ENGINEERING 40 ore (intensivo)
Giovedì 15 settembre
Inizio corso DBIII PLUS PROGRAMMAZIONE 24 ore (serale)
Lunedì 19 settembre
Inizio corso DESK TOP PUBLISHING 40 ore (serale)
Inizio corso TRASDUTTORI, SENSORI, ATTUATORI 20 ore (serale)
Mercoledì 21 settembre
Inizio corso ELETTRONICA DIGITALE 60 ore (intensivo)
Lunedì 26 settembre
Inizio corso PROGRAMMAZIONE IN COBOL 60 ore (intensivo)
Mercoledì 28 settembre
Inizio corso CONTROLLORI LOGICI PROGRAMMABILI 40 ore (serale)
Lunedì 3 ottobre Inizio corso ARCHITETTURA SNA 32 ore (intensivo)
Inizio corso INTEGRAZIONE EDP E TLC NELL'OFFICE AUTOMATION 32 ore (intensivo)
Inizio corso APPLICAZIONI INDUSTRIALI DELLE TECNICHE LASER DI BASSA POTENZA 32 ore (intensivo)
Inizio corso VENTURA 24 ore (intensivo)
Lunedì 10 ottobre
Inizio corso APPARATI E SISTEMI PER LE RETI DI COMPUTER 40 ore (intensivo)
Inizio corso MICROPROCESSORI BASE 40 ore (intensivo)
Inizio corso PAGE MAKER 24 ore (intensivo)
Lunedì 17 ottobre Inizio corso PROGRAMMAZIONE IN C 80 ore (intensivo)
Inizio corso PROGRAMMAZIONE WINDOWS BASE 80 ore (intensivo)
Inizio corso PROCESSORI DI SEGNALE DIGITALE 60 ore (intensivo)
Inizio corso MANUSCRIPT 24 ore (intensivo)
Lunedì 24 ottobre Inizio corso ELEMENTI BASE DI ROBOTICA 20 ore (serale)
Lunedì 7 novembre Inizio corso PC/MS-DOS 24 ore
Inizio corso PIANIFICAZIONE RETICOLARE COL PC 24 ore (intensivo)
Inizio corso OFFICE COMMUNICATION 24 ore (intensivo)
Inizio corso INFOCENTER 32 ore (intensivo)
Inizio corso TECNICHE BASE E SISTEMI PER TRASMISSIONI DATI 80 ore (serale)
Inizio corso MICROPROCESSORI EVOLUTO 40 ore (serale)
Inizio corso INTRODUZIONE ALL'INTELLIGENZA ARTIFICIALE E AI SISTEMI ESPERTI 40 ore (intensivo)
Inizio corso INFORMIX/SQL 50 ore (serale)
Inizio corso ARCHITETTURA OS/2 40 ore (intensivo)
Inizio corso MICROPROCESSORI A 16 BIT 60 ore (intensivo)
Inizio corso AFFIDABILITÀ DEI CIRCUITI E DEI COMPONENTI ELETTRONICI 24 ore (intensivo)
Mercoledì 9 novembre Inizio corso WORD 24 ore
Lunedì 14 novembre
Inizio corso RETI DI COMUNICAZIONE NELLA FABBRICA AUTOMATIZZATA 20 ore (serale)
Inizio corso MODELLI PREVISIONALI COL PC 24 ore (intensivo)
Inizio corso USO DEL PC NELL'AREA PRODUZIONE 24 ore (intensivo)
Inizio corso PROGETTAZIONE DEI MODERNI CIRCUITI STAMPATI 24 ore (intensivo)
Lunedì 21 novembre
Inizio corso RETI A COMMUTAZIONE DI PACCHETTO 40 ore (intensivo)
Inizio corso PROGRAMMAZIONE IN LISP 40 ore (intensivo)
Inizio corso MODELLI DECISIONALI COL PC 24 ore (intensivo)
Inizio corso USO DEL PC NELL'AREA MARKETING 24 ore (intensivo)
Lunedì 28 novembre Inizio corso MULTIPLAN 24 ore
Inizio corso PROGRAMMAZIONE IN PROLOG 40 ore (intensivo)
Inizio corso AUTO-CAD 32 ore (serale)
Inizio corso SERVIZI A VALORE AGGIUNTO SULLE RETI X25 24 ore (intensivo)
Mercoledì 30 novembre Inizio corso LOTUS 1-2-3 24 ore
Lunedì 12 dicembre
Seminario con WORK-SHOP SUI LINGUAGGI DELLA IV GENERAZIONE 24 ore (intensivo)
Inizio corso IL MODELLO OSI 32 ore (intensivo)
Inizio corso SYMPHONY 40 ore (intensivo)
Inizio corso OTTIMIZZAZIONE E DEBUGGING "C" 40 ore (intensivo)
Inizio corso MICROPROCESSORI EVOLUTO 40 ore (intensivo)

SCUOLA DI ALTE TECNOLOGIE APPLICATE

Per le modalità di iscrizione e richiesta di programmi dettagliati, telefonare alla DIVISIONE FORMAZIONE E PRODOTTI PER LA DIDATTICA del Gruppo Editoriale Jackson
Via Imperia 2 Milano
Telefono 8467500

S.A.T.A.

SÌ INVIATEMI I VOLUMI SOTTOELENCATI:

INDICARE CHIARAMENTE CODICE, QUANTITÀ E PREZZO DEI VOLUMI RICHIESTI

Codice	Quantità	Prezzo		Codice	Quantità	Prezzo

Ordine minimo L. 30.000 + L. 3.500 per contributo fisso spese di spedizione

☐ **Non sono abbonato** a riviste Jackson

☐ **Sono abbonato** alla/e seguente/i rivista/e Jackson _____

_____ e ho quindi diritto allo sconto del **10%**

MODALITÀ DI PAGAMENTO

☐ Allego assegno n. _____ di L. _____
della Banca _____

☐ Ho effettuato il pagamento di L. _____ a mezzo:
☐ vaglia postale ☐ vaglia telegrafico ☐ versamento sul c/c postale n. 11666203 intestato a Gruppo Editoriale Jackson SpA Milano e allego fotocopia della ricevuta.

☐ Pagherò al postino l'importo di L. _____
al ricevimento dell'opera.

☐ Vi autorizzo ad addebitare l'importo di L. _____
sulla carta di credito: ☐ Visa ☐ American Express ☐ Diners Club ☐ Carta SI
conto n. _____ data di scadenza _____

☐ Richiedo l'emissione della fattura (formula riservata alle aziende) e
comunico il numero di Partita IVA _____

DATA _____ FIRMA _____

SI INVIATEMI LE SEGUENTI "GRANDI OPERE JACKSON"

☐ **EI - Enciclopedia di Elettronica e Informatica** 10 volumi	L. 476.000
☐ **SOFTWARE** 5 volumi	L. 236.000
☐ **DEI - Dizionario di Elettronica e Informatica** 10 volumi	L. 276.000
☐ **Enciclopedia Monografica di Elettronica e Informatica** 2 volumi	L. 116.000
☐ **ABC Personal Computer** 4 volumi	L. 136.000
☐ **VIDEOBASIC MSX** 20 lezioni + 20 cassette	L. 176.000
☐ **VIDEOBASIC SPECTRUM** 20 lezioni + 20 cassette	L. 176.000
☐ **VIDEOBASIC** C64/C128/64PC 20 lezioni + 20 cassette	L. 176.000
☐ **VIDEOBASIC** C64/C128/64PC 20 lezioni + 10 floppy	L. 176.000
☐ **VIDEOBASIC** C16/PUS4 20 lezioni + 20 cassette	L. 176.000
☐ **VIDEOBASIC** VIC20 20 lezioni + 20 cassette	L. 176.000
☐ **CORSO DI GRAFICA** C64/C128/64PC 10 lez. + 10 cass.	L. 96.000
☐ **A SCUOLA DI SCACCHI** C64/C128/64PC 10 lezioni + 10 cassette	L. 96.000
☐ **7 NOTE BIT** C64/C128/64PC 15 lezioni + 15 cassette	L. 156.000
☐ **LABORATORIO DI ELETTRONICA** 5 volumi (disp. da giugno 1988)	L. 236.000
☐ **BYTES** 6 volumi (disp. da giugno 1988)	L. 276.000

☐ **PAGAMENTO RATEALE**

MODALITÀ DI PAGAMENTO IN UN'UNICA SOLUZIONE

☐ Allego assegno n. _____ di L. _____
della Banca _____

☐ Ho effettuato il pagamento di L. _____ a mezzo:
☐ vaglia postale ☐ vaglia telegrafico ☐ versamento sul c/c postale n. 11666203 intestato a Gruppo Editoriale Jackson SpA Milano e allego fotocopia della ricevuta.

☐ Pagherò al postino l'importo di L. _____
al ricevimento dell'opera.

☐ Vi autorizzo ad addebitare l'importo di L. _____
sulla carta di credito: ☐ Visa ☐ American Express ☐ Diners Club ☐ Carta SI
conto n. _____ data di scadenza _____

☐ Richiedo l'emissione della fattura (formula riservata alle aziende) e
comunico il numero di Partita IVA _____

DATA _____ FIRMA _____

CEDOLA INFORMAZIONI E AGGIORNAMENTI

☐ Desidero riceve informazioni sulle **riviste** JACKSON
☐ Desidero ricevere una copia saggio della **rivista** JACKSON _____
(indicare titolo)
☐ Desidero ricevere il **catalogo libri tecnici** JACKSON
☐ Desidero ricevere il **catalogo libri scolastici** JACKSON
☐ Desidero ricevere il **catalogo grandi opere** JACKSON e una copia saggio _____
(indicare titolo)
☐ Desidero ricevere informazioni per l'acquisto in **forma rateale delle Grandi Opere** JACKSON e una copia saggio _____
(indicare titolo)
☐ Desidero ricevere il **catalogo occasioni** JACKSON

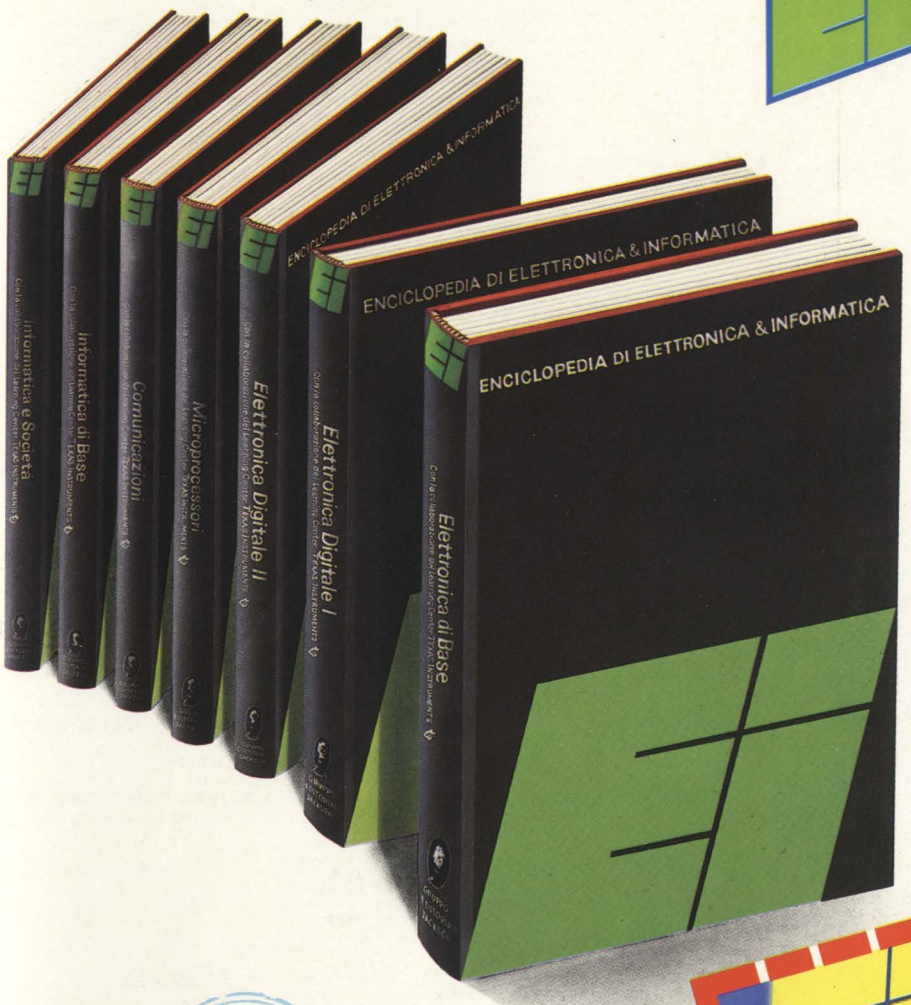
Desidero ricevere maggiori informazioni sulla **Divisione Formazione e Prodotti per la Didattica**
☐ Area ELETTRONICA E MICROPROCESSORI
☐ Area TELECOMUNICAZIONI E TELEMATICA
☐ Area AUTOMAZIONE INDUSTRIALE E ROBOTICA
☐ Area INFORMATICA E INTELLIGENZA ARTIFICIALE
☐ Area ALTE TECNOLOGIE SPECIALI
☐ Laboratorio di INFORMATICA
☐ Laboratorio di TELEMATICA
☐ Laboratorio di OFFICE AUTOMATION
☐ Laboratorio di ELETTRONICA
☐ Laboratorio di MICROPROCESSORI
☐ Laboratorio di AUTOMAZIONE INDUSTRIALE

DAL 12 OTTOBRE IN EDICOLA

UN GRANDE RITORNO

Nuova edizione

ENCICLOPEDIA DI ELETTRONICA INFORMATICA E COMUNICAZIONI



Fascicolo dopo fascicolo, in sole 30 settimane, si completano i 7 grandi volumi di EI: l'Enciclopedia Jackson di Elettronica, Informatica e Comunicazioni.
EI è l'appuntamento settimanale in edicola con la tecnologia più avanzata; ogni fascicolo l'emozione di nuove scoperte; ogni pagina la certezza di vivere da protagonista la rivoluzione tecnologica in atto. EI, oggi in edicola, domani nella tua libreria.

GRUPPO EDITORIALE JACKSON
DIVISIONE GRANDI OPERE

Il 1° e il 2° fascicolo a sole L. 500

È JACKSON

LA TUA ENCICLOPEDIA

FINALMENTE IN EDICOLA UNA VERA RIVISTA DI 96 PAGINE PIÙ DISCHETTO SOFTWARE PER IL TUO COMPUTER.

LINGUAGGI

BASIC • C • FORTH • MODULA 2

APPLICAZIONI

GRAFICA • MUSICA • DESK – TOP PUBLISHING

RECENSIONI

HARDWARE E SOFTWARE

...E IN PIÙ

TOOLS - GAMES... E ALTRO ANCORA PER IL PROFESSIONISTA E L'HOBBISTA.

Nel nuovo numero in edicola dal 1 novembre

- SMAU 1988
- Lettore di files .BMAP in Amiga Basic
- Le stampanti Amiga
- Giocare con il controller del disco
- Icone per tutti
- Form ILBM
- Kindwords
- Pixmate

e tanti altri interessanti articoli e programmi

AMIGA MAGAZINE

IN EDICOLA SCEGLI IL MEGLIO, SCEGLI JACKSON